Project-15 : Fuzzy-CNN for Medical Image Segmentation

Objective: Develop a model that combines fuzzy logic with CNNs to improve the segmentation of medical images, such as MRI scans or X-rays, where boundaries between tissues may be unclear.

Preprocess the medical images

In [22]:
```
import os  # Import for operating system interactions
import numpy as np  # Import NumPy for array and numerical operations
from PIL import Image  # Import PIL for image processing

# Function to list all files in a specific subfolder
def list_files(base_path, subfolder):
    folder_path = os.path.join(base_path, subfolder)  # Create the full path to the folder
    if not os.path.exists(folder_path):  # Check if the folder exists
        print(f"Error: Path not found -> {folder_path}")  # Print error if path doesn't exist
        return []  # Return an empty list if the folder is missing
    # Return a list of file paths in the folder
    return [os.path.join(folder_path, file) for file in os.listdir(folder_path) if os.path.isfile(os.path.join(folder_

# Function to load and normalize image pixel values
def load_and_normalize_image(filepath):
    image = Image.open(filepath).convert('L')  # Open the image and convert to grayscale
    pixel_data = np.array(image, dtype=np.float32)  # Convert image to a NumPy array of floats
    min_val, max_val = pixel_data.min(), pixel_data.max()  # Find minimum and maximum pixel values
    normalized = (pixel_data - min_val) / (max_val - min_val)  # Normalize pixels to range [0, 1]
    return normalized  # Return the normalized array

# Define the base path to the main folder
base_path = '/Users/pro/Downloads/DRIVE'

# Get all training image and mask file paths
training_image_paths = list_files(base_path, 'training/images')
training_mask_paths = list_files(base_path, 'training/mask')

# Get all test image and mask file paths
test_image_paths = list_files(base_path, 'test/images')
test_mask_paths = list_files(base_path, 'test/mask')
```

```
# Debugging: Print the paths of training images and masks
print("Training Images:", training_image_paths)
print("Training Masks:", training_mask_paths)

# Preprocess the first training image if images exist
if training_image_paths:
    sample_image_path = training_image_paths[0]  # Get the first training image path
    preprocessed_image = load_and_normalize_image(sample_image_path)  # Load and normalize the image
    print(f"Processed Image Shape: {preprocessed_image.shape}")  # Print the shape of the processed image
else:
    print("No training images found.")  # Print a message if no training images are available
```

```
Training Images: ['/Users/pro/Downloads/DRIVE/training/images/29_training.tif', '/Users/pro/Downloads/DRIVE/training/
images/28_training.tif', '/Users/pro/Downloads/DRIVE/training/images/22_training.tif', '/Users/pro/Downloads/DRIVE/tr
aining/images/25_training.tif', '/Users/pro/Downloads/DRIVE/training/images/30_training.tif', '/Users/pro/Downloads/D
RIVE/training/images/37_training.tif', '/Users/pro/Downloads/DRIVE/training/images/40_training.tif', '/Users/pro/Down
loads/DRIVE/training/images/36_training.tif', '/Users/pro/Downloads/DRIVE/training/images/31_training.tif', '/Users/p
ro/Downloads/DRIVE/training/images/24_training.tif', '/Users/pro/Downloads/DRIVE/training/images/23_training.tif', '/
Users/pro/Downloads/DRIVE/training/images/33_training.tif', '/Users/pro/Downloads/DRIVE/training/images/34_training.t
if', '/Users/pro/Downloads/DRIVE/training/images/21_training.tif', '/Users/pro/Downloads/DRIVE/training/images/26_tra
ining.tif', '/Users/pro/Downloads/DRIVE/training/images/27_training.tif', '/Users/pro/Downloads/DRIVE/training/image
s/35_training.tif', '/Users/pro/Downloads/DRIVE/training/images/32_training.tif', '/Users/pro/Downloads/DRIVE/trainin
g/images/38_training.tif', '/Users/pro/Downloads/DRIVE/training/images/39_training.tif']
Training Masks: ['/Users/pro/Downloads/DRIVE/training/mask/37_training_mask.gif', '/Users/pro/Downloads/DRIVE/trainin
g/mask/32_training_mask.gif', '/Users/pro/Downloads/DRIVE/training/mask/36_training_mask.gif', '/Users/pro/Downloads/
DRIVE/training/mask/33_training_mask.gif', '/Users/pro/Downloads/DRIVE/training/mask/35_training_mask.gif', '/Users/p
ro/Downloads/DRIVE/training/mask/29_training_mask.gif', '/Users/pro/Downloads/DRIVE/training/mask/30_training_mask.gi
f', '/Users/pro/Downloads/DRIVE/training/mask/34_training_mask.gif', '/Users/pro/Downloads/DRIVE/training/mask/28_tra
ining_mask.gif', '/Users/pro/Downloads/DRIVE/training/mask/31_training_mask.gif', '/Users/pro/Downloads/DRIVE/trainin
g/mask/38_training_mask.gif', '/Users/pro/Downloads/DRIVE/training/mask/24_training_mask.gif', '/Users/pro/Downloads/
DRIVE/training/mask/21_training_mask.gif', '/Users/pro/Downloads/DRIVE/training/mask/39_training_mask.gif', '/Users/p
ro/Downloads/DRIVE/training/mask/25_training_mask.gif', '/Users/pro/Downloads/DRIVE/training/mask/40_training_mask.gi
f', '/Users/pro/Downloads/DRIVE/training/mask/26_training_mask.gif', '/Users/pro/Downloads/DRIVE/training/mask/23_tra
ining_mask.gif', '/Users/pro/Downloads/DRIVE/training/mask/27_training_mask.gif', '/Users/pro/Downloads/DRIVE/trainin
g/mask/22_training_mask.gif']
Processed Image Shape: (584, 565)
```

Fuzzify pixel intensities using membership functions.

In [10]:
```
import numpy as np  # Import NumPy for array operations

# Define the Gaussian Membership Function
def gaussian_membership(x, c, sigma):
    # Compute membership value using the Gaussian formula
```

```
    return np.exp(-((x - c)**2) / (2 * sigma**2))

# Function to fuzzify an image using Gaussian membership functions
def fuzzify_image(image, centers, sigmas):
    """
    Fuzzify a 2D or 3D image using Gaussian membership functions.
    Args:
        image: Input image (2D or 3D array).
        centers: List of center values for Gaussian functions.
        sigmas: List of standard deviations for Gaussian functions.
    Returns:
        fuzzified_image: Output fuzzified image with membership values.
    """
    if image.ndim == 3:  # Check if the image has multiple channels
        height, width, channels = image.shape  # Get image dimensions
        fuzzified_image = np.zeros((height, width, len(centers), channels))  # Initialize 4D array for fuzzification
        for c in range(channels):  # Loop through each channel
            for k, (center, sigma) in enumerate(zip(centers, sigmas)):  # Loop through each fuzzy set
                # Compute Gaussian membership for each pixel
                fuzzified_image[:, :, k, c] = np.exp(-((image[:, :, c] - center) ** 2) / (2 * sigma ** 2))
        return fuzzified_image.mean(axis=-1)  # Average membership values across channels
    else:  # For single-channel (grayscale) images
        height, width = image.shape  # Get image dimensions
        fuzzified_image = np.zeros((height, width, len(centers)))  # Initialize 3D array for fuzzification
        for k, (center, sigma) in enumerate(zip(centers, sigmas)):  # Loop through each fuzzy set
            # Compute Gaussian membership for each pixel
            fuzzified_image[:, :, k] = np.exp(-((image - center) ** 2) / (2 * sigma ** 2))
        return fuzzified_image  # Return the fuzzified image

# Define centers (mean values) for the Gaussian membership functions
centers = [0.2, 0.5, 0.8]  # Low, Medium, High intensity centers
sigmas = [0.1, 0.1, 0.1]   # Standard deviations for the Gaussian functions

# Fuzzify the preprocessed image (normalized grayscale image)
fuzzified_image = fuzzify_image(preprocessed_image, centers, sigmas)

# Output the shape of the fuzzified image
print(fuzzified_image.shape)  # Prints (height, width, num_fuzzy_sets), e.g., (584, 565, 3)
```

```
(584, 565, 3)
```

Train a CNN on the fuzzified images.

In [11]:
```
import numpy as np  # Import NumPy for numerical operations

# Define a 2D convolution function for multi-channel images
def convolve2d_multichannel(image, kernel, stride=1, padding=0):
    """
    Perform 2D convolution on multi-channel images (e.g., RGB).
    Args:
        image: Input image (height, width, num_channels).
        kernel: Convolution kernel (kernel_height, kernel_width).
        stride: Step size for moving the kernel.
        padding: Border padding for the image.
    Returns:
        Convolved output image (output_height, output_width, num_channels).
    """
    # Add zero-padding to the image based on the given padding size
    image = np.pad(image, ((padding, padding), (padding, padding), (0, 0)), mode='constant', constant_values=0)

    # Get dimensions of the padded image
    image_height, image_width, num_channels = image.shape  # Height, width, and number of channels
    kernel_height, kernel_width = kernel.shape  # Dimensions of the kernel

    # Calculate output dimensions after convolution
    output_height = (image_height - kernel_height) // stride + 1  # Height of the output image
    output_width = (image_width - kernel_width) // stride + 1  # Width of the output image

    # Initialize an empty array for the output image
    output = np.zeros((output_height, output_width, num_channels))  # Shape: (output_height, output_width, num_channe

    # Loop through each channel of the image
    for c in range(num_channels):  # Iterate over each channel
        # Perform convolution operation for the current channel
        for i in range(0, output_height * stride, stride):  # Iterate over image rows
            for j in range(0, output_width * stride, stride):  # Iterate over image columns
                # Apply the kernel to the corresponding region and sum the result
                output[i // stride, j // stride, c] = np.sum(image[i:i+kernel_height, j:j+kernel_width, c] * kernel)

    return output  # Return the convolved output image
```

In [12]:
```
# ReLU activation function
def relu(x):
    return np.maximum(0, x)
```

In [13]:
```
# Define a max pooling function for multi-channel images
def max_pooling_multichannel(image, pool_size=2, stride=2):
    # Get the dimensions of the input image
    image_height, image_width, num_channels = image.shape  # Height, width, and number of channels

    # Calculate the output dimensions after pooling
    output_height = (image_height - pool_size) // stride + 1  # Height of the pooled output
    output_width = (image_width - pool_size) // stride + 1  # Width of the pooled output

    # Initialize the output feature map with zeros
    output = np.zeros((output_height, output_width, num_channels))  # Shape: (output_height, output_width, num_channe

    # Perform max pooling for each channel
    for c in range(num_channels):  # Loop through each channel
        for i in range(output_height):  # Iterate over rows of the output
            for j in range(output_width):  # Iterate over columns of the output
                # Extract the current region of the image for pooling
                region = image[i * stride:i * stride + pool_size, j * stride:j * stride + pool_size, c]
                # Find the maximum value in the region and store it in the output
                output[i, j, c] = np.max(region)

    return output  # Return the pooled output
```

In [14]:
```
# Fully connected layer (dense layer)
def fully_connected(input_data, weights, bias):
    return np.dot(input_data, weights) + bias
```

In [15]:
```
# Define the softmax function for classification outputs
def softmax(x):
    exp_x = np.exp(x - np.max(x))  # Compute exponentials with stability trick to avoid overflow
    return exp_x / np.sum(exp_x, axis=-1, keepdims=True)  # Normalize to get probabilities across classes
```

In [16]:
```
# Define the Mean Squared Error (MSE) loss function
def mse_loss(predictions, targets):
    """
    Compute Mean Squared Error (MSE) loss.
    Args:
        predictions (numpy.ndarray): Predicted values (e.g., probabilities or logits).
        targets (numpy.ndarray): True values (e.g., one-hot encoded or raw values).
    Returns:
        float: Computed MSE loss value.
    """
```

```python
    m = targets.shape[0]  # Get the number of examples (batch size)
    loss = np.sum((predictions - targets) ** 2) / (2 * m)  # Apply the MSE formula
    return loss  # Return the computed loss
```

In [17]:
```python
# Define backpropagation for a fully connected layer
def backprop_fc_layer(input_data, output_error, weights, learning_rate=0.01):
    """
    Perform backpropagation for a fully connected layer.
    Args:
        input_data: Input data to the layer (1, flattened_size).
        output_error: Gradient of loss w.r.t. layer output (1, num_outputs).
        weights: Current weights of the layer.
        learning_rate: Learning rate for updating weights.
    Returns:
        Updated weights and biases after applying gradient descent.
    """
    # Compute gradient of weights: input transposed dot product with output error
    d_weights = np.dot(input_data.T, output_error)  # Shape: (flattened_size, num_outputs)

    # Compute gradient of biases: sum output error across samples
    d_bias = np.sum(output_error, axis=0, keepdims=True)  # Shape: (1, num_outputs)

    # Update weights using gradient descent
    weights -= learning_rate * d_weights  # Adjust weights by scaled gradient

    # Update bias using gradient descent
    bias = -learning_rate * d_bias  # Adjust bias by scaled gradient (you might want to pass bias explicitly)

    return weights, bias  # Return updated weights and biases
```

In [18]:
```python
# Initialize random filters (kernels) globally
kernel1 = np.random.randn(3, 3)  # First convolution kernel (randomly initialized)
kernel2 = np.random.randn(3, 3)  # Second convolution kernel (randomly initialized)

# CNN training function
def cnn_train(fuzzified_image, target_mask, num_epochs=10, learning_rate=0.001):
    # Initialize random filters (kernels) locally
    kernel1 = np.random.randn(3, 3)  # Random initialization for kernel1
    kernel2 = np.random.randn(3, 3)  # Random initialization for kernel2

    # Compute dimensions of the feature map after each layer
    input_height, input_width, num_channels = fuzzified_image.shape  # Input dimensions
    after_pool1_height = (input_height - 3 + 1) // 2  # Height after first pooling
```

```
    after_pool1_width = (input_width - 3 + 1) // 2   # Width after first pooling
    after_pool2_height = (after_pool1_height - 3 + 1) // 2  # Height after second pooling
    after_pool2_width = (after_pool1_width - 3 + 1) // 2   # Width after second pooling

    # Calculate flattened feature map size
    flattened_size = after_pool2_height * after_pool2_width * num_channels

    # Initialize weights and biases for the fully connected layer
    fc_weights = np.random.randn(flattened_size, 10)  # Weights for fully connected layer
    fc_bias = np.zeros((1, 10))  # Bias for fully connected layer

    # Training loop for the given number of epochs
    for epoch in range(num_epochs):
        # Forward Pass: Convolution, ReLU, and Max Pooling
        x = convolve2d_multichannel(fuzzified_image, kernel1)  # First convolution
        x = relu(x)  # Apply ReLU activation
        x = max_pooling_multichannel(x)  # First max pooling

        x = convolve2d_multichannel(x, kernel2)  # Second convolution
        x = relu(x)  # Apply ReLU activation
        x = max_pooling_multichannel(x)  # Second max pooling

        # Flatten the feature map for the fully connected layer
        flattened = x.flatten().reshape(1, -1)  # Reshape into a 1D array

        # Fully connected layer forward pass
        fc_output = fully_connected(flattened, fc_weights, fc_bias)  # Compute FC layer output

        # Softmax activation for output
        predictions = softmax(fc_output)  # Convert logits to probabilities

        # Calculate loss (Mean Squared Error)
        loss = mse_loss(predictions, target_mask)  # Compute the MSE loss

        # Backpropagation for the fully connected layer
        output_error = predictions - target_mask  # Compute gradient of loss w.r.t FC output
        fc_weights, fc_bias = backprop_fc_layer(flattened, output_error, fc_weights, learning_rate)  # Update weights

        # Print the loss at each epoch
        print(f'Epoch {epoch + 1}/{num_epochs}, Loss: {loss}')  # Display progress

    # Return the final weights and biases of the fully connected layer
    return fc_weights, fc_bias
```

In [19]:
```
# Example target mask for training
target_mask = np.zeros((1, 10))
target_mask[0, 3] = 1  # Assume class index 3 is the correct class

# Train the CNN model
cnn_train(fuzzified_image, target_mask, num_epochs=10, learning_rate=0.001)
```

```
Epoch 1/10, Loss: 0.9880225077336462
Epoch 2/10, Loss: 0.9576941430225838
Epoch 3/10, Loss: 0.8811030525402959
Epoch 4/10, Loss: 0.7900513806759587
Epoch 5/10, Loss: 0.7556679825887177
Epoch 6/10, Loss: 0.7505788553521758
Epoch 7/10, Loss: 0.7499970421695191
Epoch 8/10, Loss: 0.7498755573080977
Epoch 9/10, Loss: 0.7497423978587601
Epoch 10/10, Loss: 0.7494913406745703
```

Out[19]:
```
(array([[ 0.18663723, -0.43192835,  0.62251832, ..., -0.86900559,
          0.23392835, -1.73231429],
        [ 0.96589655, -1.40289967,  0.01570025, ..., -0.03861078,
          1.76953623, -1.71505445],
        [ 0.70145913, -0.93775833,  1.13752878, ...,  0.30714502,
         -0.30863337, -0.2393441 ],
        ...,
        [-0.80067368,  1.29736664, -0.88497382, ...,  1.0417221 ,
          1.05210144,  0.41969502],
        [ 1.34139933, -1.94748505,  0.74678467, ..., -0.1077711 ,
         -0.63357223, -0.89744965],
        [ 0.35964011,  0.40382343,  0.5723985 , ...,  0.50579528,
         -1.48032475,  1.3799546 ]]),
 array([[-7.68173360e-08, -3.93757044e-29, -4.99226714e-04,
          1.00000000e-03, -1.06839948e-23, -4.99754416e-04,
         -1.94101741e-25, -9.42052962e-07, -7.71817052e-39,
         -1.00023921e-22]]))
```

Use the CNN to segment the image into different regions

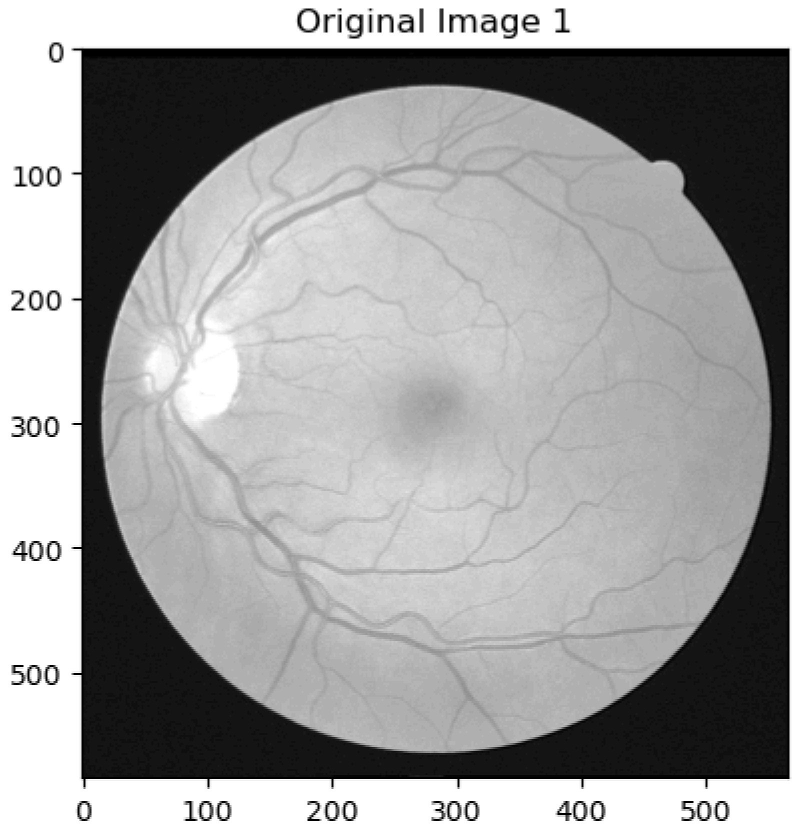
In [23]:
```
def cnn_inference(fuzzified_image, fc_weights, fc_bias, kernel1, kernel2, pool_size=2):
    """
    Perform inference on a fuzzified image using the trained CNN model.
    Args:
        fuzzified_image: Preprocessed and fuzzified input image.
        fc_weights: Trained weights for the fully connected layer.
```

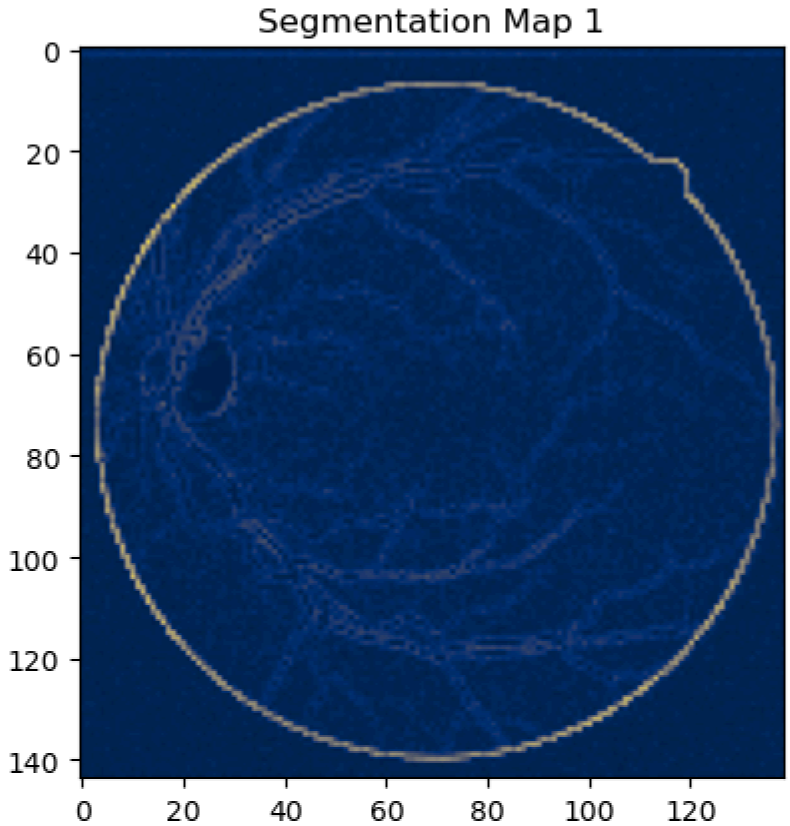
```
        fc_bias: Trained bias for the fully connected layer.
        kernel1, kernel2: Trained filters (kernels) for convolution layers.
        pool_size: Size of the pooling layer (default 2x2).
    Returns:
        The predicted class probabilities for the input image.
    """
    # Step 1: Convolution and pooling (Convolution -> ReLU -> Max Pooling)
    x = convolve2d_multichannel(fuzzified_image, kernel1)  # Convolve with the first kernel
    x = relu(x)  # Apply ReLU activation
    x = max_pooling_multichannel(x, pool_size)  # Apply max pooling to down-sample

    x = convolve2d_multichannel(x, kernel2)  # Convolve with the second kernel
    x = relu(x)  # Apply ReLU activation
    x = max_pooling_multichannel(x, pool_size)  # Apply max pooling to further down-sample

    # Step 2: Flatten the feature map (for fully connected layer)
    flattened = x.flatten().reshape(1, -1)  # Flatten the output of the convolutions into a vector

    # Step 3: Fully connected layer (Dense layer)
    fc_output = fully_connected(flattened, fc_weights, fc_bias)  # Perform the fully connected operation

    # Step 4: Apply softmax to get class probabilities
    predictions = softmax(fc_output)  # Apply softmax to get class probabilities

    return predictions
```

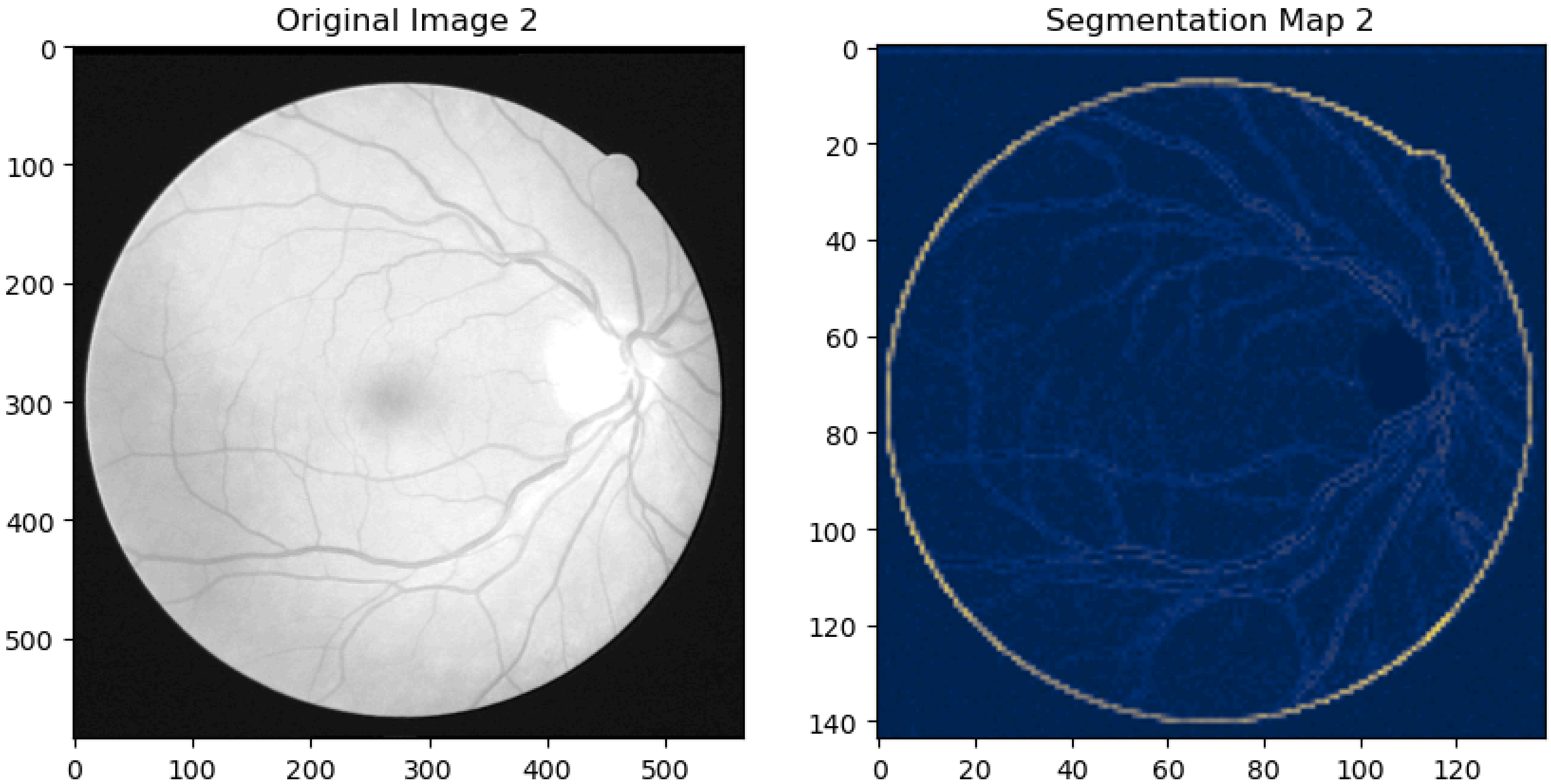
In [20]:
```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

# Function to load, normalize, and increase intensity of the image
def load_and_normalize_image(filepath, intensity_factor=1.5):
    image = Image.open(filepath).convert('L')  # Load and convert the image to grayscale
    pixel_data = np.array(image, dtype=np.float32)  # Convert image to a numpy array

    # Normalize pixel values to the range [0, 1]
    pixel_data /= 255.0  # Normalize pixel values to [0, 1]

    # Increase intensity by scaling
    pixel_data *= intensity_factor  # Increase intensity by multiplying with the factor
    pixel_data = np.clip(pixel_data, 0, 1)  # Ensure pixel values are within [0, 1]

    # Apply gamma correction for contrast adjustment
```

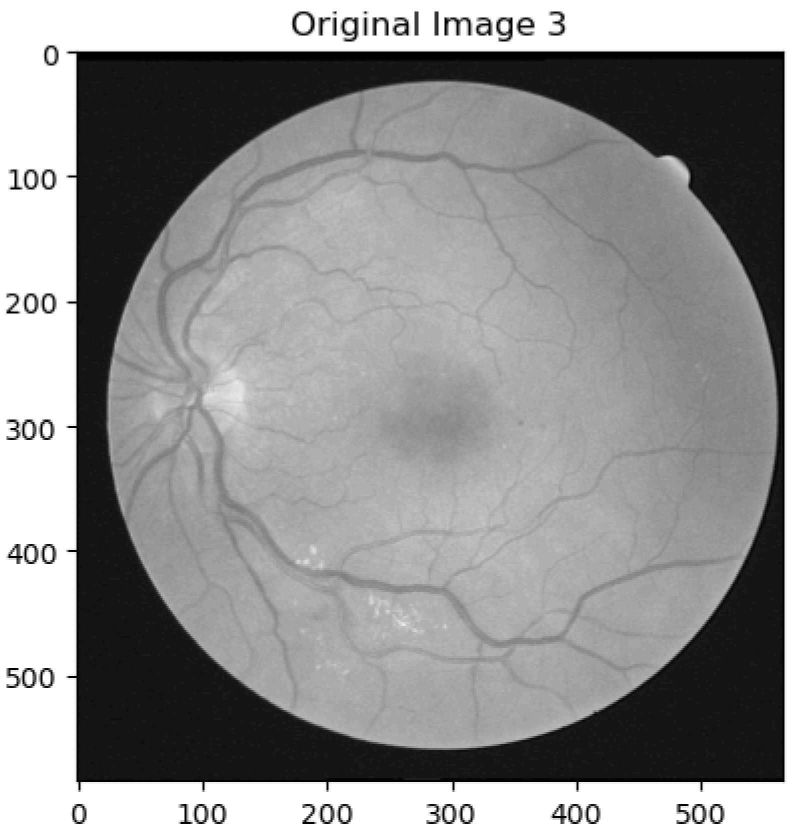
```
    pixel_data = np.power(pixel_data, 0.8)  # Apply gamma correction to adjust contrast

    # Ensure the image has 3 channels (for compatibility with convolution function)
    pixel_data = np.expand_dims(pixel_data, axis=-1)  # Add a channel dimension (height, width, 1)
    pixel_data = np.repeat(pixel_data, 3, axis=-1)  # Repeat the single channel to make it RGB (height, width, 3)

    return pixel_data  # Return the processed image

# Laplacian kernel for edge detection
laplacian_kernel = np.array([[0, -1, 0],  # Define the Laplacian kernel
                              [-1, 4, -1],
                              [0, -1, 0]])

# Function to create segmentation map without fully connected layers
def create_segmentation_map_without_fc(image, kernel1, kernel2):
    x = convolve2d_multichannel(image, kernel1)  # Apply first convolution using kernel1
    x = relu(x)  # Apply ReLU activation function
    x = max_pooling_multichannel(x)  # Apply max pooling

    x = convolve2d_multichannel(x, kernel2)  # Apply second convolution using kernel2
    x = relu(x)  # Apply ReLU activation function
    x = max_pooling_multichannel(x)  # Apply second max pooling

    # Normalize for visualization
    x_min = np.min(x)  # Find the minimum value in the feature map
    x_max = np.max(x)  # Find the maximum value in the feature map
    x_normalized = (x - x_min) / (x_max - x_min)  # Normalize the feature map to the range [0, 1]

    return x_normalized[..., 0]  # Return the first channel (grayscale) of the normalized map

# Paths to test images from the DRIVE dataset
test_image_paths = [
    '/Users/pro/Downloads/DRIVE/test/images/01_test.tif',  # List of paths to test images
    '/Users/pro/Downloads/DRIVE/test/images/02_test.tif',
    '/Users/pro/Downloads/DRIVE/test/images/03_test.tif',
    '/Users/pro/Downloads/DRIVE/test/images/04_test.tif',
    '/Users/pro/Downloads/DRIVE/test/images/05_test.tif',
    '/Users/pro/Downloads/DRIVE/test/images/06_test.tif',
    '/Users/pro/Downloads/DRIVE/test/images/07_test.tif',
    '/Users/pro/Downloads/DRIVE/test/images/08_test.tif',
    '/Users/pro/Downloads/DRIVE/test/images/09_test.tif',
    '/Users/pro/Downloads/DRIVE/test/images/10_test.tif'
]
```

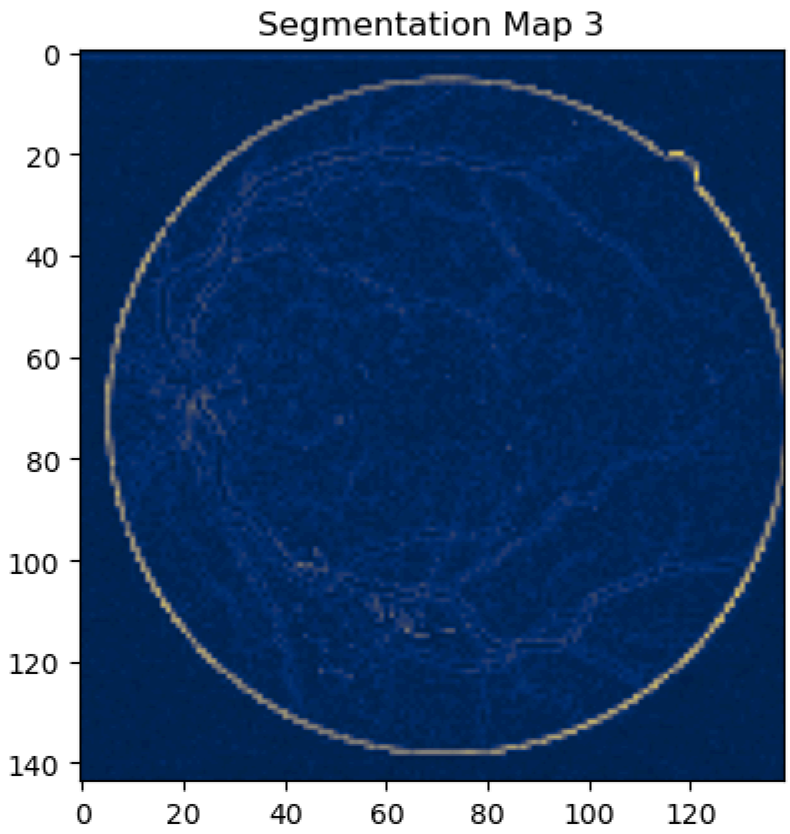
```python
# Example kernels for testing
kernel1 = np.random.rand(3, 3)  # Random kernel1 for testing
kernel2 = laplacian_kernel  # Use Laplacian kernel2 for edge detection

# Loop through test images
for i, test_image_path in enumerate(test_image_paths):
    # Load and preprocess the test image with increased intensity
    test_image = load_and_normalize_image(test_image_path, intensity_factor=1.5)  # Process each test image

    # Create the segmentation map without fully connected layers
    segmentation_map = create_segmentation_map_without_fc(test_image, kernel1, kernel2)  # Get segmentation map

    # Visualize the results
    plt.figure(figsize=(10, 5))  # Create a figure to display images side by side
    plt.subplot(1, 2, 1)  # Create the first subplot for the original image
    plt.title(f"Original Image {i+1}")  # Set the title for the original image
    plt.imshow(test_image[..., 0], cmap='gray')  # Display the grayscale version of the test image
    plt.subplot(1, 2, 2)  # Create the second subplot for the segmentation map
    plt.title(f"Segmentation Map {i+1}")  # Set the title for the segmentation map
    plt.imshow(segmentation_map, cmap='cividis')  # Display the segmentation map using 'cividis' colormap
    plt.show()  # Show the plots
```

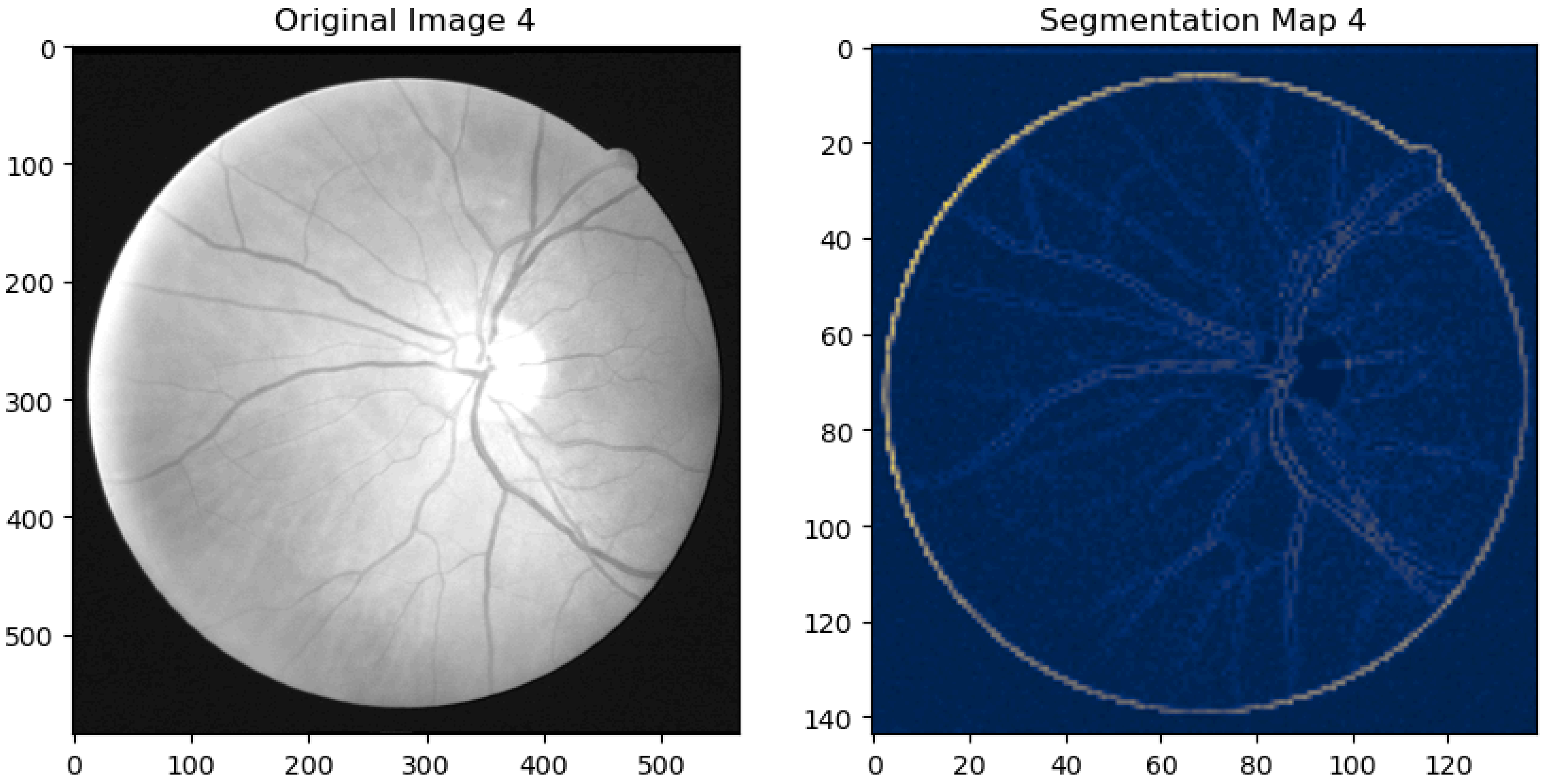

Original Image 1
0
100
200
300
400
500
0
100
200
300
400
500

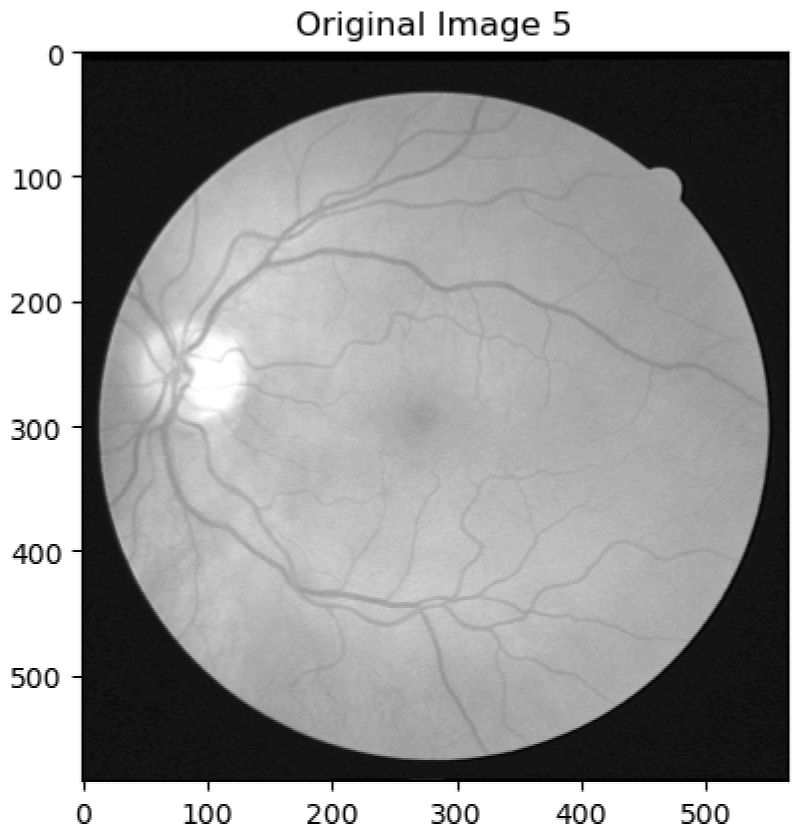

Segmentation Map 1
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120

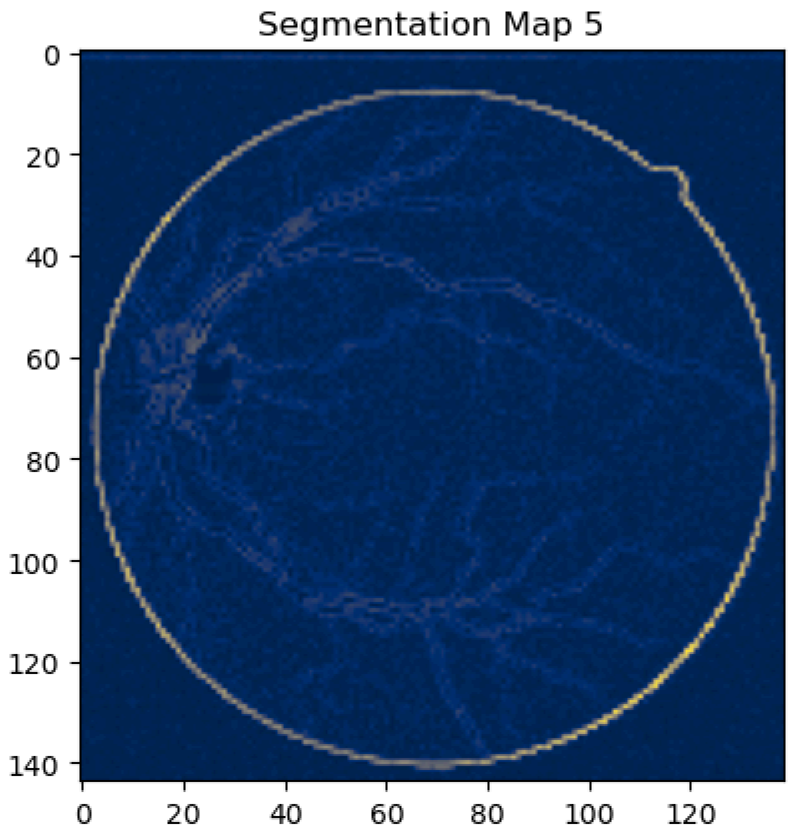

Original Image 2
0
100
200
300
400
500
0
100
200
300
400
500
Segmentation Map 2
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120

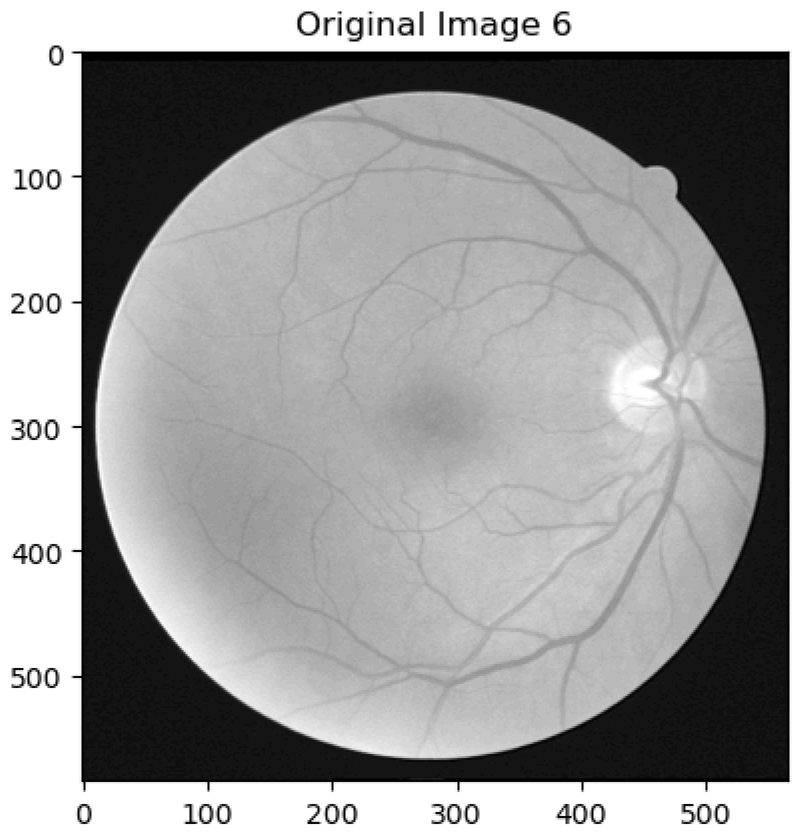

Original Image 3
0
100
200
300
400
500
0
100
200
300
400
500

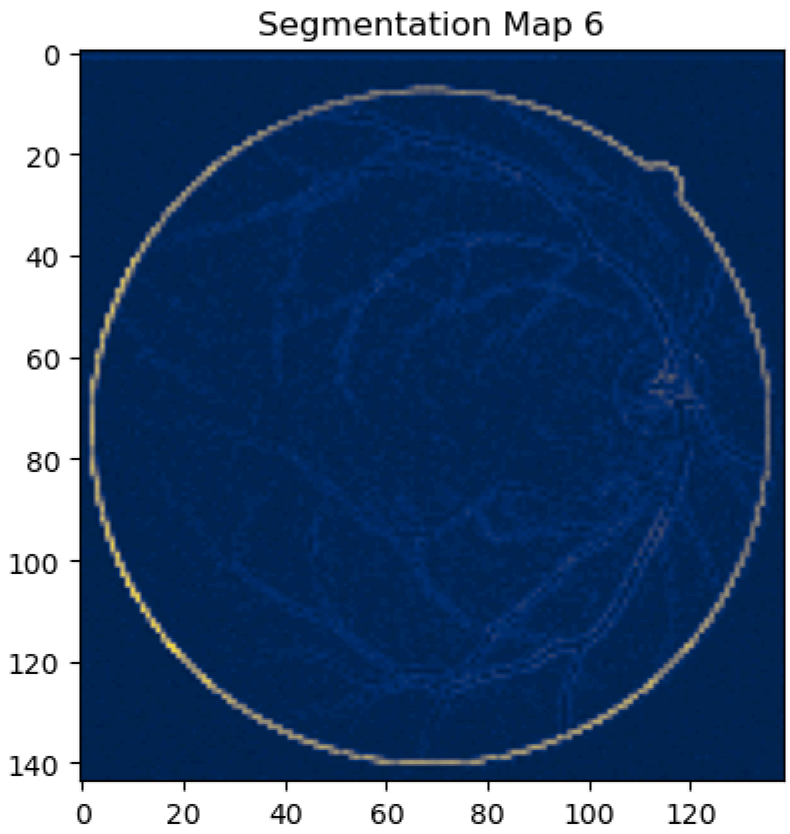

Segmentation Map 3
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120

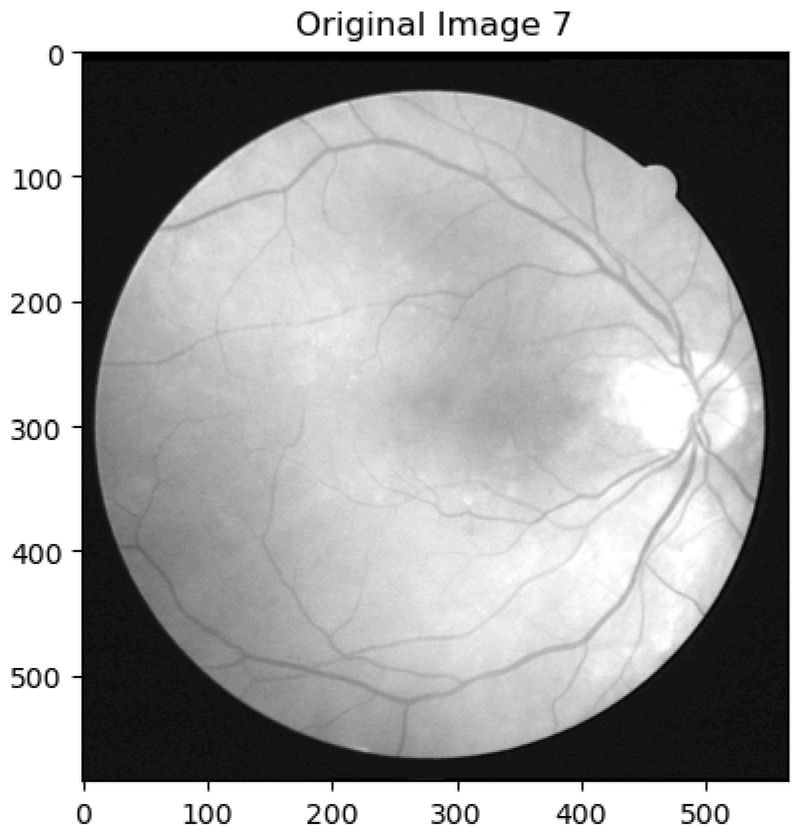

Original Image 4
0
100
200
300
400
500
0
100
200
300
400
500
Segmentation Map 4
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120

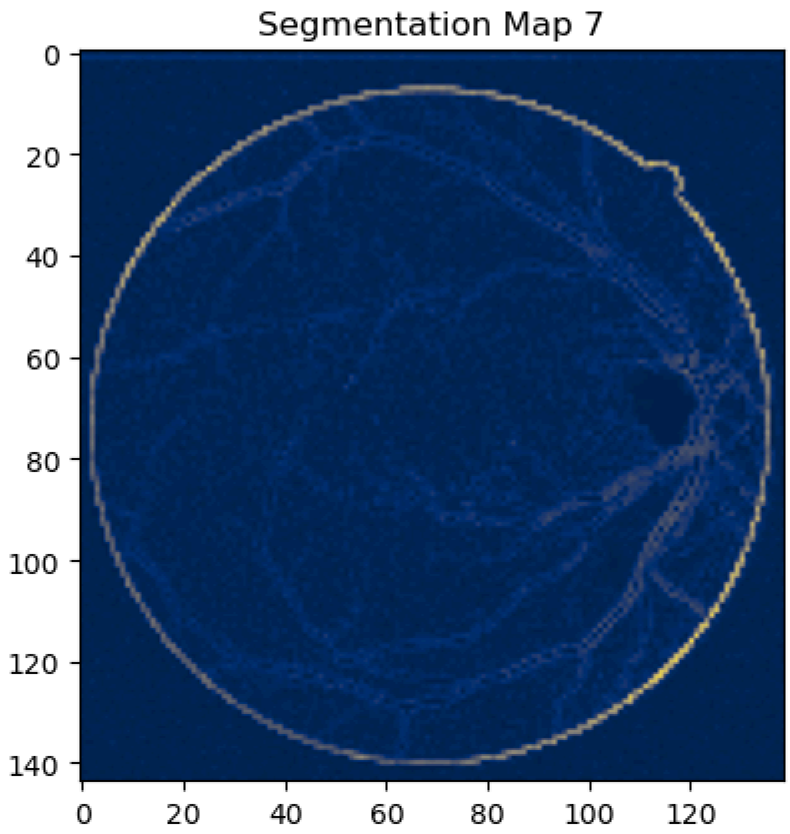

Original Image 5
0
100
200
300
400
500
0
100
200
300
400
500

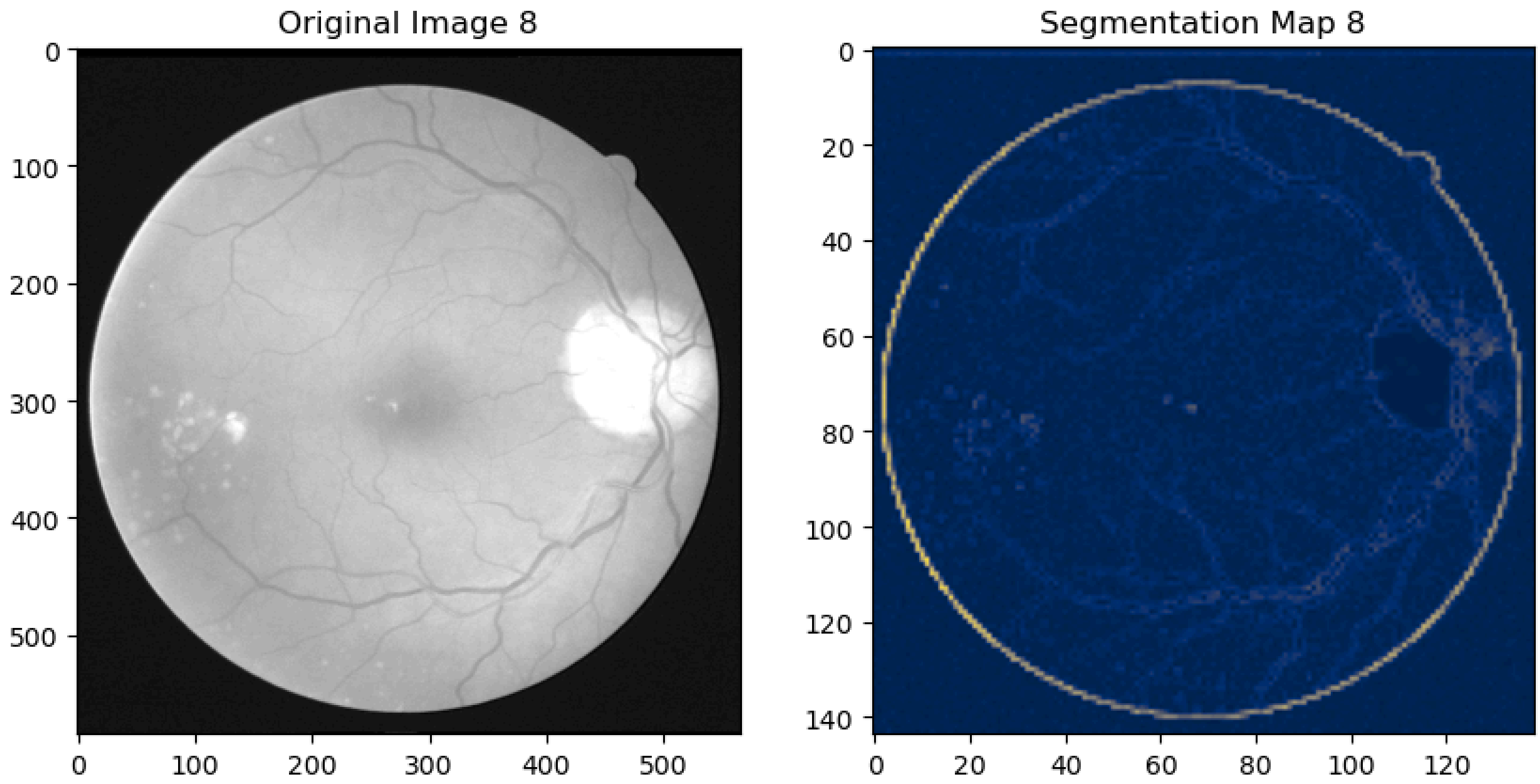

Segmentation Map 5
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120

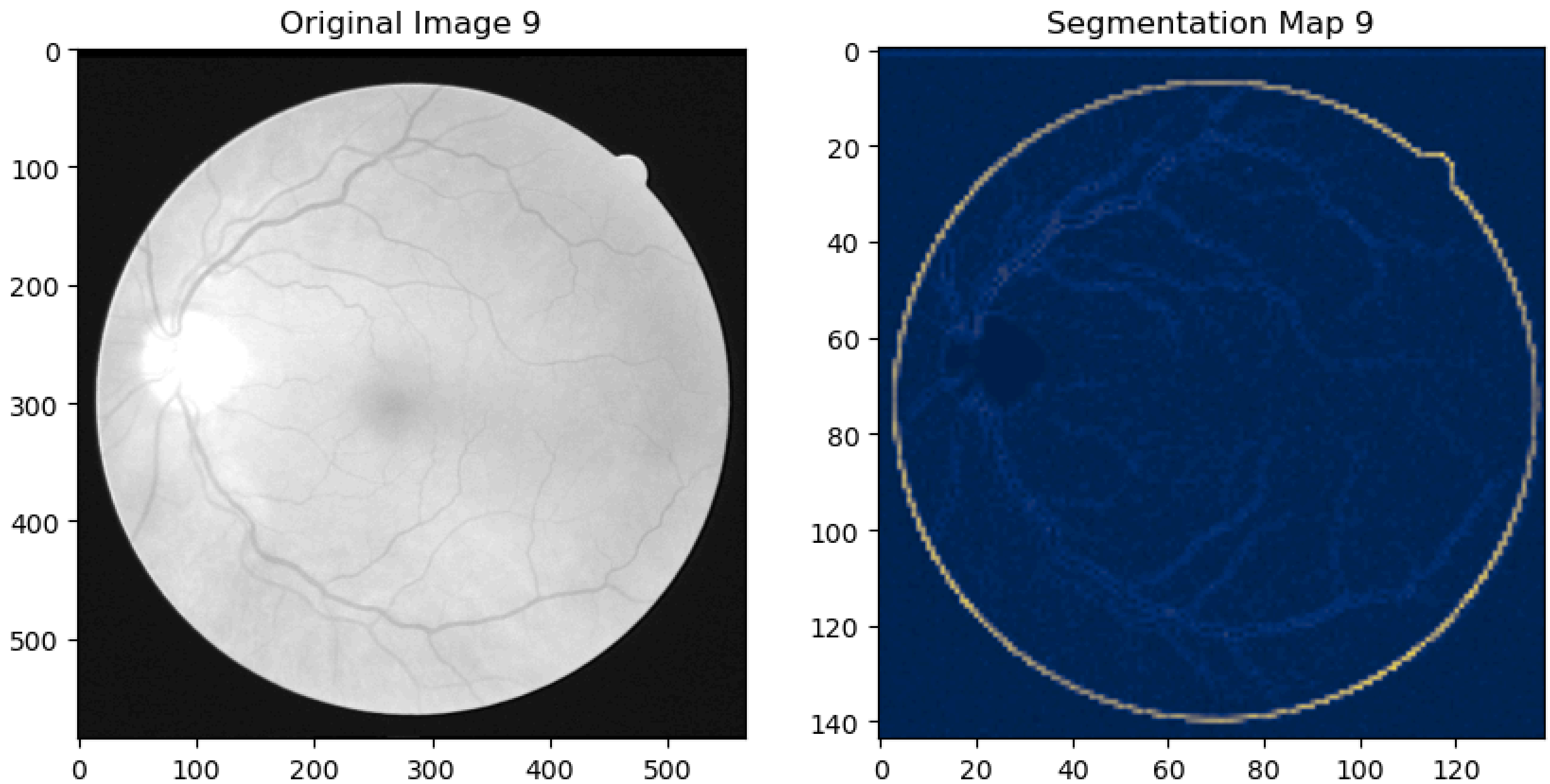

Original Image 6
0
100
200
300
400
500
0
100
200
300
400
500

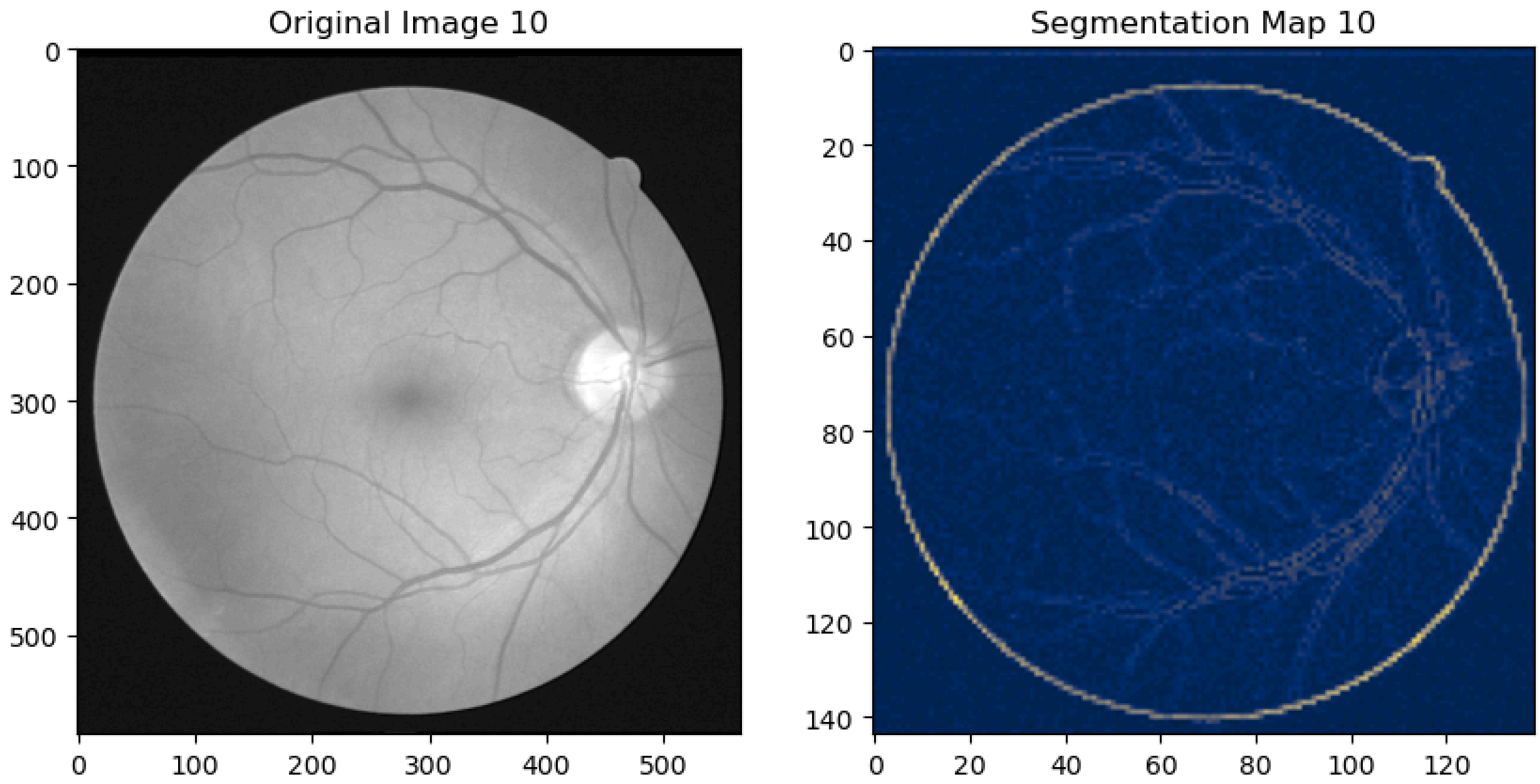

Segmentation Map 6
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120


Original Image 7
0
100
200
300
400
500
0
100
200
300
400
500


Segmentation Map 7
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120


Original Image 8
0
100
200
300
400
500
0 100 200 300 400 500
Segmentation Map 8
0
20
40
60
80
100
120
140
0 20 40 60 80 100 120


Original Image 9
0
100
200
300
400
500
0
100
200
300
400
500
Segmentation Map 9
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120


Original Image 10
0
100
200
300
400
500
0
100
200
300
400
500
Segmentation Map 10
0
20
40
60
80
100
120
140
0
20
40
60
80
100
120

```
In [ ]:
```