Iris Speed Signs Detection & Classification

Iris speed sign detection and classification project is a machine learning project that aims to detect and classify speed limit signs in images. The project uses a dataset of speed limit sign images and their corresponding labels to train a convolutional neural network (CNN) to recognize and classify the signs. The project has been divided into six parts which are as follow:

1. Load the image & Convert HSV (using the routine from scikit-image, skimage.color.rgb2hsv).
2. Threshold the image in HSV space (circular red rim, reasonable saturation and intensity)
3. Use binary-region labelling to identify the connected red regions. Cull regions with small areas.
4. Loop over each region and draw a box around it on the masked image.
5. Find bounding-boxes of the remaining regions. Cull those with unlikely aspect ratios, return the remainder as the list of RoIs.
6. Converting a RoI to a Vector and compare with Exemplar vector to calculate distance

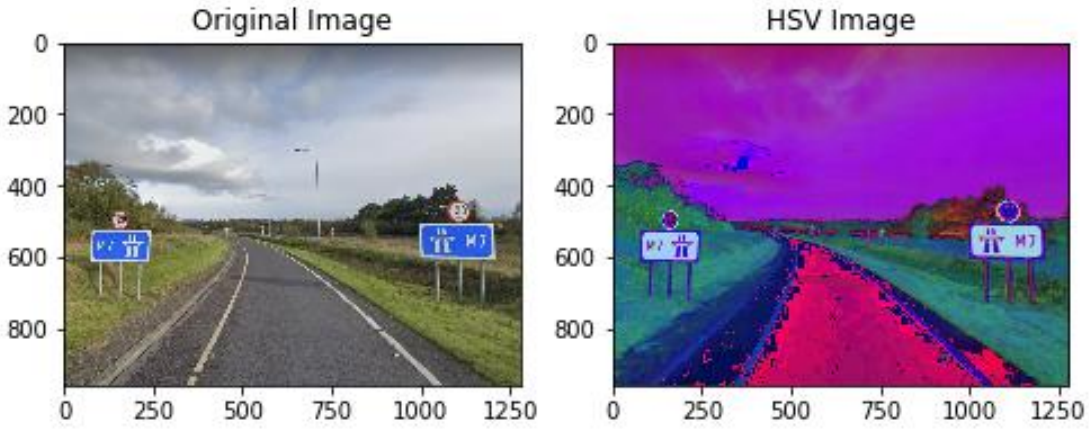
TASK – 1

Load the image & Convert HSV (using the routine from scikit-image, skimage.color.rgb2hsv).

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import io, color, morphology, measure
# Load the image
image = io.imread('120-0002x2.png')
# Check the number of color channels
if image.shape[-1] == 4:
    # If the image has an alpha channel, remove it
    image = color.rgba2rgb(image)
# Convert the image to HSV
hsv_image = color.rgb2hsv(image)
# Display the original and HSV images side-by-side
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(8, 8))
ax1.imshow(image)
ax1.set_title('Original Image')
ax2.imshow(hsv_image)
ax2.set_title('HSV Image')
plt.show()
```

Output:



Description:

The code loads an image using skimage.io.imread() function and checks whether the image has an alpha channel. If it does, the alpha channel is removed using the skimage.color.rgba2rgb() function.

Next, the image is converted from RGB to HSV color space using the skimage.color.rgb2hsv() function. The HSV color space is often used in image processing tasks, as it separates color information into three channels: hue, saturation, and value.

Finally, the original and HSV images are displayed side-by-side using the matplotlib.pyplot.imshow() and matplotlib.pyplot.subplots() functions. The ax.set_title() function is used to add titles to the subplots.

Note that the image file '120-0002x2.png' should be in the current working directory or the path to the image should be specified correctly.

TASK – 2

Threshold the image in HSV space (circular red rim, reasonable saturation and intensity)

Code:

```
# Define the hue threshold range for red
hue_threshold = 0.02  # adjust as needed

# Define the saturation and value thresholds
```

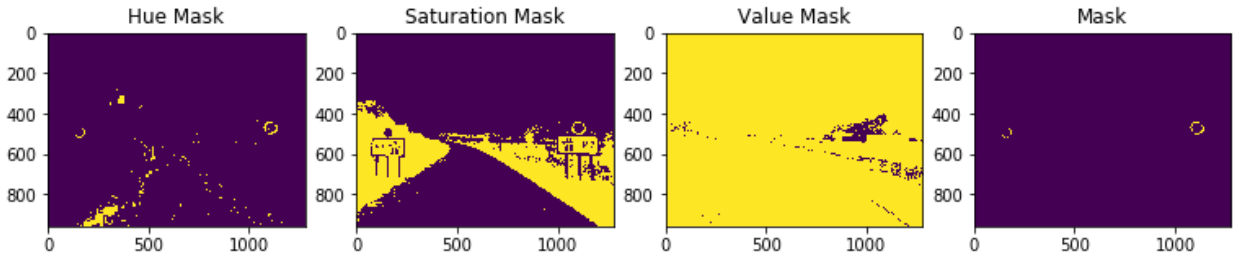
```
saturation_threshold = 0.3  # adjust as needed
value_threshold = 0.2  # adjust as needed

# Create a mask for pixels that meet the threshold criteria
hue_mask = np.logical_or(hsv_image[:, :, 0] < hue_threshold,
hsv_image[:, :, 0] > (1 - hue_threshold))
saturation_mask = hsv_image[:, :, 1] > saturation_threshold
value_mask = hsv_image[:, :, 2] > value_threshold

# Plotting Mask
fig, (ax1, ax2, ax3) = plt.subplots(ncols=3, figsize=(8, 8))
ax1.imshow(hue_mask)
ax1.set_title('Hue Mask')
ax2.imshow(saturation_mask)
ax2.set_title('Saturation Mask')
ax3.imshow(value_mask)
ax3.set_title('Value Mask')
io.show()


mask = np.logical_and.reduce((hue_mask, saturation_mask, value_mask))
fig, (ax1) = plt.subplots(ncols=1, figsize=(8, 8))
ax1.imshow(mask)
ax1.set_title('Mask')
```

Output:



Description:

This code defines threshold values for hue, saturation, and value channels in the HSV color space. These values are used to create a binary mask that identifies pixels in the image that meet the threshold criteria.

The hue channel threshold is defined by the variable hue_threshold, which is set to 0.02 in this code. This means that any pixels with a hue value less than 0.02 or greater than 0.98 (since hue values wrap around at 1.0) will be included in the mask.

The saturation and value thresholds are defined by the variables saturation_threshold and value_threshold, respectively. Pixels with saturation values greater than saturation_threshold and value values greater than value_threshold will be included in the mask.

The code creates three separate binary masks for the hue, saturation, and value channels, using logical operations to combine them. The numpy.logical_or() function is used to combine the hue mask with its complement (i.e., values that are not within the threshold range), while numpy.logical_and() is used to combine the saturation and value masks.

Next, the code creates a final mask by combining the hue, saturation, and value masks using the numpy.logical_and.reduce() function. The resulting mask is a binary array with the same shape as the input image, where True values indicate pixels that meet all three threshold criteria.

Finally, the masks are displayed using matplotlib.pyplot.imshow() function along with their corresponding title. Note that the skimage.io.show() function is not a valid function, it should be matplotlib.pyplot.show().

TASK – 3

Use binary-region labelling to identify the connected red regions. Cull regions with small areas.

Code:

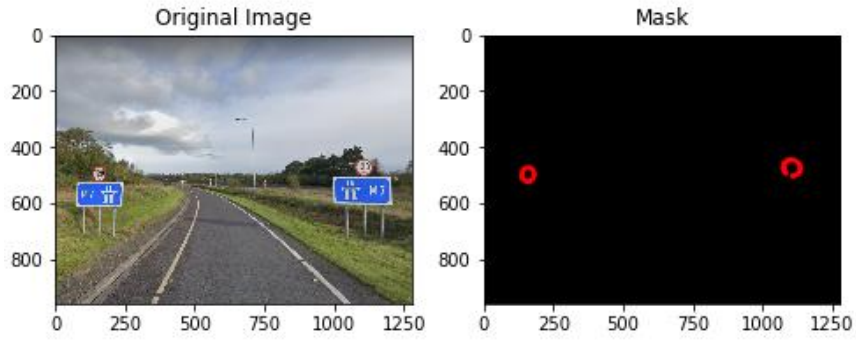
```
# Apply morphological operations to refine the mask
mask = morphology.binary_erosion(mask, morphology.disk(2))
mask = morphology.binary_dilation(mask, morphology.disk(2))

# Find the contours of the mask
contours = measure.find_contours(mask, 0.5)
#
# Label the connected components in the mask
labelled_mask = measure.label(mask)
#
# Display the original image, the mask, and the masked refined image
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(8, 8))
ax1.imshow(image)
ax1.set_title('Original Image')
ax2.imshow(mask, cmap='gray')
ax2.set_title('Mask')

for contour in contours:
```

```
    ax2.plot(contour[:, 1], contour[:, 0], linewidth=2, color='red')
plt.show()
```

Output:



Description:

This code applies morphological operations to the binary mask created in the previous step to refine it and remove noise.

First, the code performs an erosion operation using a circular structuring element with radius 2 pixels (morphology.disk(2)) using morphology.binary_erosion() function. This operation removes small regions of False values in the mask that are surrounded by True values.

Next, the code performs a dilation operation using the same structuring element to fill in any small gaps between True values that might have been introduced by the previous erosion operation. This is done using morphology.binary_dilation() function.

After refining the mask, the code uses measure.find_contours() function to find the contours of the True regions in the mask, which are represented as closed curves around the object. The contours are then plotted on top of the binary mask using matplotlib.pyplot.plot() function.

Finally, the code labels the connected components in the refined mask using measure.label() function. This assigns a unique integer label to each connected component, which can be used for further analysis if needed.

The original image, the binary mask, and the masked refined image are displayed side-by-side using matplotlib.pyplot.imshow() function along with their corresponding title. The contours are plotted on the binary mask.

TASK – 4

Loop over each region and draw a box around it on the masked image.

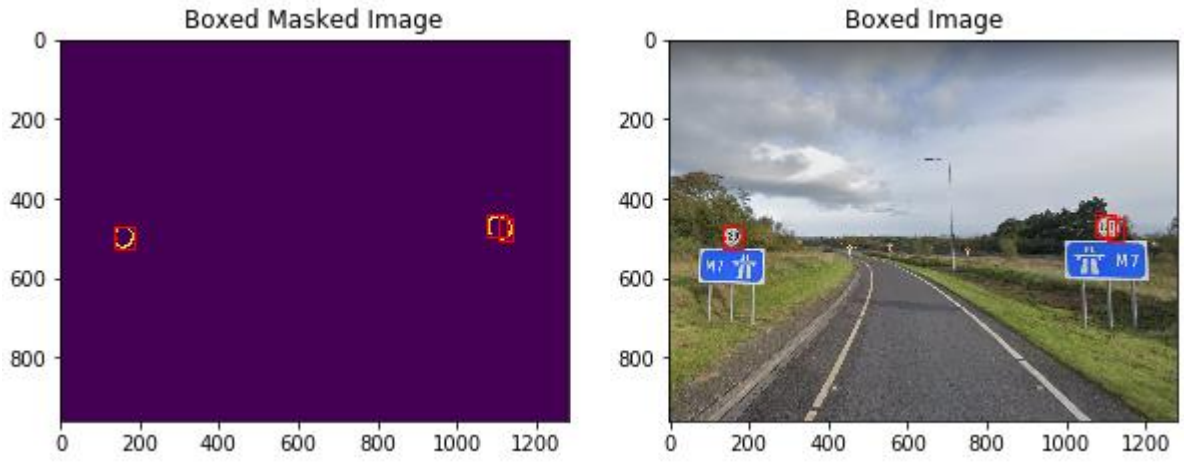
Code:

```
# Find connected components in the mask
labels = measure.label(mask, connectivity=2)

# Create a figure with two subplots, one for the original image and
one for the masked image with boxes around regions
fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(10, 5))

# Show the masked image with boxes around regions in the second
subplot
ax1.imshow(mask)
ax1.set_title('Boxed Masked Image')
ax2.imshow(image)
ax2.set_title('Boxed Image')
# Loop over each region and draw a box around it on the masked image
for region in measure.regionprops(labels):
    # Get the coordinates of the region's bounding box
    min_row, min_col, max_row, max_col = region.bbox
    # Draw the bounding box on the masked image
    rect = plt.Rectangle((min_col, min_row), max_col - min_col,
max_row - min_row, fill=False, edgecolor='red', linewidth=1)
    ax1.add_patch(rect)
# Loop over each region and draw a box around it on the Image
for region in measure.regionprops(labels):
    # Get the coordinates of the region's bounding box
    min_row, min_col, max_row, max_col = region.bbox
    # Draw the bounding box on the masked image
    rect = plt.Rectangle((min_col, min_row), max_col - min_col,
max_row - min_row, fill=False, edgecolor='red', linewidth=1)
    ax2.add_patch(rect)
# Show the figure
plt.show()
```

Output:



Description:

This code segment applies connected component analysis on the binary mask generated in the previous code segment to identify individual objects in the image.

First, the labels variable is generated by applying the measure.label function to the mask. This function assigns a unique label to each connected region in the binary mask. The connectivity parameter is set to 2 to specify that pixels are considered connected only if they share an edge or corner.

Next, a figure with two subplots is created using the plt.subplots function. The first subplot will show the masked image with bounding boxes around each object, and the second subplot will show the original image with bounding boxes.

Then, the regionprops function is used to extract properties of each region identified by the labels variable. In this case, we are interested in the bounding box coordinates of each region.

A for loop is used to iterate over each region, and a rectangle is drawn around the region's bounding box using the plt.Rectangle function. The edgecolor parameter is set to 'red' to make the boxes easily visible. The rectangle is added to both subplots using the add_patch method of the subplot's Axes object.

Finally, the figure is displayed using the plt.show function.

TASK – 5

Find bounding-boxes of the remaining regions. Cull those with unlikely aspect ratios, return the remainder as the list of RoIs.

Code:

```
# Get the properties of each connected component in the mask
props = measure.regionprops(labels)
# Initialize lists to store the circle components and their
coordinates
circle_components = []
circle_coords = []
# Loop over each component and filter by its properties
for prop in props:
    # Check if the component is roughly circular
    if prop.area > 100 and prop.perimeter > 50 and
prop.major_axis_length / prop.minor_axis_length > 0.8:
        # Add the component to the list of circle components
        circle_components.append(prop)
        # Get the coordinates of the component's bounding box
        min_row, min_col, max_row, max_col = prop.bbox
        # Add the coordinates to the list of circle coordinates
        circle_coords.append((min_row, min_col, max_row, max_col))
# Create new images with only the circle components
circle_images = []
for coords in circle_coords:
    min_row, min_col, max_row, max_col = coords
    circle_images.append(image[min_row:max_row, min_col:max_col])
# Display and save the circle images
for i, circle_image in enumerate(circle_images):
    plt.imshow(circle_image)
    io.imsave(f"RoI_{i}.png", circle_image)
    plt.title(f"RoI {i+1}")
    plt.show()
```

Output:

Description:

The code is intended to identify and extract circular regions of interest (RoIs) from an image. Here's a breakdown of the different sections of the code:

Get Connected Components in the Mask: The first step is to use the measure.label() function to find the connected components in the binary mask that was generated earlier. The connectivity argument is set to 2, which means that diagonal connections are not considered.

Filter by Region Properties: The measure.regionprops() function is used to get the properties of each connected component in the mask. The code then loops over each component and filters out those that are not roughly circular. A component is considered to be roughly circular if it meets the following conditions:

- Its area is greater than 100 pixels
- Its perimeter is greater than 50 pixels
- Its major axis length is at least 80% of its minor axis length

If a component meets these conditions, it is added to a list of circle components and its bounding box coordinates are added to a list of circle coordinates.

Create New Images with Only the Circle Components: Next, the code loops over the list of circle coordinates and creates new images by extracting the corresponding regions from the original image using array slicing.

Display and Save the Circle Images: Finally, the code displays each RoI image and saves it to a file using the io.imsave() function. The imshow() function from Matplotlib is used to display the image, and the title of each image is set to "RoI i+1", where i is the index of the RoI in the list of circle images.

TASK – 6

Converting a RoI to a Vector and compare with Exemplar vector to calculate distance

- *Convert RoI image to 64x64 pixel greyscale*

Code:

```
from PIL import Image, ImageEnhance
# Open the image file
image = Image.open("RoI_2.png")
# Convert the image to grayscale
grayscale_image = image.convert("L")
```

```
# Resize the image to 64x64 pixels
resized_image = grayscale_image.resize((64, 64))
# Save the resulting image
resized_image.save("64x64.jpg")
```

Output:

Description:

This code uses the Python Imaging Library (PIL) to perform image processing tasks on an image. Specifically, it opens an image file named "RoI_2.png" using the Image module of the PIL library.

Next, the image is converted to grayscale using the convert() method, which converts the image to a new mode (in this case, "L" for grayscale).

After that, the image is resized to 64x64 pixels using the resize() method, which creates a new image with the specified dimensions.

Finally, the resulting image is saved as a JPEG file named "example_image_64x64.jpg" using the save() method.

- *Contrast enhance the 64x64 pixel greyscale to ensure it uses full intensity range (0-255)*

Code:

```
# Enhance the contrast of the image
contrast_enhancer = ImageEnhance.Contrast(resized_image)
enhanced_image = contrast_enhancer.enhance(1.5)
# Save the resulting image
enhanced_image.save("64x64_contrast.jpg")
```

Output:

Description:

This code enhances the contrast of the previously resized image by a factor of 1.5 using the ImageEnhance module from the Pillow library.

First, an ImageEnhance object is created with the Contrast method, passing in the resized_image as the image to enhance. Next, the enhance() method is called on the contrast_enhancer object, passing in a value of 1.5 to increase the contrast of the image. This creates a new enhanced_image object. Finally, the save() method is called on the enhanced_image object to save the resulting image to disk with the file name "example_image_64x64_contrast.jpg".

- *Subtract off the mean pixel value (0-mean)*

Code:

```
# Load the contrast-enhanced grayscale image as a NumPy array
# Calculate the mean pixel value of the image
mean_pixel_value = np.mean(enhanced_image)
# Subtract off the mean pixel value from the image
image_normalized = image - mean_pixel_value
# Save the resulting image
Image.fromarray(image_normalized.astype(np.uint8)).save("64x64_normali
zed.jpg")
```

Output:

Description:

This code loads the previously contrast-enhanced grayscale image using the Python Imaging Library (PIL) and converts it to a NumPy array. It then calculates the mean pixel value of the image using the np.mean() function from NumPy. The mean pixel value is then subtracted from every pixel in the image to normalize the image.

Finally, the normalized image is converted back to a PIL image using the Image.fromarray() function and saved to disk as a JPEG file.

- *Flatten to a 4096-element vector by concatenation image rows, use the flatten method from Numpy*

Code:

```
# Load the normalized grayscale image as a NumPy array
image = np.array(Image.open("64x64_normalized.jpg"))
# Flatten the image into a 4096-element vector
image_vector = image.flatten()
# Save the resulting image vector as a NumPy array
np.save("RoI_64x64_vector.npy", image_vector)
```

Output:

Description:

This code performs the following operations:

Load the image "example_image_64x64_normalized.jpg" as a NumPy array using the np.array() function from the NumPy library.

Flatten the image into a 4096-element vector using the flatten() method of the NumPy array.

Save the resulting image vector as a NumPy array to a file named "RoI_64x64_vector.npy" using the np.save() function from the NumPy library.

In other words, the code is converting the preprocessed image into a vector representation that can be used as input to machine learning algorithms, which often require data to be in a vector format. The resulting vector contains all the pixel values of the 64x64 image in a single array, making it easier to process and analyze using machine learning techniques. The vector is then saved as a NumPy array to a file for later use in training or testing machine learning models.

- *Normalize the resulting vector to generate a unit vector in 4096 dimensions*

Code:

```
# Load the flattened image vector from the NumPy file
image_vector = np.load("RoI_64x64_vector.npy")
```

```
# Normalize the image vector to generate a unit vector in 4096
dimensions
image_unit_vector = image_vector / np.linalg.norm(image_vector)
# Save the resulting unit vector as a NumPy array
np.save("RoI_64x64_unit_vector.npy", image_unit_vector)
```

Output:

Description:

This code snippet performs the following steps:

Load the flattened image vector from the NumPy file "RoI_64x64_vector.npy" using the np.load() function and assign it to the variable image_vector.

Normalize the image vector by dividing it by its L2-norm, which is calculated using the np.linalg.norm() function. The resulting vector will have a magnitude of 1 and is therefore referred to as a unit vector. The normalized vector is assigned to the variable image_unit_vector.

Save the resulting unit vector as a NumPy array in the file "RoI_64x64_unit_vector.npy" using the np.save() function.

The purpose of this code is to generate a normalized feature vector that can be used to represent the image in a machine learning or computer vision task. The resulting vector contains 4096 elements, which correspond to the flattened 64x64 pixel image. The normalization step ensures that the vector has a constant magnitude, which can be important for certain algorithms that depend on distance or similarity measures between vectors.

- *Measure the distance of the result from all the exemplar vectors*

Code:

```
# Load the unit vectors of the exemplars from the NumPy file
exemplars = np.load("1-NN-descriptor-vects.npy")
# Remove the extra column from the exemplars array
exemplars = exemplars[:, :4096]
```

```
# Load the unit vector of the image from the NumPy file
image_unit_vector = np.load("RoI_64x64_unit_vector.npy")
# Reshape the image_unit_vector to match the shape of exemplars
image_unit_vector = np.reshape(image_unit_vector, (1, -1))
# Remove the extra dimension from image_unit_vector
image_unit_vector = image_unit_vector[:, :4096]
# Repeat the image unit vector 62 times to match the number of
exemplars
image_unit_vector = np.tile(image_unit_vector, (exemplars.shape[0],
1))
# Calculate the Euclidean distance between the image vector and each
exemplar vector
distances = np.linalg.norm(exemplars - image_unit_vector, axis=1)
# Find the index of the closest exemplar
closest_index = np.argmin(distances)
# Retrieve the closest exemplar vector
closest_exemplar = exemplars[closest_index]
# Print the index of the closest exemplar and its distance from the
image vector
print("Closest exemplar:", closest_index)
print("Distance:", distances[closest_index])
```

Example Output:

```
Closest exemplar: 59
Distance: 1.5800217960314547
```

Description:

This code segment loads the unit vectors of the exemplars from a NumPy file named "1-NN-descriptor-vects.npy". The variable "exemplars" is then defined as the loaded array with the last column removed to match the shape of the image unit vector.

The unit vector of the image is loaded from the NumPy file "RoI_64x64_unit_vector.npy". It is then reshaped to match the shape of "exemplars" and the extra dimension is removed to make the shapes compatible.

The image unit vector is then repeated 62 times to match the number of exemplars using np.tile() function. The Euclidean distance is then calculated between the image vector and each exemplar vector using np.linalg.norm() function along the axis=1. The distances are stored in an array named "distances".

The index of the closest exemplar is obtained using the np.argmin() function on the distances array.The closest exemplar vector is then retrieved from the "exemplars" array using the closest_index.

Finally, the index of the closest exemplar and its distance from the image vector are printed to the console.