Chapter 2: Handling Images

Introduction

- Before applying machine learning to images, we often need to transform the raw images to features usable
- To work with images, we will use the Open Source Computer Vision Library (OpenCV)
- OpenCV is the most popular and documented library for handling images
- conda install --channel https://conda.anaconda.org/menpo opencv3
- import cv2

Loading Images

Solution

Use OpenCV's `imread`:

```
# Load library
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image as grayscale
image = cv2.imread("images/plane.jpg", cv2.IMREAD_GRAYSCALE)
```

If we want to view the image, we can use the Python plotting library Matplotlib:

```
# Show image
plt.imshow(image, cmap="gray"), plt.axis("off")
plt.show()
```

Discussion

Fundamentally, images are data and when we use `imread` we convert that data into a data type we are very familiar with—a NumPy array:

```
# Show data type
type(image)
```

```
numpy.ndarray
```

We have transformed the image into a matrix whose elements correspond to individual pixels. We can even take a look at the actual values of the matrix:

```
# Show image data
image
```

```
array([[140, 136, 146, ..., 132, 139, 134],
       [144, 136, 149, ..., 142, 124, 126],
       [152, 139, 144, ..., 121, 127, 134],
       ...,
       [156, 146, 144, ..., 157, 154, 151],
       [146, 150, 147, ..., 156, 158, 157],
       [143, 138, 147, ..., 156, 157, 157]], dtype=uint8)
```

The resolution of our image was 3600 × 2270, the exact dimensions of our matrix:

```
# Show dimensions
image.shape
```

```
(2270, 3600)
```

In the matrix, each element contains three values corresponding to blue, green, red values (BGR):

```
# Load image in color
image_bgr = cv2.imread("images/plane.jpg", cv2.IMREAD_COLOR)

# Show pixel
image_bgr[0,0]
```

```
array([195, 144, 111], dtype=uint8)
```

One small caveat: by default OpenCV uses BGR, but many image applications—including Matplotlib—use red, green, blue (RGB), meaning the red and the blue values are swapped. To properly display OpenCV color images in Matplotlib, we need to first convert the color to RGB (apologies to hardcopy readers):

```
# Convert to RGB
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# Show image
plt.imshow(image_rgb), plt.axis("off")
plt.show()
```

Saving Images

```python
# Load libraries
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image as grayscale
image = cv2.imread("images/plane.jpg", cv2.IMREAD_GRAYSCALE)

# Save image
cv2.imwrite("images/plane_new.jpg", image)
```

```
True
```

Discussion

OpenCV's `imwrite` saves images to the filepath specified. The format of the image is defined by the filename's extension (*.jpg*, *.png*, etc.). One behavior to be careful about: `imwrite` will overwrite existing files without outputting an error or asking for confirmation.

Resizing Images

Solution

Use `resize` to change the size of an image:

```
# Load image
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image as grayscale
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Resize image to 50 pixels by 50 pixels
image_50x50 = cv2.resize(image, (50, 50))

# View image
plt.imshow(image_50x50, cmap="gray"), plt.axis("off")
plt.show()
```

Discussion

Resizing images is a common task in image preprocessing for two reasons. First, images come in all shapes and sizes, and to be usable as features, images must have the same dimensions. This standardization of image size does come with costs, however; images are matrices of information and when we reduce the size of the image we are reducing the size of that matrix and the information it contains. Second, machine learning can require thousands or hundreds of thousands of images. When those images are very large they can take up a lot of memory, and by resizing them we can dramatically reduce memory usage. Some common image sizes for machine learning are 32 × 32, 64 × 64, 96 × 96, and 256 × 256.

Cropping Images

The image is encoded as a two-dimensional NumPy array, so we can crop the image easily by slicing the array:

```
# Load image
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image in grayscale
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Select first half of the columns and all rows
image_cropped = image[:,:128]

# Show image
plt.imshow(image_cropped, cmap="gray"), plt.axis("off")
plt.show()
```

Blurring Images

To blur an image, each pixel is transformed to be the average value of its neighbors. This neighbor and the operation performed are mathematically represented as a kernel (don't worry if you don't know what a kernel is). The size of this kernel determines the amount of blurring, with larger kernels producing smoother images. Here we blur an image by averaging the values of a 5×5 kernel around each pixel:

```python
# Load libraries
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image as grayscale
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Blur image
image_blurry = cv2.blur(image, (5,5))

# Show image
plt.imshow(image_blurry, cmap="gray"), plt.axis("off")
plt.show()
```

Discussion

Kernels are widely used in image processing to do everything from sharpening to edge detection, and will come up repeatedly in this chapter. The blurring kernel we used looks like this:

```
# Create kernel
kernel = np.ones((5,5)) / 25.0

# Show kernel
kernel
```

```
array([[ 0.04,  0.04,  0.04,  0.04,  0.04],
       [ 0.04,  0.04,  0.04,  0.04,  0.04],
       [ 0.04,  0.04,  0.04,  0.04,  0.04],
       [ 0.04,  0.04,  0.04,  0.04,  0.04],
       [ 0.04,  0.04,  0.04,  0.04,  0.04]])
```

The center element in the kernel is the pixel being examined, while the remaining elements are its neighbors. Since all elements have the same value (normalized to add up to 1), each has an equal say in the resulting value of the pixel of interest. We can manually apply a kernel to an image using `filter2D` to produce a similar blurring effect:

```
# Apply kernel
image_kernel = cv2.filter2D(image, -1, kernel)
```

Sharpening Images

```
# Load image as grayscale
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Create kernel
kernel = np.array([[0, -1, 0],
                   [-1, 5,-1],
                   [0, -1, 0]])

# Sharpen image
image_sharp = cv2.filter2D(image, -1, kernel)

# Show image
plt.imshow(image_sharp, cmap="gray"), plt.axis("off")
plt.show()
```

Enhancing Contrast

Solution

Histogram equalization is a tool for image processing that can make objects and shapes stand out. When we have a grayscale image, we can apply OpenCV's `equalizeHist` directly on the image:

```
# Load libraries
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Enhance image
image_enhanced = cv2.equalizeHist(image)

# Show image
plt.imshow(image_enhanced, cmap="gray"), plt.axis("off")
plt.show()
```

However, when we have a color image, we first need to convert the image to the YUV color format. The Y is the luma, or brightness, and U and V denote the color. After the conversion, we can apply `equalizeHist` to the image and then convert it back to BGR or RGB:

```
# Load image
image_bgr = cv2.imread("images/plane.jpg")

# Convert to YUV
image_yuv = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2YUV)

# Apply histogram equalization
image_yuv[:, :, 0] = cv2.equalizeHist(image_yuv[:, :, 0])

# Convert to RGB
image_rgb = cv2.cvtColor(image_yuv, cv2.COLOR_YUV2RGB)

# Show image
plt.imshow(image_rgb), plt.axis("off")
plt.show()
```

Isolating Colors

```
# Load image
image_bgr = cv2.imread('images/plane_256x256.jpg')

# Convert BGR to HSV
image_hsv = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2HSV)

# Define range of blue values in HSV
lower_blue = np.array([50,100,50])
upper_blue = np.array([130,255,255])

# Create mask
mask = cv2.inRange(image_hsv, lower_blue, upper_blue)

# Mask image
image_bgr_masked = cv2.bitwise_and(image_bgr, image_bgr, mask=mask)

# Convert BGR to RGB
image_rgb = cv2.cvtColor(image_bgr_masked, cv2.COLOR_BGR2RGB)

# Show image
plt.imshow(image_rgb), plt.axis("off")
plt.show()
```

Binarizing Images

Thresholding is the process of setting pixels with intensity greater than some value to be white and less than the value to be black. A more advanced technique is *adaptive thresholding*, where the threshold value for a pixel is determined by the pixel intensities of its neighbors. This can be helpful when lighting conditions change over different regions in an image:

```
# Load image as grayscale
image_grey = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Apply adaptive thresholding
max_output_value = 255
neighborhood_size = 99
subtract_from_mean = 10
image_binarized = cv2.adaptiveThreshold(image_grey,
                                        max_output_value,
                                        cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                        cv2.THRESH_BINARY,
                                        neighborhood_size,
                                        subtract_from_mean)

# Show image
plt.imshow(image_binarized, cmap="gray"), plt.axis("off")
plt.show()
```

Discussion

Our solution has four important arguments in adaptiveThreshold. max_output_value simply determines the maximum intensity of the output pixel intensities. cv2.ADAPTIVE_THRESH_GAUSSIAN_C sets a pixel's threshold to be a weighted sum of the neighboring pixel intensities. The weights are determined by a Gaussian window. Alternatively we could set the threshold to simply the mean of the neighboring pixels with cv2.ADAPTIVE_THRESH_MEAN_C:

```
# Apply cv2.ADAPTIVE_THRESH_MEAN_C
image_mean_threshold = cv2.adaptiveThreshold(image_grey,
                                             max_output_value,
                                             cv2.ADAPTIVE_THRESH_MEAN_C,
                                             cv2.THRESH_BINARY,
                                             neighborhood_size,
                                             subtract_from_mean)

# Show image
plt.imshow(image_mean_threshold, cmap="gray"), plt.axis("off")
plt.show()
```

Removing Backgrounds

```python
image_bgr = cv2.imread('images/plane_256x256.jpg')
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# Rectangle values: start x, start y, width, height
rectangle = (0, 56, 256, 150)

# Create initial mask
mask = np.zeros(image_rgb.shape[:2], np.uint8)

# Create temporary arrays used by grabCut
bgdModel = np.zeros((1, 65), np.float64)
fgdModel = np.zeros((1, 65), np.float64)

# Run grabCut
cv2.grabCut(image_rgb, # Our image
            mask, # The Mask
            rectangle, # Our rectangle
            bgdModel, # Temporary array for background
            fgdModel, # Temporary array for background
            5, # Number of iterations
            cv2.GC_INIT_WITH_RECT) # Initiative using our rectangle

# Create mask where sure and likely backgrounds set to 0, otherwise 1
mask_2 = np.where((mask==2) | (mask==0), 0, 1).astype('uint8')

# Multiply image with new mask to subtract background
image_rgb_nobg = image_rgb * mask_2[:, :, np.newaxis]
```

Detecting Edges

```
# Load image as grayscale
image_gray = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Calculate median intensity
median_intensity = np.median(image_gray)

# Set thresholds to be one standard deviation above and below median intensity

 lower_threshold = int(max(0, (1.0 - 0.33) * median_intensity))
 upper_threshold = int(min(255, (1.0 + 0.33) * median_intensity))

 # Apply canny edge detector
 image_canny = cv2.Canny(image_gray, lower_threshold, upper_threshold)

 # Show image
 plt.imshow(image_canny, cmap="gray"), plt.axis("off")
 plt.show()
```

Discussion

Edge detection is a major topic of interest in computer vision. Edges are important because they are areas of high information. For example, in our image one patch of sky looks very much like another and is unlikely to contain unique or interesting information. However, patches where the background sky meets the airplane contain a lot of information (e.g., an object's shape). Edge detection allows us to remove low-information areas and isolate the areas of images containing the most information.

Detecting Corners

```python
# Load image as grayscale
image_bgr = cv2.imread("images/plane_256x256.jpg")
image_gray = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)
image_gray = np.float32(image_gray)

# Set corner detector parameters
block_size = 2
aperture = 29
free_parameter = 0.04

# Detect corners
detector_responses = cv2.cornerHarris(image_gray,
                                      block_size,
                                      aperture,
                                      free_parameter)

# Large corner markers
detector_responses = cv2.dilate(detector_responses, None)
```

```python
# Only keep detector responses greater than threshold, mark as white
threshold = 0.02
image_bgr[detector_responses >
          threshold *
          detector_responses.max()] = [255,255,255]

# Convert to grayscale
image_gray = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)

# Show image
plt.imshow(image_gray, cmap="gray"), plt.axis("off")
plt.show()
```

Discussion

The Harris corner detector is a commonly used method of detecting the intersection of two edges. Our interest in detecting corners is motivated by the same reason as for deleting edges: corners are points of high information. A complete explanation of the Harris corner detector is available in the external resources at the end of this recipe, but a simplified explanation is that it looks for windows (also called *neighborhoods* or *patches*) where small movements of the window (imagine shaking the window) creates big changes in the contents of the pixels inside the window. `cornerHarris` contains three important parameters that we can use to control the edges detected. First, `block_size` is the size of the neighbor around each pixel used for corner detection.

Second, `aperture` is the size of the Sobel kernel used (don't worry if you don't know what that is), and finally there is a free parameter where larger values correspond to identifying softer corners.

Creating Features for Machine Learning

```python
# Load image
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image as grayscale
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Resize image to 10 pixels by 10 pixels
image_10x10 = cv2.resize(image, (10, 10))

# Convert image data to one-dimensional vector
image_10x10.flatten()
```

```
array([133, 130, 130, 129, 130, 129, 129, 128, 128, 127, 135, 131, 131,
       131, 130, 130, 129, 128, 128, 128, 134, 132, 131, 131, 130, 129,
       129, 128, 130, 133, 132, 158, 130, 133, 130,  46,  97,  26, 132,
       143, 141,  36,  54,  91,   9,   9,  49, 144, 179,  41, 142,  95,
        32,  36,  29,  43, 113, 141, 179, 187, 141, 124,  26,  25, 132,
       135, 151, 175, 174, 184, 143, 151,  38, 133, 134, 139, 174, 177,
       169, 174, 155, 141, 135, 137, 137, 152, 169, 168, 168, 179, 152,
       139, 136, 135, 137, 143, 159, 166, 171, 175], dtype=uint8)
```

If the image is in color, instead of each pixel being represented by one value, it is represented by multiple values (most often three) representing the channels (red, green, blue, etc.) that blend to make the final color of that pixel. For this reason, if our 10×10 image is in color, we will have 300 feature values for each observation:

One of the major challenges of image processing and computer vision is that since every pixel location in a collection of images is a feature, as the images get larger, the number of features explodes:

```
# Load image in grayscale
image_256x256_gray = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Convert image data to one-dimensional vector, show dimensions
image_256x256_gray.flatten().shape
```

```
(65536,)
```

And the number of features only intensifies when the image is in color:

```
# Load image in color
image_256x256_color = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_COLOR)

# Convert image data to one-dimensional vector, show dimensions
image_256x256_color.flatten().shape
```

```
(196608,)
```

Encoding Mean Color as a Feature

```
# Load image as BGR
image_bgr = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_COLOR)

# Calculate the mean of each channel
channels = cv2.mean(image_bgr)

# Swap blue and red values (making it RGB, not BGR)
observation = np.array([(channels[2], channels[1], channels[0])])

# Show mean channel values
observation
```

```
array([[  90.53204346,  133.11735535,  169.03074646]])
```

Discussion

The output is three feature values for an observation, one for each color channel in the image. These features can be used like any other features in learning algorithms to classify images according to their colors.

Encoding Color Histograms as Features

```python
# Load image
image_bgr = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_COLOR)

# Convert to RGB
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# Create a list for feature values
features = []

# Calculate the histogram for each color channel
colors = ("r","g","b")

# For each channel: calculate histogram and add to feature value list
for i, channel in enumerate(colors):
    histogram = cv2.calcHist([image_rgb], # Image
                        [i], # Index of channel
                        None, # No mask
                        [256], # Histogram size
                        [0,256]) # Range
    features.extend(histogram)

# Create a vector for an observation's feature values
observation = np.array(features).flatten()
```

Discussion

In the RGB color model, each color is the combination of three color channels (i.e., red, green, blue). In turn, each channel can take on one of 256 values (represented by an integer between 0 and 255). For example, the top-leftmost pixel in our image has the following channel values:

```
# Show RGB channel values
image_rgb[0,0]
```

```
array([107, 163, 212], dtype=uint8)
```

A histogram is a representation of the distribution of values in data. Here is a simple example:

Q&A

Thank you!