Computer Vision

Muchang Bahng

October 19, 2023

Contents

1 Cameras and Transformations **2**
1.1 Lie Groups and Lie Algebras . . . 2
1.2 Camera Parameterizations . . . 2
1.3 Sampling and Quantization . . . 2
1.4 Image Compression . . . 2

2 Image Processing **2**
2.1 Basic Functionality . . . 2
2.1.1 Color Histograms . . . 2
2.2 Drawing and Transformations . . . 3
2.2.1 Masking . . . 3
2.3 Kernels . . . 3

3 Region Based Object Detection **6**
3.1 RCNN . . . 6
3.2 Fast RCNN . . . 6
3.3 Faster RCNN . . . 6
3.4 Mask RCNN . . . 6

4 YOLO **6**

5 Simple Online Realtime Tracking **6**
5.1 SORT . . . 6
5.2 DeepSORT . . . 6
5.3 StrongSORT . . . 6

6 ByteTrack **6**

1 Cameras and Transformations

1.1 Lie Groups and Lie Algebras

1.2 Camera Parameterizations

1.3 Sampling and Quantization

1.4 Image Compression

2 Image Processing

2.1 Basic Functionality

A pixel can be really represented by a number. More specifically, a grayscale pixel is a number between 0 (black) and 255 (white), and a color pixel is in the RGB format, represented by a 3-tuple. Therefore, a grayscale image is really represented by a $H \times W$ matrix, and a RBG image by a $3 \times H \times W$ tensor. We can see this when opening up an image in OpenCV with the following code:

```
import cv2

PATH = "park.jpg"

img = cv2.imread(PATH)
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

cv2.imshow("Park", img)     # Show RGB image
cv2.imshow('Gray', gray)    # Show gray image
cv2.waitKey(0)              # Wait time until image closes
```

The results are shown in Figure **??**.

The `img` object called by `cv2.imread()` actually outputs a numpy array directly. This makes it easy for us to access the resolution of the image and to crop it by slicing across the dimensions. We can also rescale it accordingly using `cv.resize()`.

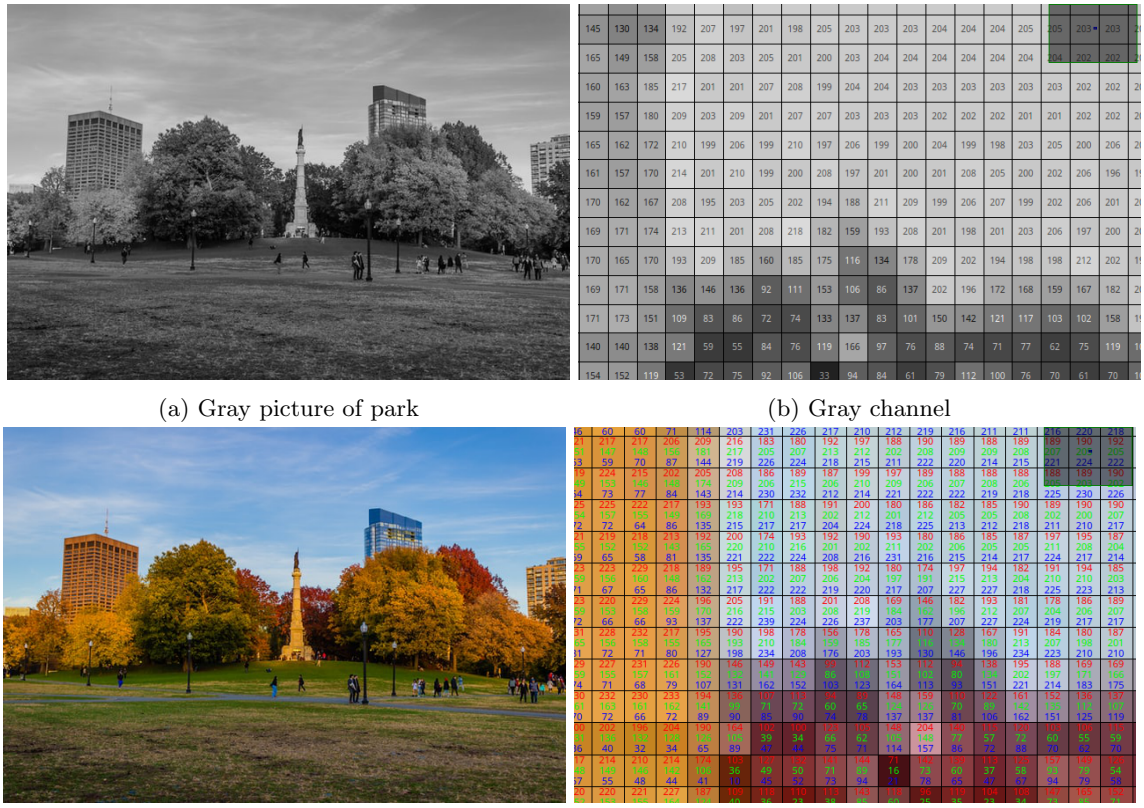
```
print(img.shape)        <class 'numpy.ndarray'> (427, 640, 3)
print(gray.shape)       <class 'numpy.ndarray'> (427, 640)

# Crop it
img_cropped = img[100:200, 100:200, :]

# Resize the image
width, height = int(frame.shape[1] * 0.6), int(frame.shape[0] * 0.6)
dimensions = (width, height)
img_resized = cv2.resize(img, dimensions, interpolation=cv2.INTER_AREA)
```

2.1.1 Color Histograms

You can also find the distribution of the color channels in an image with a histogram. A code snippet is shown below, along with the corresponding generated plots in Figure **??**.



(a) Gray picture of park (b) Gray channel

(c) Colored picture of park (d) RGB channels

Figure 1: Different color channel representations of an image.

```
img = cv2.imread("Cats.jpg)
cv2.imshow("Cats", img)

# Color histogram
colors = ('b', 'g', 'r')
for i, col in enumerate(colors):
    hist = cv2.calcHist([img], [i], None, [256], [0, 256])
    plt.plot(hist, color=col)
    plt.xlim([0, 256])

plt.show()
cv2.waitKey(0)
```

2.2 Drawing and Transformations

2.2.1 Masking

2.3 Kernels

Now a convolution is described by a **kernel**, also called a **filter**, which is simply a $K \times K$ matrix. It does not have to be square but is conventionally so. It goes through a grayscale image at every point and compute the dot product of the kernel with the overlapping portion of the image, creating a new pixel. This can be shown in Figure 3.

Now if this was a color image, then the $K \times K$ kernel $\mathcal{K}$ would dot over all 3 layers, without changing

(a) Gray picture of park (b) Gray channel

Figure 2: RBG Channel Histograms for Cats.png image.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Image →

-1	0	1
-2	0	2
-1	0	1

Kernel →

20	30	40
45	55	65
70	80	90

Figure 3: Convolution using a kernel on an image.

over all 3 layers. This is equivalent to applying the kernel over all 3 channels separately, and then combining them together into one. Another thing to note is that the output image of a kernel would be slightly smaller than the input image, since the kernel cannot go over the edge. However, there are padding schemes to preserve the original dimensions. To construct our custom kernel, we can simply create a custom matrix:

```
img = cv2.imread("cats.jpg")

# create custom 5x5 kernel
kernel = (1/25) * np.ones((5, 5), dtype=np.float32)

# apply to image
dst = cv2.filter2D(img, -1, kernel)
cv2.imshow("Park", dst)
cv2.waitKey(0)
```

Note that the kernel matrix may have the property that all of its entries sum to 1, meaning that on average, the expected value of the brightness of each pixel will be 0, and the values will be left unchanged on average. However, this is not a requirement.

Example 2.1 (Mean Blur, Gaussian Blur)**.** The mean and Gaussian blur is defined with kernels that are distributed uniformly and normally across the entire matrix. You can see how this would blur an image

since for every pixel, we take the weighted average over all of its surrounding pixels.

$$\text{mean} = \frac{1}{25}\begin{bmatrix}1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\end{bmatrix}, \quad \text{Gaussian} = \frac{1}{273}\begin{bmatrix}1 & 4 & 7 & 4 & 1\\ 4 & 16 & 26 & 16 & 4\\ 7 & 26 & 41 & 26 & 7\\ 4 & 16 & 26 & 16 & 4\\ 1 & 4 & 7 & 4 & 1\end{bmatrix}$$

On a large scale, there really aren't any discernable differences, as seen in Figure 4, but the Guassian blur is known to be a more realistic representation of how humans receive blur.

(a) Original image. (b) 5×5 mean blur applied. (c) 5×5 Gaussian blur applied.

Figure 4: Comparison of blurring kernels on image.

Example 2.2 (Sharpening)**.** A sharpening of an image would be the opposite of a blur, meaning that we emphasize the center pixel and reduce the surrounding pixels.

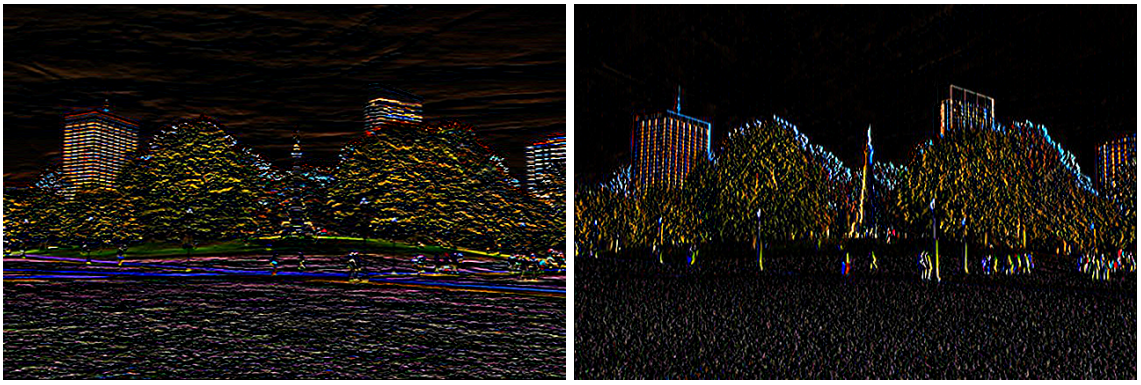
$$\text{Sharpen} = \begin{bmatrix}0 & -1 & 0\\ -1 & 5 & -1\\ 0 & -1 & 0\end{bmatrix}$$

(a) Original image. (b) 3×3 sharpening applied.

Figure 5: Sharpening kernels applied to image.

Example 2.3 (Edge Detection)**.** The edge detecting kernel looks like the following, which differs for horizontal and vertical edge detection. Note that the sum of all of its values equal 0, which means that for areas that have a relatively constant value of pixels, all the surrounding ones will "cancel" out and the kernel will output a value of 0, corresponding to black. This is why we see mostly black in the photo.

$$\text{Horizontal} = \begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix}, \quad \text{Vertical} = \begin{bmatrix}-1 & -2 & -1\\ 0 & 0 & 0\\ 1 & 2 & 1\end{bmatrix}$$



(a) 3×3 horizontal edge detecting kernel applied. (b) 3×3 vertical edge detecting kernel applied.

Figure 6: Edge detecting kernels applied to image.

3 Region Based Object Detection

So far, we can use CNNs to either classify or regress an input image. However, the task of **object detection** requires us to draw bounding boxes around an arbitrary number of objects within an image *and* correctly label each one. This seems like quite an enormous task, but we can build it up step by step.

3.1 RCNN

3.2 Fast RCNN

3.3 Faster RCNN

3.4 Mask RCNN

4 YOLO

5 Simple Online Realtime Tracking

5.1 SORT

5.2 DeepSORT

5.3 StrongSORT

6 ByteTrack