Computer Vision

Muchang Bahng

March 12, 2024

Contents

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 ??.

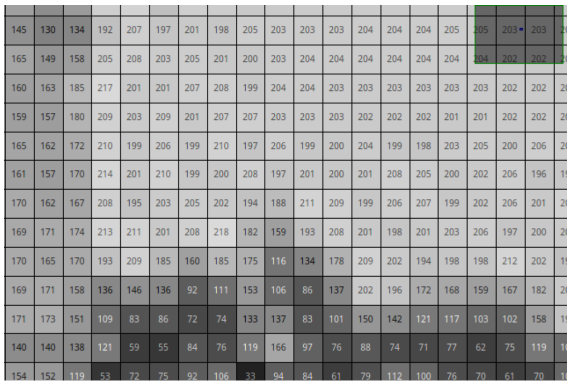
The `img` object called by `cv2.imread()` actually outputs a numpy array directly. This makes it easy for us to access the resolution of the image and to crop it by slicing across the dimensions. We can also rescale it accordingly using `cv.resize()`.

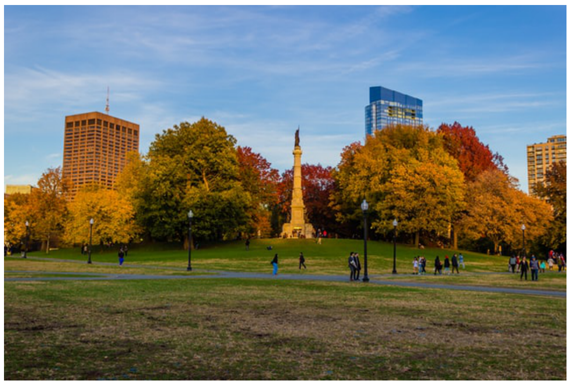
```
print(img.shape)        <class 'numpy.ndarray'> (427, 640, 3)
print(gray.shape)       <class 'numpy.ndarray'> (427, 640)

# 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

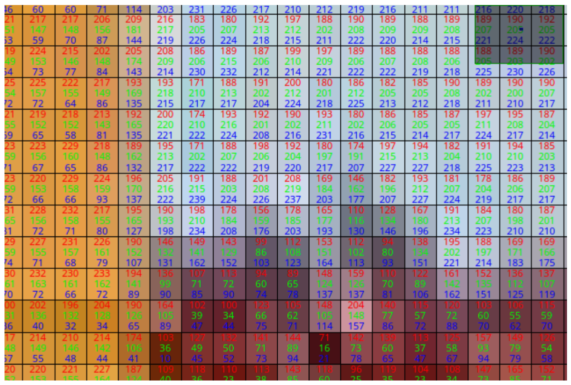

(d) RGB channels

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

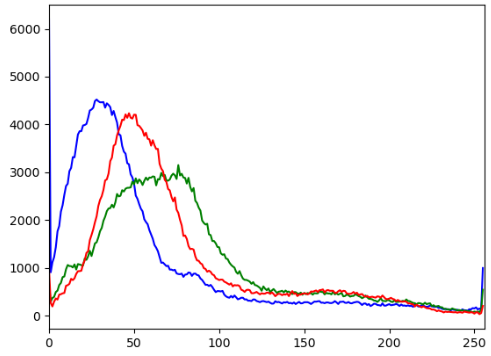
2.2.1 Masking

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.

$$\begin{array}{|c|c|c|c|c|}\hline 1 & 2 & 3 & 4 & 5 \\ \hline 6 & 7 & 8 & 9 & 10 \\ \hline 11 & 12 & 13 & 14 & 15 \\ \hline 16 & 17 & 18 & 19 & 20 \\ \hline 21 & 22 & 23 & 24 & 25 \\ \hline \end{array} \xrightarrow{\text{Image}} \begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline -2 & 0 & 2 \\ \hline -1 & 0 & 1 \\ \hline \end{array} \xrightarrow{\text{Kernel}} \begin{array}{|c|c|c|}\hline 20 & 30 & 40 \\ \hline 45 & 55 & 65 \\ \hline 70 & 80 & 90 \\ \hline \end{array}$$

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)
```

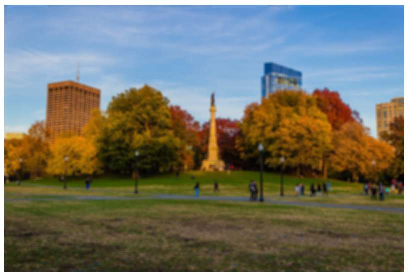
Note that the kernel matrix may have the property that all of its entries sum to 1, meaning that on average, the expected value of the brightness of each pixel will be 0, and the values will be left unchanged on average. However, this is not a requirement.

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.

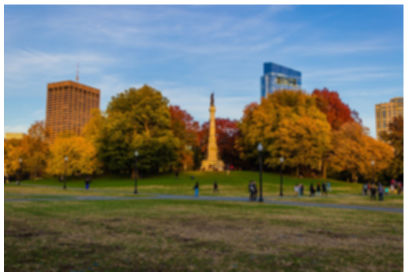
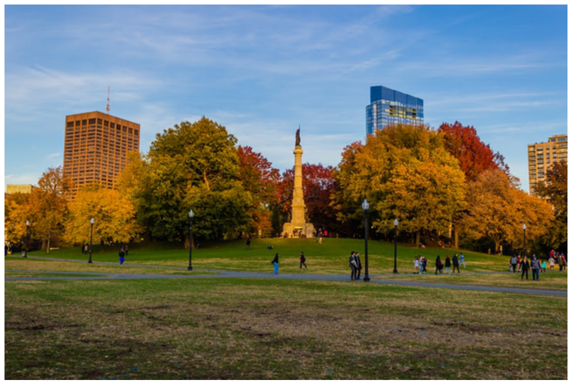

(c) 5×5 Gaussian blur applied.

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.

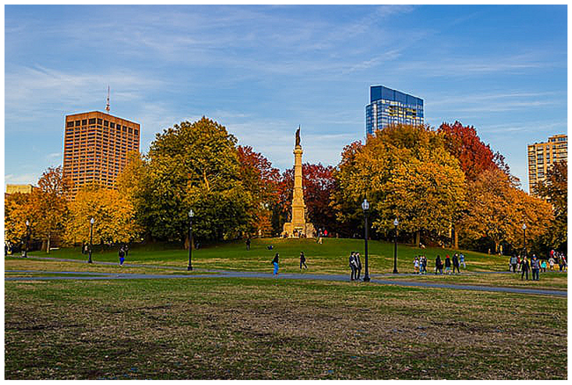

(b) 3×3 sharpening applied.

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.

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

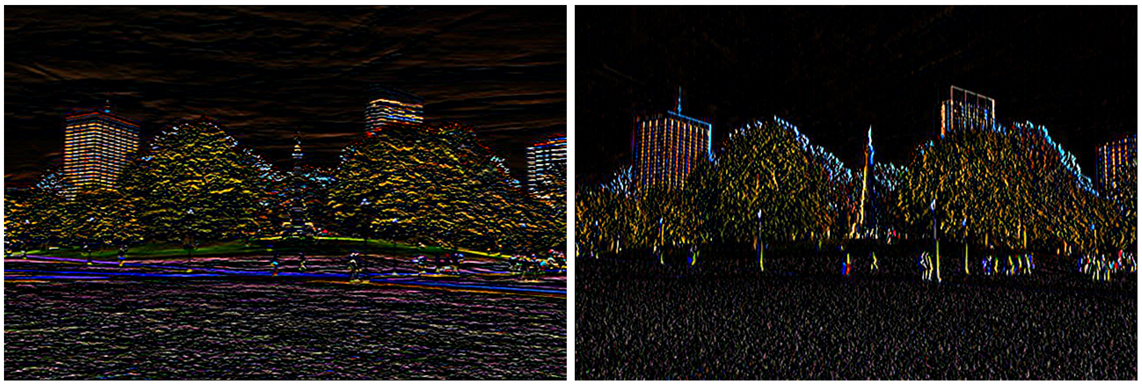

(a) 3×3 horizontal edge detecting kernel applied. (b) 3×3 vertical edge detecting kernel applied.

Figure 6: Edge detecting kernels applied to image.

3 Object Detection

So far, we have talked about classification, but this is fundamentally a different problem than object detection. Many aspects will be shared between the two problems. Essentially in detection algorithms, we try to draw a bounding box around the object of interest to locate it within the image. There could be many bounding boxes representing different objects of interest, and what makes this so challenging is that we don't know how many beforehand. Because of these bounding boxes, we must use models that predict also the number and the dimensions of these boxes, use datasets that provide these bounding boxes in addition to labels (e.g. COCO, VOC), and finally we must construct loss functions that take into consideration these boxes.

Therefore, the major reason why you cannot proceed with this problem by building a standard CNN followed by a fully connected layer is that the length of the output layer is variable. A naive approach to solve this problem would be to take different regions of interest from an image and use a CNN to classify the presence of the object within that region. That is, we do the following steps.

1. We have an input image $\mathbf{x}$, and we want to define a set of subimages $\mathcal{R}$. Each element of $\mathcal{R}$ is essentially a box that segments out a portion of the image $\mathbf{x}$, and we can parameterize this box with 4 numbers. Two of the most popular ones is to take the center of the box plus the width and height (c_x, c_y, w, h) or to take the top-left and bottom-right corners $(x_{min}, y_{min}, x_{max}, y_{max})$. Essentially, we have a giant set of vectors in $\mathbb{R}^4$, and each element is called a **region of interest**.

2. We now take each ROI and input this to our trusty CNN and do the classification task that we have studied up until now. This should tell us whether there is our object of interest in the image or not, and if it passes a certain threshold, we can have it infer that this ROI contains the object. If multiple ROIs pass the threshold, then we can pick the ROI for which the CNN outputs the highest probability of the object existing in.

This workflow essentially just extracts the regions of interest and does vanilla classification on them. This is an intuitive extension of what we know, but there are a few problems with this approach:

1. A small problem is that the ROIs in $\mathcal{R}$ may not be the same size, while our CNN feature extractor requires its inputs to be fixed. However, this isn't too bad since we can just resize the ROIs for preprocessing. We could also just have $\mathcal{R}$ contain all images of the same size, but this will really hamper the robustness of the model. Objects can be of different size, and so intuitively, we must have different sized ROIs.

2. There is the bigger question of how we actually construct $\mathcal{R}$. We have a tradeoff. If $\mathcal{R}$ is too small, it may not capture the structure of the objects of interest. If it is too large, then we may have a combinatoral explosion of ROIs to look for. For example, if we simply have a sliding window with a stride of 1 that adds each image into $\mathcal{R}$, for an $N \times N$ image we would have $O(N^2)$ ROIs, and this

is only for one resolution! This may have been feasible if we were working with linear models, but in the deep learning regime this is not. This second problem is what we will focus on when explaining R-CNNs.

img/sliding_window.png

3.1 Region-Based CNN

The region-based CNN first bounds $|\mathcal{R}| \leq 2000$ and uses classical image processing techinques to reduce the number of ROIs to less than 2000.

1. We start off by using the **selective search algorithm**. It over-segments the image into seed regions, with each region generating a bounding box. This leads to a ton of regions to look for, and to reduce $|\mathcal{R}|$, we use a greedy algorithm to recursively combine similar regions into larger ones. The segmentation algorithm is the basis of the region proposals, since it matches similar pixels together for a good intialization, giving us the best balance between computational feasibility and quality of proposals. It is specifically designed to have high recall, but low precision, which means that it returns many false positive regions, but are quite certain that they contain the objects of interest.

   img/selective_search.png

2. This usually leads to about 2000 ROIs, and for each ROI, we warp them into a square and feed them into a CNN, which produces a 4096-dimensional feature vector as output.

3. The CNN acts as a feature extractor and the output dense layer consists of the feature extracted from the image. This 4096-vector is fed into a support vector machine to classify the presence of the object within that candidate region proposal.

   img/rcnn_diagram.png

4. In addition to predicting the presence of an object within the region proposals, the algorithm also predicts 4 values which are offset values to increase the precision of the bounding box. For example, given a region proposal, the algorithm would have predicted the presence of a person but the face of that person within that region proposal could have been cut in half. Therefore, these offset values can help in adjusting the bbox.

   img/rcnn_diagram2.png

However, there are still several problems:

1. While we have improved the computational cost by a lot, we still have to run the CNN feature extractor on *each* of the 2000 region proposals, which is still too slow for real time inference (takes about 47 seconds), and it takes too long to train.

2. When we reshape each region proposal into a square, it may warp the image, causing it to lose its original features.

3. The selective search algorithm is not a learning algorithm, so it may generate bad candidate region proposals for various types of images.

4. Finally, the R-CNN model is not trained on an end-to-end fashion (the SVM) is trained separately, so this may reduce potential performance.

Definition 3.1 (IOU)**.** To provide a measure of how good these boxes are, we use the **intersection over union (IOU)** metric. If B is the true bounding box and $\tilde{B}$ our estimate of it, then we have

$$\text{IoU} = \frac{\mu(B \cap \tilde{B})}{\mu(B \cup \tilde{B})}$$

The closer it is to 1, the better, and a "good" match is defined to be 0.7 or above.

3.2 Fast RCNN

Fast RCNN is basically just RCNN but with a few tweaks to make it fast. The computational bottleneck came from having to run a CNN on each of the 2000 region proposals, but now, to speed things up, we have a CNN extract features from the whole image first. We describe the steps below.

1. We take the image $\mathbf{x} \in \mathbb{R}^{n \times m}$ and use the CNN f to extract features from it, resulting in say $\boldsymbol{\phi} \in \mathcal{F}$.
2. At the same time, we run the selective search algorithm on $\mathbf{x}$ to get our region proposals $\mathcal{R}$.
3. For each region of interest $\mathbf{r} \in \mathcal{R}$, we project the resulting regions coordinate into the feature map $\mathcal{F}$. Note that we are not running inference here, just projecting, so this is computationally cheap and allows us to reduce the computational cost of extracting features by an order of 2000. Specifically, if there is a convolution on a ROI with stride 1 and the proper padding, the regions will have the same coordinates. Meanwhile, if we have a max pooling layer, then each coordinate (x, y) will get mapped to $(\lceil x/2 \rceil, \lceil y/2 \rceil)$.
4. After this projection, for every $\mathbf{r} \in \mathcal{R}$, we have its projections $p(\mathbf{r}) \in \mathcal{F}$. However, this projection is not the same le,gth for all region proposals, so we have a **RoI pooling layer** which takes each projected region, divides it into a fixed number of bins (independent of the input shape), and does max pooling over each bin to generate a fixed feature vector for each region proposal.

   img/roi_pooling.png

5. Then we take these fixed size feature vectors and feed them through a fully connected neural network to get a feature vector where we can do softmax classification on, along with regression of the bboxes dimensions.

   img/fast_rcnn.png

Regarding the problems, this first improves the computational cost compared to R-CNN. By utilizing the IOU pooling layers, we have resized the images after the CNN feature extractor rather than distorting the original input itself. Moreover, we have replaced the SVM classifiers with neural nets. However, we still have the problem that the region proposal algorithm is an external algorithm that's not learned, and this selective search method still turns out to be quite slow for real-time inference.

3.3 Faster RCNN

Therefore, Ren et al, came up with an object detection algorithm that eliminates the selective search algorithm and lets the network learn the region proposals. The Faster RCNN model essentially repalces the region proposal algorithm with a significantly faster neural network that can learn to propose better regions for the task at hand.

img/faster_rcnn_diagram.png

Let's walk through the steps of this algorithm:

1. We take the image $\mathbf{x}$ and run it through a backbone convolutional network (usually the first few layers of ResNet or VGG), generating a feature map $\phi \in \mathbb{R}^{K \times K}$. There are K^2 "points" or "pixels" in this feature representation of $\mathbf{x}$, and **anchor points** are generated on ϕ . If they are projected back into the original image, we would have K^2 equally spaced points over $\mathbf{x}$.

   `img/anchor_points.png`

2. For each anchor point, we have k predefined boxes of different sizes and aspect ratios that are used to generate region proposals that may or may not contain objects of interest. For example, we can have $k = 4$ in the picture below.

   `img/anchor_boxes.png`

3. At this point, we have a total of kK^2 total anchor boxes on the feature space. We project all this back to the original image space, and we look for the samples that have a good IoU (calculated in the feature space with the projected ground truth boxes!) with the ground truth bbox. We generate (usually an equal amount) of positive and negative samples from the kK^2 total boxes.

   `img/gt_sampling.png`

4. For each of the kK^2 boxes, we want to classify each as an object or background, along with predicting their offsets from the corresponding gt bboxes. To do this, we use 1×1 convolutional layers. We take the feature map, which is of shape $(C, K, K) = (2048, 8, 8)$ and use the kernel to give us an output of size (k, K, K). This output basically labels each of the kK^2 anchor boxes with some scalar that represents whether we think it is an object or background. We take a second 1×1 kernel and have it output $(4k, K, K)$ representing the offsets of the bounding boxes. This is called the regression head, and we need $4k$ since there are 4 degrees of freedom in adjusting each bounding box.

5. Now we have the scores and offsets of all the anchor boxes. But during training we only select the positive and negative anchor boxes to compute classification loss, and only positive boxes to compute the L2 regression loss.

6. In this second stage, we receive region proposals and predict the category of the object in the proposals. These region proposals (due to the anchor boxes being different sizes) are not the same size, so we use RoI pooling just like in Fast RCNN. After this, it's the same: we pass them through a fully connected MLP with some softmax and regressor link.

7. During inference, we pass the image through the backbone network to generate anchor boxes. Then, we select only the top 300 boxes that get a high classification score and qualify them for the next stage. We then predict the final categories and offsets, performing an extra post-processing step called non-max surpression to remove duplicate bounding boxes.

Note that since we are using convolutions, this makes the object detection algorithm translationally invariant, and rotational invariance can be achieved through data augmentation.

3.4 Measuring Performance

Recall that if we are just predicting the presence of 1 class, then we can talk about the recall or precision. A better way is to look at the mean precision, which is found by taking the integral of the precision recall curve. Other metrics include the F_1 score.

$$\text{Precision} = \frac{TP}{TP + FP}, \text{ Recall} = \frac{TP}{TP + FN}, \; F_1 = 2\frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

For multiple classes, we can use the **mean average precision**, which is basically the average precision for multiple classes.

4 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.

4.1 Mask RCNN

5 YOLO

6 Simple Online Realtime Tracking

6.1 SORT

6.2 DeepSORT

6.3 StrongSORT

7 ByteTrack