# Computer Vision

Muchang Bahng

June 23, 2023

## Contents

We will use OpenCV and PyTorch.

# 1 Image Processing

## 1.1 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 1.

(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.

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)
```

### 1.1.1 Color Histograms

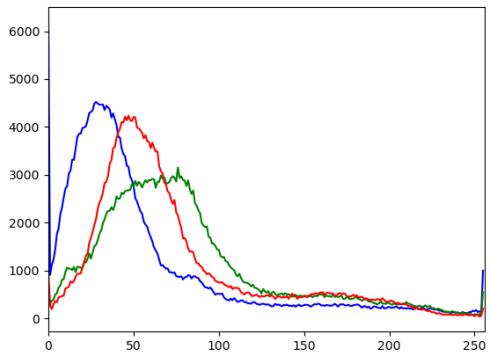
You can also find the distribution of the color channels in an image with a histogram. A code snippet is shown below, along with the corresponding generated plots in Figure **??**.

```
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)
```

(a) Gray picture of park




(b) Gray channel

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

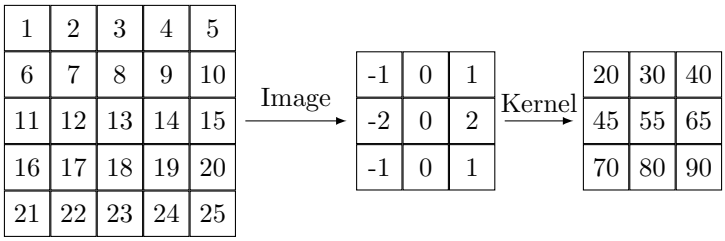



Figure 3: Convolution using a kernel on an image.

## 1.2 Drawing and Transformations

### 1.2.1 Masking

## 1.3 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 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 1.1** (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 **??**, but the Guassian blur is known to be a more realistic representation of how humans receive blur.

**Example 1.2** (Sharpening)**.** A sharpening of an image would be the opposite of a blur, meaning that we

(a) Original image. (b) $5 \times 5$ mean blur applied. (c) $5 \times 5$ Gaussian blur applied.

Figure 4: Comparison of blurring kernels on image.

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) $3 \times 3$ sharpening applied.

Figure 5: Sharpening kernels applied to image.

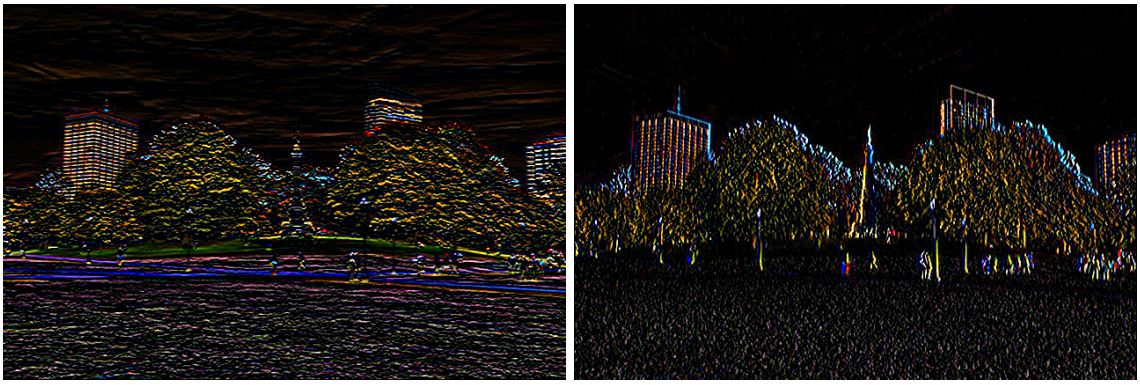
**Example 1.3** (Edge Detection)**.** The edge detecting kernel looks like the following, which differs for horizontal and vertical edge detection. Note that the sum of all of its values equal 0, which means that for areas that have a relatively constant value of pixels, all the surrounding ones will "cancel" out and the kernel will output a value of 0, corresponding to black. This is why we see mostly black in the photo.

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

# 2 Convolutional Neural Networks

## 2.1 Convolutional Layers

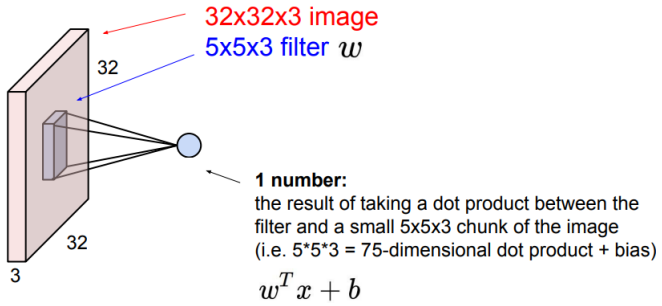
Given an image of dimension $W \times H \times D$ (where $D$ is the depth, or number of color channels), we can take a convolution over this image by multiplying the matrix $\mathcal{K} \in \mathbb{R}^{K \times K}$ over each depth channel. Before, we have said that $\mathcal{K}$ is applied to each color channel layer, so the total number of parameters that govern this



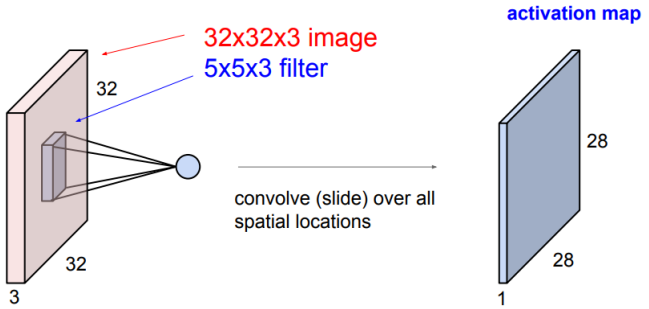
(a) $3 \times 3$ horizontal edge detecting kernel applied. (b) $3 \times 3$ vertical edge detecting kernel applied.

Figure 6: Edge detecting kernels applied to image.

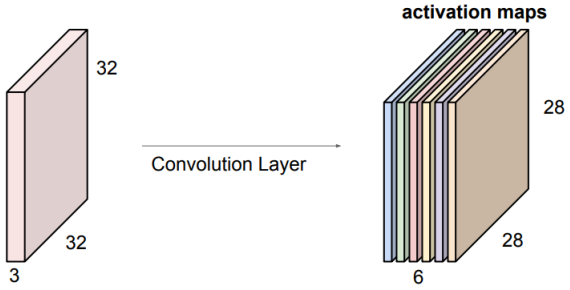
convolution is $K^2$. For greater flexibility, we would like our kernel to be of form $\mathcal{K} = \mathbb{R}^{K \times K \times D}$, where the kernel applied in each color channel could be different. Furthermore, we would like a bias term $b \in \mathbb{R}$, giving us a total of $DK^2 + 1$ parameters to optimize over to get the "best possible" convolution.




It is almost always the case that $D = 3$, so we will keep it to $3K^2 + 1$. A possible hyperparameter that needs to be set is the value of $K$, determining the **width** of the kernel. Another hyperparameter is the **stride** of the kernel, which determines how many pixels you want the filter to move. The filter will start on the top left of the image and stride (at a certain rate) to the right and down until it reaches the bottom right. Once finished, we should have a transformed image, which we can invoke our activation function $\boldsymbol{\sigma}$ on for each element (pixel) to introduce some nonlinearity.




If one filter $\mathcal{K}$ gives us one output image, then a collection of filters $\mathcal{K}_1, \dots, \mathcal{K}_l$ (almost always assumed to be of the same size) will give us a collection of $l$ images, or also known as one image of depth $l$.




These convolutions help extract some sort of information, and our choice of convolutions (i.e. the $3K^2$ numbers) will extract different information. Therefore, this is called a **convolution layer**. As we have more and more convolution layers, we are able to extract from low-level, to mid-level, to high-level features in an image that we can ultimately train on. Ultimately, we would like to stack these layers together enough so that we can have our features, and then we run a few layers of MLP to get our prediction.

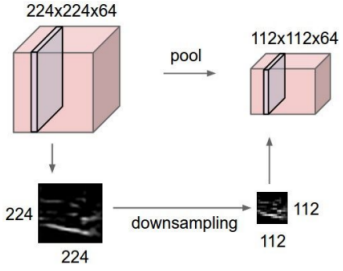
### 2.1.1 Convolutions as Sparase Matrix Multiplication

In a way, this is really just a giant sparse matrix multiplication since we can stretch the image out, and each convolution on a pixel is just a linear combination of the pixels around it (plus the ones around it in other channels). In a vanilla MLP, you would optimize all the parameters possible of a linear map over the input images, but a convolutional one takes a very small subset of parameters to optimize, setting everything else to 0. There are three reasons:

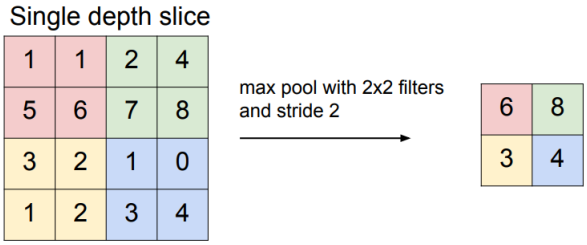
1. If the dimensions of the input image was $D_1$ and that of the output image was $D_2$, then we would have to optimize a matrix with $D_1 \times D_2$ parameters. Note that the smallest images in today's standards are $3 \times 256 \times 256 \approx 200,000$ pixels, and so $D_1 \times D_2 \approx 4 \cdot 10^{10}$, which is too many even for one layer.
2. There is an MLP way of doing computer vision, and we have performed MLP on very small grayscale images like the MNIST with decent results. Though MLP is more generalized, CNNs have comparable performance at a fraction of the computational cost.
3. The images are spatially ordered data. That is, the order in which the pixels are arranged to form the matrix provides a great help in features extraction, so this sense of locality in convolutions help.

## 2.2 Pooling Layers

Eventually, we would like to use convolutional layers to extract perhaps a few hundred or thousand features that we can then run a MLP on for the last few layers, followed by whatever final activation function needed for prediction. The problem is the high-dimensionality of our inputs. Recall that even the smallest images are $3 \times 256 \times 256 \approx 200,000$ pixels. Even after multiple convolutional layers, the dimensions may not decrease as fast as we want, so we introduce **pooling layers**, which are efficient layers that decrease the dimension of its inputs in a controlled way, called **downsampling**.

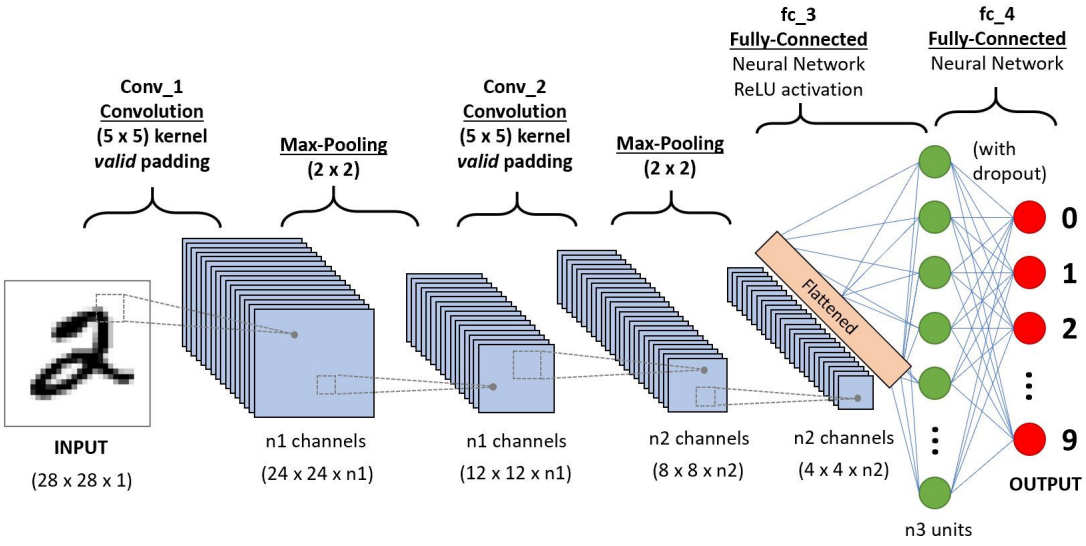



We can think of this as decreasing the resolution of the image, and the most common way is through **max pooling**. You basically have a $P \times P$ square window with some stride, and for each stride, we take the largest value in the window.




## 2.3 Total Architecture

We have a bunch of convolutional layers with RELU, along with some pooling layers in between. Once we've done enough pooling, we just stretch the resulting image out and run a MLP on the rest with a softmax link function.




We can implement it directly in PyTorch by calling `nn.Conv2d()`, where the parameters are number of input channels, number of output channels, and size of kernel (along with optional stride, padding, bias parameters), and `nn.MaxPool2d()`, which take in the kernel size and stride. The rest of the code is completely the same.

```
class NeuralNetwork(nn.Module):
    # Convolutional Neural Network

    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)


    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

# 3 Network Training

## 3.1 Backpropagation

## 3.2 Implementation from Scratch

## 3.3 CNN Architectures