Lecture 12: Convolutional Neural Networks

ENEE 691 – Machine Learning and Photonics @ UMBC

Ergun Simsek, Ph.D. and Masoud Soroush, Ph.D.

April 17, 2023

1 Convolutional Neural Networks

In this chapter[1], we introduce another important building block of neural networks that is called **convolutional neural networks** or CNN for short, which has a wide range of applications in optical character recognition and photonics computing.

1.1 Background

Convolution is an operation which finds its roots in mathematics. In the context of functional analysis, convolution is an operation that receives two functions f and g as inputs, and produces a third function $f \odot g$ as the output of the operation. The graph of the output function $f \odot g$ indicates how the graph of f is modified by the shape of the graph of g, when the two graphs (graphs of f and g) are combined. It turns out that convolutions have a wide range of applications in many fields of research including in **probability theory**, **statistics**, **acoustics**, **spectroscopy**, **signal processing and image processing**, **geophysics**, **physics**, and **computer vision**. In the context of neural networks, we will be mainly interested in convolutions applied to images. Convolutions enable us to construct more accurate models for both classification and regression, in particular when the features are 2D or 3D images.

Brief History: In 1980s, Dr. Kunihiko Fukushima designs an artificial neural network that mimics the functioning of simple and complex cells. While S-cells operate as artificial simple cells, C-cells operate as artificial complex cells. They are artificial because they are not biological neurons, but instead, they mimic the algorithmic structure of simple and complex cells. The main idea of Fukushima's Neocognitron was simple: Capture complex patterns (e.g., a dog) using complex cells that gather their information from other lower-level complex cells or simple cells that detect simpler patterns (e.g., a tail).[2]

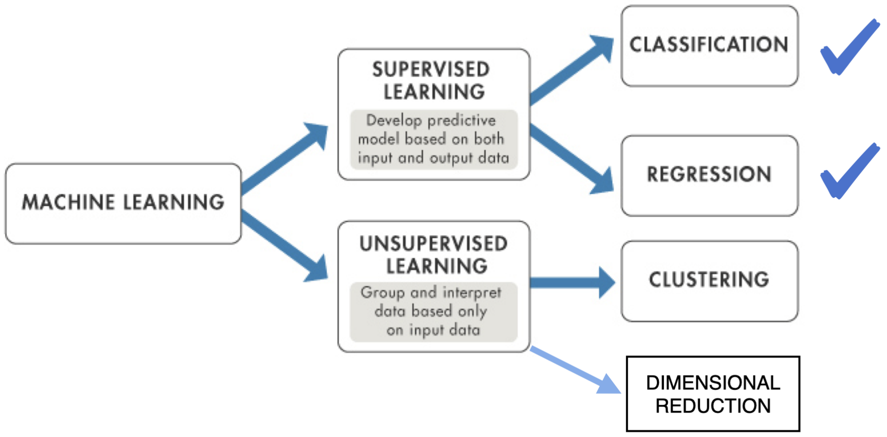
Although the work of Fukushima was very powerful in the newly developing field of artificial intelligence, the first modern application of convolutional neural networks was implemented in the 90s by Yann LeCun *et al.* in their paper Gradient-Based Learning Applied to Document Recognition, which is probably by far the most popular AI paper from the 90s (cited 54k+ times as of April 2023).[3] YaCun trained a convolutional neural network with the MNIST dataset of handwritten digits. The idea was a follow-up of Fukushima's Neocognitron: Aggregating simpler features into more complicated features using complex artificial cells. LeCun's implementation set the standards for today's computer vision and image processing applications.

Convolutional operations on images have a longstanding history, but they resurfaced at around 2010 in the context of deep learning with a huge success on image classification tasks. In 2012, a deep convolutional neural network architecture called AlexNex achieved a 16% error rate (10% lower than the runner-up) by

[1] This product is a property of UMBC, and no distribution is allowed. These notes are solely for your own use, as a registered student in this class.

[2] https://www.rctn.org/bruno/public/papers/Fukushima1980.pdf

[3] http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf



utilizing GPUs. The use of GPUs for computer vision tasks became standard after AlexNext's incredible achievement for its time.[4] Since then convolutional neural networks have been one of the prominent architectures in deep learning, and they have offered a vast range of applications in the field of computer vision. In 2017, 29 of 38 competing teams at the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) achieved less than 5% error.[5]

Today, we will introduce a particular type of convolution applied to 2D images, and will build neural networks based on this type of convolution. This chapter is just an introduction to the subject, and our treatment is by no means comprehensive. Interested readers should aim for more advanced courses on deep learning to explore the subject further.

1.2 Discrete Convolutions

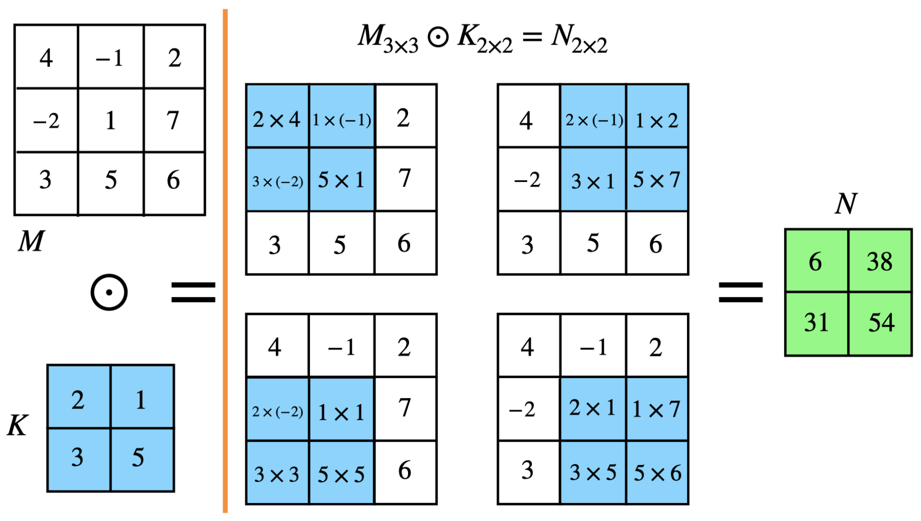
We start by introducing the a special type of convolution operation that is commonly used for constructing neural networks to analyze 2D images. This type of convolution is referred to as the *discrete convolution*. It should be noted that discrete convolutions can be defined for 1D, 2D, and 3D inputs (they can even be generalized to higher dimensions than 3). However, in this chapter, we will focus on discrete convolutions in 2D. The inputs of a two-dimensional discrete convolution operation are two matrices. The matrix with the larger size encodes a two-dimensional image, and the matrix with the smaller size is referred to as the *kernel* (or the *filter*) of the convolution operation.

Before we offer a general definition for the two-dimensional discrete convolution operation, we first explain the basic idea through a simple but explicit example. Suppose M and K are 3×3 and 2×2 matrices, respectively. Explicitly, M and K are given by

$$M = \begin{pmatrix} 4 & -1 & 2 \\ -2 & 1 & 7 \\ 3 & 5 & 6 \end{pmatrix}, \qquad K = \begin{pmatrix} 2 & 1 \\ 3 & 5 \end{pmatrix}. \tag{1}$$

We would like to convolve matrix M with kernel K, and find matrix $N = M \odot K$ (*i.e.* the result of the 2D discrete convolution). The following picture depicts how the two-dimensional convolution works.

[4] https://papers.nips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf
[5] https://www.image-net.org/challenges/LSVRC/



As is clear from above picture, in order to perform the convolution, we first place the kernel on the top left corner of matrix M. We then multiply the entries of kernel K with those entries of M that lie underneath of the kernel. We then add the four entries after multiplication and this would form the very first entry of the final result N. Once this is complete, we move the kernel one cell to the right, and perform the same operation and find the next entry of N. Once the entries of the first row of M are all covered, we slide the kernel down by one cell. We repeat the same process and find new entries of N until all entries of the second row are covered. At the end of the process, we find all elements of the 2×2 matrix N. In general, when matrices M and K have arbitrary entries

$$M = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix}, \qquad K = \begin{pmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{pmatrix}, \tag{2}$$

the resulting 2×2 convolved matrix $N = M \odot K$ is given by

$$\begin{aligned} M \odot K &= \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} \odot \begin{pmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{pmatrix} \\ &= \begin{pmatrix} k_{11}m_{11} + k_{12}m_{12} + k_{21}m_{21} + k_{22}m_{22} & k_{11}m_{12} + k_{12}m_{13} + k_{21}m_{22} + k_{22}m_{23} \\ k_{11}m_{21} + k_{12}m_{22} + k_{21}m_{31} + k_{22}m_{32} & k_{11}m_{22} + k_{12}m_{23} + k_{21}m_{32} + k_{22}m_{33} \end{pmatrix}. \end{aligned} \tag{3}$$

Although what we did above concerned a specific example with the image and kernel with some fixed sizes, we can easily generalize the above operation to images and kernels of arbitrary size. Now, suppose that the image A is given by an $m \times n$ matrix, and the kernel K is a $k \times \ell$ matrix, with $k \leq m$ and $\ell \leq n$. We would like to convolve matrix A with kernel K. The result of the convolution $A \odot K$ turns out to be an $(m - k + 1) \times (n - \ell + 1)$ matrix. The following formula gives the entries of the convolved matrix $A \odot K$ explicitly

$$\boxed{(A \odot K)_{rs} = \sum_{i=1}^{k} \sum_{j=1}^{\ell} k_{ij}\, a_{i+r-1,j+s-1}\,, \qquad \text{for} \quad \begin{cases} r = 1, 2, \cdots, m - k + 1\,, \\ s = 1, 2, \cdots, n - \ell + 1\,. \end{cases}} \tag{4}$$

Equation (4) is completely general, and calculates all entries of the convolved matrix $A \odot K$ through a double sum formula. We can easily define basic python functions that implement (4) and calculate the convolved matrix. In the following, we have done this task in two steps. We first implement the discrete convolution in 1D by sliding the kernel horizontally, and then include sliding the kernel vertically in the next step.

[1]:
```
import numpy as np

def convolve_1d(image, kernel):
    ks = kernel.shape[0]
    final_length = image.shape[0] - ks + 1
    return np.array([(image[i:i+ks]*kernel).sum() for i in range(final_length)])

def convolve_2d(image, kernel):
    ks = kernel.shape[1]
    final_height = image.shape[1] - ks + 1
    return np.array([convolve_1d(image[:, i:i+ks], kernel) for i in␣
 ↪range(final_height)]).T
```

To make sure the above functions work in the right way, we can feed the example in equation (1) to the functions and calculate the convolved matrix.

[2]:
```
M = np.array([[4, -1, 2], [-2, 1, 7], [3, 5, 6]])  # Defining matrix M as an numpy␣
 ↪array
K = np.array([[2, 1], [3, 5]])                     # Defining matrix K as an numpy␣
 ↪array

convolve_2d(M, K)   # Computing the convolved matrix
```

[2]:
```
array([[ 6, 38],
       [31, 54]])
```

As is expected, the result of the python function for convolution coincides with the result presented in the first picture. Now, we would like to have an example where a rectangular matrix is convolved with a rectangular kernel. To make sure that the definition of the 2D discrete convolution has been appreciated, the reader is recommended to do the arithmetic in the following example by hand and compare the result with the outcome of our python function convolve_2d.

Example: Let A and K be $A = \begin{pmatrix} 1 & 2 & 2 & 1 & 3 \\ 2 & 4 & 6 & 1 & 3 \\ -3 & 1 & 4 & 2 & 4 \\ 7 & -2 & -3 & -1 & 5 \\ 2 & 1 & 5 & 7 & 1 \\ 4 & -2 & 1 & 2 & 5 \end{pmatrix}$ and $K = \begin{pmatrix} 1 & -2 & 1 \\ 3 & 2 & -2 \end{pmatrix}$. Convolve matrix A with filter K to find $A \odot K$.

As is clear, A is a 6×5 matrix, and the kernel is a 2×3 matrix. According to formula (4), the convolved matrix $A \odot K$ must be a $(4 - 2 + 1) \times (5 - 3 + 1) = 5 \times 3$ matrix. We can easily find the result using convlove_2d function.

[3]:
```
A = np.array([[1, 2, 2, 1, 3],
              [2, 4, 6, 1, 3],
              [-3, 1, 4, 2, 4],
              [7, -2, -3, -1, 5],
              [2, 1, 5, 7, 1],
              [4, -2, 1, 2, 5]])

K = np.array([[1, -2, 1], [3, 2, -2]])
```

```
print('Size of A =', A.shape)
print('Size of K =', K.shape)
print('Size of A convolved by K =', convolve_2d(A, K).shape)

convolve_2d(A, K)
```

```
Size of A = (6, 5)
Size of K = (2, 3)
Size of A convolved by K = (5, 3)
```

```
[3]: array([[  1,  21,  17],
            [-15,   0,  15],
            [ 22, -15, -17],
            [  6,   2,  31],
            [ 11, -10, -11]])
```

1.3 Padding Technique

As is seen in previous examples, whenever we apply a kernel whose size is greater than 1×1, the size of the convolved image is smaller than the size of the original image. This can potentially be an issue when we design convolutional neural networks. For a convolutional network with many hidden layers, convolutions are performed on the input images in a successive manner. If we perform discrete convolutions multiple times, then the size of the convolved image gets smaller, and at some point, it may become even smaller than the size of the kernel in which case, we will not be able to continue to perform convolutions anymore! Therefore, it would be highly desirable if discrete convolutions can be performed in a way so that the size of the output will be the same as the size of the original image. Fortunately, this issue can be taken care of in a straightforward manner. A simple technique, called the **padding technique**, performs discrete convolutions in a way that the convolved images have the same size as the original input images.

Suppose we are using a kernel of size $k \times \ell$ to perform a 2D convolution with $k > 1$ and $\ell > 1$. Let $p = \left[\frac{k}{2}\right]$ (*i.e.* the integer part of $k/2$). Then before performing the convolution, we enlarge the original input image as follows. First, we increase the number of rows of the original image according to the following prescription:

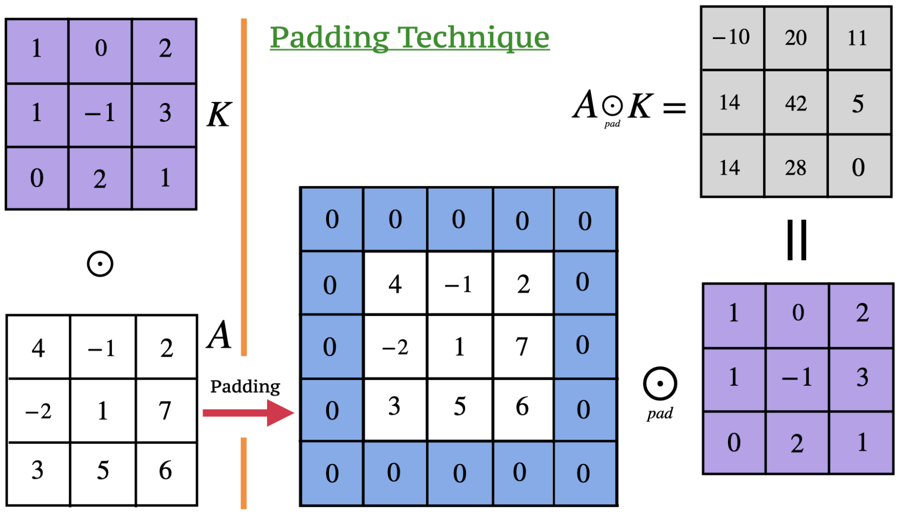
- If k is even, and p is even, then add p zero rows to the top and $p - 1$ zero rows to the bottom of the image.
- If k is even, and p is odd, then add $2\left[\frac{p}{2}\right] + 1$ zero rows to the top and $2\left[\frac{p}{2}\right]$ zero rows to the bottom of the image.
- If k is odd, then add p zero rows to the top and p zero rows to the bottom of the image.

We similarly increase the number of columns of the original matrix provided that $\ell > 1$. Let $q = \left[\frac{\ell}{2}\right]$. We increase the number of columns of the original image according to the following prescription:

- If ℓ is even, and q is even, then add q zero columns to the left side and $q - 1$ zero columns to the right side of the image.
- If ℓ is even, and q is odd, then add $2\left[\frac{q}{2}\right] + 1$ zero columns to the left side and $2\left[\frac{q}{2}\right]$ zero columns to the right side of the image.
- If ℓ is odd, then add q zero columns to the left side and q zero columns to the right side of the image.

The following picture depicts the application of the padding technique to a filter of size 3×3. In the following toy example, the size of the original image A is the same as the size of the kernel K (*i.e.* 3×3).

Therefore, if no padding is applied, we will be able to apply the given 3×3 kernel K to the image A *only once* (in which case the size of the convolved image $A \odot K$ will be 1×1).



To perform the padding technique, we note that $k = \ell = 3$. Since k and ℓ are odd and $p = \left\lfloor \frac{k}{2} \right\rfloor = q = \left\lfloor \frac{\ell}{2} \right\rfloor = 1$, we need to add one zero row to the top, one zero row to the bottom, one zero column to the left side, and one zero column to the right side of matrix A. Adding the necessary zero rows and columns, the padded image is of size 5×5. Now, applying kernel K, the convolved image will be of size 3×3, which is exactly the same as the size of the original matrix A. Let us use our python function convolve_2d to confirm the obtained result for the convolved matrix after padding.

[4]:
```
A = np.array([[0, 0, 0, 0, 0], [0, 4, -1, 2, 0], [0, -2, 1, 7, 0], [0, 3, 5, 6, 0],␣
 ↪[0, 0, 0, 0, 0]])
K = np.array([[1, 0, 2], [1, -1, 3], [0, 2, 1]])

convolve_2d(A, K)
```

[4]:
```
array([[-10,  20,  11],
       [ 14,  42,   5],
       [ 14,  28,   0]])
```

1.4 Applications of Kernels

In the previous sections, we saw how discrete convolutions were defined. We also learned how we can implement discrete convolutions in practice. We are now at a point we can ask an important question. This question may be expressed in a number of different forms:

- **What are discrete convolutions good for?**
- **What do kernels do for us?**
- **Why should we be interested in discrete convolutions by different kernels?**

The short answer is that kernels, through the implementation of discrete convolutions, can perform a number interesting operations on images. To illustrate this point further, we take the MNIST dataset that includes 70000 images of handwritten digits, and perform 2D convolutions on some images of this dataset with a number of specified kernels. The MNIST dataset is already built in pytorch through torchvision. We download this dataset and readily transform the images into 28×28 matrices, using the transforms module of torchvision.

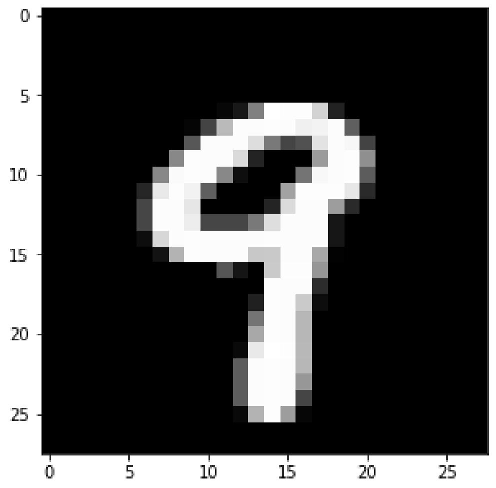
```
[5]: import torch
     import torchvision
     from torchvision import transforms
     import matplotlib.pyplot as plt

     MNIST_train = torchvision.datasets.MNIST("./data", train=True, download=True,␣
      ↪transform=transforms.ToTensor())
     MNIST_test = torchvision.datasets.MNIST("./data", train=False, download=True,␣
      ↪transform=transforms.ToTensor())

     print('Size of MNIST train =', len(MNIST_train))
     print('Size of MNIST test =', len(MNIST_test))
```

```
Size of MNIST train = 60000
Size of MNIST test = 10000
```

Using the matplotlib library and the imshow command, we can represent the images explicitly. Pay special attention to the tensor structure of the images.

```
[6]: img_num = 39532

     X_sample, y_sample = MNIST_train[img_num]

     plt.figure(figsize=(5, 5))
     plt.imshow(X_sample[0,:], vmin=0, vmax=1, cmap='gray')
     print('Tensor shape of the image =', X_sample.shape)
     print('Digit =', y_sample)
     plt.show()
```

```
Tensor shape of the image = torch.Size([1, 28, 28])
Digit = 9
```



Now, we would like to introduce a number of interesting kernels, and perform 2D convolutions on MNIST images by these kernels. The following exercise should hint toward a cogent answer to the questions we posed at the beginning of this section.

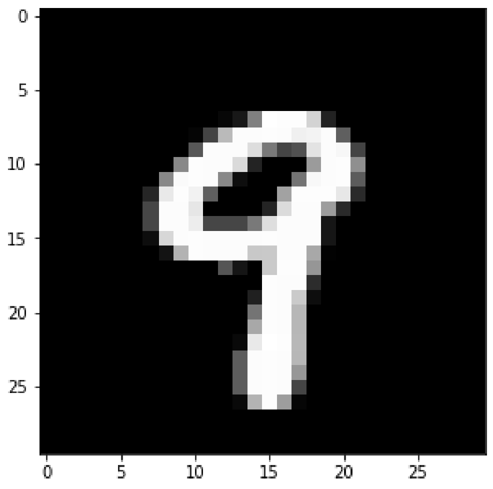
Example: Apply the following kernels to some of the MNIST images and explain the action of each kernel on the images.

$$K_0 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad K_1 = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}, \quad K_2 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix},$$
$$K_3 = \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}, \quad K_4 = \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}, \quad K_5 = \frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}. \tag{5}$$

To perform the convolution with padding, we take the advantage of the convolve command from the signal module of the scipy library. We write a simple function that receives the image number for the MNIST dataset, and the kernel as inputs and displays the convolved image with the kernel as the output.

```
[7]: def display_kernel_action(img_num, ker):

         import matplotlib.pyplot as plt
         from scipy.signal import convolve

         X_sample, y_sample = MNIST_train[img_num]     # Pick the image from MNIST
         print('Digit =', y_sample)                    # Display the label
         conv_image = convolve(X_sample[0], ker)       # Convolve the (padded) image with␣
      ↪the kernel
         plt.figure(figsize=(5, 5))
         plt.imshow(conv_image, vmin=0, vmax=1, cmap='gray')  # Display the convolved image
         plt.show()
         return

     kernel_0 = np.array([[0, 0, 0], [0, 1, 0], [0, 0, 0]])

     display_kernel_action(img_num, kernel_0)
```

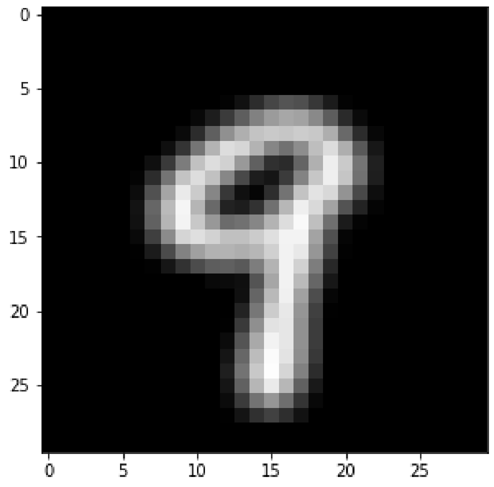
```
Digit = 9
```



As is obvious from above, there is no difference between the original image and the convolved image with kernel K_0. In fact, kernel K_0 is called the *identity kernel*, as it makes no changes to the given image.

[8]:
```
kernel_1 = np.array([[1, 1, 1], [1, 1, 1], [1, 1, 1]])/9

display_kernel_action(img_num, kernel_1)
```

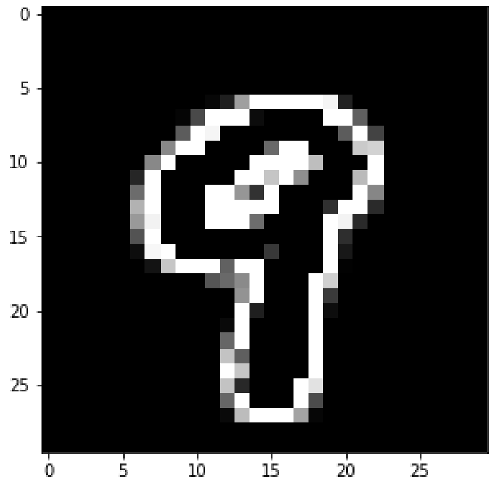
```
Digit = 9
```



As is seen from above, kernel K_1 *blurs* the original image.

[9]:
```
kernel_2 = np.array([[1, 1, 1], [1, -8, 1], [1, 1, 1]])

display_kernel_action(img_num, kernel_2)
```

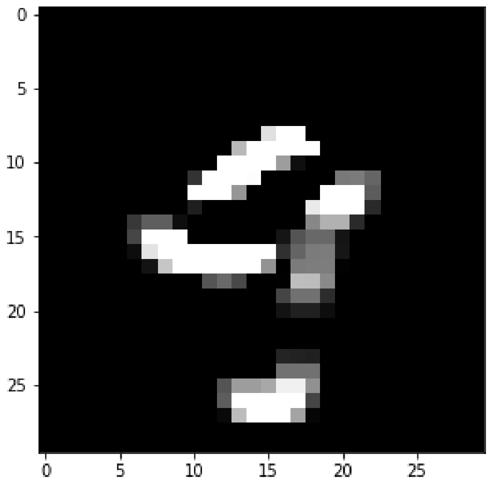
```
Digit = 9
```



The action of kernel K_2 is *edge detection*. Kernel K_2 disregards the interior of the image and only keeps the boundary between the image and the background.

```
[10]: kernel_3 = np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1]])

      display_kernel_action(img_num, kernel_3)
```

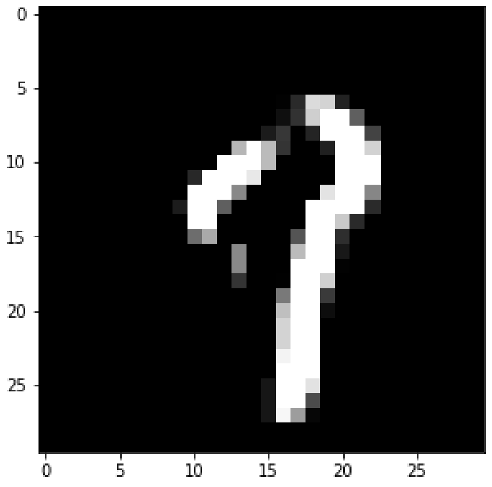
```
Digit = 9
```



Kernel K_3 identifies the horizontal edges of the image. Kernel K_3 is a *horizontal edge detector*.

```
[11]: kernel_4 = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])

      display_kernel_action(img_num, kernel_4)
```

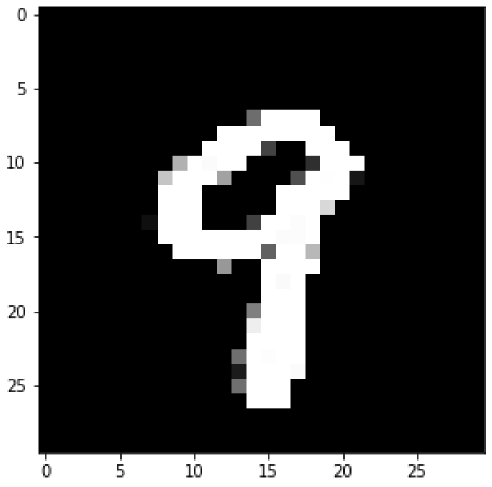
```
Digit = 9
```



Kernel K_4 identifies the vertical edges of the image. Kernel K_4 is a *vertical edge detector*.

```
[12]: kernel_5 = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])

      display_kernel_action(img_num, kernel_5)
```

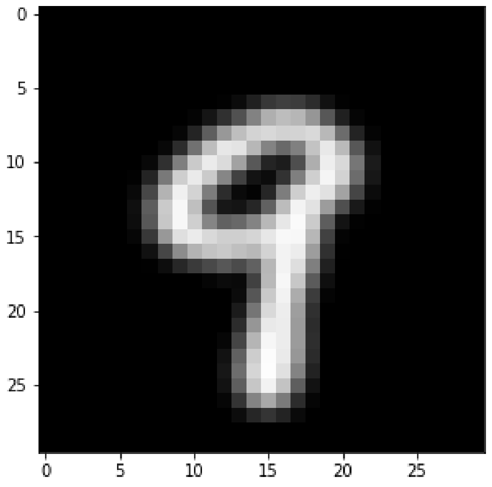
```
Digit = 9
```



As realized from the above image, kernel K_5 *sharpens the interior* of the image.

```
[13]: kernel_6 = np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]])/16

      display_kernel_action(img_num, kernel_6)
```

```
Digit = 9
```



Kernel K_6 is called the 3×3 Gaussian blur kernel, as it implements the blurring in a specific manner.

As we found out from the above example, kernels can perform interesting operations on images. Now, suppose we want to use discrete convolutions as building blocks of certain neural networks. The critical question is now *what kernels are the most appropriate ones for the specific task* (*e.g.* classification or regression on some target variable) we would like to perform? The answer is that we should allow the network to learn the best filters through the training for the specific task we would like to perform! To be more precise, the *number* and the *size of the filters* in our neural network will be regarded as *hyperparameters* of the model that we need to specify prior to the beginning of the training process. However, the *entries of the filters* will be among the *parameters* of the model that are learned through the training process. In fact this is exactly how *convolutional neural networks* work. We can now state an algorithm by which CNN models models work:

1. As the user, feed the input images as tensors (with appropriate dimensions) to the CNN model.
2. As the user, specify the number and the size of the filters for each convolutional hidden layer.
3. The model will find the optimal values of the entries of filters through the training process by minimizing the loss function of the model.
4. Based on the found optimal values for the entries of the filters (*i.e.* weights of the CNN model), the model makes predictions for test samples.

Now that we know how convolutional neural networks are structured, we should answer an important question.

Question: *What is the main difference between the fully connected and convolutional neural networks?*

Answer: As we saw in the previous chapter, in a fully connected neural network, every neuron receives contribution from *all other neurons* in the previous layer. However, in a convolutional neural network, the interactions are very *local*. This means that for a given pixel/point (*i.e.* entry of its corresponding matrix) p of an image, only the pixels in the vicinity of p can interact with it. The pixels that are outside of the chosen filter, when the the filter includes p itself, have no interaction with p. In other words, the distance of the farthest point that can have interaction with p cannot be greater than the length of the diagonal of the chosen filter. The following picture indicates a farthest point that can have a nontrivial interaction with point p. As shown in the picture, there is no interaction between points p and q, as it is impossible to place the filter in a position that will include both p and q.

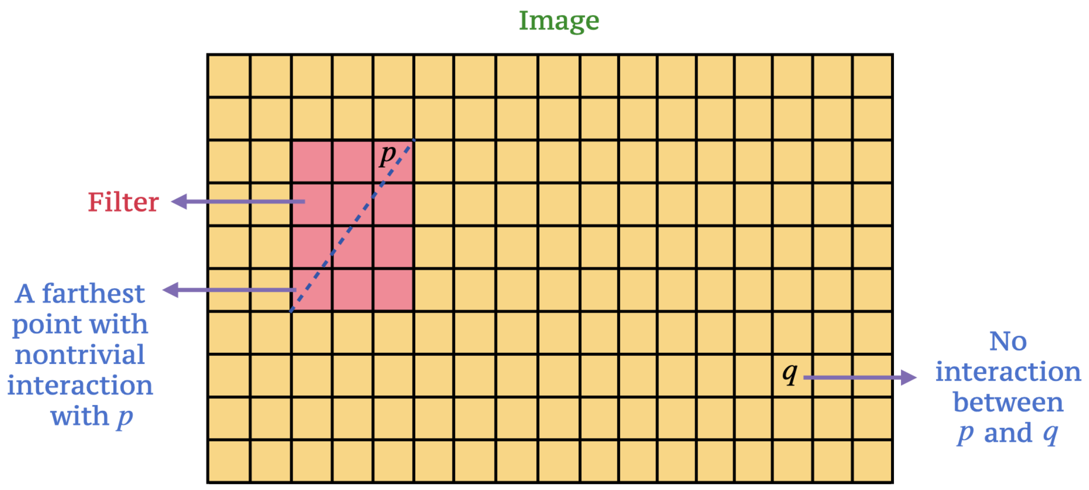


Image
p
Filter
A farthest point with nontrivial interaction with p
q
No interaction between p and q