UNIT-III

Overall Architecture – The convolution layer – The pooling layer – Implementing the Convolution and Pooling Layers – Implementing a CNN – Visualizing a CNN – Typical CNNs. CHAPTER – 7 (T1)

CONVOLUTIONAL NEURAL NETWORKS

A **Convolutional Neural Network (CNN)** is a type of Deep Learning neural network architecture commonly used in Computer Vision. Computer vision is a field of Artificial Intelligence that enables a computer to understand and interpret the image or visual data.

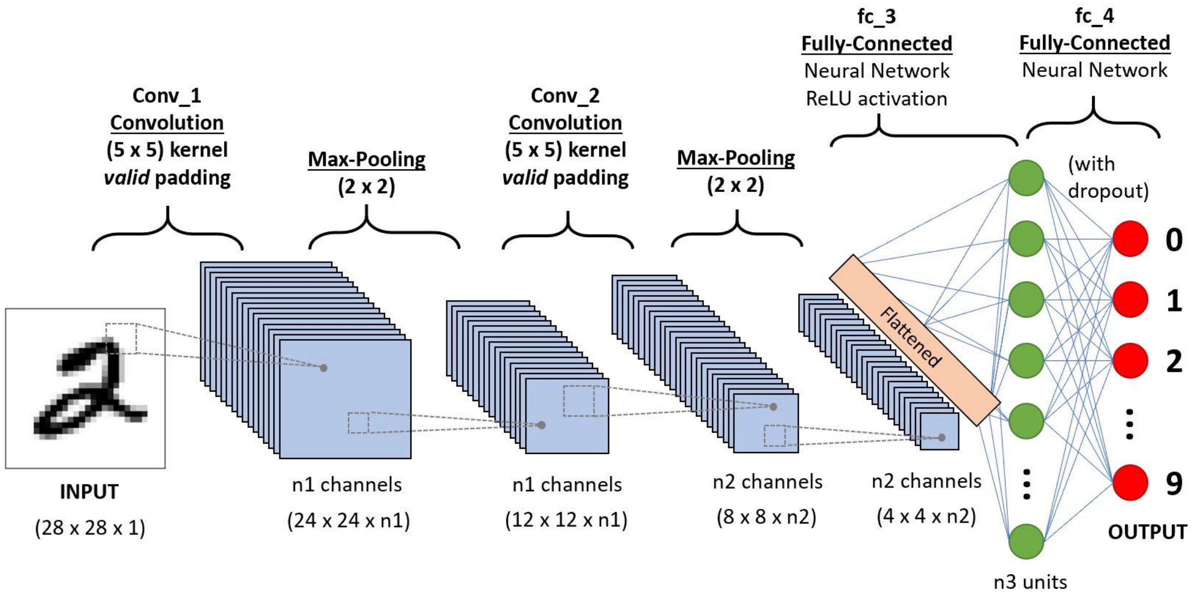
Convolutional Neural Network or CNN is a type of artificial neural network, which is widely used for image/object recognition and classification. Deep Learning thus recognizes objects in an image by using a CNN. CNNs are playing a major role in diverse tasks/functions like image processing problems, computer vision tasks like localization and segmentation, video analysis, to recognize obstacles in self-driving cars, as well as speech recognition in natural language processing. As CNNs are playing a significant role in these fast-growing and emerging areas, they are very popular in Deep Learning

CNN Application

a. Decoding Facial Recognition
b. Analyzing Documents
c. Historic and Environmental Collections
d. Understanding Climate
e. Grey Areas
f. Advertising
g. Other Interesting Fields

Overall Architecture

A CNN can be created by combining layers, much in the same way as the neural networks. CNNs have other layers as well: a convolution layer and a pooling layer.



Convolutional Neural Network consists of multiple layers like the input layer, Convolutional layer, Pooling layer, and fully connected layers. The Convolutional layer applies filters to the input image to extract features, the Pooling layer down samples the image to reduce computation, and the fully connected layer makes the final prediction. The network learns the optimal filters through backpropagation and gradient descent.

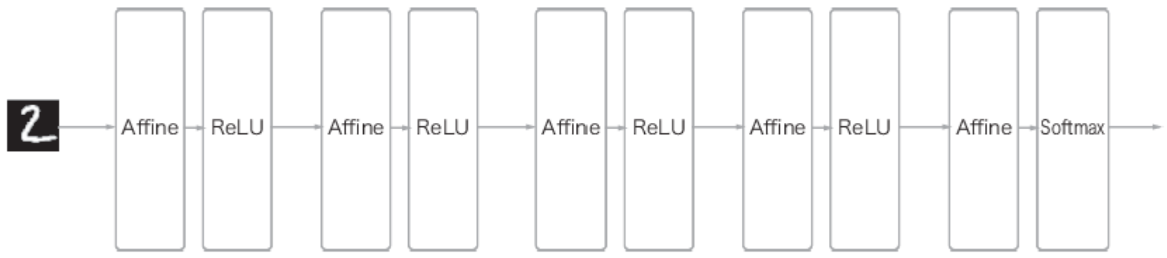
- **Input Layers:** It’s the layer in which we give input to our model. In CNN, Generally, the input will be an image or a sequence of images. This layer holds the raw input of the image with width 32, height 32, and depth 3.
- **Convolutional Layers:** This is the layer, which is used to extract the feature from the input dataset. It applies a set of learnable filters known as the kernels to the input images. The filters/kernels are smaller matrices usually 2×2, 3×3, or 5×5 shape. it slides over the input image data and computes the dot product between kernel weight and the corresponding input image patch. The output of this layer is referred ad feature maps. Suppose we use a total of 12 filters for this layer we’ll get an output volume of dimension 32 x 32 x 12.
- **Activation Layer:** By adding an activation function to the output of the preceding layer, activation layers add nonlinearity to the network. it will apply an element-wise activation function to the output of the convolution layer. Some common activation functions are **RELU**: max (0, x), Tanh, **Leaky RELU**, etc. The volume remains unchanged hence output volume will have dimensions 32 x 32 x 12.

- **Pooling layer:** This layer is periodically inserted in the converts and its main function is to reduce the size of volume which makes the computation fast reduces memory and prevents overfitting. Two common types of pooling layers are **max pooling** and **average pooling**. If we use a max pool with 2 x 2 filters and stride 2, the resultant volume will be of dimension 16x16x12.
- Flattening: The resulting feature maps are flattened into a one-dimensional vector after the convolution and pooling layers so they can be passed into a completely linked layer for categorization or regression.
- Fully Connected Layers: It takes the input from the previous layer and computes the final classification or regression task.
- Output Layer: The output from the fully connected layers is then fed into a logistic function for classification tasks like sigmoid or SoftMax which converts the output of each class into the probability score of each class.

How layers are combined to create a CNN

In the neural networks that we have seen so far, all the neurons in adjacent layers are connected. These layers are called **fully connected** layers, and we implemented them as Affine layers.

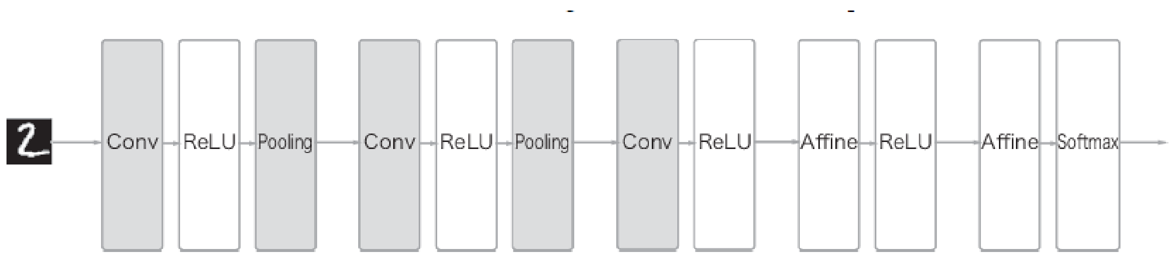
You can use Affine layers to create a neural network consisting of five fully connected layers, for example



Sample network consisting of fully connected layers (Affine layers)

after four pairs of **Affine – ReLU** layers, comes the Affine layer, which is the fifth layer. And finally, the Softmax layer outputs the final result (probability)

what architecture does a CNN have?



Sample CNN – convolution and pooling layers are added

CNN has additional convolution and pooling layers. In the CNN, layers are connected in the order of **Convolution – ReLU – (Pooling)** (a pooling layer is sometimes omitted). We can consider the previous **Affine – ReLU** connection as being replaced with "Convolution – ReLU – (Pooling)."

the layers near the output are the previous "Affine – ReLU" pairs, while the last output layers are the previous "Affine – Softmax" pairs. This is the structure often seen in an ordinary CNN.

Issues with the Fully Connected Layer

In a fully connected layer, all the neurons in the adjacent layer are connected, and the number of outputs can be determined arbitrarily.

The issue with a fully connected layer, though, is that the shape of the data is *ignored*. For example, when the input data is an image, it usually has a three-dimensional shape, determined by the height, the width, and the channel dimension. However, three-dimensional data must be converted into one-dimensional flat data when it is provided to a fully connected layer.

Let's say an image has a three-dimensional shape and that the shape contains important spatial information. Essential patterns to recognize this information may hide in three-dimensional shapes. Spatially close pixels have similar values, the RBG channels are closely related to each other, and the distant pixels are not related. However, a fully connected layer ignores the shape and treats all the input data as equivalent neurons (neurons with the same number of dimensions), so it cannot use the information regarding the shape.

On the other hand, a convolution layer maintains the shape. For images, it receives the input data as three-dimensional data and outputs three-dimensional data to the next layer. Therefore, CNNs can understand data with a shape, such as images, properly.

The input data for a convolution layer is called an **input feature map**, while the output data for a convolution layer is called an **output feature map**.

Workflow of CNN

Convolution, Padding, Stride, and Pooling in CNN

Convolution operation

The convolution is a mathematical operation used to extract features from an image. The convolution is defined by an image kernel. The image kernel is nothing more than a small matrix. Most of the time, a 3x3 kernel matrix is very common.

In the below fig, the green matrix is the original image, and the yellow moving matrix is called kernel, which is used to learn the different features of the original image. The kernel first moves horizontally, then shift down and again moves horizontally. The sum of the dot product of the image pixel value and kernel pixel value gives the output matrix. Initially, the kernel value initializes randomly, and its a learning parameter.

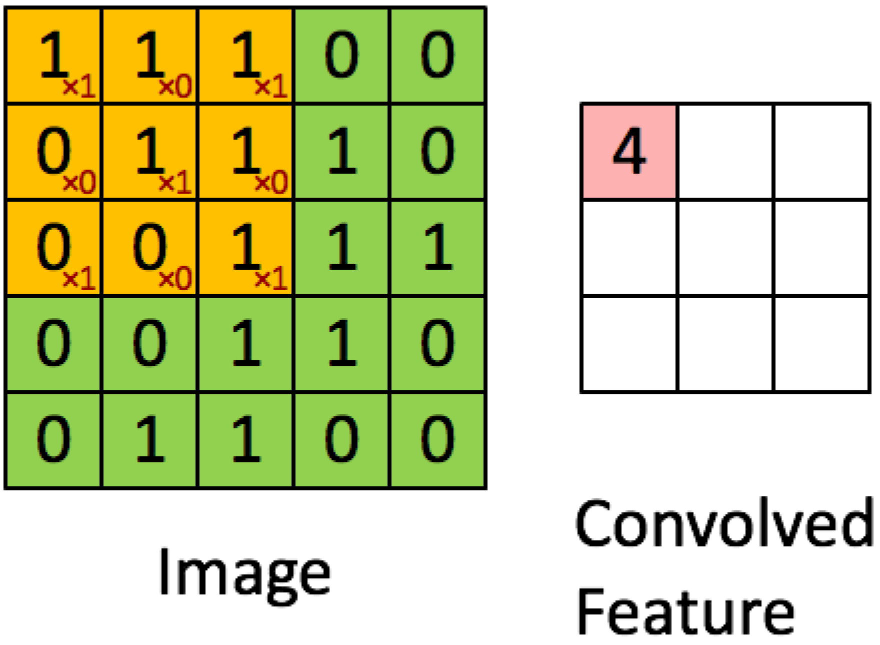
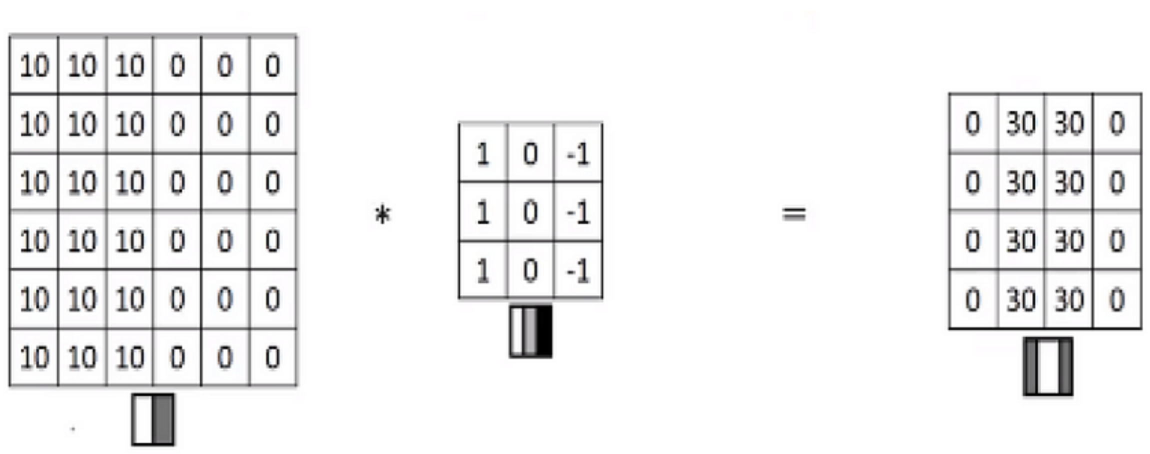


Illustration of the convolution operation

There are some standard filters like **Sobel filter,** contains the value 1, 2, 1, 0, 0, 0, -1, -2, -1, the advantage of this is it puts a little bit more weight to the central row, the central pixel, and this makes it maybe a little bit more robust. Another filter used by computer vision researcher is instead of a 1, 2, 1, it is 3, 10, 3 and then -3, -10, -3, called a **Scharr filter**. And this has yet other slightly different properties and this can be used for vertical edge detection. If it is flipped by 90 degrees, the same will act like horizontal edge detection.

In order to understand the concept of edge detection, taking an example of a simplified image.

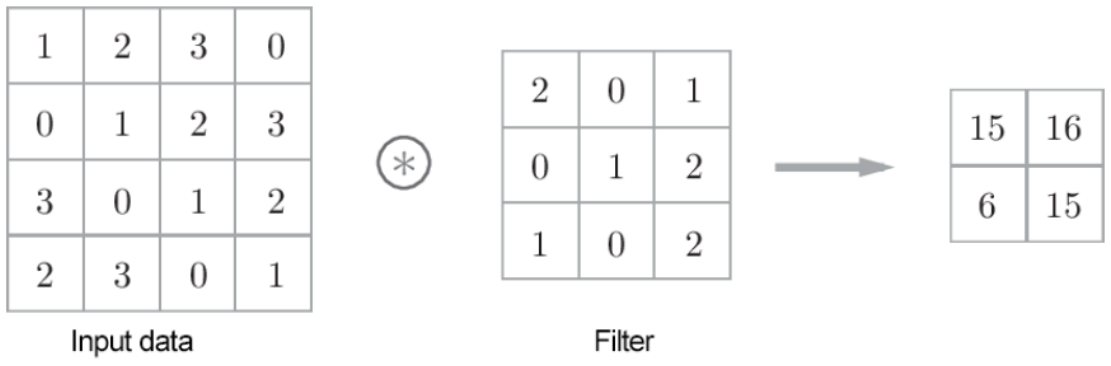


A 6∗6 image convolved with 3∗3 kernel

So if a 6*6 matrix convolved with a 3*3 matrix output is a 4*4 matrix. To generalize this if a $m * m$ image convolved with $n * n$ kernel, the output image is of size $\mathbf{(m - n + 1) * (m - n + 1)}$.

Convolution Operations

The processing performed in a convolution layer is called a "convolution operation" and is equivalent to the "filter operation" in image processing.



Convolution operation – the ⊛ symbol indicates a convolution operation

a convolution operation applies a filter to input data

the shape of the input data has a height and width

the input size is (4, 4), the filter size is (3, 3), and the output size is (2, 2)

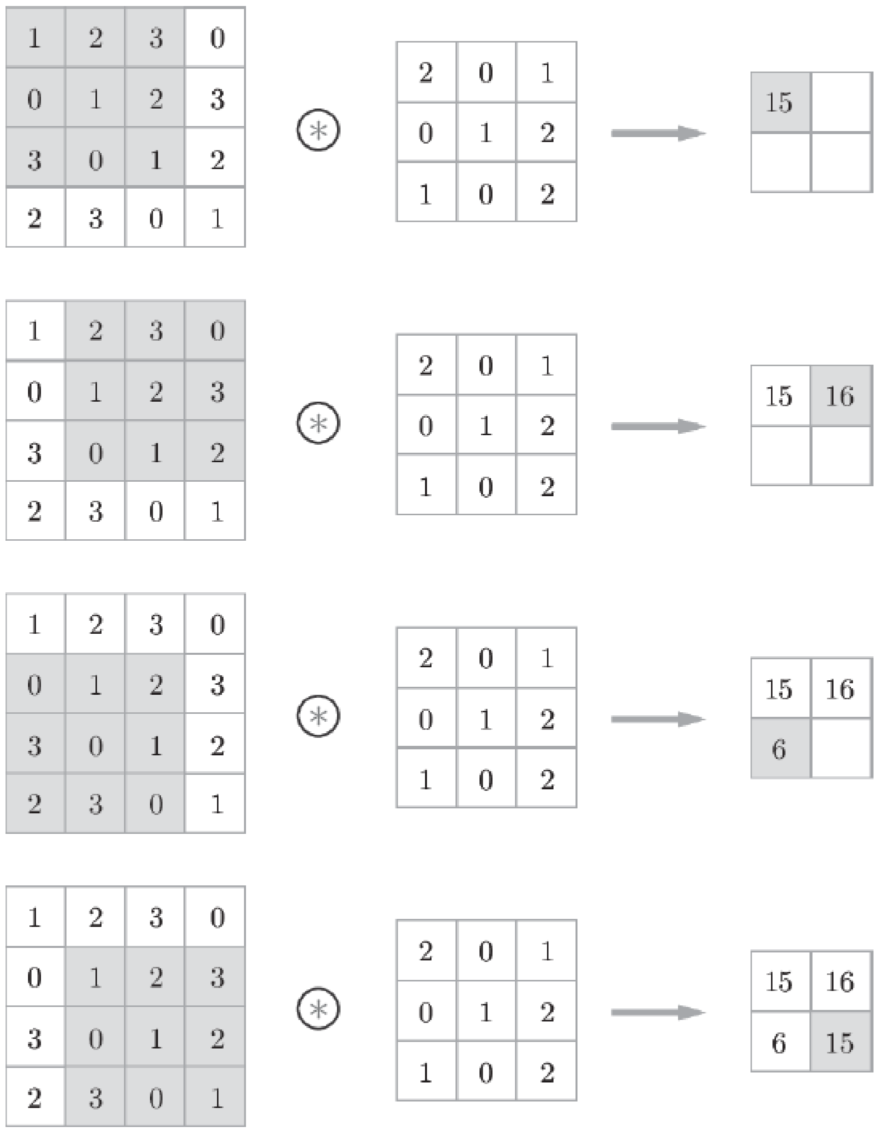
uses the word "kernel" for the term "filter."

A convolution operation is applied to the input data while the filter window is shifted at a fixed interval. The window here indicates the gray 3x3 section

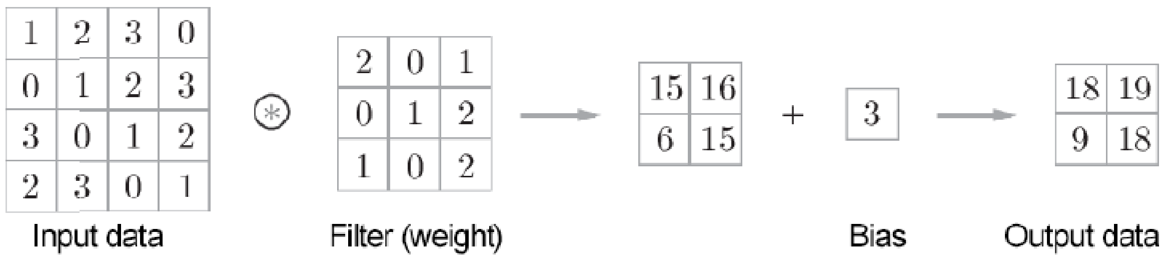
the element of the filter and the corresponding element of the input are multiplied and summed at each location (this calculation is sometimes called a **multiply-accumulate**

operation). The result is stored in the corresponding position of the output. The output of the convolution operation can be obtained by performing this process at all locations.

A fully connected neural network has biases as well as weight parameters. In a CNN, the filter parameters correspond to the previous "weights." It also has biases



Calculation procedure of a convolution operation



Bias in a convolution operation – a fixed value (bias) is added to the element after the filter is applied

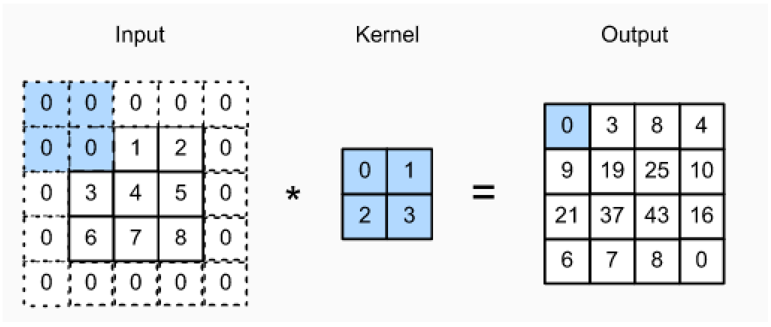
Here, the bias is always only one (1x1) where one bias exists for the four pieces of data after the filter is applied. This one value is added to all the elements after the filter is applied.

Padding

There are two problems arises with convolution:

1. Every time after convolution operation, original image size getting shrinks, as we have seen in above example six by six down to four by four and in image classification task there are multiple convolution layers so after multiple convolution operation, our original image will really get small, but we don't want the image to shrink every time.
2. The second issue is that, when kernel moves over original images, it touches the edge of the image a smaller number of times and touches the middle of the image more number of times and it overlaps also in the middle. So, the corner features of any image or on the edges aren't used much in the output.

So, to solve these two issues, a new concept is introduced called **padding.** Padding preserves the size of the original image.

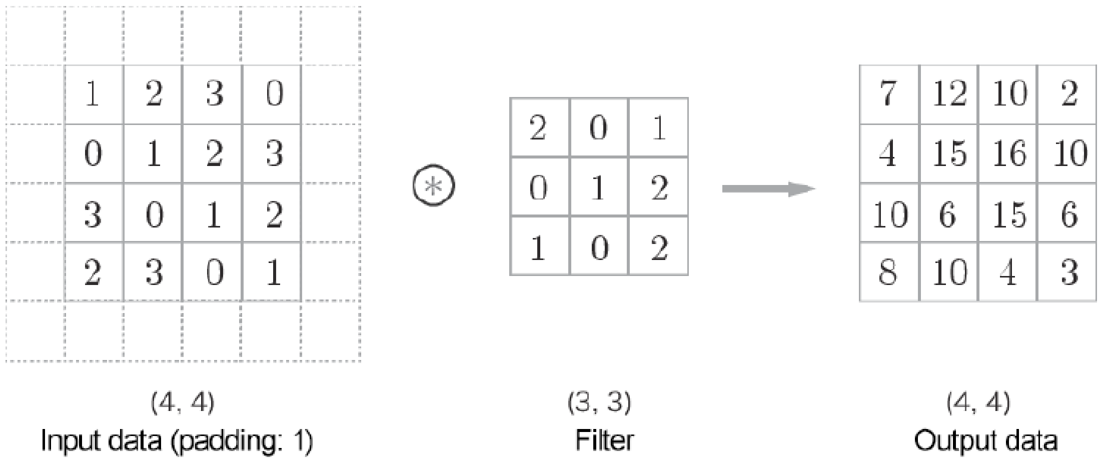


Padded image convolved with 2*2 kernel

Before a convolution layer is processed, fixed data (such as 0) is sometimes filled around the input data. This is called **padding** and is often used in a convolution operation

Padding is used mainly for adjusting the output size.

For example, in the below *Figure* , padding of 1 is applied to the (4, 4) input data. The padding of 1 means filling the circumference with zeros with the width of one pixel

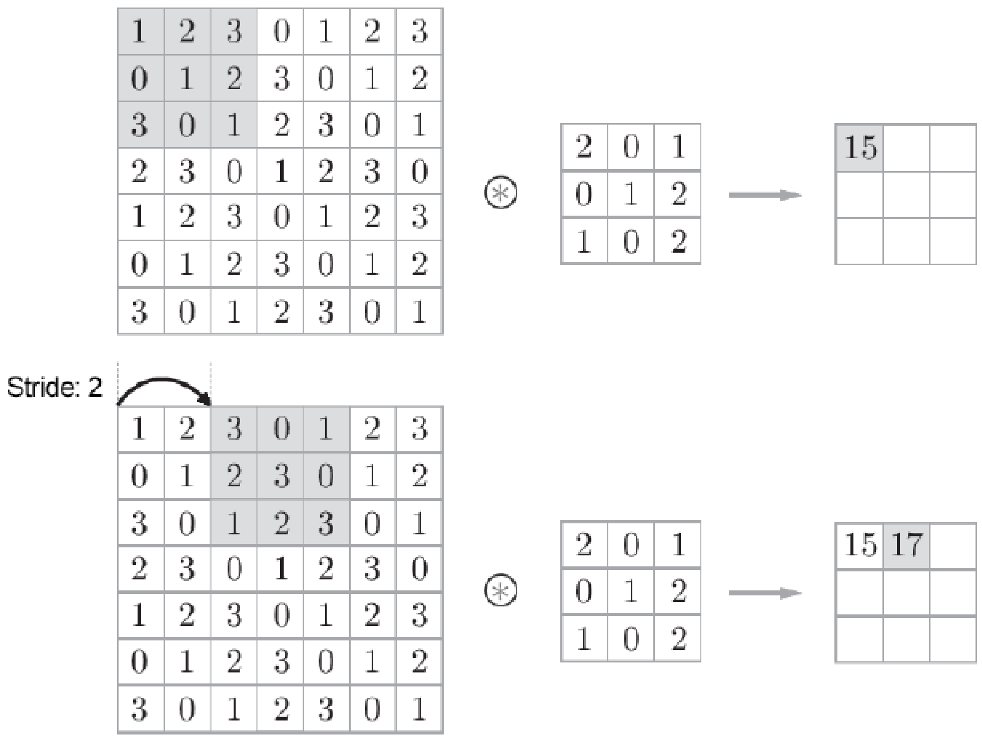


Padding in a convolution operation – add zeros around the input data (padding is shown by dashed lines here, and the zeros are omitted)

padding converts the (4, 4) input data into (6, 6) data. After the (3, 3) filter is applied, (4, 4) output data is generated. In this example, the padding of 1 was used. You can set any integer, such as 2 or 3, as the padding value. If the padding value was 2, the size of the input data would be (8, 8). If the padding was 3, the size would be (10, 10).

Stride

The interval of the positions for applying a filter is called a **stride**. In all previous examples, the stride was 1. When the stride is 2, for example, the interval of the window for applying a filter will be two elements.



Sample convolution operation where the stride is 2

a filter is applied to the (7, 7) input data with the stride of 2. When the stride is 2, the output size becomes (3, 3). Thus, the stride specifies the interval for applying a filter.

the larger the stride, the smaller the output size, and the larger the padding, the larger the output size

the input size is (*H*, *W*), the filter size is (*FH*, *FW*), the output size is (*OH*, *OW*), the padding is *P*, and the stride is *S*. In this case, you can calculate the output size with the following equation

$$OH = \frac{H + 2P - FH}{S} + 1$$
$$OW = \frac{W + 2P - FW}{S} + 1$$

Example 1: Example is shown in Figure 7.6

Input size: (4, 4), padding: 1, stride: 1, filter size: (3, 3):

$$OH = \frac{4 + 2 \cdot 1 - 3}{1} + 1 = 4$$
$$OW = \frac{4 + 2 \cdot 1 - 3}{1} + 1 = 4$$

2. **Example 2: Example is shown in Figure 7.7**

Input size: (7, 7), padding: 0, stride: 2, filter size: (3, 3):

3. **Example 3**

Input size: (28, 31), padding: 2, stride: 3, filter size:(5, 5):

note that you must assign values so that and in equation (7.1) are divisible. If the output size is not divisible (i.e., the result is a decimal), you must handle that by generating an error. Some deep learning frameworks advance this process without generating an error; for example, they round the value to the nearest integer when it is not divisible.

So if a $n*n$ matrix convolved with an f*f matrix the with padding p then the size of the output image will be (n + 2p — f + 1) * (n + 2p — f + 1) where p =1 in this case.

Stride is the number of pixels shifts over the input matrix. For padding p, filter size $f*f$ and input image size $n * n$ and stride 's' our output image dimension will be [{(n + 2p − f + 1) / s} + 1] ∗ [{(n + 2p − f + 1) / s} + 1].

Pooling

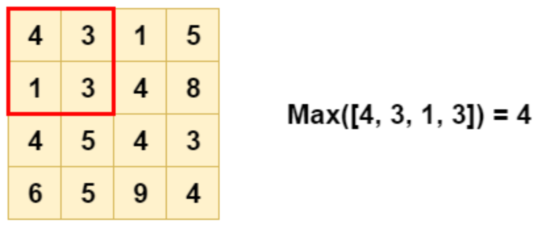
A pooling layer is another building block of a CNN**.** Pooling Its function is to progressively reduce the spatial size of the representation to reduce the network complexity and computational cost.

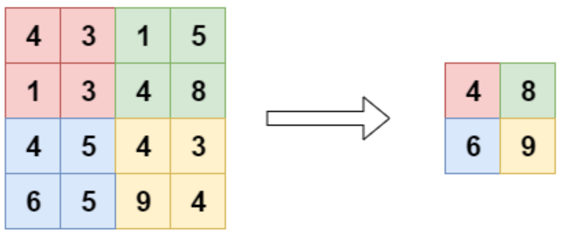
There are two types of widely used pooling in CNN layer:

1. Max Pooling
2. Average Pooling

Max Pooling

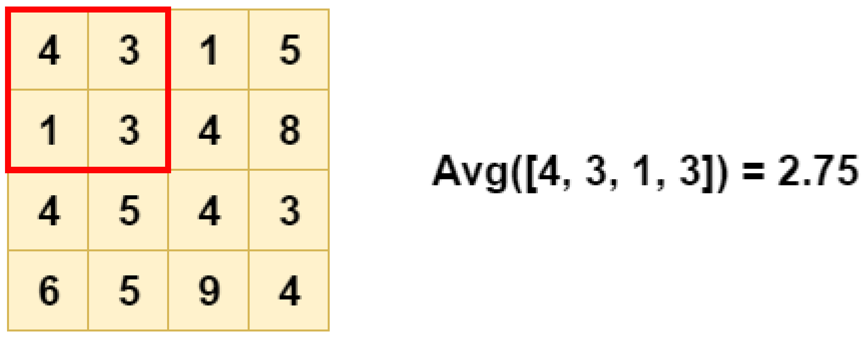
Max pooling is simply a rule to take the maximum of a region and it helps to proceed with the most important features from the image. Max pooling selects the brighter pixels from the image. It is useful when the background of the image is dark and we are interested in only the lighter pixels of the image.

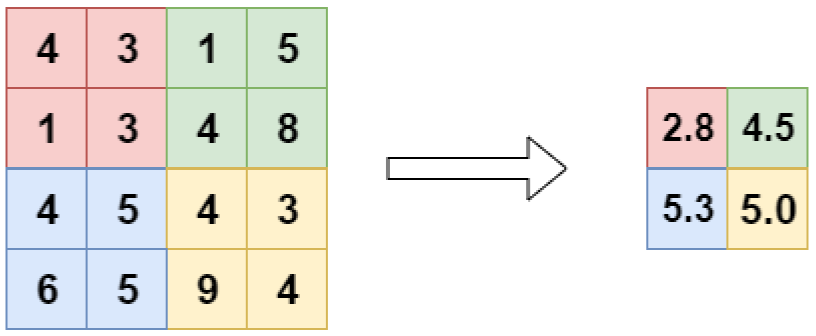




Average Pooling

Average Pooling is different from Max Pooling in the sense that it retains much information about the "less important" elements of a block, or pool. Whereas Max Pooling simply throws them away by picking the maximum value, Average Pooling blends them in. This can be useful in a variety of situations, where such information is useful.





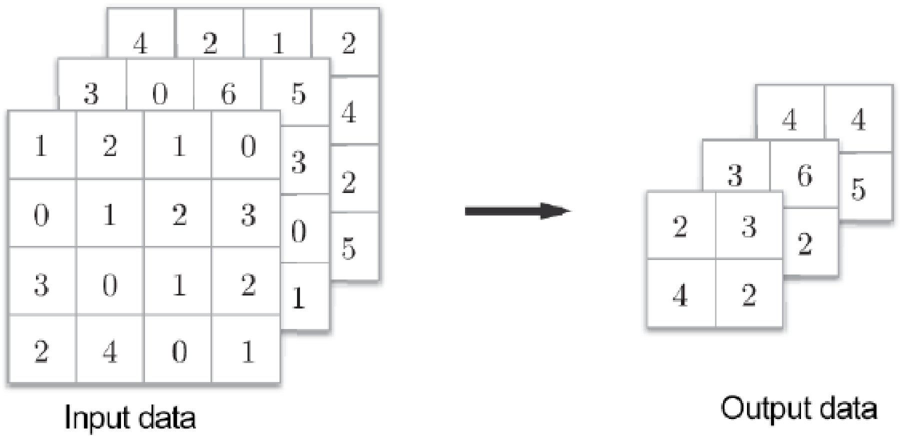
Characteristics of a Pooling Layer

There are no parameters to learn.

Pooling has no parameters to learn because it only takes the maximum value (or averages the values) in the target region.

The number of channels does not change.

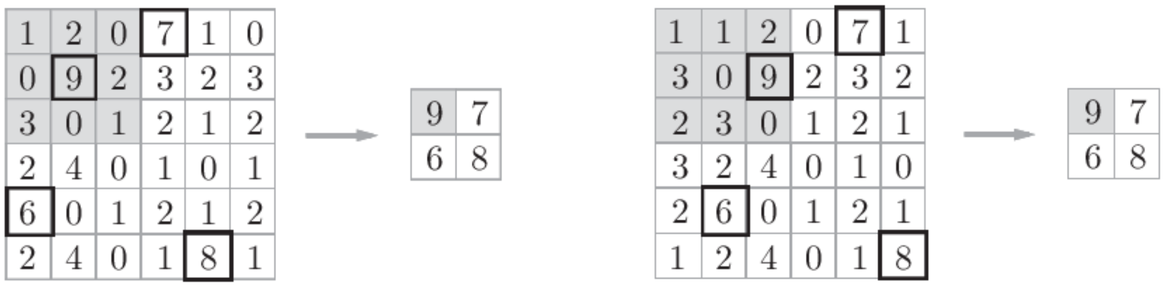
In pooling, the number of channels in the output data is the same as that in the input data.



Pooling does not change the number of channels.

It is robust to a tiny position change.

Pooling returns the same result, even when the input data is shifted slightly. Therefore, it is robust to a tiny shift of input data. For example, in 3 x 3 pooling, pooling absorbs the shift of input data



Even when the input data is shifted by one element in terms of width, the output is the same (it may not be the same, depending on the data)

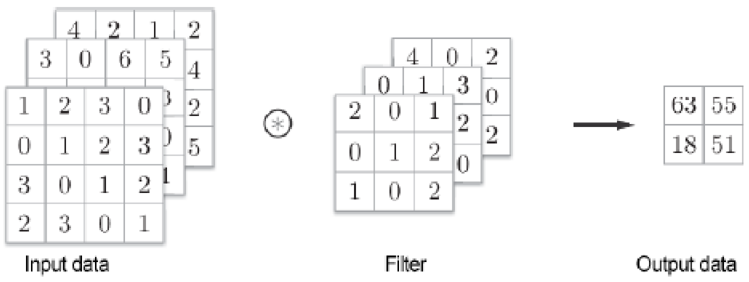
Advantages of Convolutional Neural Networks (CNNs):

1. Good at detecting patterns and features in images, videos, and audio signals.
2. Robust to translation, rotation, and scaling invariance.
3. End-to-end training, no need for manual feature extraction.
4. Can handle large amounts of data and achieve high accuracy.

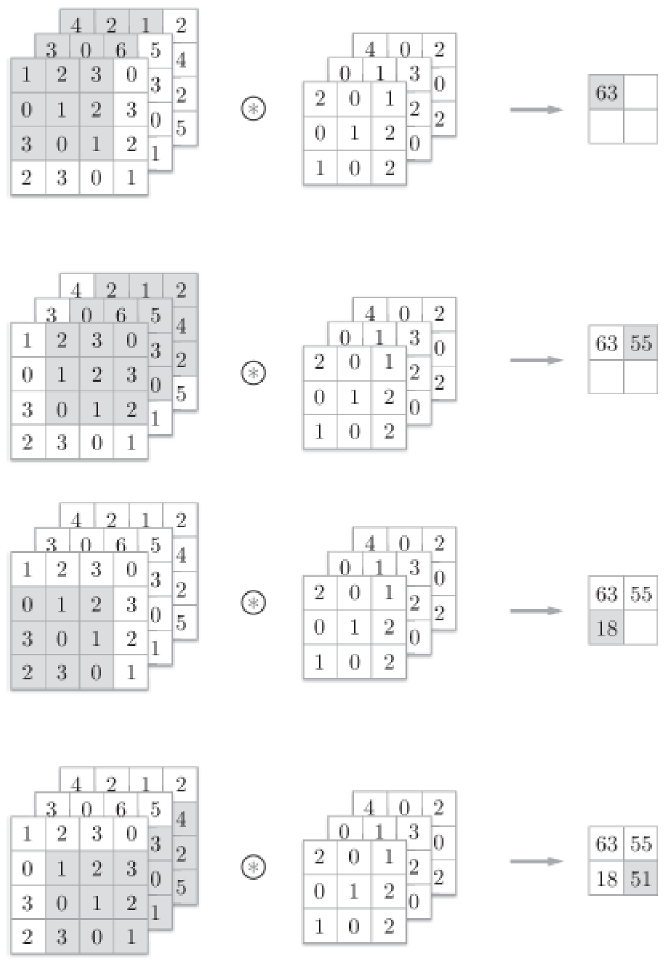
Disadvantages of Convolutional Neural Networks (CNNs):

1. Computationally expensive to train and require a lot of memory.
2. Can be prone to overfitting if not enough data or proper regularization is used.
3. Requires large amounts of labelled data.
4. Interpretability is limited, it’s hard to understand what the network has learned.

Performing a Convolution Operation on Three-Dimensional Data



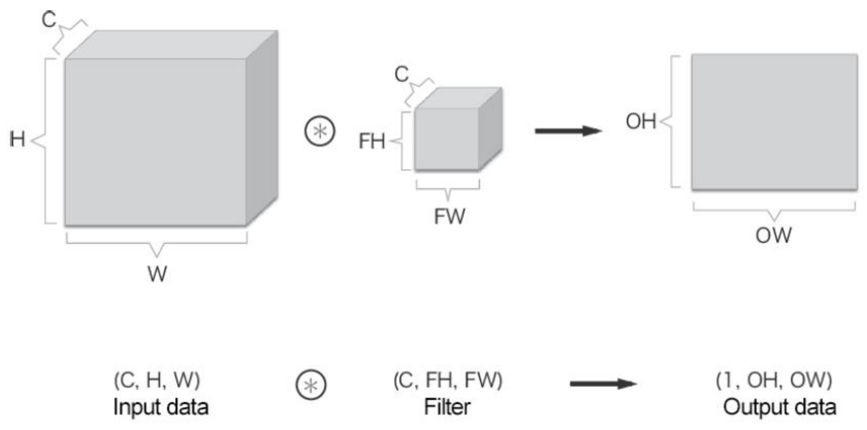
Convolution operation for three-dimensional data



Calculation procedure of the convolution operation for three-dimensional data

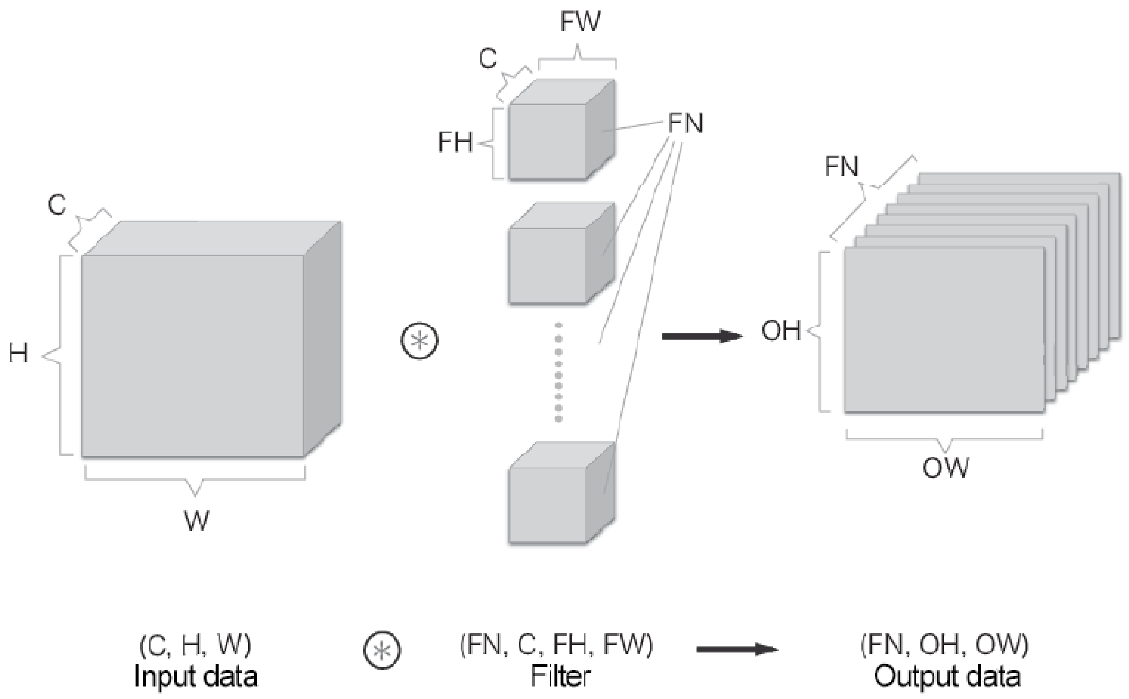
Thinking in Blocks

A block here is a three-dimensional cuboid which represent three-dimensional data as a multidimensional array in the order channel, height, width. So, when the number of channels is C, the height is H, and the width is W for shape, it is represented as (C, H, W). Represent a filter in the same order so that when the number of channels is C, the height is **FH** (**Filter Height**), and the width is **FW** (**Filter Width**) for a filter, it is represented as (C, FH, FW)



Using blocks to consider a convolution operation

The data's output is one feature map. One feature map means that the size of the output channel is one. So, how can we provide multiple outputs of convolution operations in the channel dimension? To do that, we use multiple filters (weights).

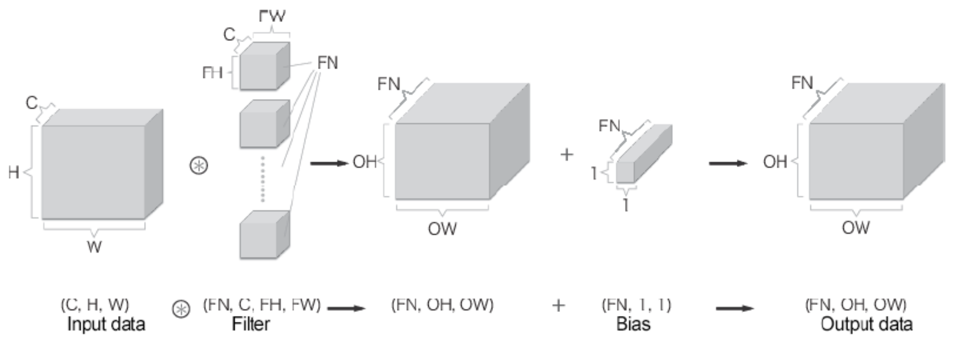


Sample convolution operation with multiple filters

when the number of filters applied is FN, the number of output maps generated is also FN. By combining FN maps, you can create a block of the shape (FN, OH, OW). Passing this completed block to the next layer is the process a CNN.

You must also consider the number of filters in a convolution operation. To do that, we will write the filter weight data as four-dimensional data (output_channel, input_ channel, height, width). For example, when there are 20 filters with three channels where the size is 5 x 5, it is represented as (20, 3, 5, 5).

convolution operation has biases



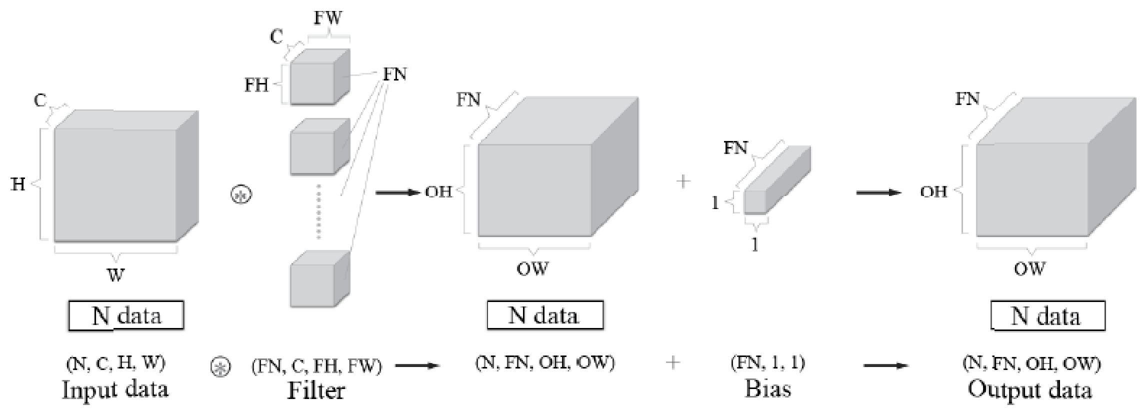
Process flow of a convolution operation (the bias term is also added)

Batch Processing

Input data is processed in batches in neural network processing.

The batch processing in a convolution operation by storing the data that flows through each layer as four-dimensional data. Specifically, the data is stored in the order (batch_num, channel, height, width). For example, when the processing shown in *Figure above is* conducted for N data in batches, the shape of the data becomes as follows.

In the data flow for batch processing shown here, the dimensions for the batches are added at the beginning of each piece of data. Thus, the data passes each layer as four-dimensional data. Please note that four-dimensional data that flows in the network indicates that a convolution operation is performed for N data; that is, N processes are conducted at one time



Process flow of a convolution operation (batch processing)

Implementing the Convolution and Pooling Layers

Four-Dimensional Arrays

For example, when the shape of the data is (10, 1, 28, 28), it indicates that ten pieces of data with a height of 28, width of 28, and 1 channel exist.

```
>>> x = np.random.rand(10, 1, 28, 28) # Generate data randomly
>>> x.shape
(10, 1, 28, 28)
```

To access the first piece of data, you can write **x[0]** (the index begins at 0 in Python). Similarly, **x[1]** to access the second piece of data:

```
>>> x[0, 0] # or x[0][0]
```

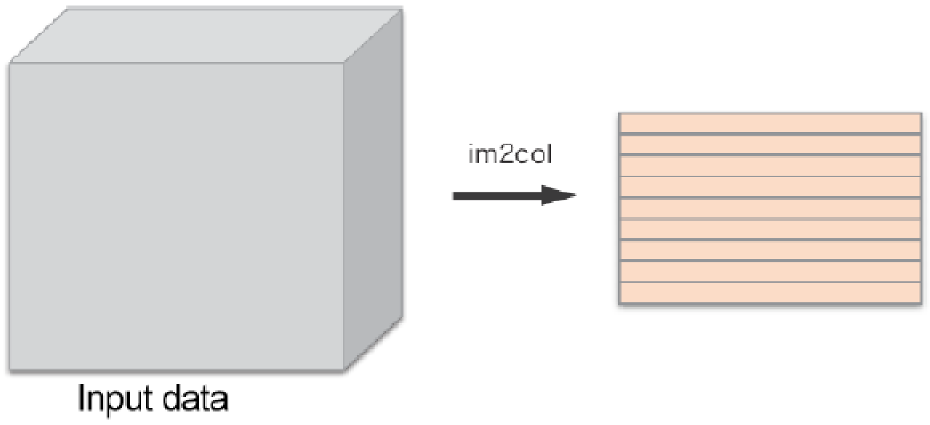
To access the spatial data in the first channel of the first piece of data

```
>>> x[0, 0] # or x[0][0]
```

implementing a convolution operation may be complicated. However, a "trick" called **im2col** makes this task easy.

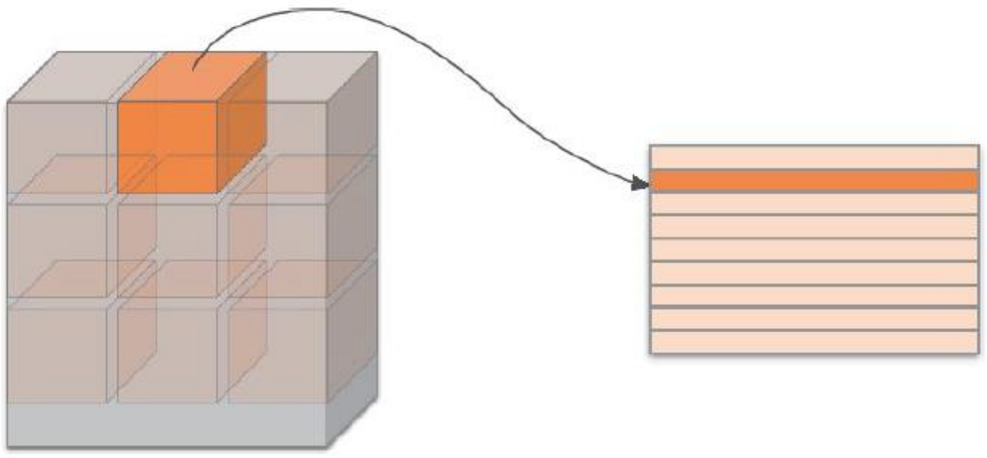
Expansion by im2col

The **im2col** function expands input data conveniently for a filter (weight). As shown in *Figure* , **im2col** converts three-dimensional input data into a two-dimensional matrix (to be exact, it converts four-dimensional data, including the number of batches, into two-dimensional data).



Overview of im2col

im2col expands the input data conveniently for a filter (weight). Specifically, it expands the area that a filter will be applied to in the input data (a three-dimensional block) into a row, as shown in *Figure* . **im2col** expands all the locations to apply a filter to.

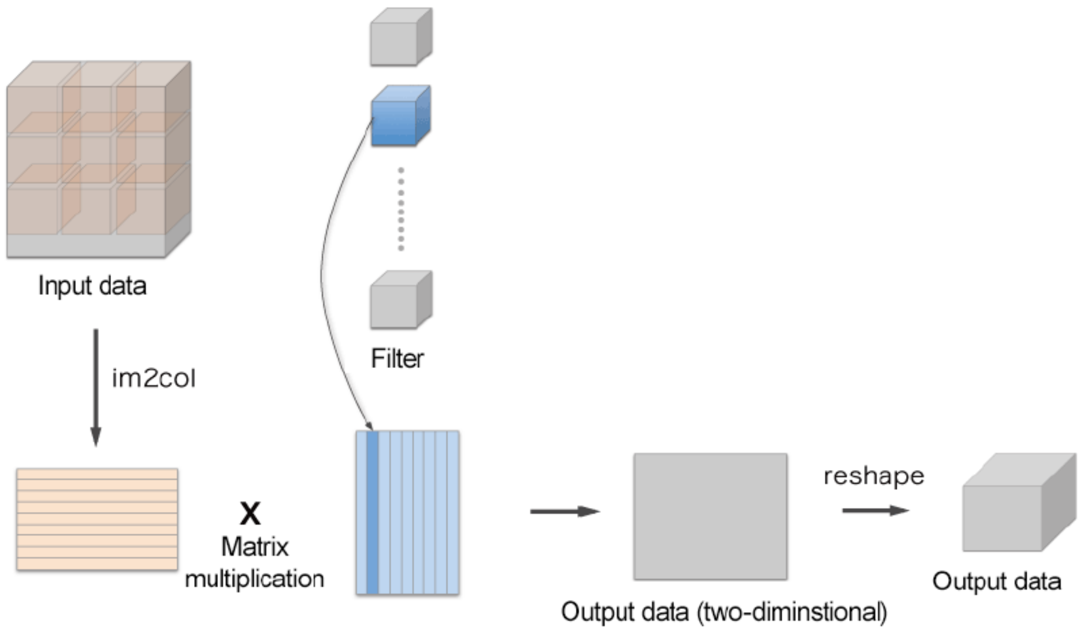


Expanding the filter target area from the beginning in a row

Note

The name **im2col** is an abbreviation of "image to column," meaning the conversion of images into matrices. Deep learning frameworks such as Caffe and Chainer provide the **im2col** function, which is used to implement a convolution layer.

After using **im2col** to expand input data, all you have to do is expand the filter (weight) for the convolution layer into a row and multiply the two matrices . This process is almost the same as that of a fully connected Affine layer



Details of filtering in a convolution operation – expand the filter into a column and multiply the matrix by the data expanded by im2col. Lastly, reshape the result of the size of the output data.

the output of using the **im2col** function is a two-dimensional matrix. You must transform two-dimensional output data into an appropriate shape because a CNN stores data as four-dimensional arrays.

Implementing a Convolution Layer

The **im2col** implementation is located at **common/ util.py**.

This **im2col** function has the following interface

im2col (input_data, filter_h, filter_w, stride=1, pad=0)

input_data: Input data that consists of arrays of four dimensions (amount of data, channel, height, breadth)

- **filter_h**: Height of the filter

filter_w: Width of the filter

- **stride**: Stride
- **pad**: Padding

The **im2col** function considers the "filter size," "stride," and "padding" to expand input data into a two-dimensional array, as follows

```
import sys, os
sys.path.append(os.pardir)
from common.util import im2col

x1 = np.random.rand(1, 3, 7, 7)
col1 = im2col(x1, 5, 5, stride=1, pad=0)
print(col1.shape) # (9, 75)

x2 = np.random.rand(10, 3, 7, 7)
col2 = im2col(x2, 5, 5, stride=1, pad=0)
print(col2.shape) # (90, 75)
```

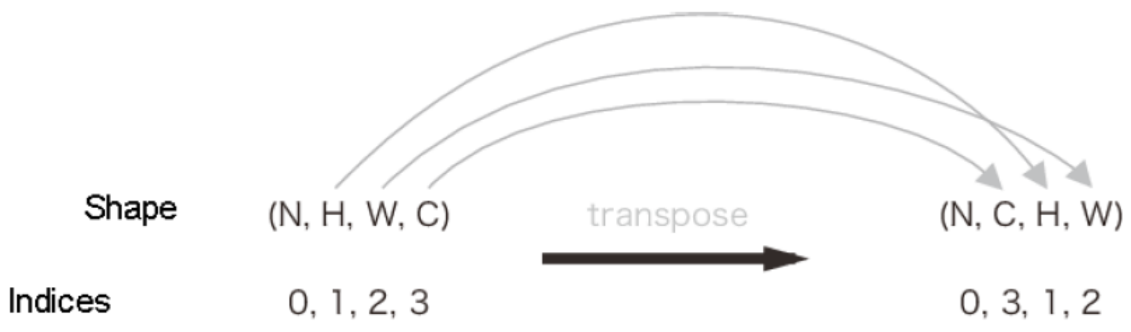
The first one uses 7x7 data with a batch size of 1, where the number of channels is 3. The second one uses data of the same shape with a batch size of 10. When we use the **im2col** function, the number of elements in the second dimension is 75 in both cases. This is the total number of elements in the filter (3 channels, size 5x5). When the batch size is 1, the result from **im2col** is (9, 75) in size. On the other hand, it is (90, 75) in the second example because the batch size is 10. It can store 10 times as much data.

use **im2col** to implement a convolution layer as a class called **Convolution**

```
class Convolution:
    def __init__(self, W, b, stride=1, pad=0):
        self.W = W
        self.b = b
        self.stride = stride
        self.pad = pad

    def forward(self, x):
        FN, C, FH, FW = self.W.shape
        N, C, H, W = x.shape
        out_h = int(1 + (H + 2*self.pad - FH) / self.stride)
        out_w = int(1 + (W + 2*self.pad - FW) / self.stride)

        col = im2col(x, FH, FW, self.stride, self.pad)
```

```
col_W = self.W.reshape(FN, -1).T # Expand the filter
out = np.dot(col, col_W) + self.b

out = out.reshape(N, out_h, out_w, -1).transpose(0, 3, 1, 2)

return out
```

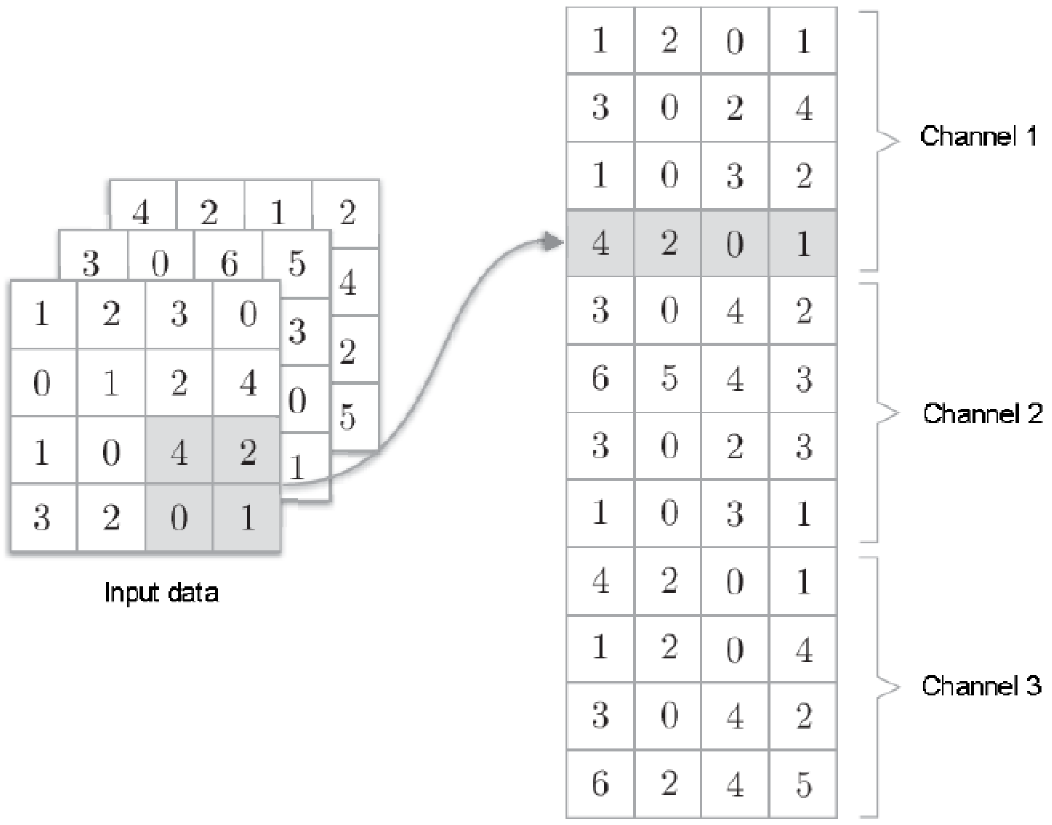
The **transpose** function changes the order of axes in a multidimensional array



Using NumPy's transpose to change the order of the axes – specifying the indices (numbers) to change the order of axes

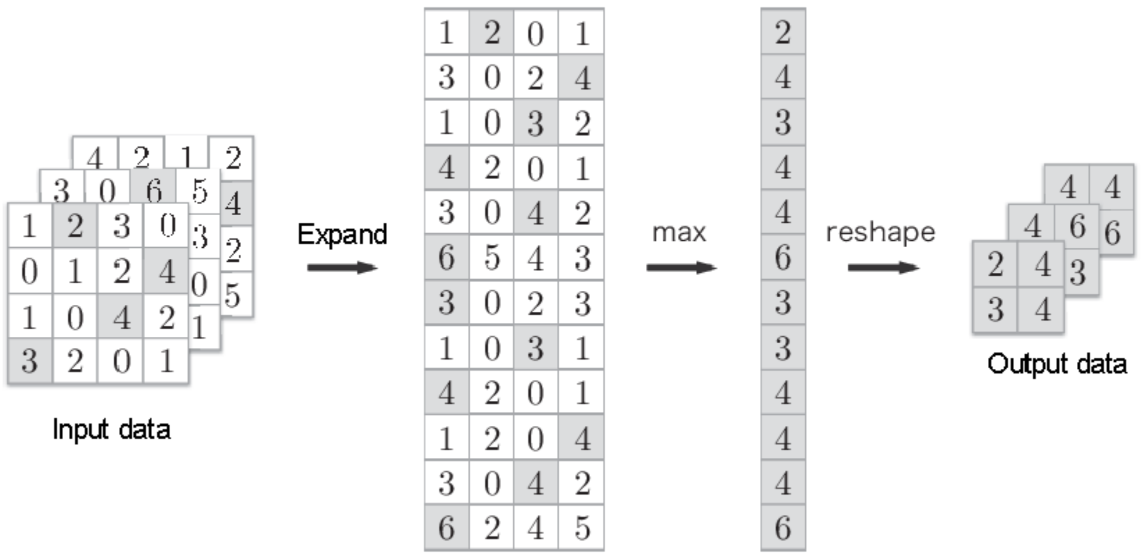
Implementing a Pooling Layer

use **im2col** to expand the input data when implementing a pooling layer, as in the case of a convolution layer. What is different is that pooling is independent of the channel dimension, unlike a convolution layer.



Expanding the target pooling area of the input data (pooling of 2x2)

After this expansion, you have only to take the maximum value in each row of the expanded matrix and transform the result into an appropriate shape



Flow of implementation of a pooling layer – the maximum elements in the pooling area are shown in gray

```
class Pooling:
    def __init__(self, pool_h, pool_w, stride=1, pad=0):
        self.pool_h = pool_h
        self.pool_w = pool_w
        self.stride = stride
        self.pad = pad

    def forward(self, x):
        N, C, H, W = x.shape
        out_h = int(1 + (H - self.pool_h) / self.stride)
        out_w = int(1 + (W - self.pool_w) / self.stride)
        # Expansion (1)

        col = im2col(x, self.pool_h, self.pool_w, self.stride, self.pad)
        col = col.reshape(-1, self.pool_h*self.pool_w)

        # Maximum value (2)
  out = np.max(col, axis=1)
  # Reshape (3)
  out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2)

        return out
```

there are three steps when it comes to implementing a pooling layer:

1. Expand the input data.
2. Take the maximum value in each row.
3. Reshape the output appropriately.

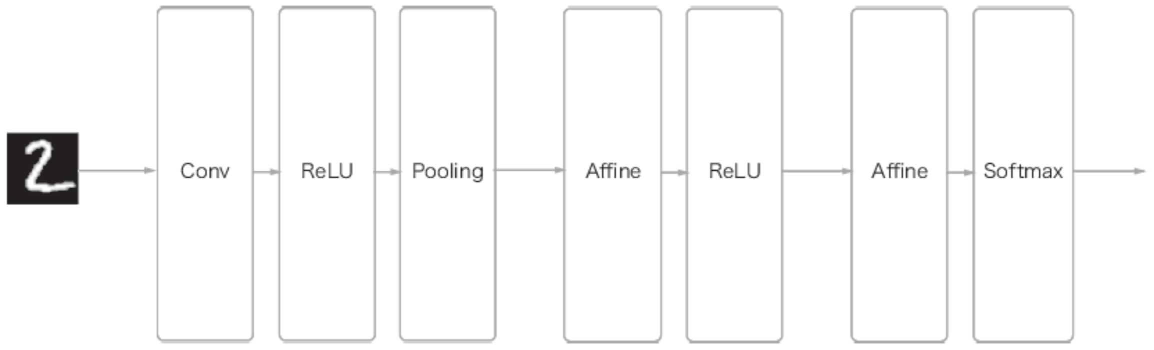
The implementation of each step is simple and is only one or two lines in length

That's all for the forward process in a pooling layer. As shown here, after expanding the input data into a shape that's suitable for pooling, subsequent implementations of it are very simple

Implementing a CNN

So far, we have implemented convolution and pooling layers. Now, we will combine these layers to create a CNN that recognizes handwritten digits and implement it

the network consists of "Convolution – ReLU – Pooling – Affine – ReLU – Affine – Softmax" layers



Network configuration of a simple CNN

Now, let's look at the initialization of **SimpleConvNet (__init__)**. It takes the following arguments:

- **input_dim**: Dimensions of the input data (**channel**, **height**, **width**).
- **conv_param**: Hyperparameters of the convolution layer (dictionary). The following are the dictionary keys:
- **filter_num**: Number of filters
- **filter_size**: Size of the filter
- **stride**: Stride
- **pad**: Padding
- **hidden_size**: Number of neurons in the hidden layer (fully connected)
- **output_size**: Number of neurons in the output layer (fully connected)

• **weight_init_std**: Standard deviation of the weights at initialization

Here, the hyperparameters of the convolution layer are provided as a dictionary called **conv_param**. We assume that the required hyperparameter values are stored using **{'filter_num':30, 'filter_size':5, 'pad':0, 'stride':1}**.

The implementation of the initialization of **SimpleConvNet** is a little long, so here it's divided into three parts to make this easier to follow. The following code shows the first part of the initialization process:

```
class SimpleConvNet:
    def __init__(self, input_dim=(1, 28, 28),
                conv_param={'filter_num':30, 'filter_size':5,
                    'pad':0, 'stride':1},
                hidden_size=100, output_size=10, weight_init_std=0.01):
        filter_num = conv_param['filter_num']
        filter_size = conv_param['filter_size']
        filter_pad = conv_param['pad']
        filter_stride = conv_param['stride']
        input_size = input_dim[1]
        conv_output_size = (input_size - filter_size + 2*filter_pad) / \
                        filter_stride + 1
        pool_output_size = int(filter_num * (conv_output_size/2) *(conv_
output_size/2))
```

Here, the hyperparameters of the convolution layer that are provided by the initialization argument are taken out of the dictionary (so that we can use them later). Then, the output size of the convolution layer is calculated. The following code initializes the weight parameters:

```
   self.params = {}
   self.params['W1'] = weight_init_std * \
   np.random.randn(filter_num, input_dim[0],
   filter_size, filter_size)
   self.params['b1'] = np.zeros(filter_num)
   self.params['W2'] = weight_init_std * \
   np.random.randn(pool_output_size,hidden_size)
   self.params['b2'] = np.zeros(hidden_size)
   self.params['W3'] = weight_init_std * \
   np.random.randn(hidden_size, output_size)
   self.params['b3'] = np.zeros(output_size)
```

The parameters required for training are the weights and biases of the first (convolution) layer and the remaining two fully connected layers. The parameters are stored in the instance dictionary variable, **params**. The **W1** key is used for the weight, while the **b1** key is used for the bias of the first (convolution) layer. In the same way, the **W2** and **b2** keys are used for the weight and bias of the second (fully connected) layer and the **W3** and **b3** keys are used for the weight and bias of the third (fully connected) layer, respectively. Lastly, the required layers are generated, as follows:

```
    self.layers = OrderedDict( )
    self.layers['Conv1'] = Convolution(self.params['W1'],
                                       self.params['b1'],
                                       conv_param['stride'],
                                       conv_param['pad'])
    self.layers['Relu1'] = Relu( )
    self.layers['Pool1'] = Pooling(pool_h=2, pool_w=2, stride=2)
    self.layers['Affine1'] = Affine(self.params['W2'],
                                       self.params['b2'])
    self.layers['Relu2'] = Relu( )
    self.layers['Affine2'] = Affine(self.params['W3'],
                                       self.params['b3'])

    self.last_layer = SoftmaxWithLoss( )
```

Layers are added to the ordered dictionary (**OrderedDict**) in an appropriate order. Only the last layer, **SoftmaxWithLoss**, is added to another variable, **last-layer**.

This is the initialization of **SimpleConvNet**. After the initialization, you can implement the **predict** method for predicting and the **loss** method for calculating the value of the loss function, as follows:

```
def predict(self, x):
    for layer in self.layers.values( ):
        x = layer.forward(x)
    return x

def loss(self, x, t):
    y = self.predict(x)
return self.lastLayer.forward(y, t)
```

Here, the **x** argument is the input data and the **t** argument is the label. The **predict** method only calls the added layers in order from the top, and passes the result to the next layer. In addition to forward processing in the **predict** method, the **loss** method performs forward processing until the last layer, **SoftmaxWithLoss**.

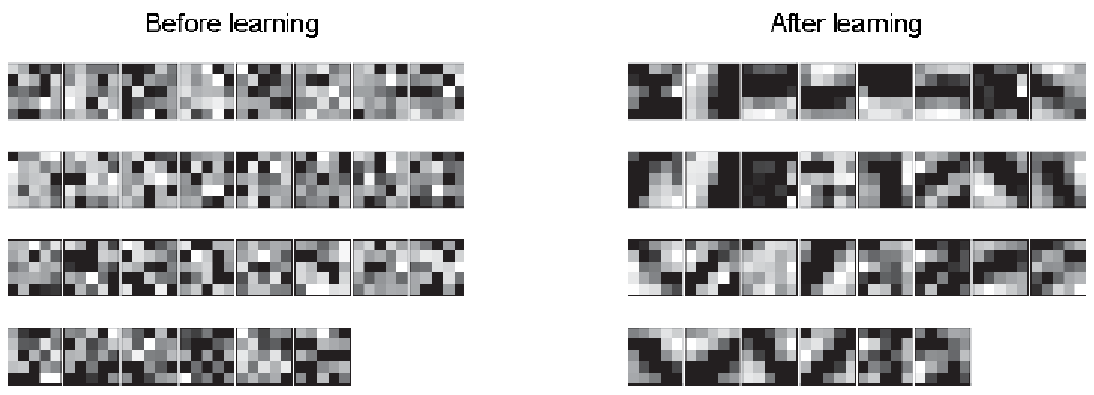
The following implementation obtains the gradients via backpropagation, as follows:

```
def gradient(self, x, t):
    # forward
    self.loss(x, t)

    # backward
    dout = 1
    dout = self.lastLayer.backward(dout)

    layers = list(self.layers.values( ))
    layers.reverse( )
    for layer in layers:
        dout = layer.backward(dout)
    # Settings
    grads = {}
    grads['W1'] = self.layers['Conv1'].dW
    grads['b1'] = self.layers['Conv1'].db
    grads['W2'] = self.layers['Affine1'].dW
    grads['b2'] = self.layers['Affine1'].db
    grads['W3'] = self.layers['Affine2'].dW
    grads['b3'] = self.layers['Affine2'].db

    return grads
```

Visualizing a CNN

Visualizing the Weight of the First Layer

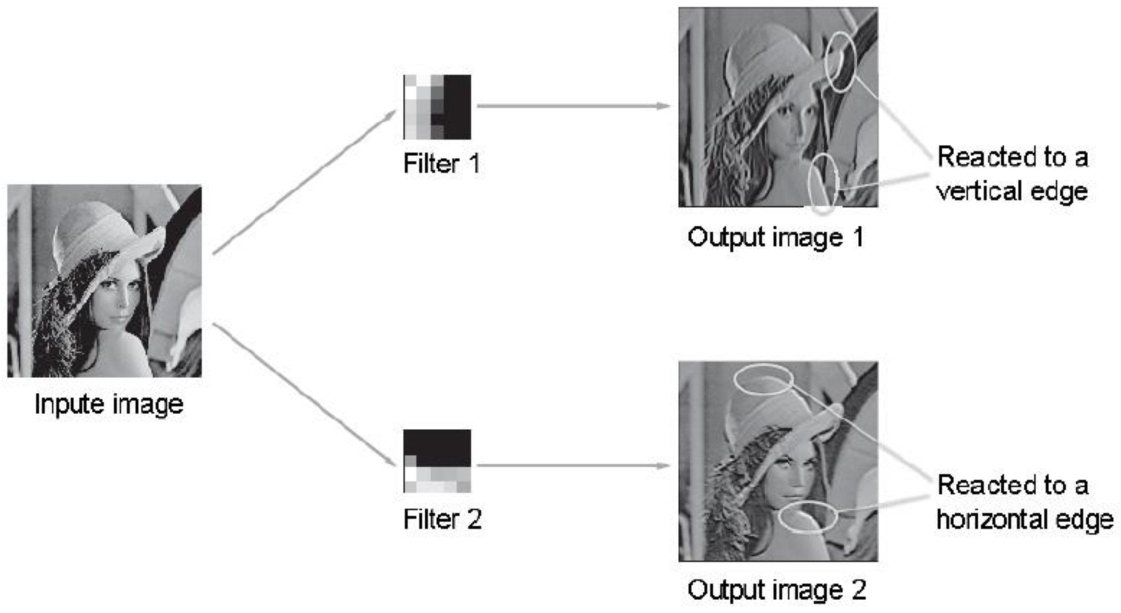
we conducted simple CNN training for the MNIST dataset. The shape of the weight of the first (convolution) layer was (30, 1, 5, 5). It was 5x5 in size, had 1 channel, and 30 filters. When the filter is 5x5 in size and has 1 channel, it can be visualized as a one-channel gray image. Now, let's show the filters of the convolution layer (the first layer) as images. Here, we will compare the weights before and after training



Weight of the first (convolution) layer before and after training. The elements of the weight are real numbers, but they are normalized between 0 and 255 to show the images so that the smallest value is black (0) and the largest value is white (255)

the filters before training are initialized randomly. Black-and-white shades have no pattern. On the other hand, the filters after training are images with a pattern. Some filters have gradations from white to black, while some filters have small areas of color (called "blobs"), which indicates that training provided a pattern to the filters.

Below Figure shows the results when two learned filters are selected, and convolution processing is performed on the input image. You can see that "filter 1" reacted to a vertical edge and that "filter 2" reacted to a horizontal edge



Filters reacting to horizontal and vertical edges. White pixels appear at a vertical edge in output image 1. Meanwhile, many white pixels appear at a horizontal edge in output image 2

filters in a convolution layer extract primitive information such as edges and blobs. The CNN that was implemented earlier passes such primitive information to subsequent layers

Typical CNNs

CNNs of various architectures have been proposed so far. In this section, we will look at two important networks. One is LeNet, The other is AlexNet.

LeNet

LeNet is a network for handwritten digit recognition that was proposed in 1998. In the network, a convolution layer and a pooling layer (i.e., a subsampling layer that only "thins out elements") are repeated, and finally, a fully connected layer outputs the result.

There are some differences between LeNet and the "current CNN." One is that there's an activation function. A sigmoid function is used in LeNet, while ReLU is mainly used now.

Subsampling is used in the original LeNet to reduce the size of intermediate data, while max pooling is mainly used now:

In this way, there are some differences between LeNet and the "current CNN," but they are not significant. This is surprising when we consider that LeNet was the "first CNN" to be proposed almost 20 years ago.

AlexNet

AlexNet was published nearly 20 years after LeNet was proposed. Although AlexNet created a boom in deep learning, its network architecture hasn't changed much from LeNet:

AlexNet stacks a convolution layer and a pooling layer and outputs the result through a fully connected layer. Its architecture is not much different from LeNet, but there are some differences, as follows:

- ReLU is used as the activation function
- A layer for local normalization called **Local Response Normalization** (**LRN**) is used
- Dropout is used (see *Dropout* sub-section in *Chapter 6*, *Training Techniques*)

LeNet and AlexNet are not very different in terms of their network architectures. However, the surrounding environment and computer technologies have advanced greatly. Now, everyone can obtain a large quantity of data, and widespread GPUs that are good at large parallel computing enable massive operations at high speed. Big data and GPUs greatly motivated the development of deep learning.