

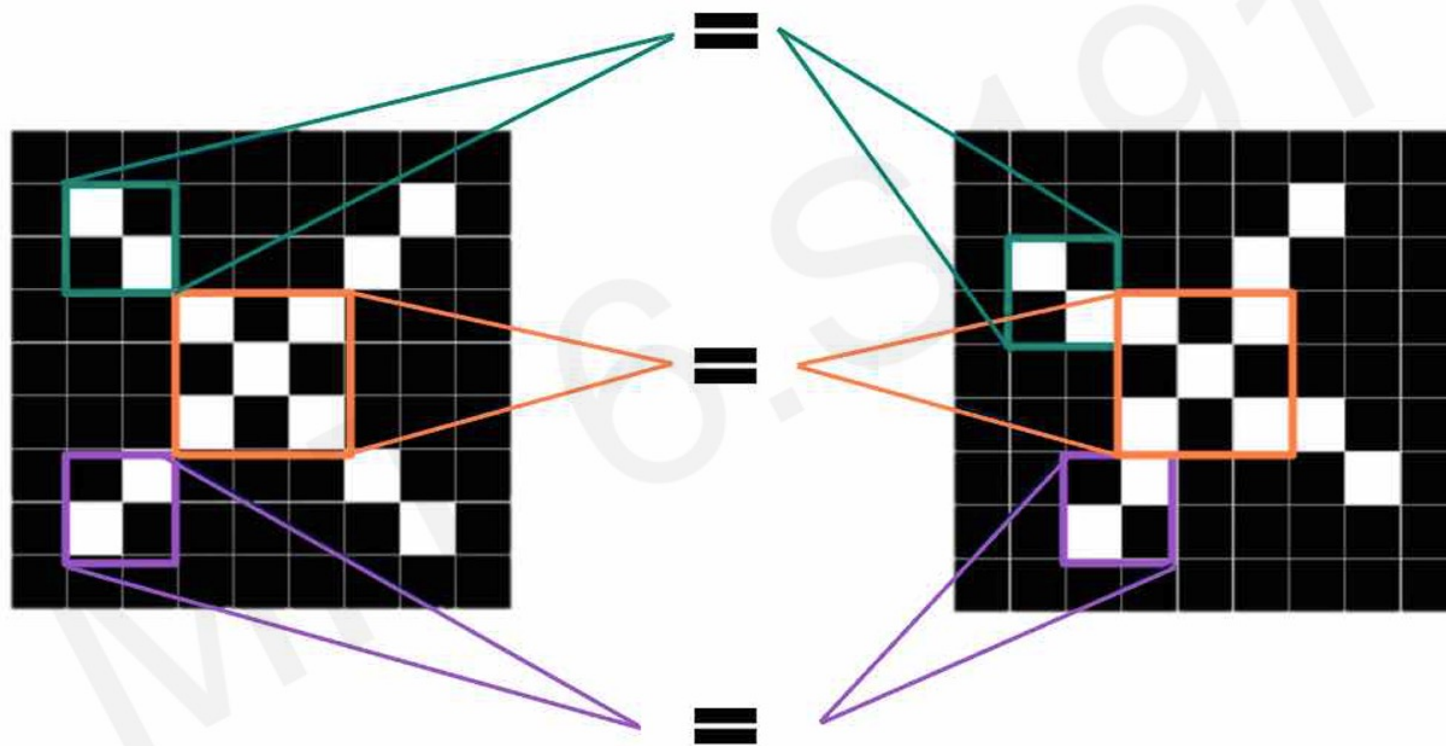
DEEP LEARNING FOR COMPUTER VISION

Week8



Dr. Tuchsanaï. PloySuwan

Feature Extraction and Convolution



-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	-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	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	-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	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	-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	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

Image is represented as matrix of pixel values... and computers are literal!
 We want to be able to classify an X as an X even if it's shifted, shrunk, rotated, deformed.

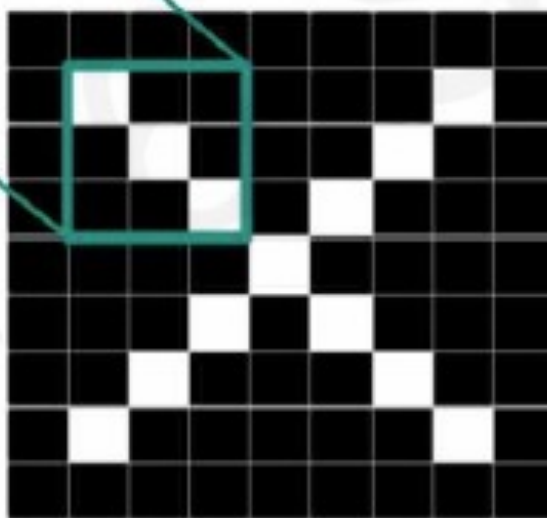
Filters to Detect X Features

filters

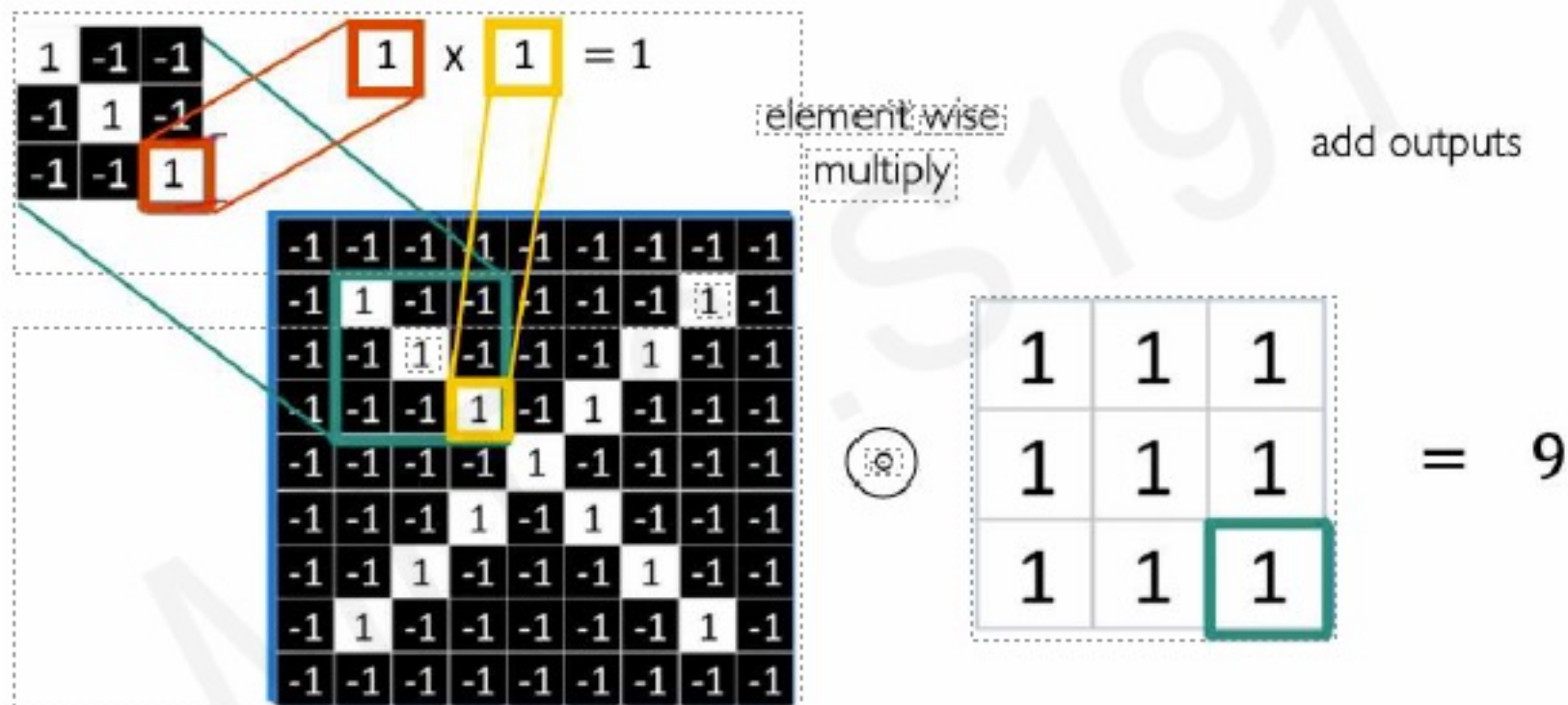
1	-1	-1
-1	1	-1
-1	-1	1

1	-1	1
-1	1	-1
1	-1	1

-1	-1	1
-1	1	-1
1	-1	-1

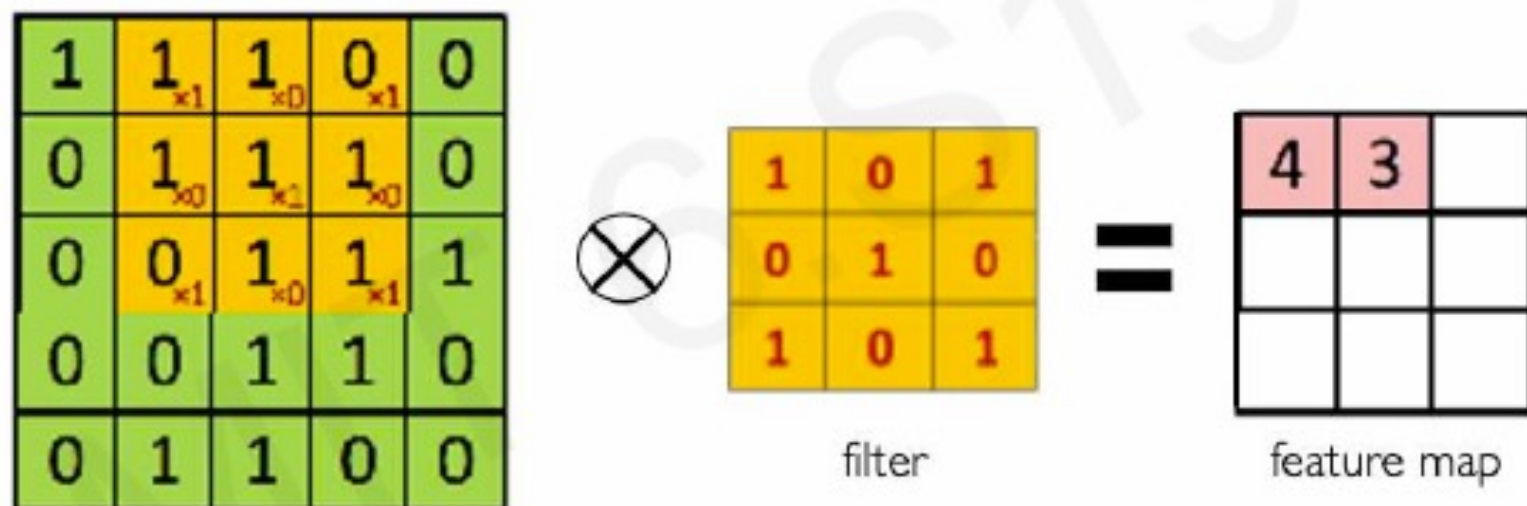


The Convolution Operation



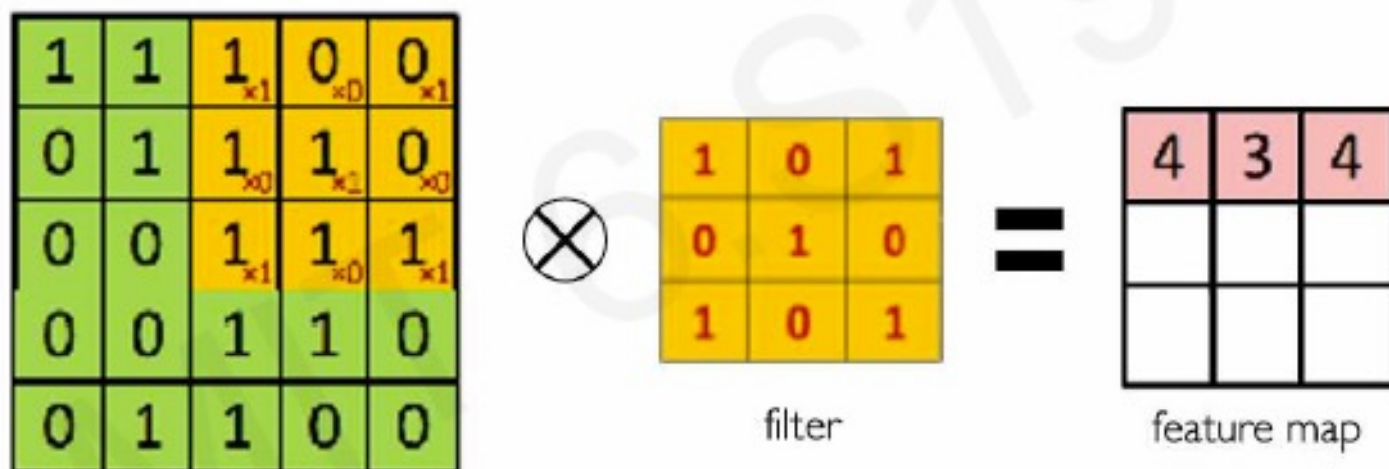
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



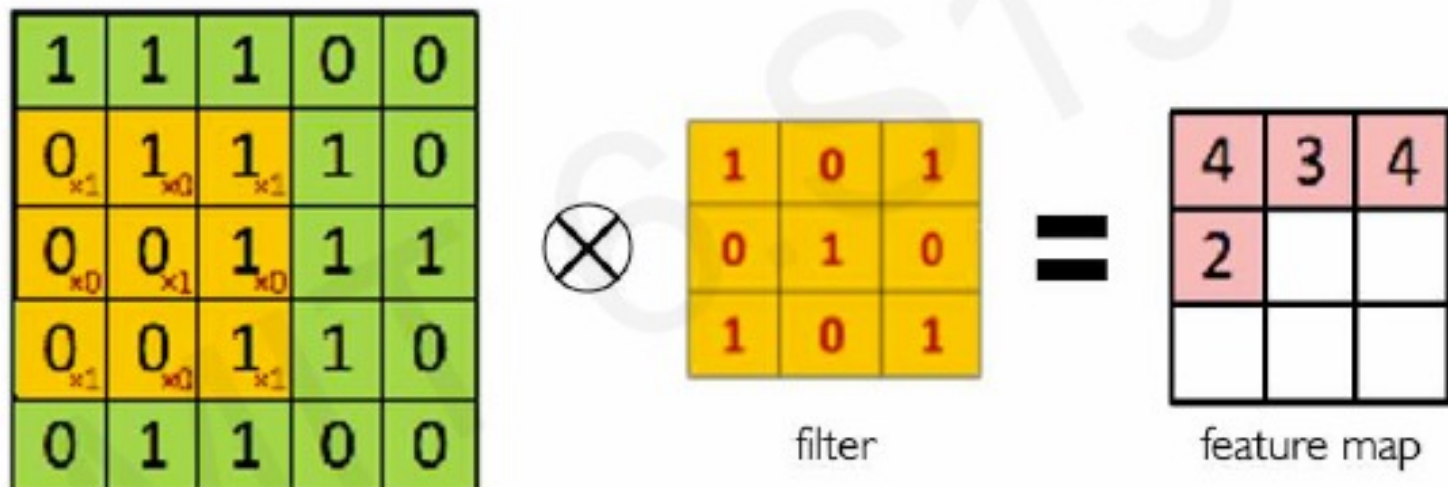
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



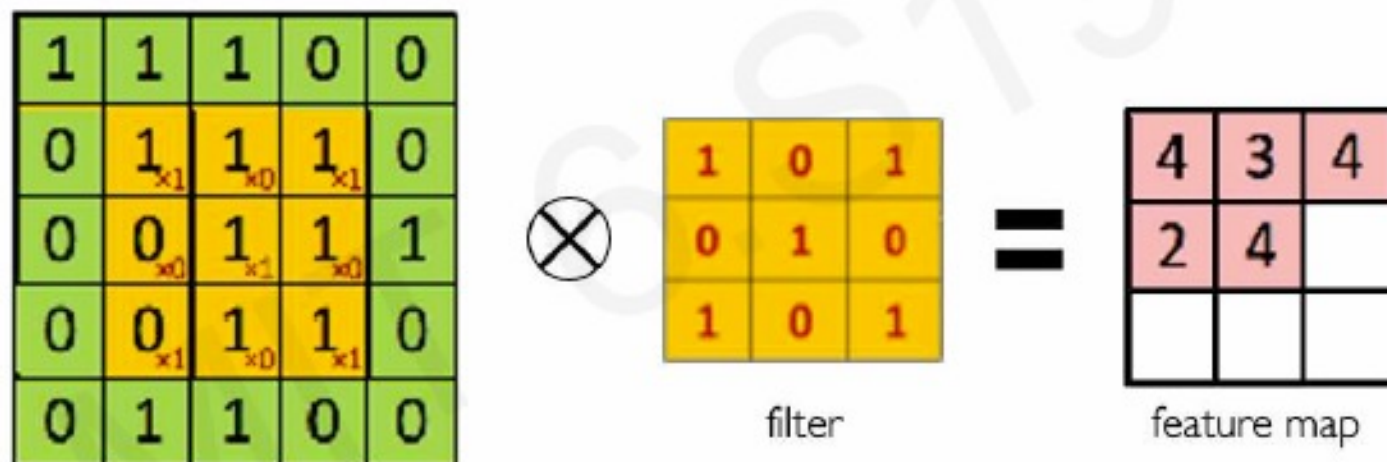
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



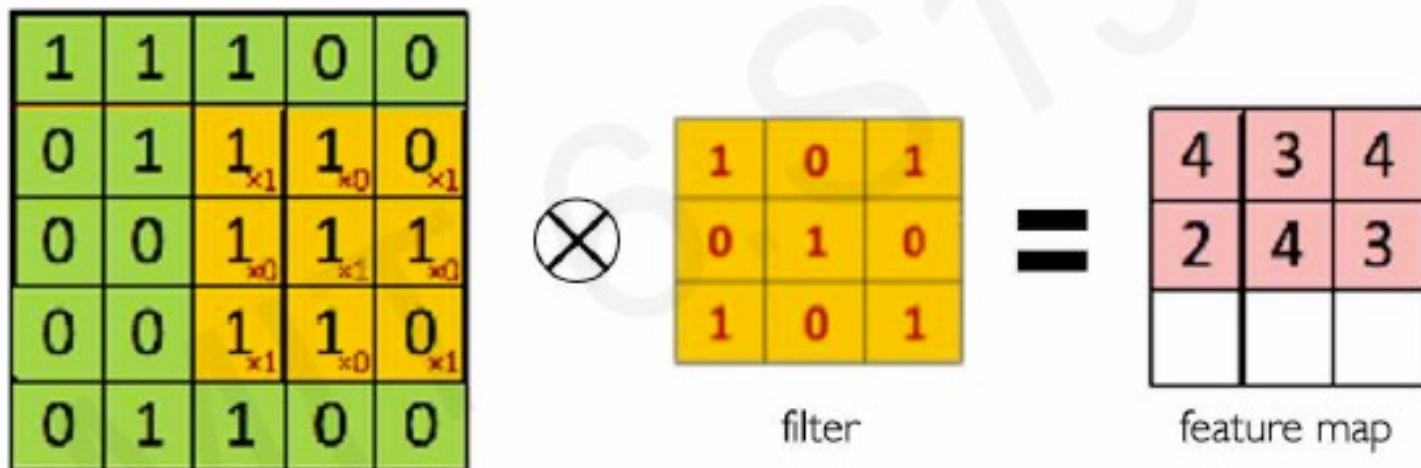
The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



The Convolution Operation

We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs

1	1	1	0	0
0	1	1	1	0
0 _{x1}	0 _{x0}	1 _{x1}	1	1
0 _{x0}	0 _{x1}	1 _{x0}	1	0
0 _{x1}	1 _{x0}	1 _{x1}	0	0



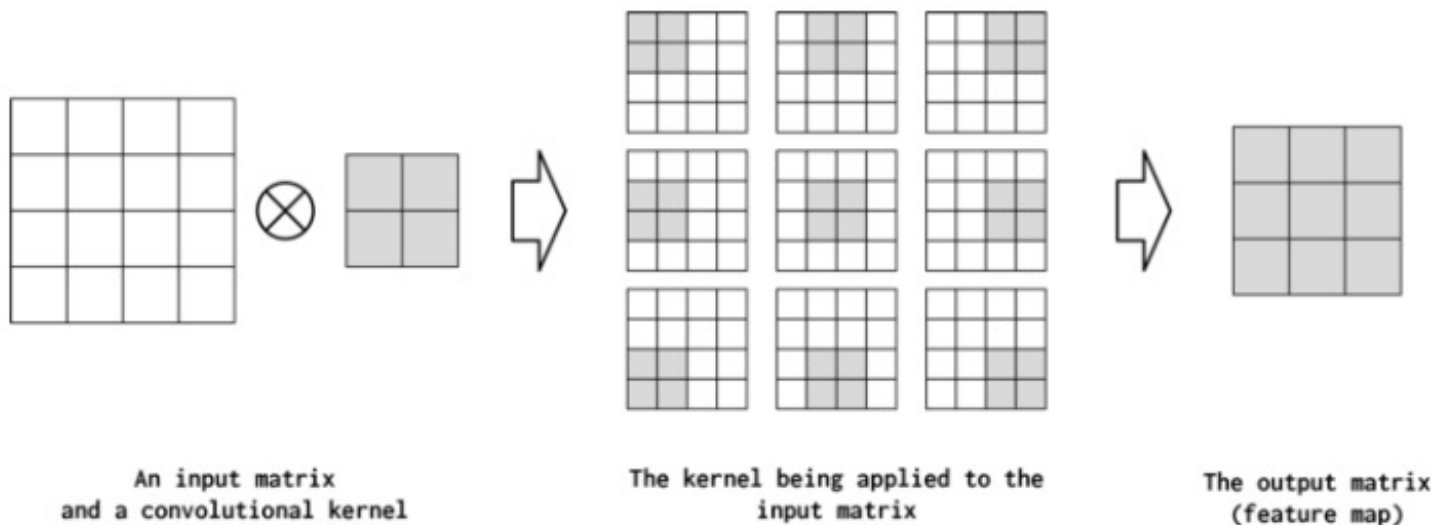
1	0	1
0	1	0
1	0	1

filter



4	3	4
2	4	3
2		

feature map



Convolution Operation

To calculate the output dimension after a convolution operation, we can use the following formula.

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1$$

n_{in} : number of input features

n_{out} : number of output features

k : convolution kernel size

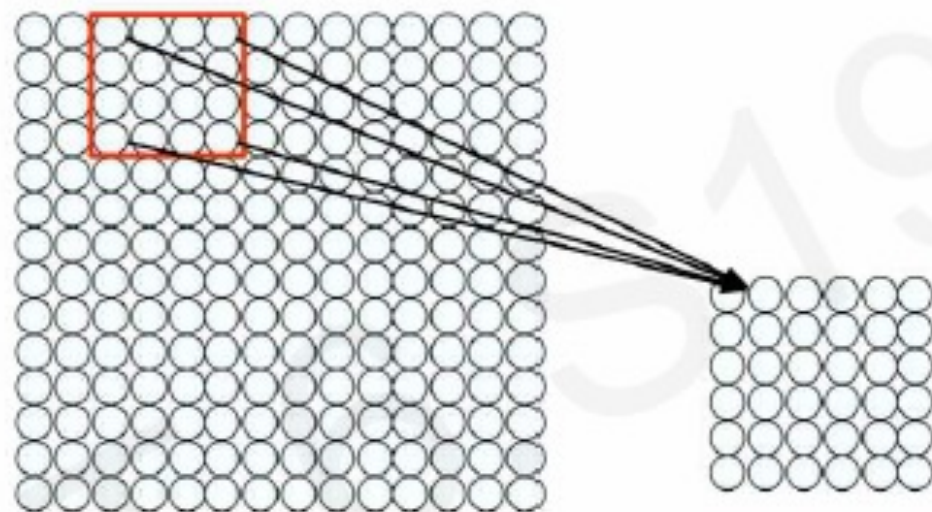
p : convolution padding size

s : convolution stride size



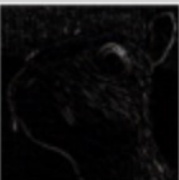
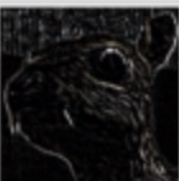

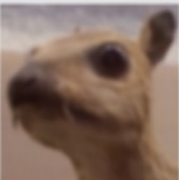
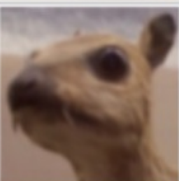
1x1	1x0	1x1	0	0
0x0	1x1	1x0	1	0
0x1	0x0	1x1	1	1
0	0	1	1	0
0	1	1	0	0

4		

Feature Extraction with Convolution



- 1) Apply a set of weights – a filter – to extract **local features**
- 2) Use **multiple filters** to extract different features
- 3) **Spatially share** parameters of each filter

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	



Original



Sharpen



Edge Detect

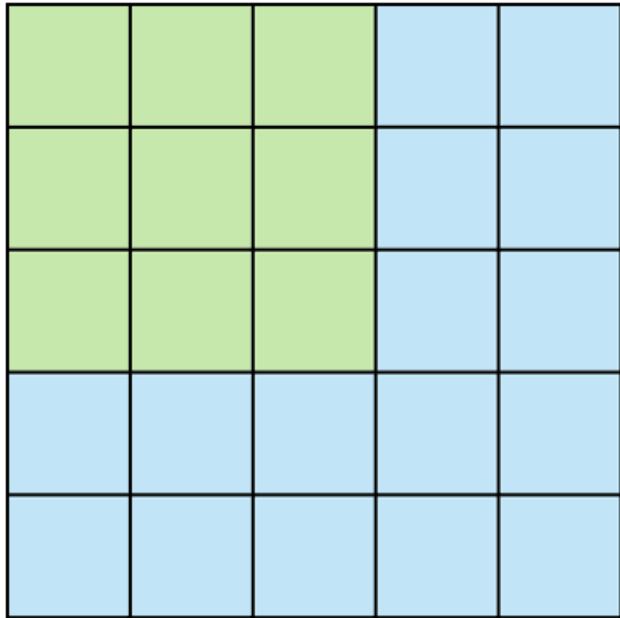


"Strong" Edge Detect

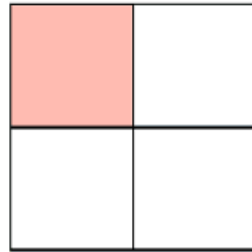
Convolution of an image with different filters

can perform operations such as edge detection, blur and sharpen by applying filters. The below example shows various convolution image after applying different types of filters (Kernels).

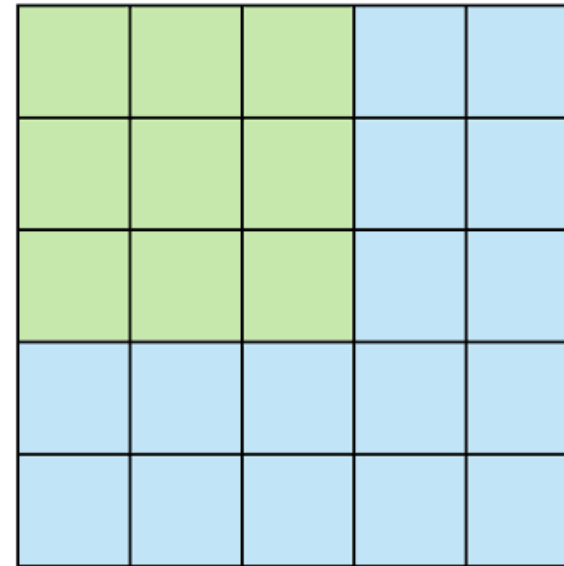
Stride : *Stride* specifies how much we move the convolution filter at each step.



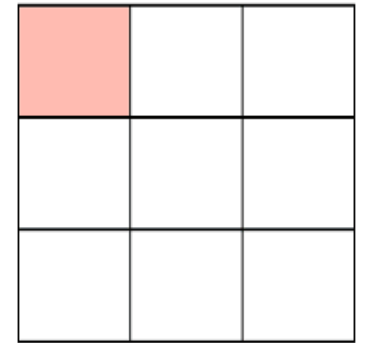
Stride 2



Feature Map



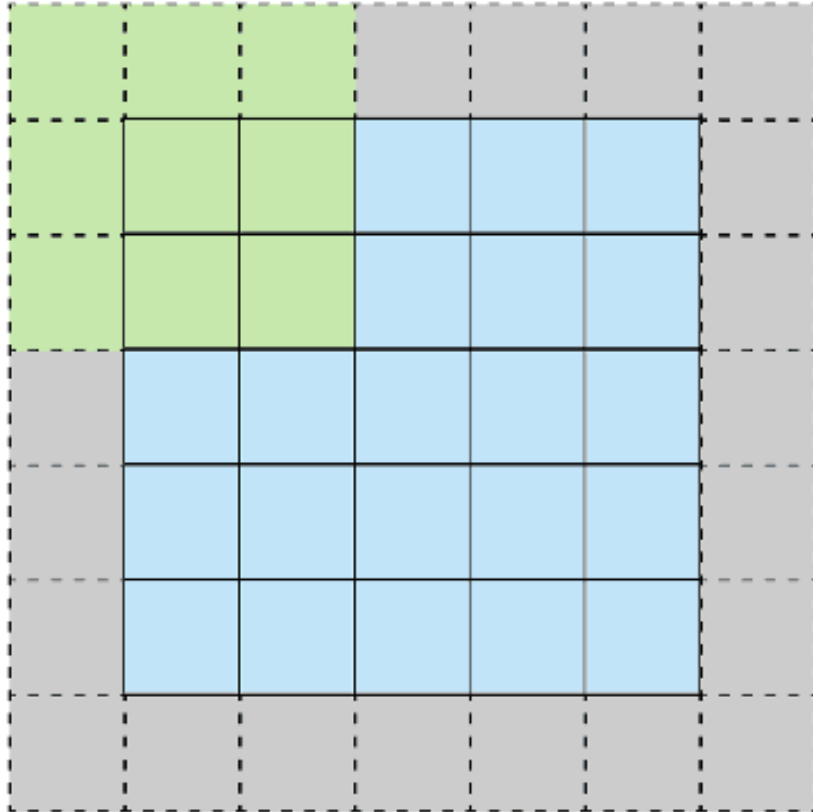
Stride 1



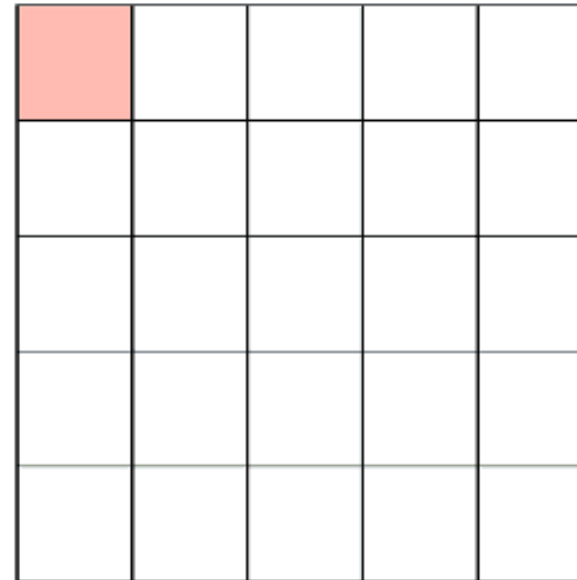
Feature Map

Padding

Here we have retained more information from the borders and have also preserved the size of the image.

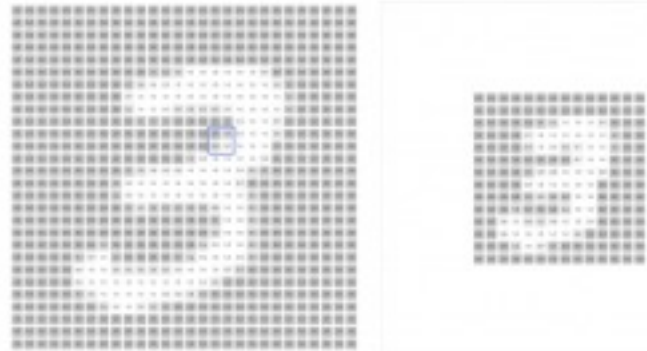
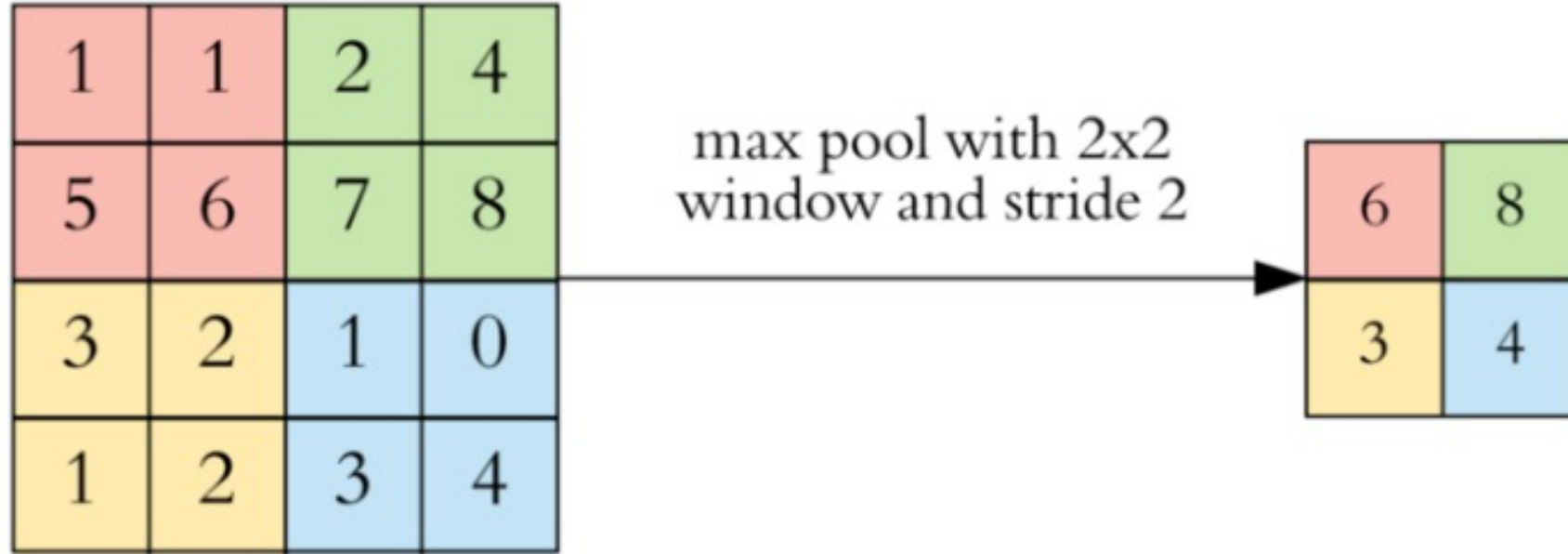


Stride 1 with Padding

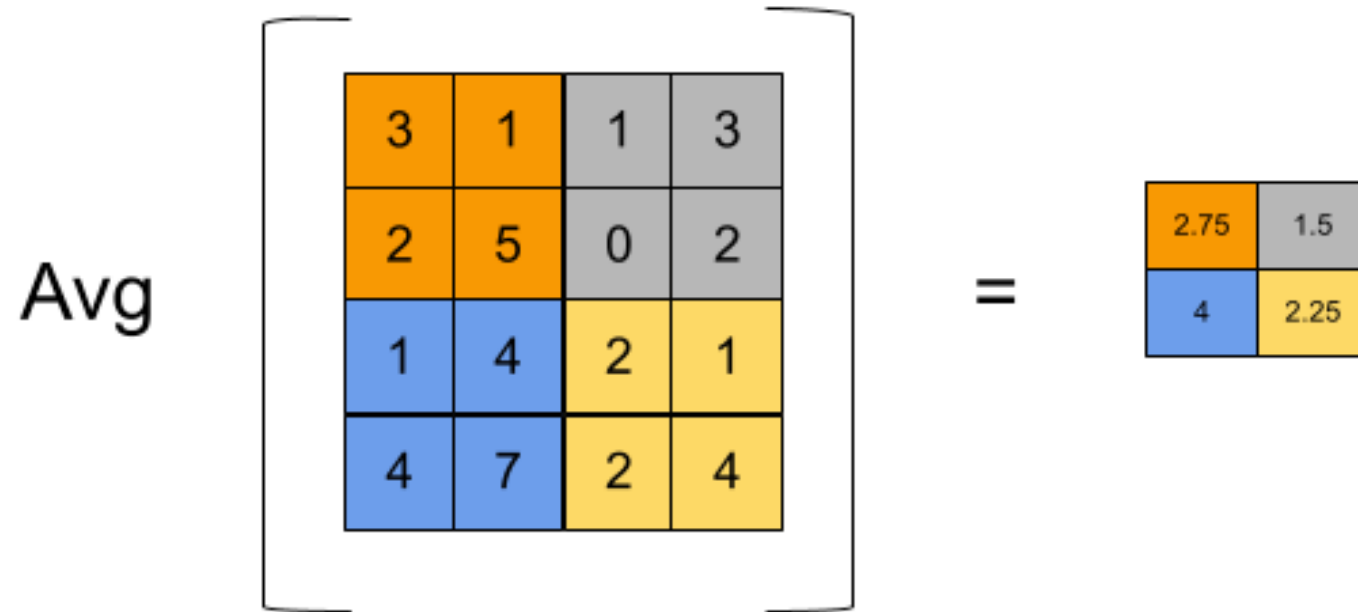


Feature Map

Max Pooling

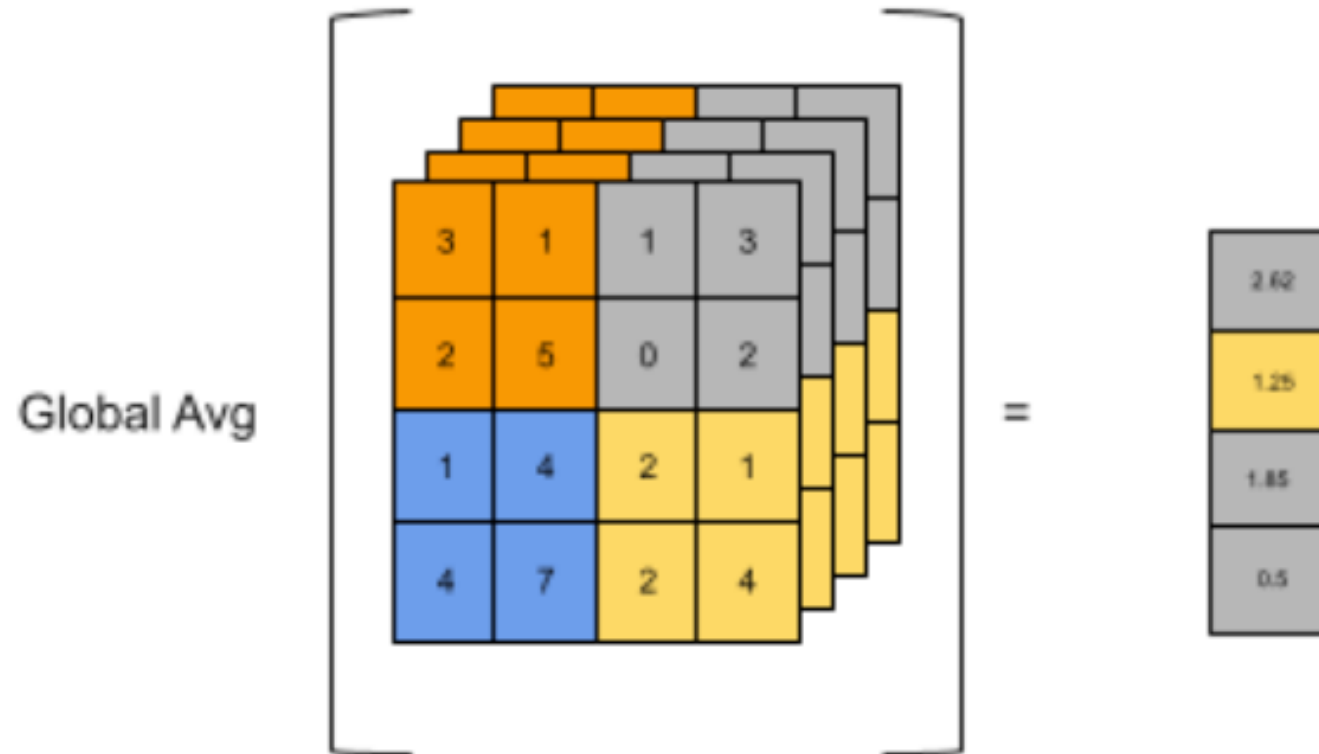


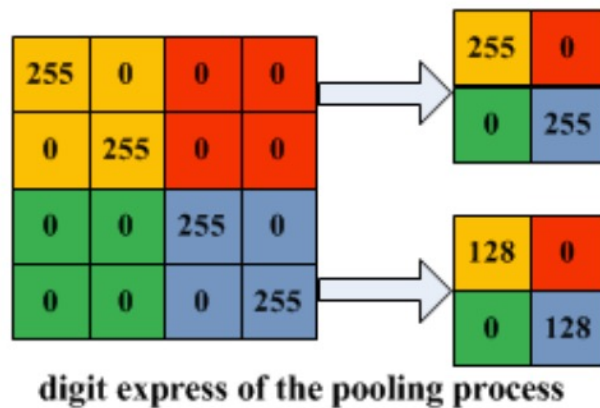
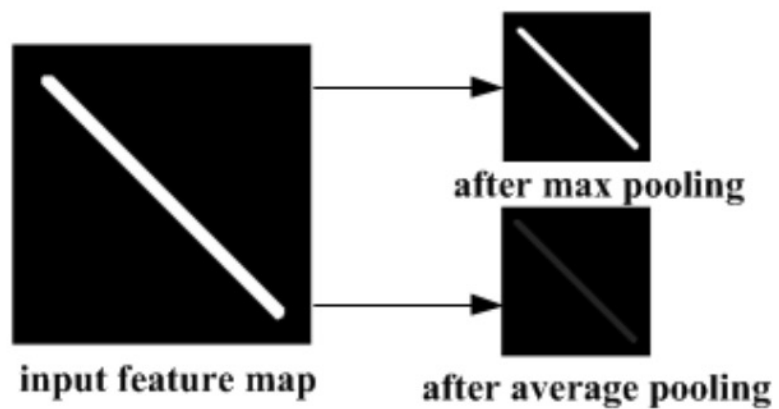
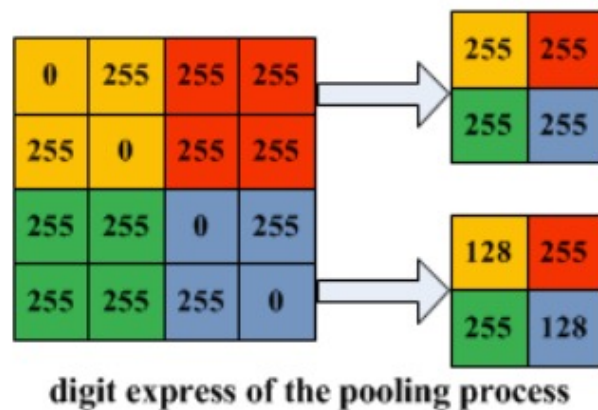
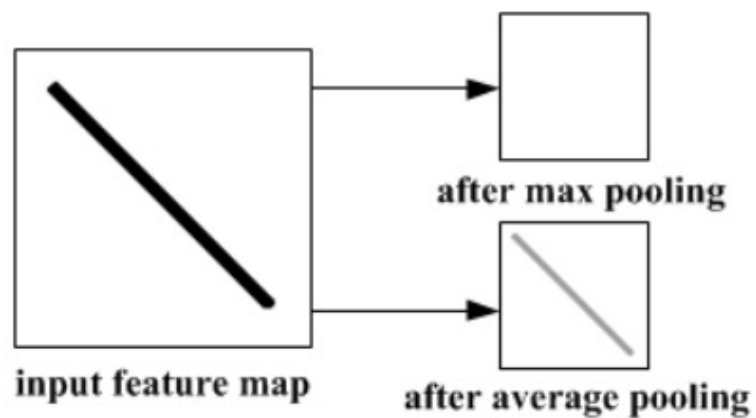
Average pooling



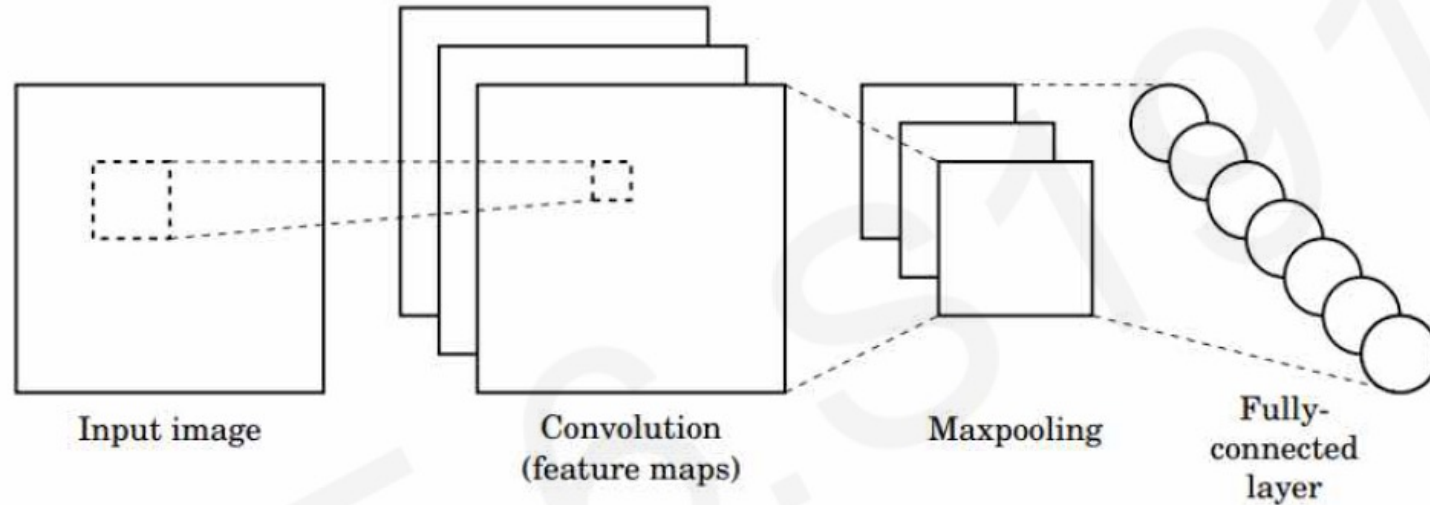
Kernel =2 strides =2

Global Average Pooling



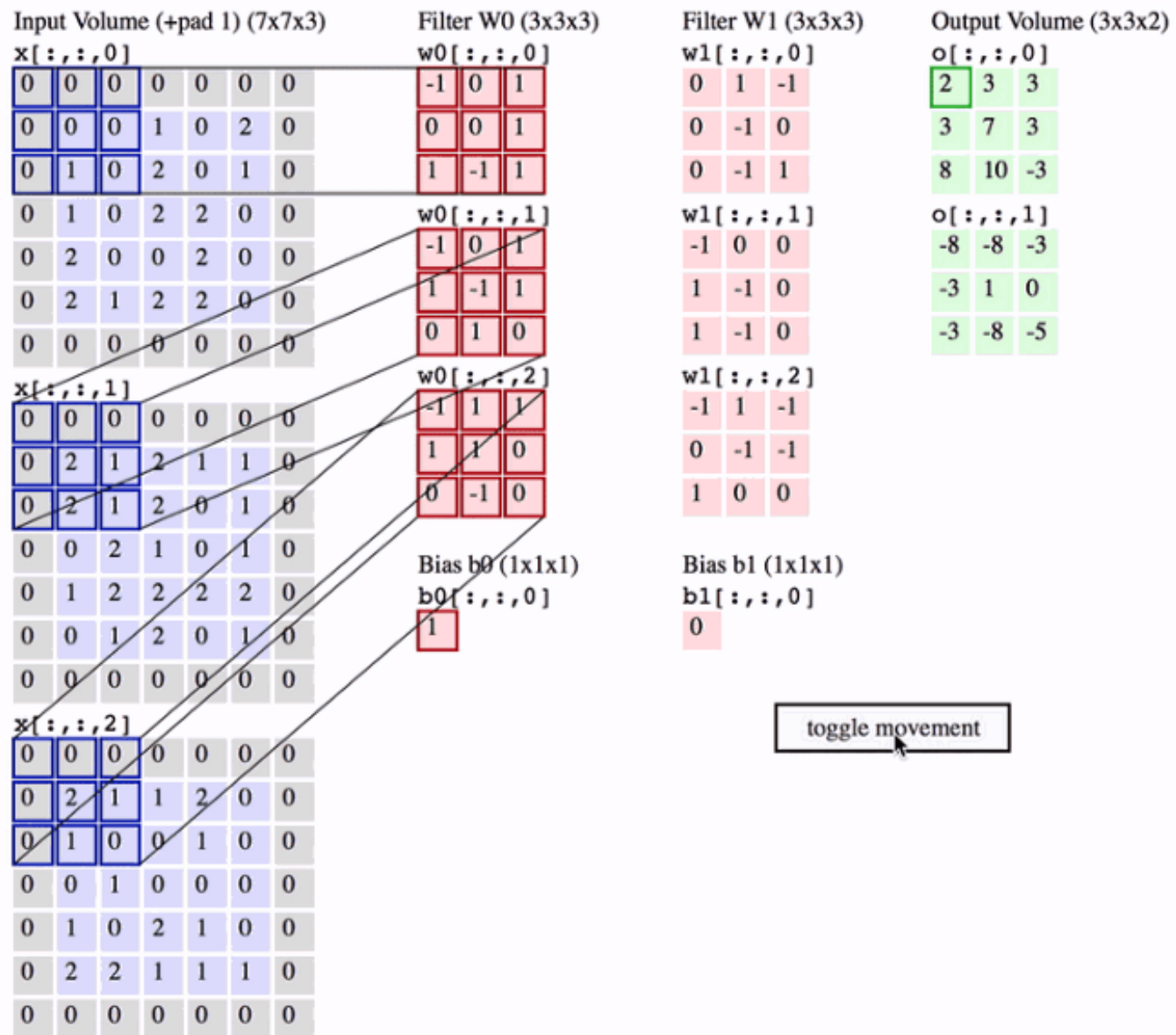


CNNs for Classification



1. **Convolution:** Apply filters to generate feature maps.
2. **Non-linearity:** Often ReLU.
3. **Pooling:** Downsampling operation on each feature map.

Train model with image data.
Learn weights of filters in convolutional layers.



```
import torch
import torch.nn as nn

✓ 0.2s

input = torch.randn(20, 3, 256, 256)

m = nn.Conv2d(in_channels=3, out_channels=28, kernel_size=3, stride=1, padding=0)
output = m(input)

print(input.shape)
print(output.shape)

✓ 0.1s

torch.Size([20, 3, 256, 256])
torch.Size([20, 28, 254, 254])
```

```
m.state_dict()['weight'].shape

✓ 0.2s

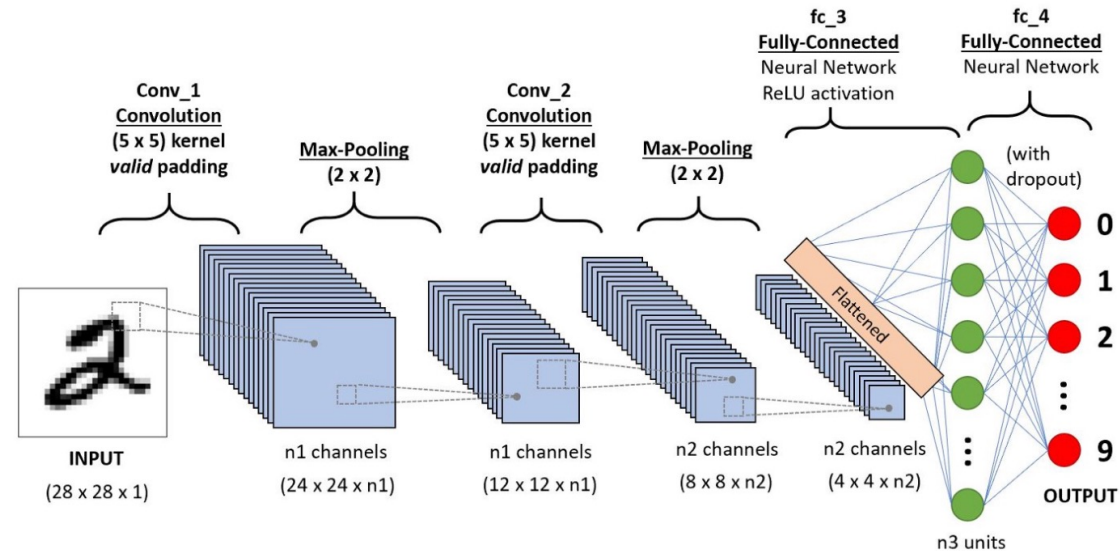
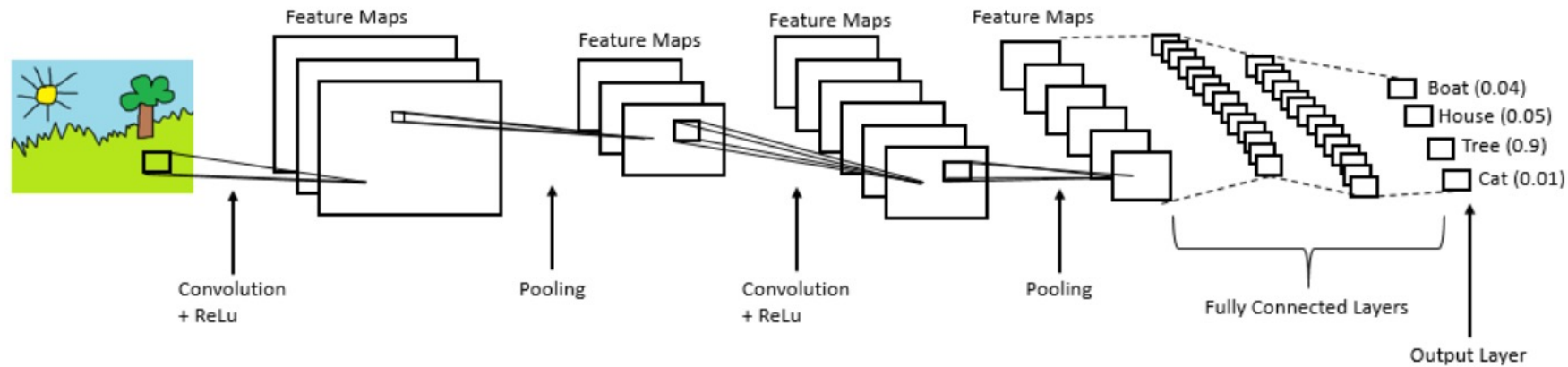
torch.Size([28, 3, 3, 3])
```

```
m.state_dict()['bias'].shape

✓ 0.2s

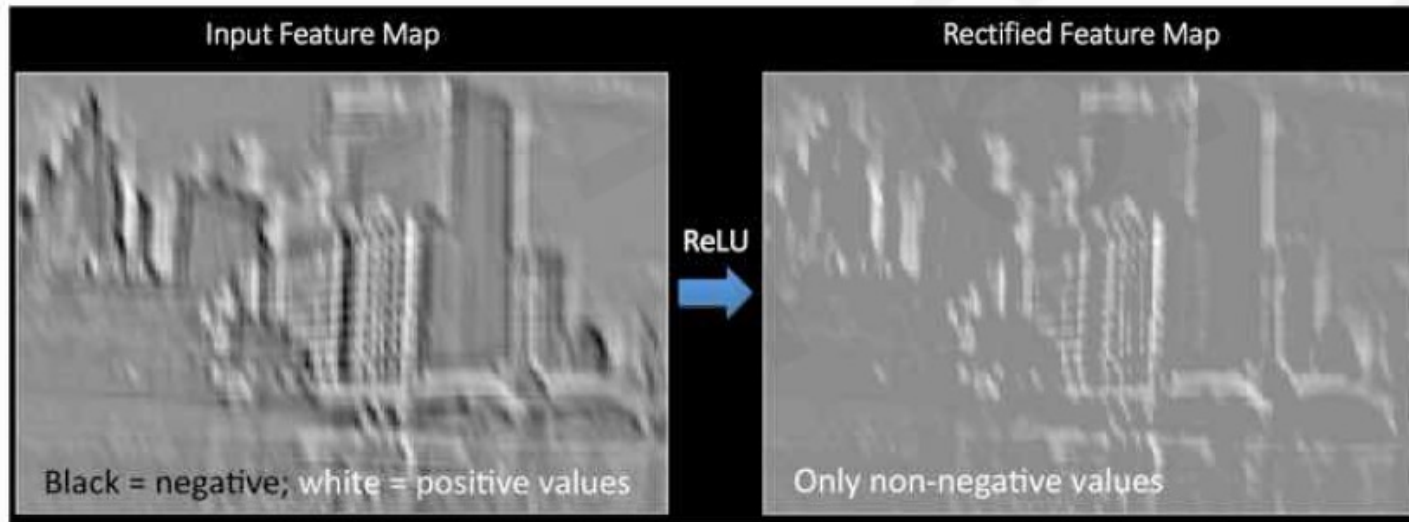
torch.Size([28])
```


In the above diagram, the feature map matrix will be converted as vector (x_1, x_2, x_3, \dots). With the fully connected layers, we combined these features together to create a model. Finally, we have an activation function such as softmax or sigmoid to classify the outputs as cat, dog, car, truck etc.,

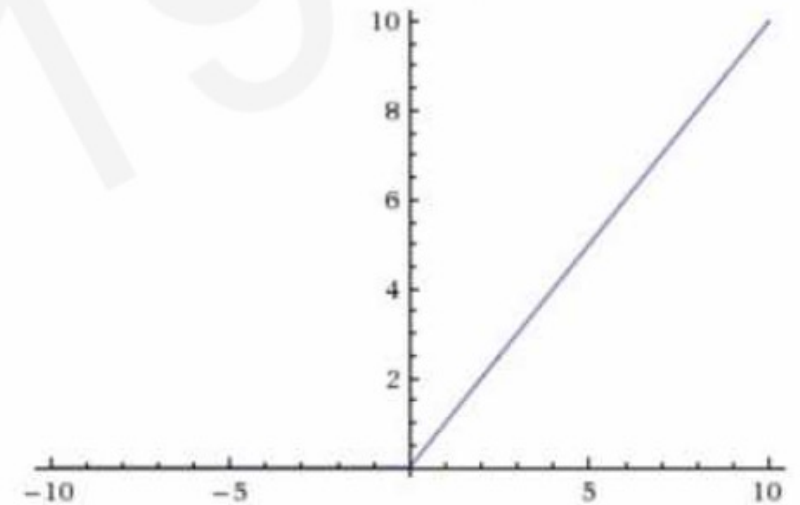


Introducing Non-Linearity

- Apply after every convolution operation (i.e., after convolutional layers)
- ReLU: pixel-by-pixel operation that replaces all negative values by zero. **Non-linear operation!**



Rectified Linear Unit (ReLU)

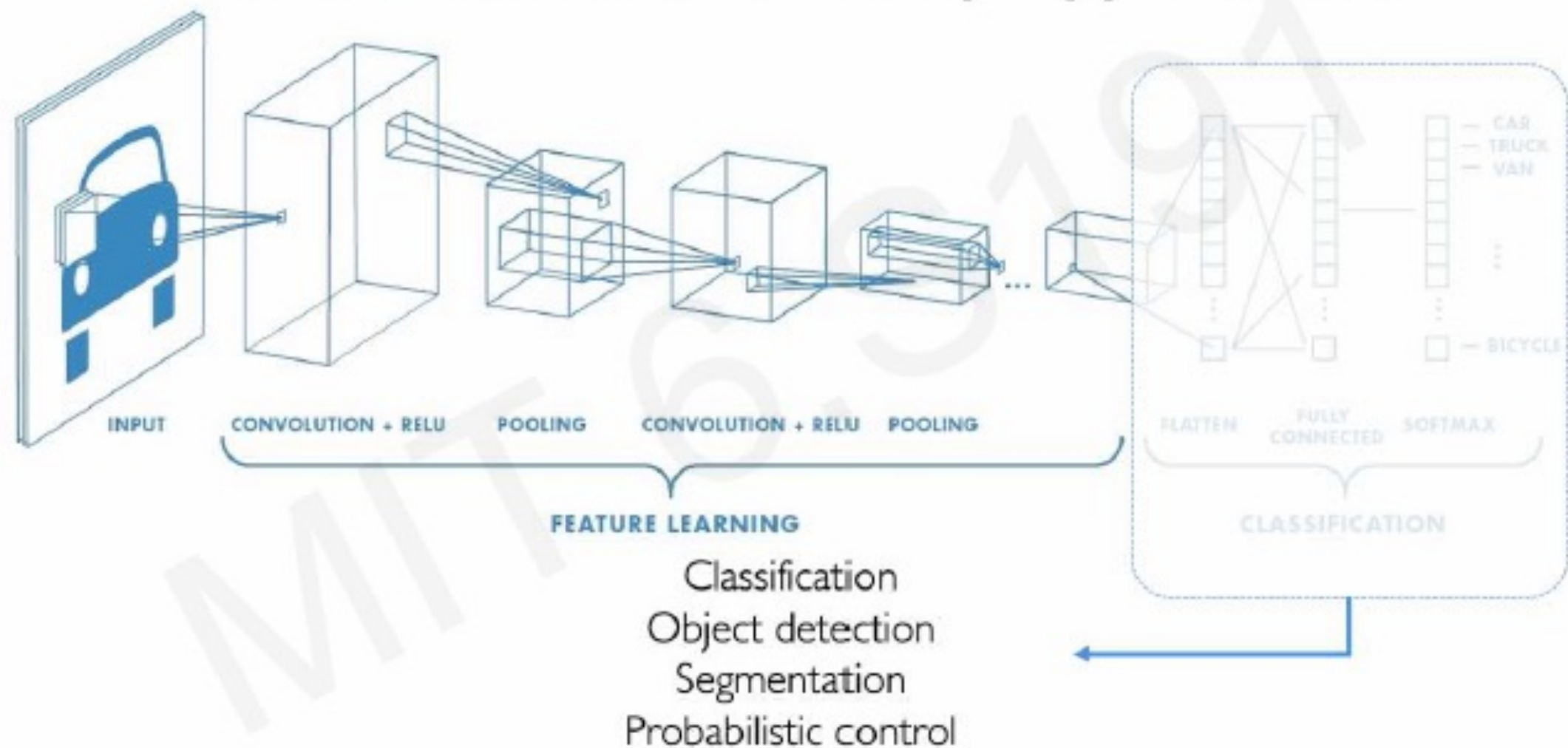


$$g(z) = \max(0, z)$$

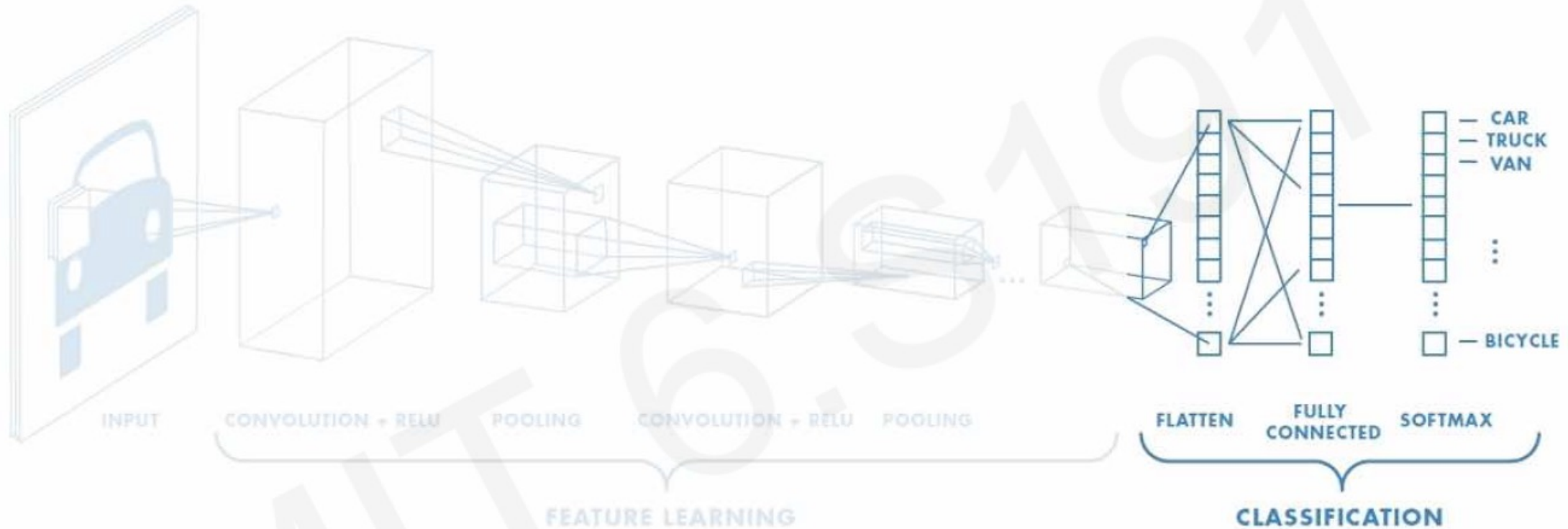


`tf.keras.layers.ReLU`

An Architecture for Many Applications



CNNs for Classification: Class Probabilities



- CONV and POOL layers output high-level features of input
- Fully connected layer uses these features for classifying input image
- Express output as **probability** of image belonging to a particular class

$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

PyTorch Conv2D

Below are the syntax and parameters of the Conv2D PyTorch function.

Syntax of Conv2D

```
torch.nn.Conv2d(in_channels: int, out_channels: int, kernel_size: Union[T, T], stride: Union[T, T], padding: Union[T, T], bias: bool = True)
```

Parameters

- **in_channels** (int) – Number of channels in the input image
- **out_channels** (int) – Number of channels produced by the convolution
- **kernel_size** (int or tuple) – Size of the convolving kernel
- **stride** (int or tuple, optional) – Stride of the convolution. Default: 1
- **padding** (int or tuple, optional) – Zero-padding added to both sides of the input. Default: 0
- **padding_mode** (string, optional) – 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'
- **dilation** (int or tuple, optional) – Spacing between kernel elements. Default: 1
- **groups** (int, optional) – Number of blocked connections from input channels to output channels. Default: 1
- **bias** (bool, optional) – If True, adds a learnable bias to the output. Default: True

We now create the instance of Conv2D function by passing the required parameters including square kernel size of 3×3 and stride = 1. We then apply this convolution to randomly generated input data.

```
m = nn.Conv2d(2, 28, 3, stride=1)

input = torch.randn(20, 2, 50, 50)

output = m(input)
```

Other Examples of Conv2D

Below are some other examples of PyTorch Conv2D function usages with different parameters.

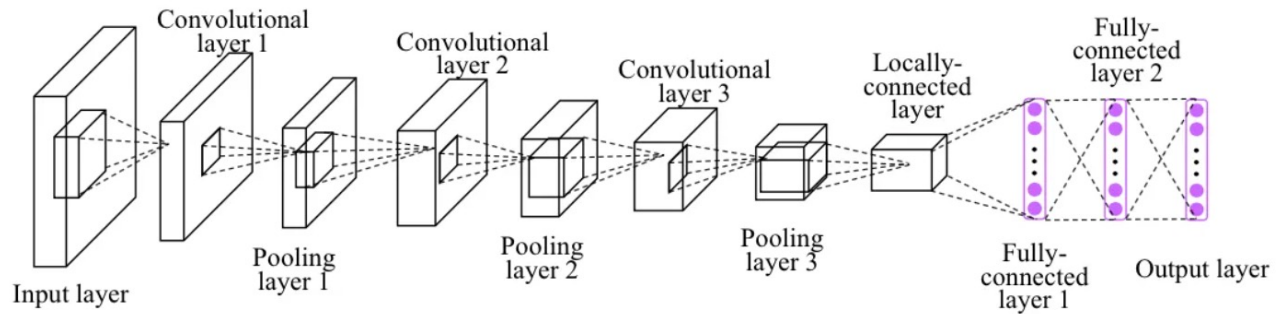
```
# With square kernels and equal stride
m = nn.Conv2d(3, 33, 3, stride=2)

# non-square kernels and unequal stride and with padding
m = nn.Conv2d(3, 33, (3, 4), stride=(3, 1), padding=(4, 2))

# non-square kernels and unequal stride and with padding and dilation
m = nn.Conv2d(3, 33, (3, 5), stride=(3, 1), padding=(4, 2), dilation=(3, 1))
```

<https://machinelearningknowledge.ai/pytorch-conv2d-explained-with-examples/>

Creating CNN Model



```
(layer1): Sequential(
  (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU()
  (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (3): Dropout(p=0, inplace=False)
)
(layer2): Sequential(
  (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU()
  (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (3): Dropout(p=0, inplace=False)
)
(layer3): Sequential(
  (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU()
  (2): MaxPool2d(kernel_size=2, stride=2, padding=1, dilation=1, ceil_mode=False)
  (3): Dropout(p=0, inplace=False)
)
(fc1): Linear(in_features=2048, out_features=625, bias=True)
(layer4): Sequential(
  (0): Linear(in_features=2048, out_features=625, bias=True)
  (1): ReLU()
  (2): Dropout(p=0, inplace=False)
)
(fc2): Linear(in_features=625, out_features=10, bias=True)
)
```

```
# Implementation of CNN/ConvNet Model
class CNN(torch.nn.Module):

    def __init__(self):
        super(CNN, self).__init__()
        # L1 ImgIn shape=(?, 28, 28, 1)
        # Conv -> (?, 28, 28, 32)
        # Pool -> (?, 14, 14, 32)
        self.layer1 = torch.nn.Sequential(
            torch.nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2),
            torch.nn.Dropout(p=1 - keep_prob))
        # L2 ImgIn shape=(?, 14, 14, 32)
        # Conv -> (?, 14, 14, 64)
        # Pool -> (?, 7, 7, 64)
        self.layer2 = torch.nn.Sequential(
            torch.nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2),
            torch.nn.Dropout(p=1 - keep_prob))
        # L3 ImgIn shape=(?, 7, 7, 64)
        # Conv -> (?, 7, 7, 128)
        # Pool -> (?, 4, 4, 128)
        self.layer3 = torch.nn.Sequential(
            torch.nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2, padding=1),
            torch.nn.Dropout(p=1 - keep_prob))

        # L4 FC 4x4x128 inputs -> 625 outputs
        self.fc1 = torch.nn.Linear(4 * 4 * 128, 625, bias=True)
        torch.nn.init.xavier_uniform(self.fc1.weight)
        self.layer4 = torch.nn.Sequential(
            self.fc1,
            torch.nn.ReLU(),
            torch.nn.Dropout(p=1 - keep_prob))
        # L5 Final FC 625 inputs -> 10 outputs
        self.fc2 = torch.nn.Linear(625, 10, bias=True)
        torch.nn.init.xavier_uniform_(self.fc2.weight) # initialize parameters

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)
        out = out.view(out.size(0), -1) # Flatten them for FC
        out = self.fc1(out)
        out = self.fc2(out)
        return out

#instantiate CNN model
model = CNN()
model
```