CNN

Computational Data Science,
Addis Ababa University

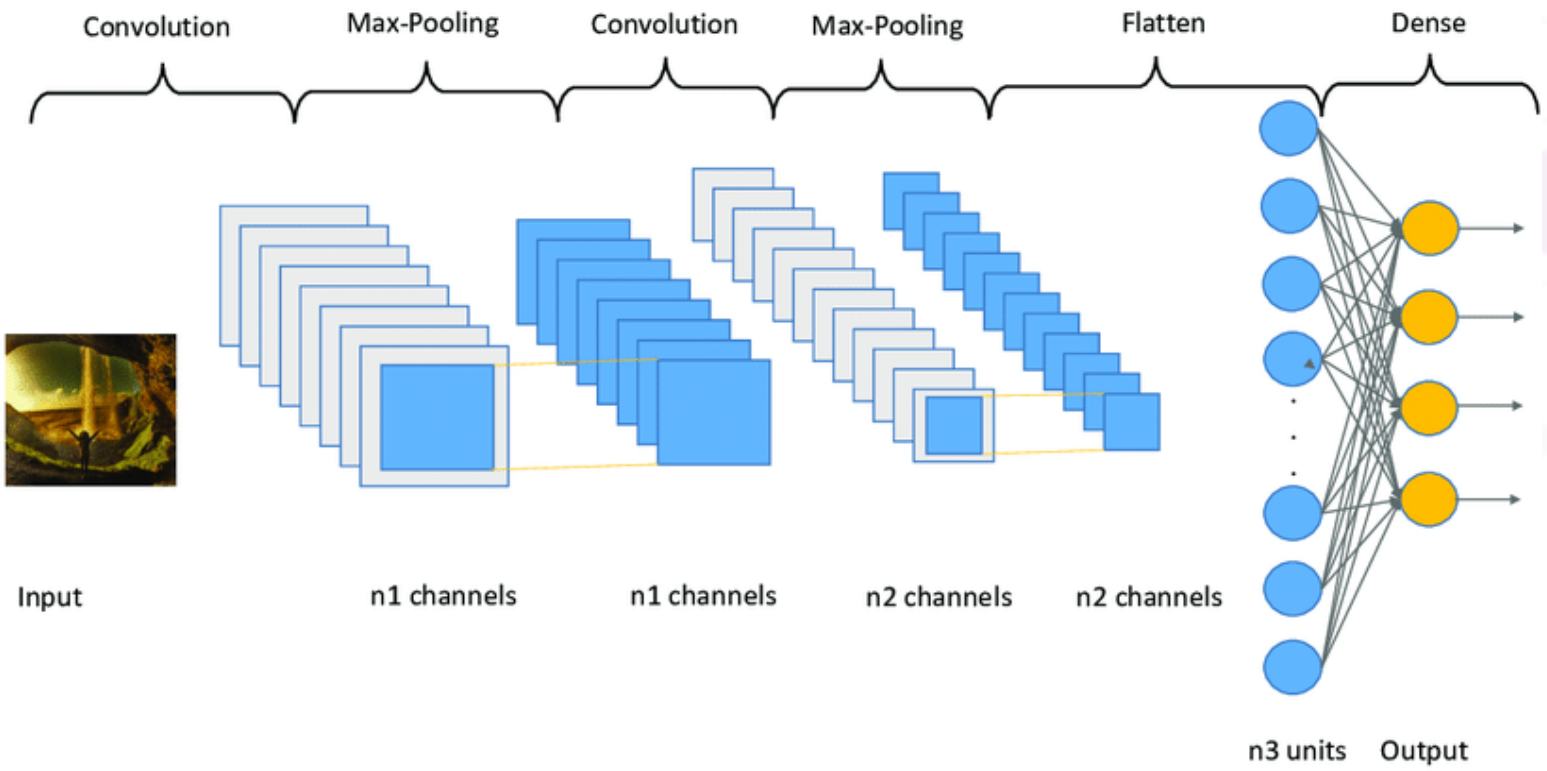
www.aau.edu.et

mesfin.diro@aau.edu.et

+251-912-086156

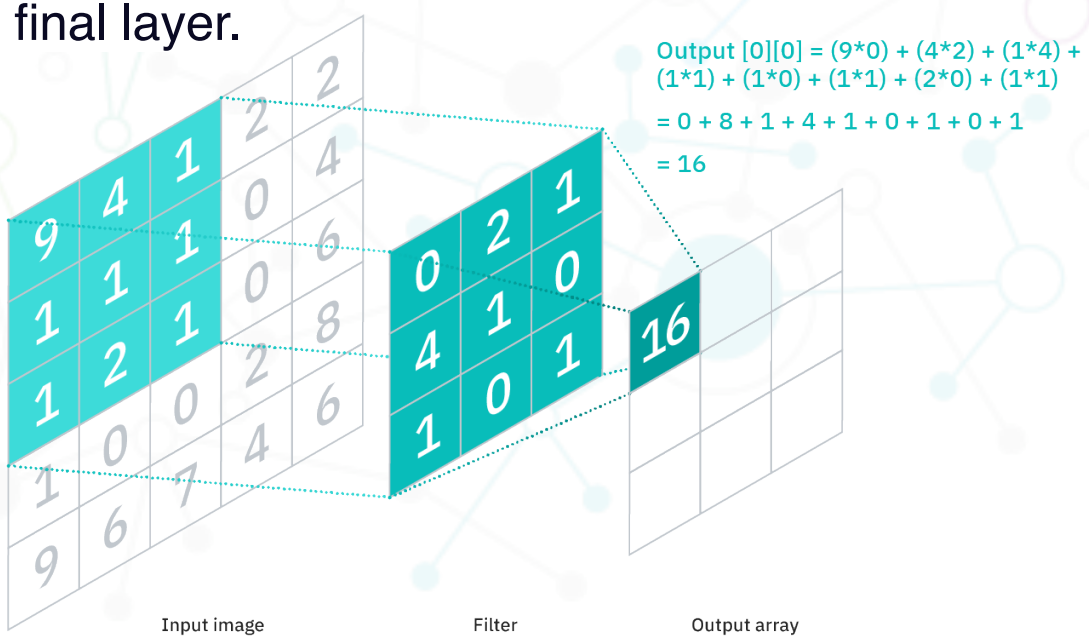
CNN

- A convolutional neural network is a specific kind of neural network with multiple layers.
- It processes data that has a grid-like arrangement then extracts important features.
- One huge advantage of using CNNs is that you don't need to do a lot of pre-processing on images.



CNN

- Convolutional neural networks are distinguished from other neural networks by their superior performance with image, speech, or audio signal inputs
- They have three main types of layers, which are:
  - Convolutional layer
  - Pooling layer
  - Fully-connected (FC) layer
- The convolutional layer is the first layer of a convolutional network.
- While convolutional layers can be followed by additional convolutional layers or pooling layers, the fully-connected layer is the final layer.



CNN

These are the network parameters to be learned.

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

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

Filter 1

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

Filter 2

Each filter detects a small pattern (3 x 3).

CNN

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

Filter 1

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

Dot product

3 -1

6 x 6 image

CNN

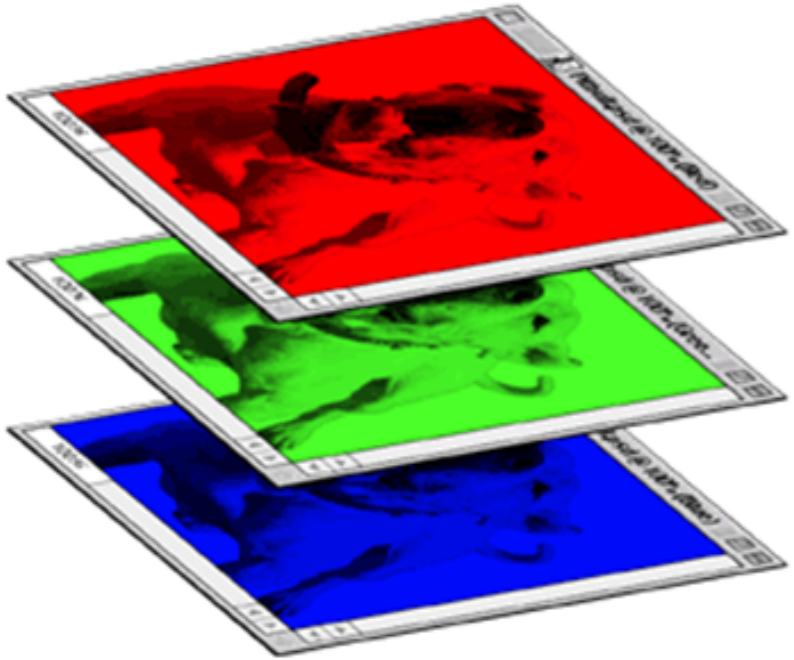
1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

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

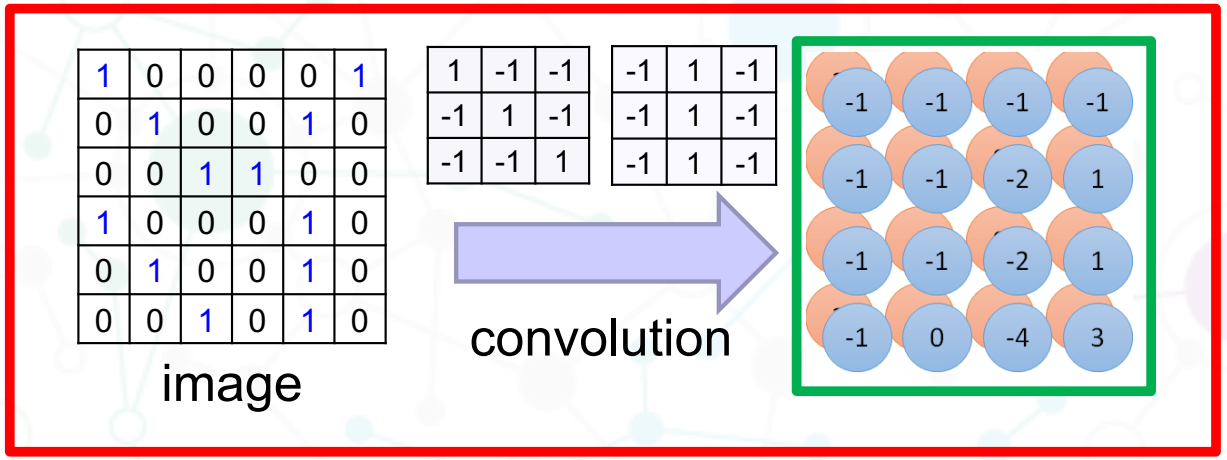
Filter 2

Color image

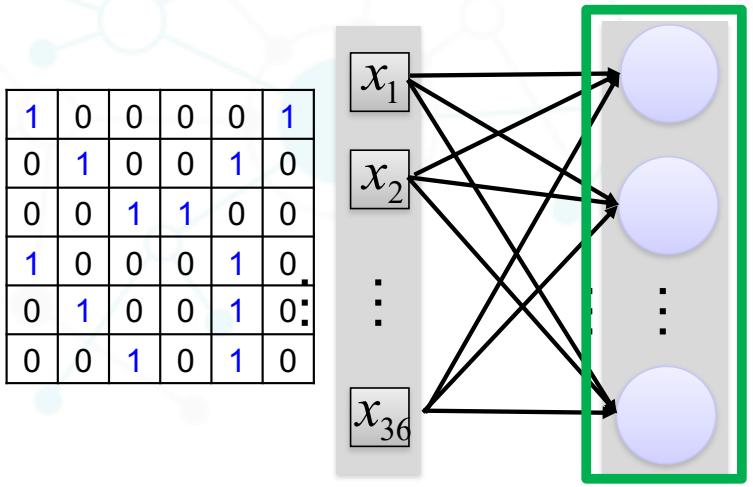


1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

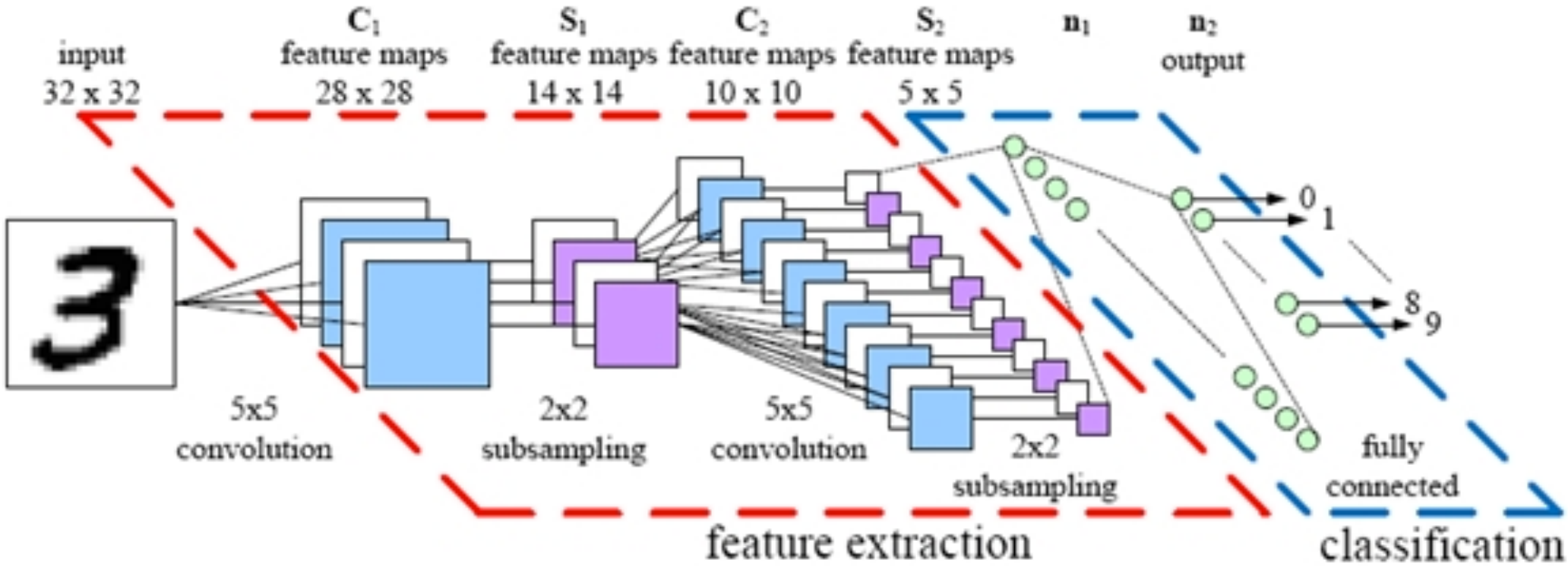
Convolution v.s. Fully Connected



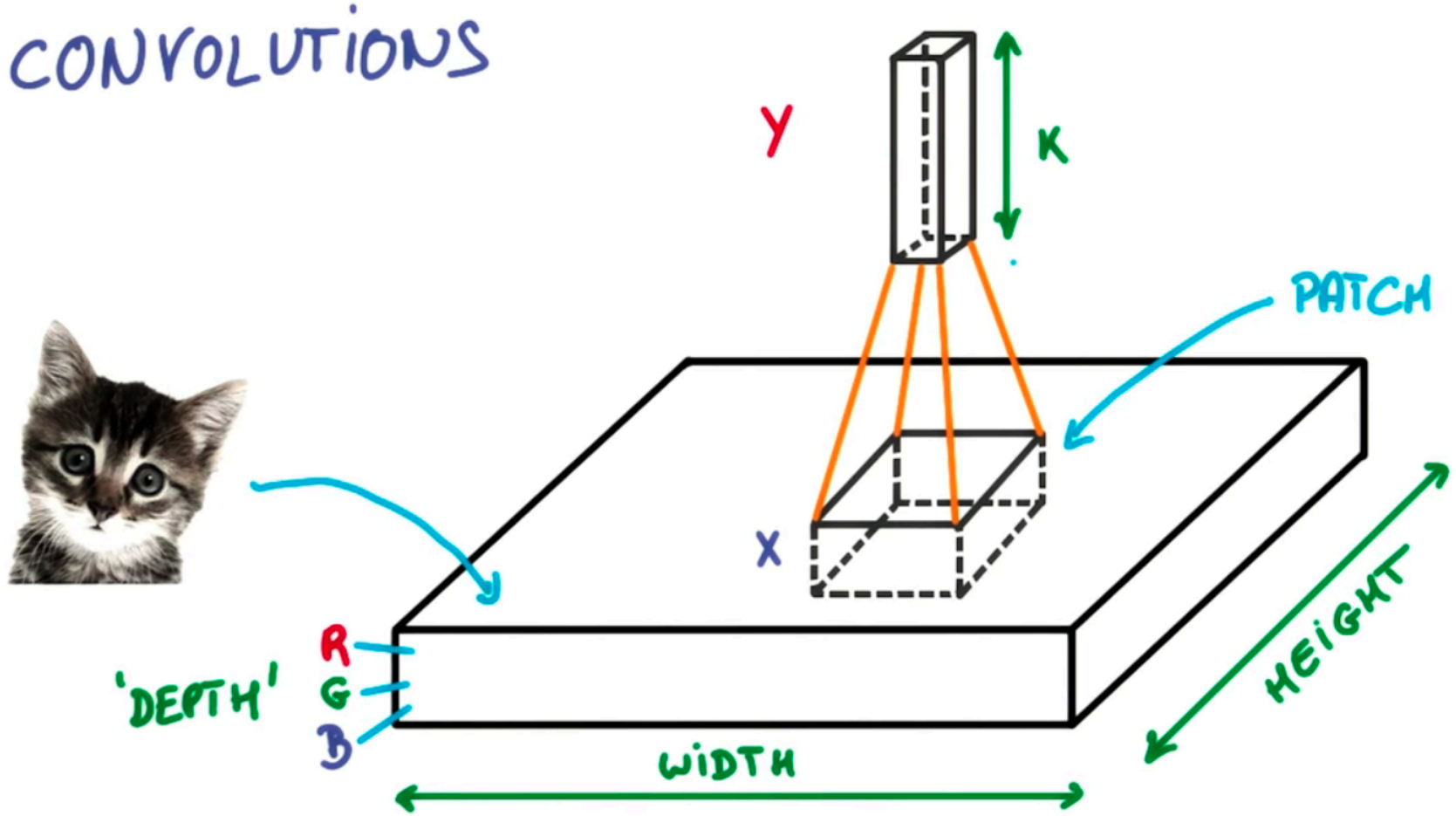
Fully-connected



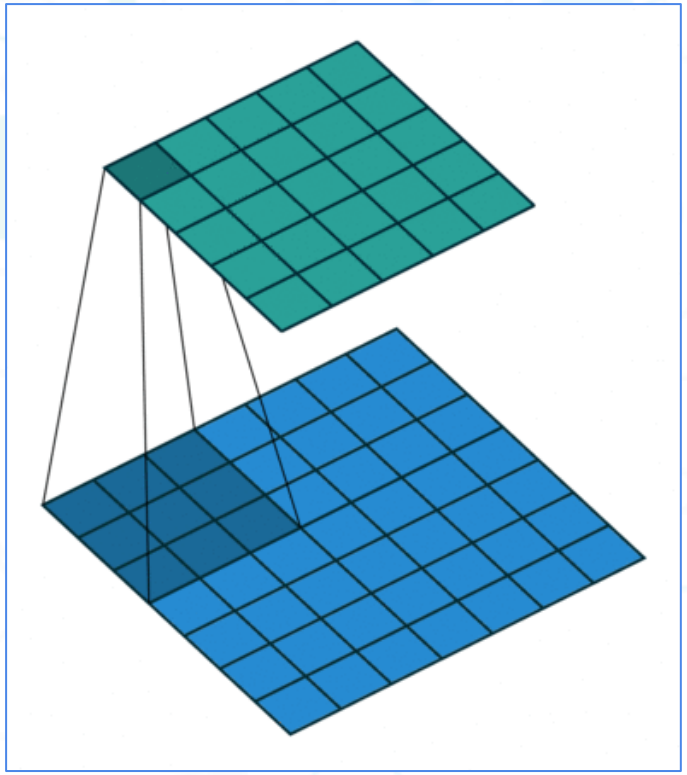
CNN


input
32 x 32
C1
feature maps
28 x 28
S1
feature maps
14 x 14
C2
feature maps
10 x 10
S2
feature maps
5 x 5
n1
n2
output
0
1
8
9
5x5
convolution
2x2
subsampling
5x5
convolution
2x2
subsampling
fully
connected
feature extraction
classification

CNN



CNN

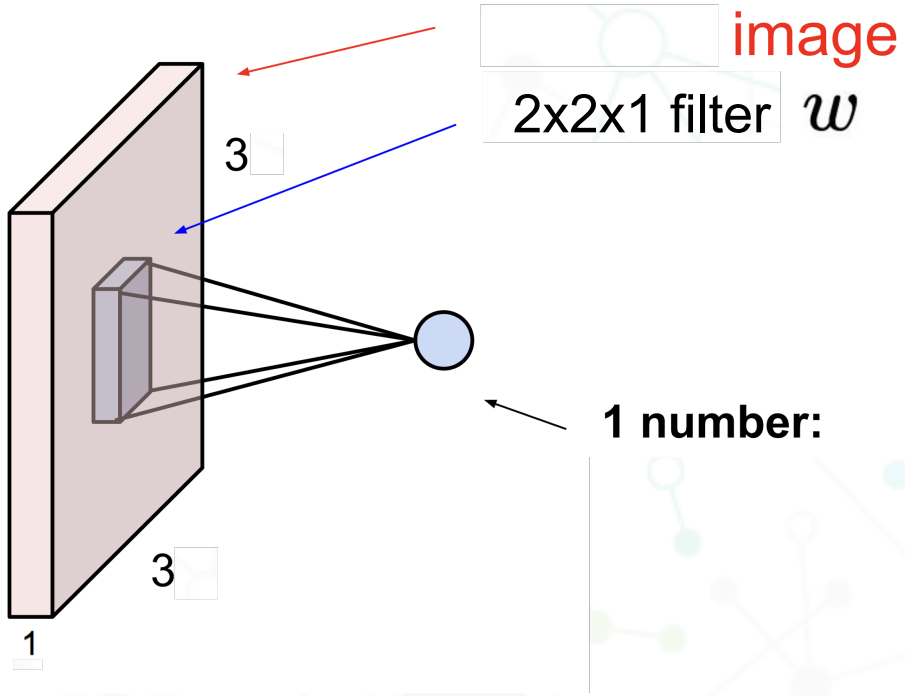


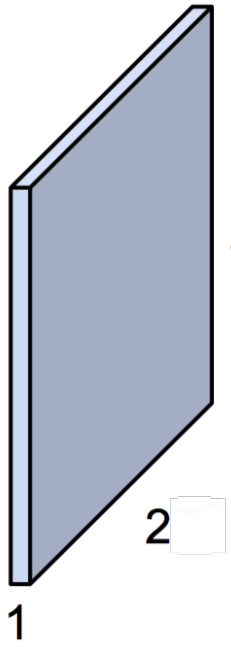
Padding

- If we have been convolving an input of 6 X 6 dimension with a 3 X 3 filter results in 4 X 4 output.
- We can generalize it and say that if the input is $n \times n$ and the filter size is $f \times f$, then the output size will be $(n-f+1) \times (n-f+1)$:
  - **Input:** $n \times n$
  - **Filter size:** $f \times f$
  - **Output:** $(n-f+1) \times (n-f+1)$
- There are primarily two disadvantages here:
  1. Every time we apply a convolutional operation, the size of the image shrinks
  2. Pixels present in the corner of the image are used only a few number of times during convolution as compared to the central pixels. Hence, we do not focus too much on the corners since that can lead to information loss
- To overcome these issues, we can pad the image with an additional border, i.e., we add one pixel all around the edges:
  - **Input:** $n \times n$
  - **Padding:** p
  - **Filter size:** $f \times f$
  - **Output:** $(n+2p-f+1) \times (n+2p-f+1)$

Strided Convolutions

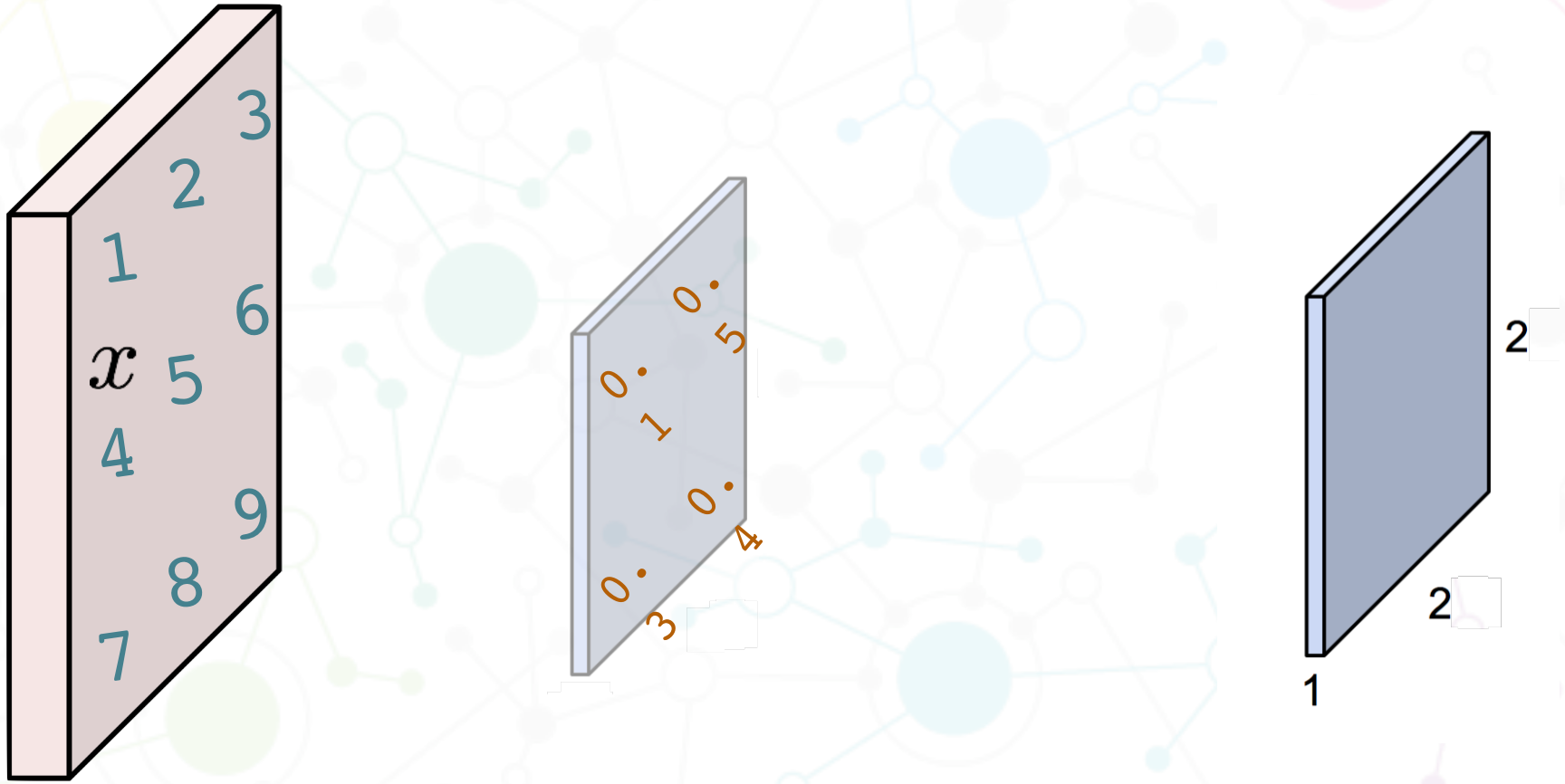
- Suppose we choose a stride of 2. So, while convoluting through the image: we will take two steps – both in the horizontal and vertical directions separately.
- The dimensions for stride s will be:
  - **Input:** $n \times n$
  - **Padding:** p
  - **Stride:** s
  - **Filter size:** $f \times f$
  - **Output:** $[(n + 2p - f)/s + 1] \times [(n + 2p - f)/s + 1]$
- Stride helps to reduce the size of the image, a particularly useful feature.





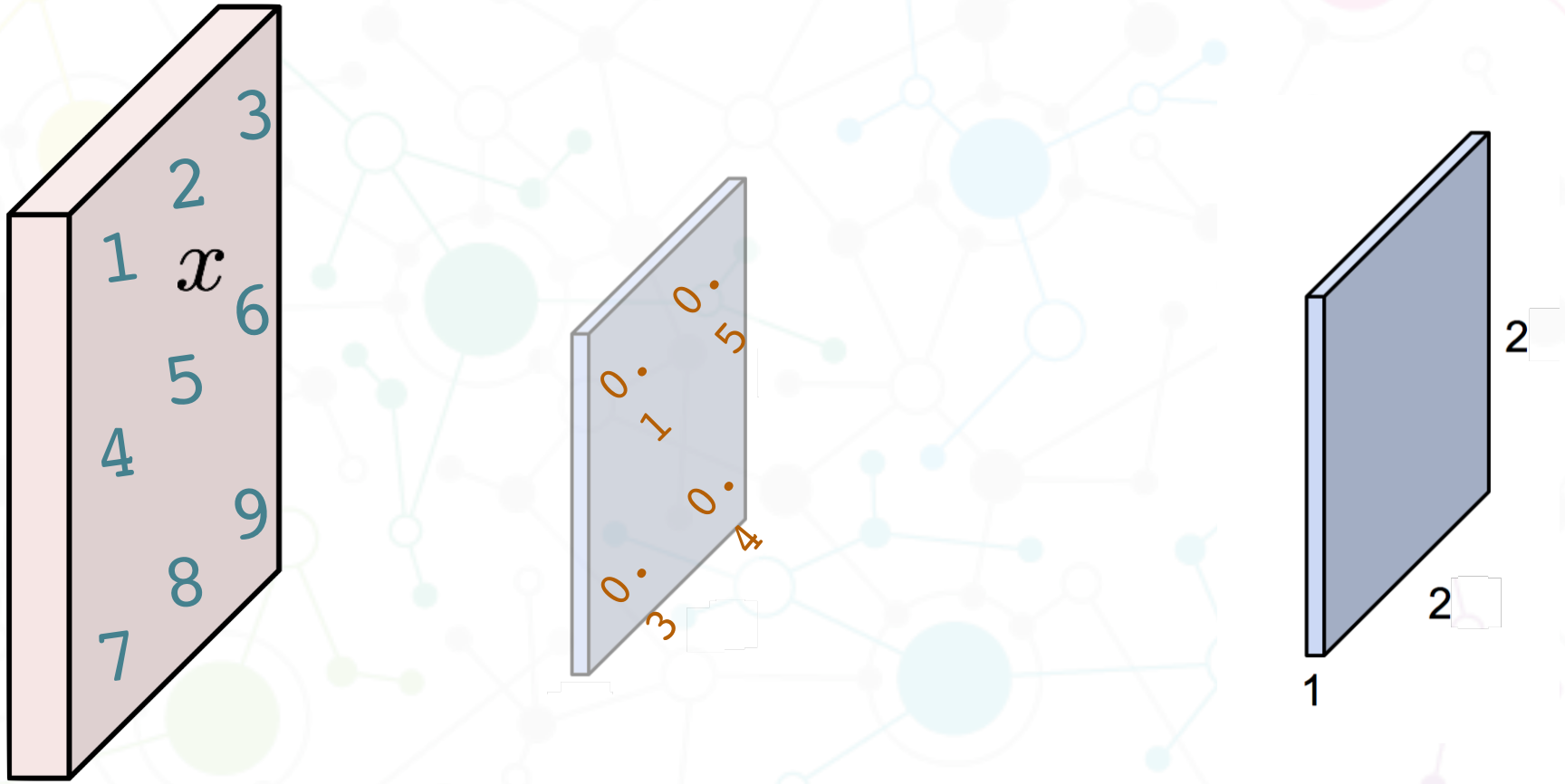
Simple convolution layer

`Stride: 1x1`



Simple convolution layer

Image: 1,3,3,1 image, Filter: 2,2,1,1, Stride: 1x1, No Padding

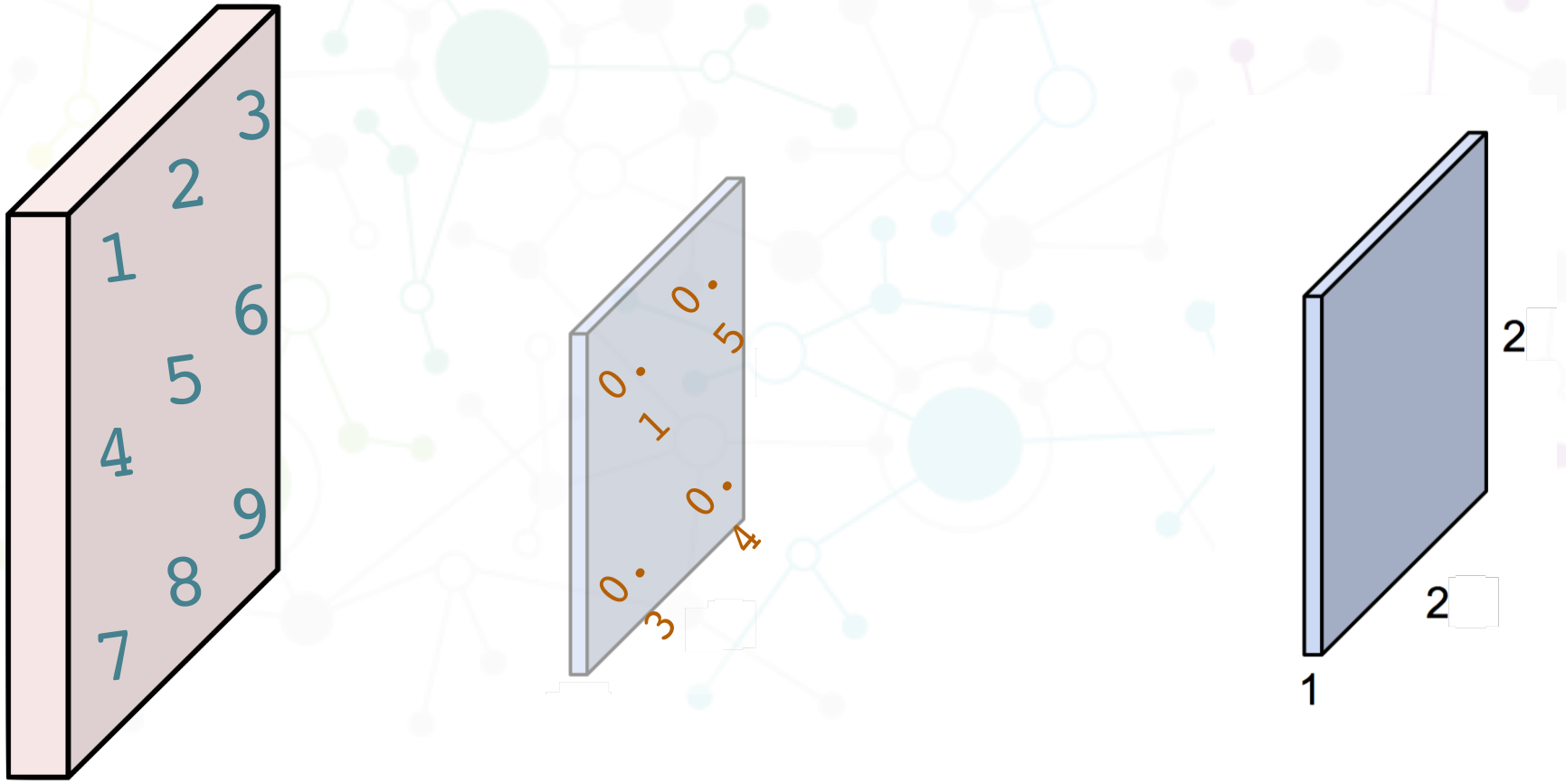


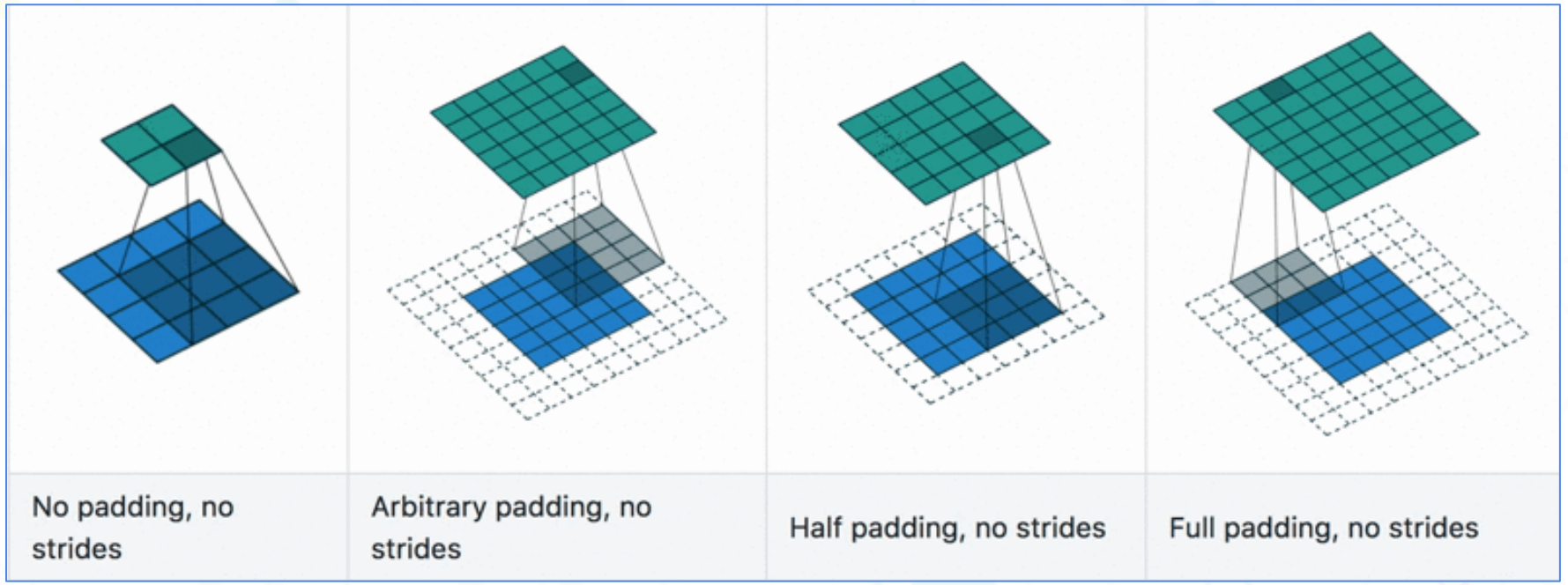
Simple convolution layer

Image: 1,3,3,1 image, Filter: 2,2,1,1, Stride: 1x1, No Padding

Simple convolution layer

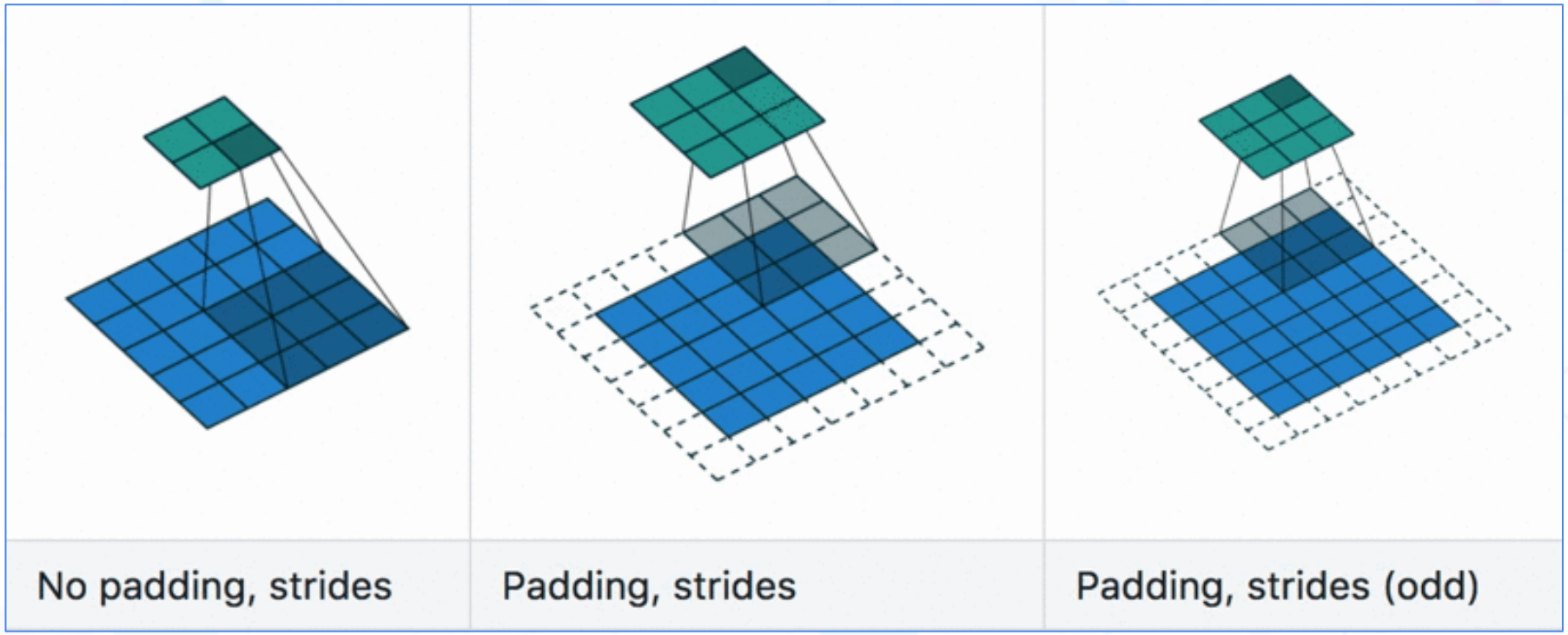
Image: 1,3,3,1 image, Filter: 2,2,1,1, Stride: 1x1, With padding





Convolution with padding in Action

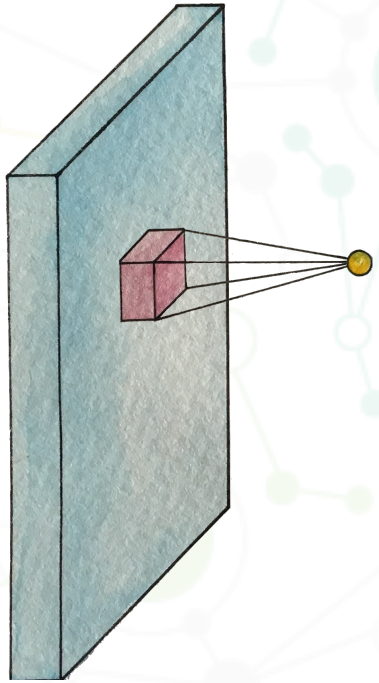
CNN



Max pooling

Pooling Layers

Pooling layers are generally used to reduce the size of the inputs and hence speed up the computation.



Single depth slice

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

max pool with 2x2 filters and stride 2 →

6	8
3	4

Why Pooling

- Subsampling pixels will not change the object

bird

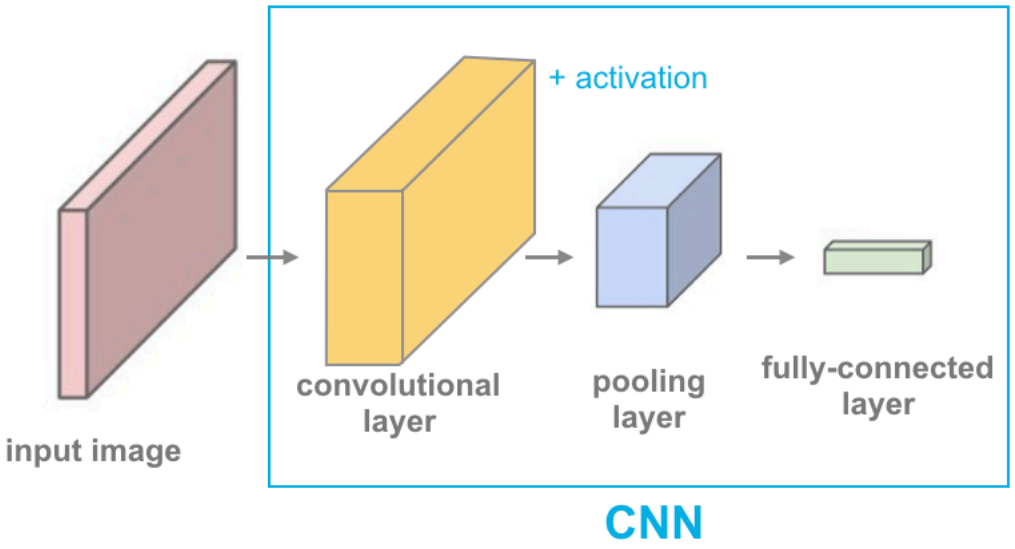
Subsampling

bird

We can subsample the pixels to make image smaller → fewer parameters to characterize the image

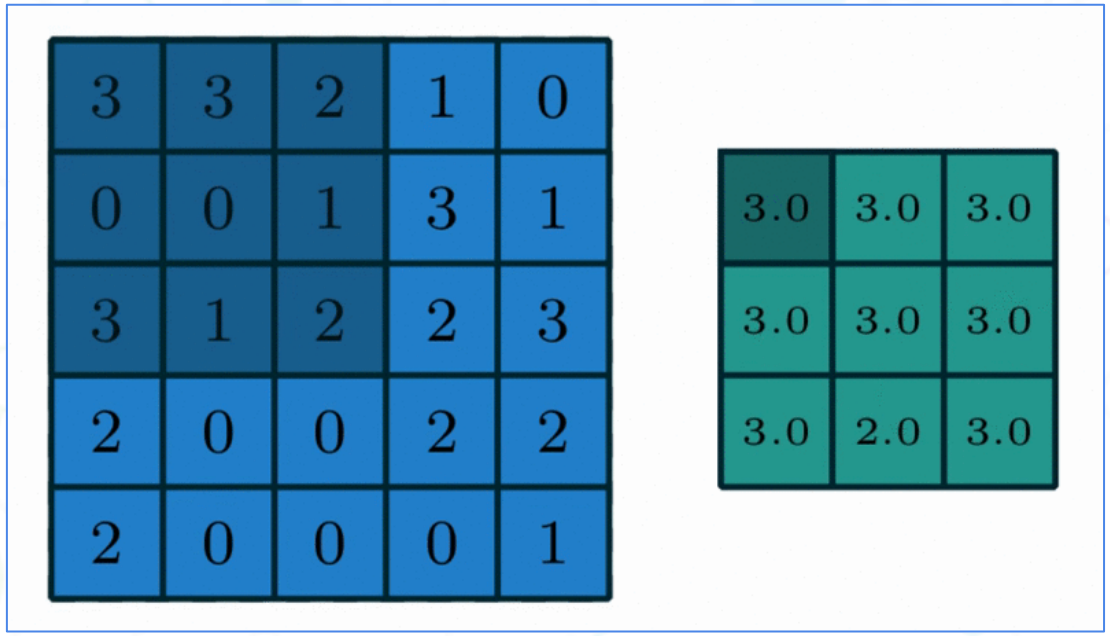
CNN

A convolutional layer + activation function, followed by a pooling layer, and a linear layer (to create a desired output size) make up the basic layers of a CNN.

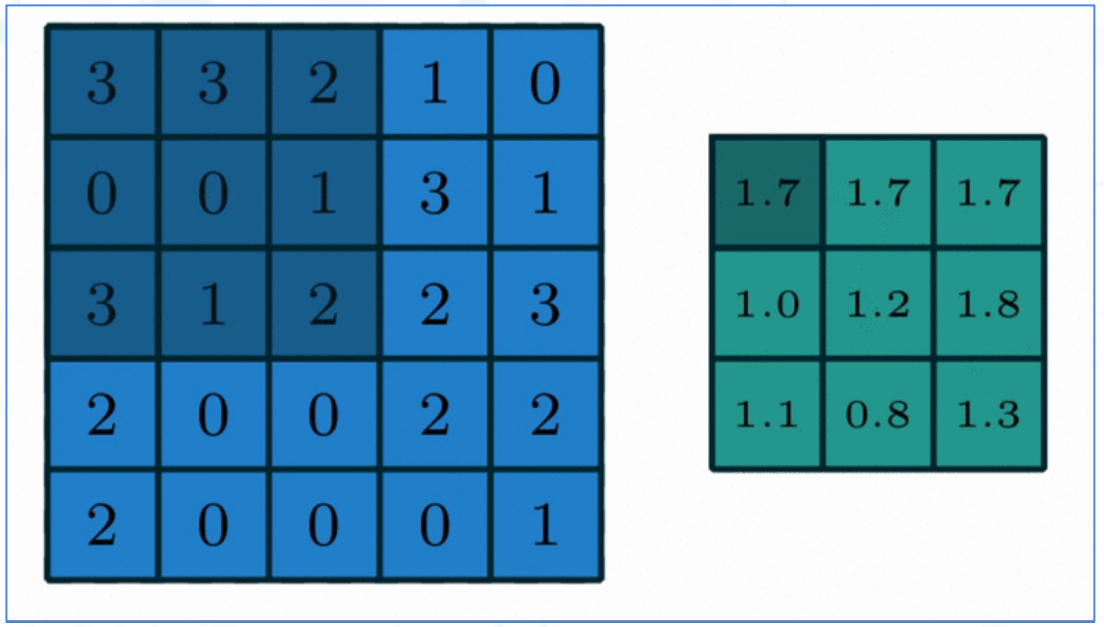


- Pooling layers, also known as downsampling, conducts dimensionality reduction, reducing the number of parameters in the input.
- There are two main types of pooling:
  - **Max pooling:** As the filter moves across the input, it selects the pixel with the maximum value to send to the output array.
  - **Average pooling:** As the filter moves across the input, it calculates the average value within the receptive field to send to the output array.
- Fully-connected layer performs the task of classification based on the features extraction through the previous layers and their different filters.
- While convolutional and pooling layers tend to use ReLu functions, FC layers usually leverage a softmax activation function to classify inputs appropriately, producing a probability from 0 to 1.

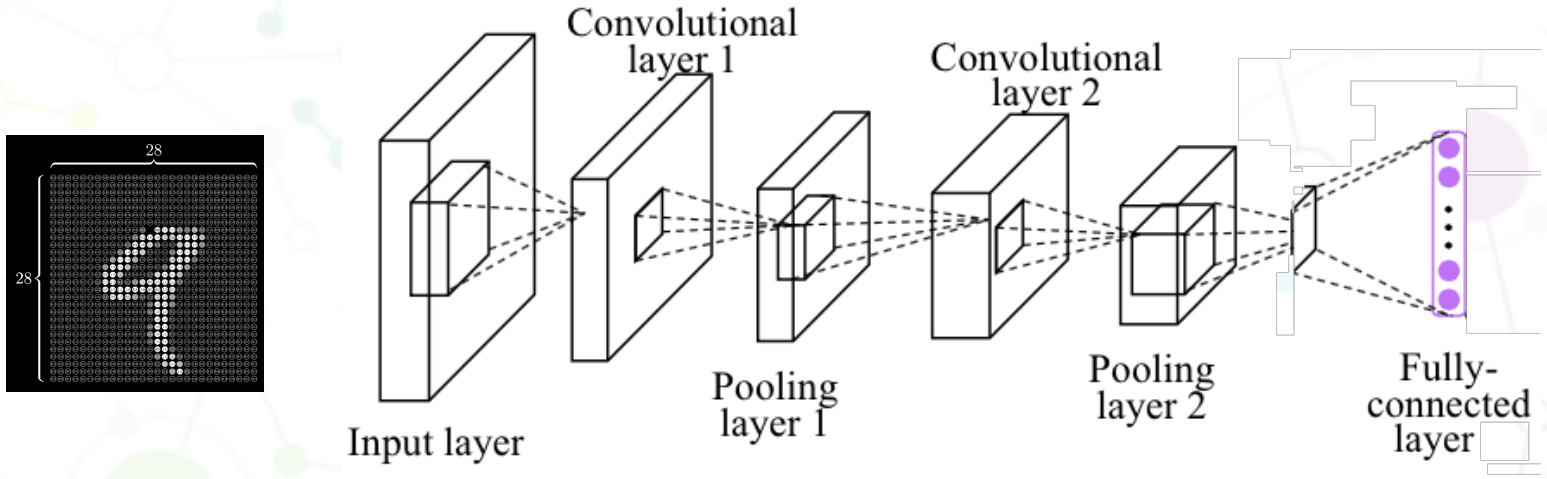
Max Pooling in Action



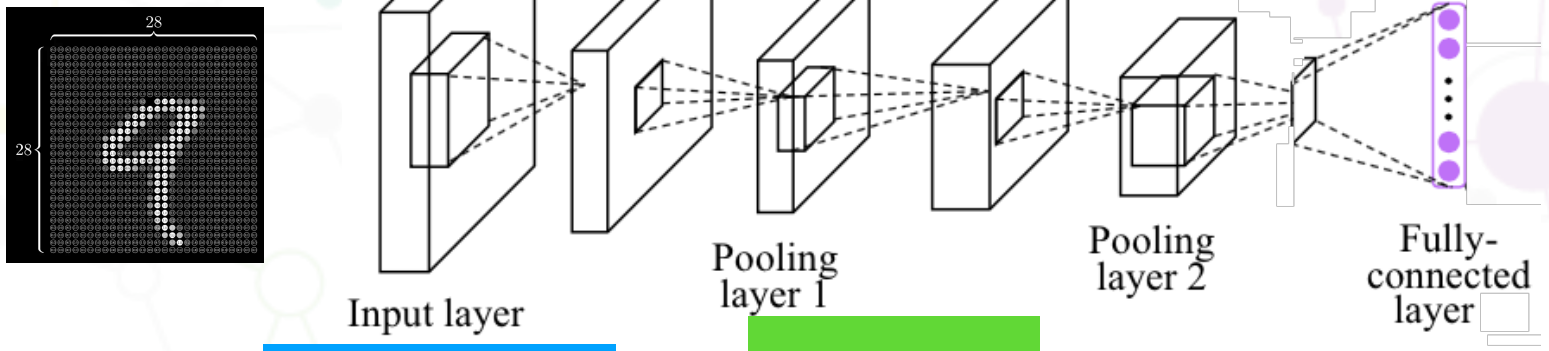
Avg Pooling in Action



Simple CNN


28
28
Convolutional layer 1
Convolutional layer 2
Input layer
Pooling layer 1
Pooling layer 2
Fully-connected layer

Simple CNN



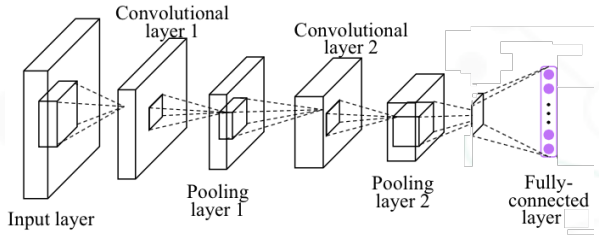

```
self.fc = nn.Linear(320, 10)
```

torch.nn.Conv2d (in_channels, out_channels, kernel_size)

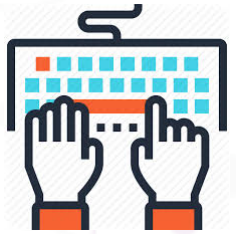
nn.MaxPool2d (kernel_size)

```
self.mp = nn.MaxPool2d(2)
```

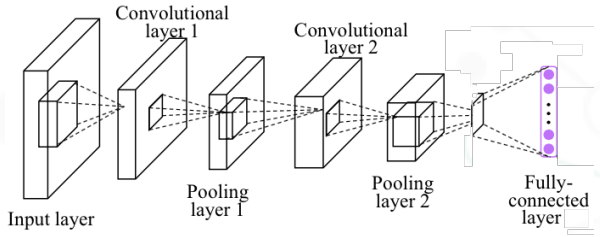
```
self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
```



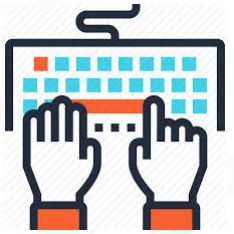
Simple CNN



```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.mp = nn.MaxPool2d(2)
        self.fc = nn.Linear(100???, 10) # ??? -> 10

    def forward(self, x):
        in_size = x.size(0)
        x = F.relu(self.mp(self.conv1(x)))
        x = F.relu(self.mp(self.conv2(x)))
        x = x.view(in_size, -1) # flatten the tensor
        x = self.fc(x)
        return F.log_softmax(x)
```

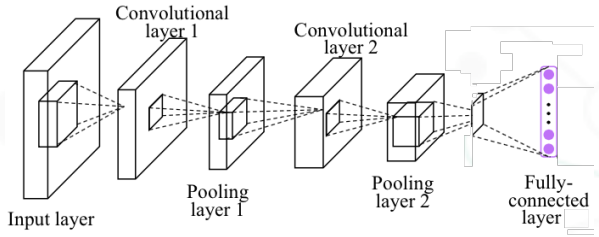


Simple CNN

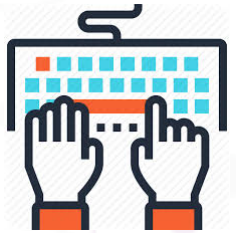


```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.mp = nn.MaxPool2d(2)
        self.fc = nn.Linear(100???, 10) # ??? -> 10

    def forward(self, x):
        in_size = x.size(0)
        x = F.relu(self.mp(self.conv1(x)))
        x = F.relu(self.mp(self.conv2(x)))
        x = x.view(in_size, -1) # flatten the tensor
        x = self.fc(x)
        return F.log_softmax(x)
```

RuntimeError: size mismatch, m1: [64 x **320**], m2: [100 x 10]

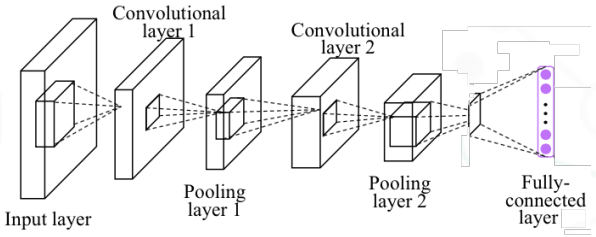


Simple CNN

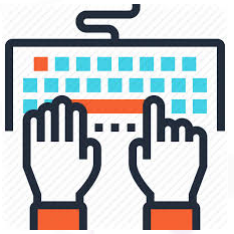


```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.mp = nn.MaxPool2d(2)
        self.fc = nn.Linear(320, 10) # ??? -> 10

    def forward(self, x):
        in_size = x.size(0)
        x = F.relu(self.mp(self.conv1(x)))
        x = F.relu(self.mp(self.conv2(x)))
        x = x.view(in_size, -1) # flatten the tensor
        x = self.fc(x)
        return F.log_softmax(x)
```

RuntimeError: size mismatch, m1: [64 x **320**], m2: [100 x 10]



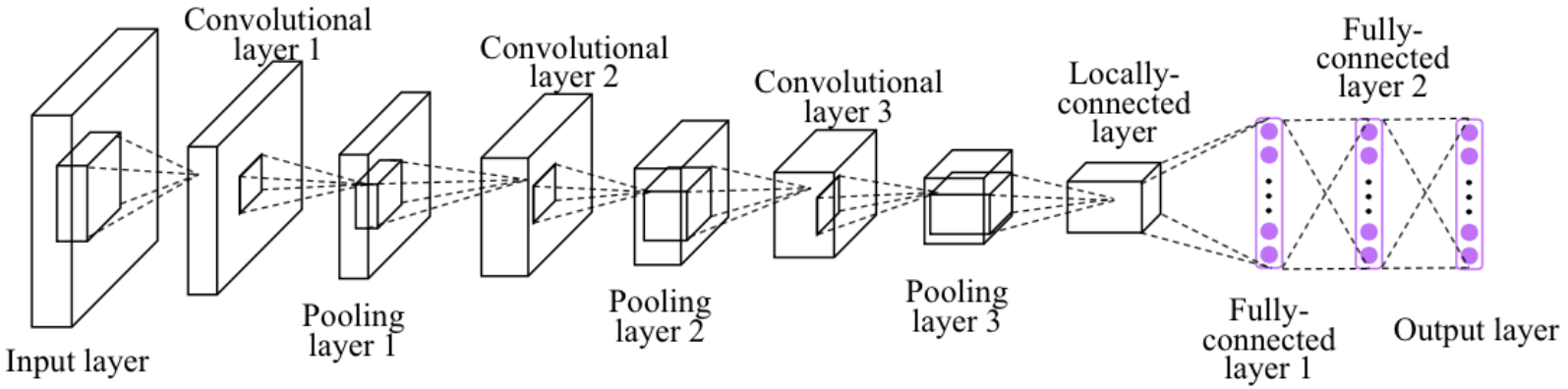
Simple CNN



```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.mp = nn.MaxPool2d(2)
        self.fc = nn.Linear(320, 10) # 320 -> 10

    def forward(self, x):
        in_size = x.size(0)
        x = F.relu(self.mp(self.conv1(x)))
        x = F.relu(self.mp(self.conv2(x)))
        x = x.view(in_size, -1) # flatten the tensor
        x = self.fc(x)
        return F.log_softmax(x)
```

```
Train Epoch: 9 [46080/60000 (77%)]      Loss: 0.108415
Train Epoch: 9 [46720/60000 (78%)]      Loss: 0.140700
Train Epoch: 9 [47360/60000 (79%)]      Loss: 0.090830
Train Epoch: 9 [48000/60000 (80%)]      Loss: 0.031640
Train Epoch: 9 [48640/60000 (81%)]      Loss: 0.014934
Train Epoch: 9 [49280/60000 (82%)]      Loss: 0.090210
Train Epoch: 9 [49920/60000 (83%)]      Loss: 0.074975
Train Epoch: 9 [50560/60000 (84%)]      Loss: 0.058671
Train Epoch: 9 [51200/60000 (85%)]      Loss: 0.023464
Train Epoch: 9 [51840/60000 (86%)]      Loss: 0.018025
Train Epoch: 9 [52480/60000 (87%)]      Loss: 0.098865
Train Epoch: 9 [53120/60000 (88%)]      Loss: 0.013985
Train Epoch: 9 [53760/60000 (90%)]      Loss: 0.070476
Train Epoch: 9 [54400/60000 (91%)]      Loss: 0.065411
Train Epoch: 9 [55040/60000 (92%)]      Loss: 0.028783
Train Epoch: 9 [55680/60000 (93%)]      Loss: 0.008333
Train Epoch: 9 [56320/60000 (94%)]      Loss: 0.020412
Train Epoch: 9 [56960/60000 (95%)]      Loss: 0.036749
Train Epoch: 9 [57600/60000 (96%)]      Loss: 0.163087
Train Epoch: 9 [58240/60000 (97%)]      Loss: 0.117539
Train Epoch: 9 [58880/60000 (98%)]      Loss: 0.032256
Train Epoch: 9 [59520/60000 (99%)]      Loss: 0.026360

Test set: Average loss: 0.0483, Accuracy: 9846/10000 (98%)
```

Exercise 10-1: Implement CNN more layers



THANK YOU!