L7 Convolutional Neural Networks

Neural network architectures

- There are many different neural network architectures
- The architecture is defined by the type of layers we implement and how layers are connected together.
- The neural network shown is known as a feed-forward network (also known as a multilayer perceptron)
- Here we have a series of fully-connected layers

Feed-forward network

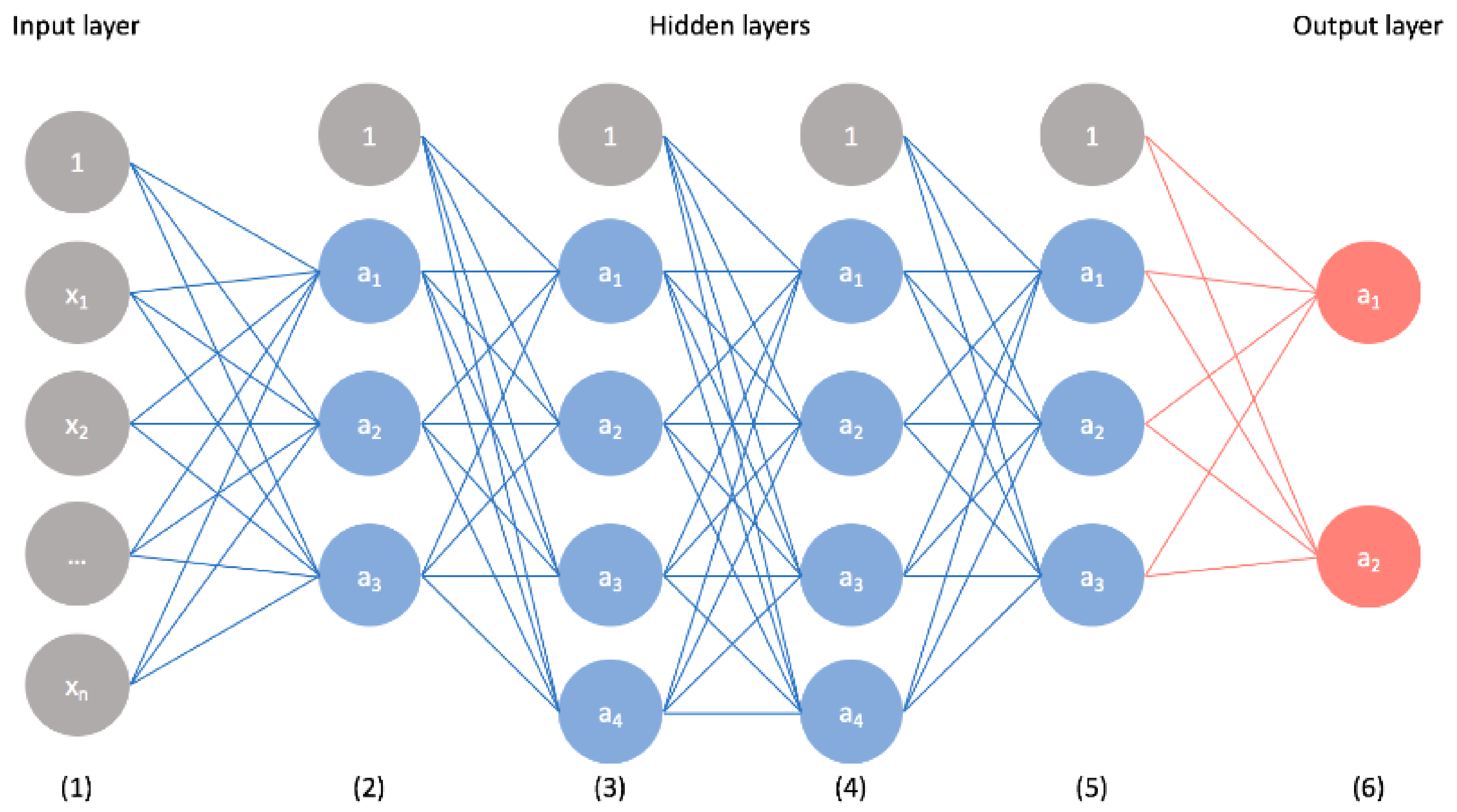


Image recognition

- To build a neural network that could recognize handwritten digits
- Ex: given the following 4 by 4 pixel image as input, our neural network should classify it as a "1“

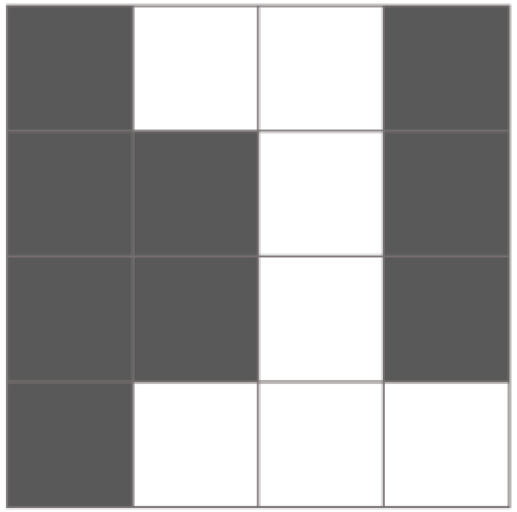
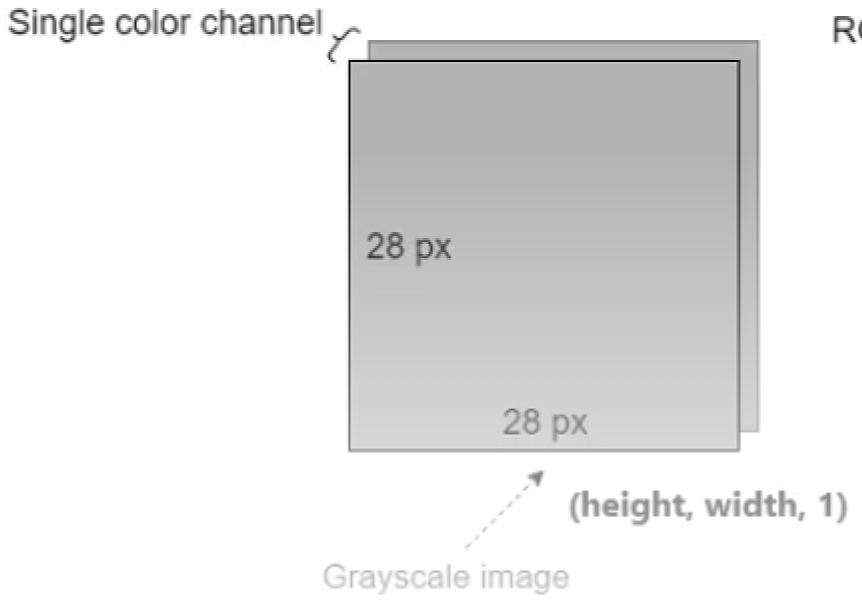
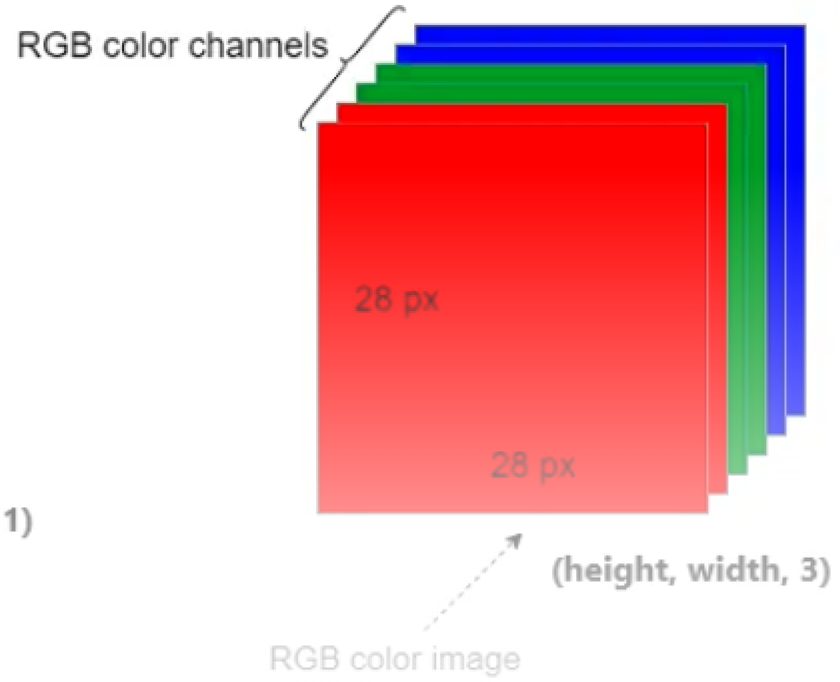


Image recognition

- Convolutional neural networks are used heavily in image recognition applications of machine learning
- Images are a matrix of values corresponding with the intensity of light
- white for highest intensity, black for lowest intensity at each pixel value.
- Grayscale images have a single value for each pixel
- while color images are typically represented by light intensity values for red, green, and blue at each pixel value.
- Thus, a 400 by 400 pixel image has the dimensions (400*400*1) for a grayscale image
- and for a color image (400*400*3)

Grayscale vs RGB images





Grayscale vs RGB images

- An image consists of pixels and images are represented as arrays of pixel values.

- There is only one color channel in a grayscale image. So, a grayscale image is represented as (1, height, width) or simply (height, width).
- We can ignore the third dimension because it is one. Therefore, a grayscale image is often represented as a 2D array (tensor).

- There are three color channels (Red, Green and Blue) in an RGB image.
- So, an RGB image is represented as (3, height, width) as a 3D array (tensor).

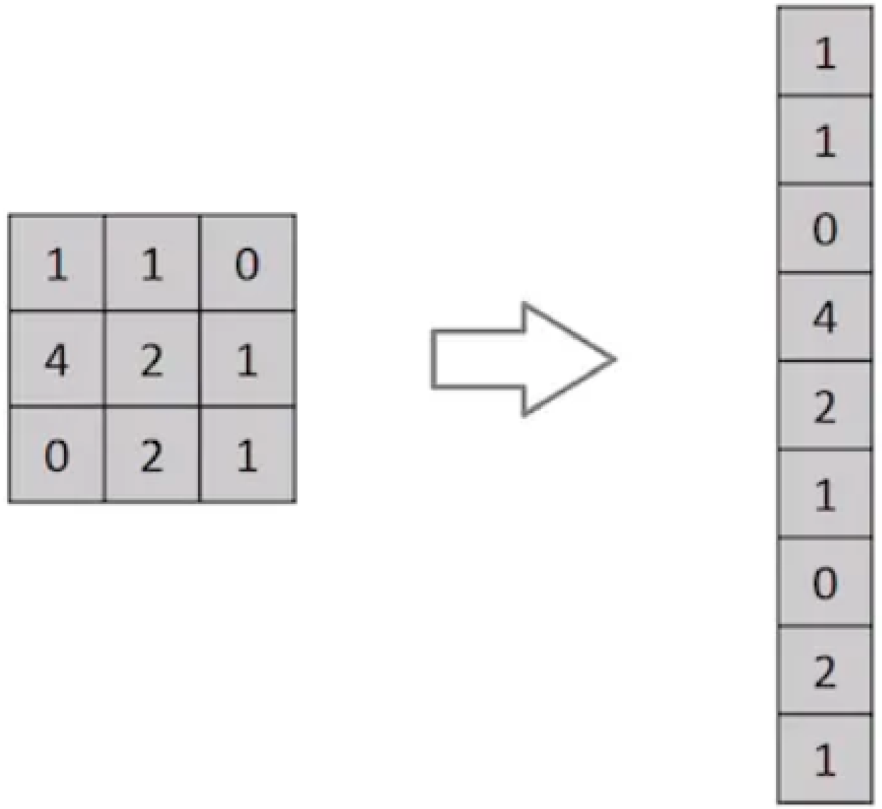
Weakness of feedforward networks

1. Cannot directly feed this 2D image into the neural network.

- A feed-forward network takes a vector of inputs, so we must flatten our 2D array of pixel values into a vector
- But we lose a great deal of information about the picture when we convert the 2D array of pixel values into a vector
- Specifically, we lose the spatial relationships within the data.
- Identifying the number in a flattened vector form without rearranging the pixels back into a 2D array is difficult
- A convolutional neural network (CNN for short) is a special type of neural network model primarily designed to process 2D image data, but which can also be used with 1D and 3D data.
- In cases of extremely basic binary images, the method might show an average precision score while performing prediction of classes but would have little to no accuracy when it comes to complex images having pixel dependencies throughout.

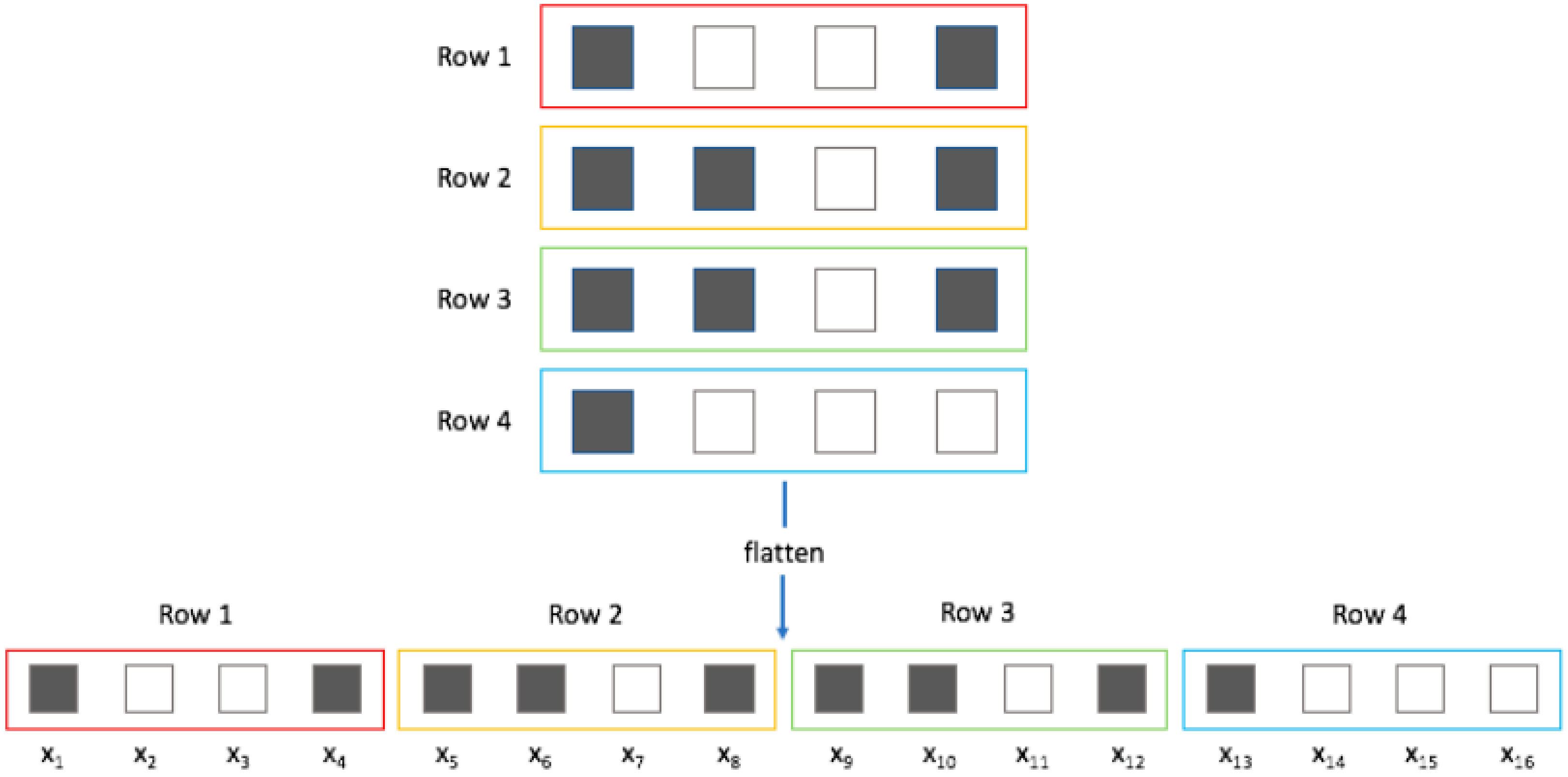
2. CNNs can reduce the number of parameters in the network significantly. So, CNNs are parameter efficient.

Flattening 2D image into a vector



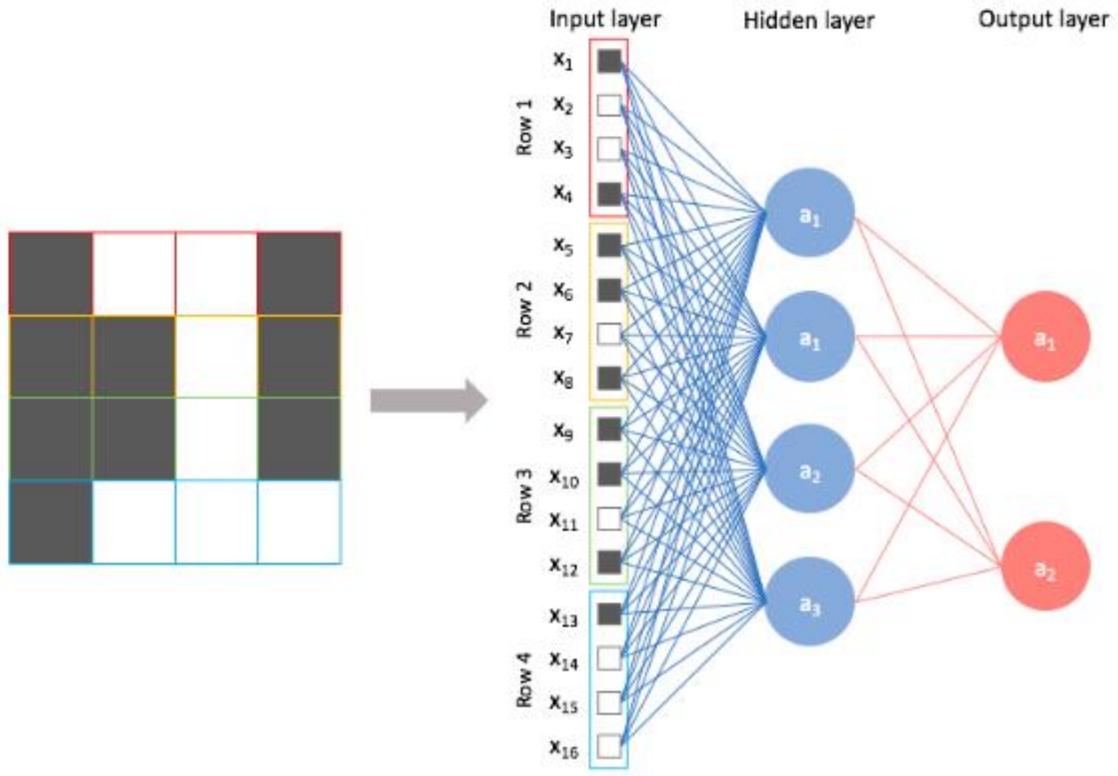
Flattening of a 3×3 image matrix into a 9×1 vector

Input to feed-forward network



Input to feed-forward network

- Now, we can treat each individual pixel value as a feature and feed it into the neural network.



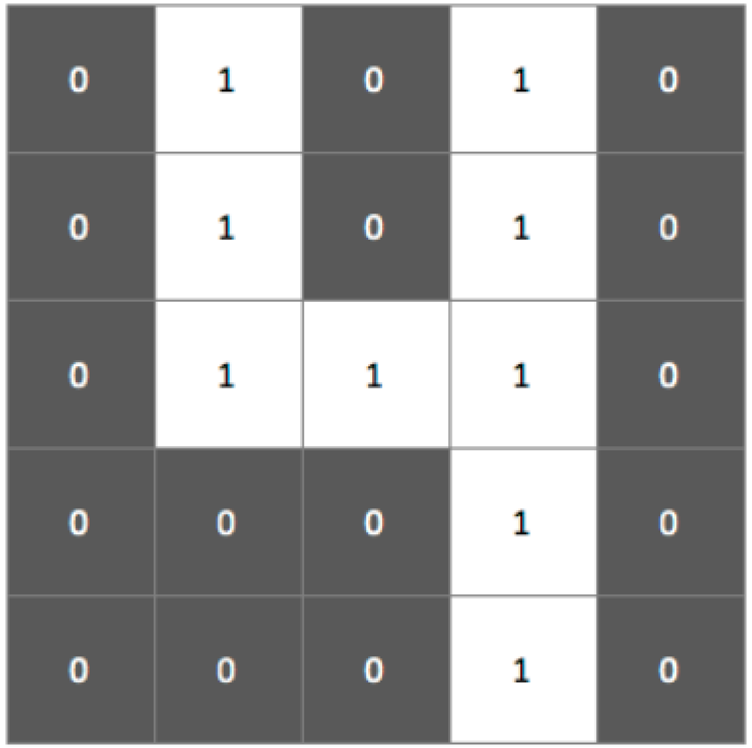
Convolution layers

- A convolution layer defines a window
- By which we examine a subset of the image, and
- Subsequently scans the entire image looking through this window.
- We can parameterize the window to look for specific features (e.g. edges) within an image.
- This window is also called a filter, since it produces an output image which focuses solely on the regions of the image which exhibited the feature it was searching for.
- The output of a convolution is referred to as a feature map
- Note: Windows, filters, and kernels all used interchangeably with respect to convolutional neural networks

Convolution layers

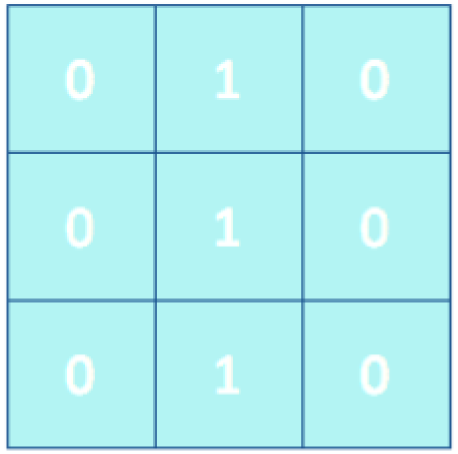
Ex: the following window searches for vertical lines in the image

5 by 5 pixel image



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

3 by 3 window

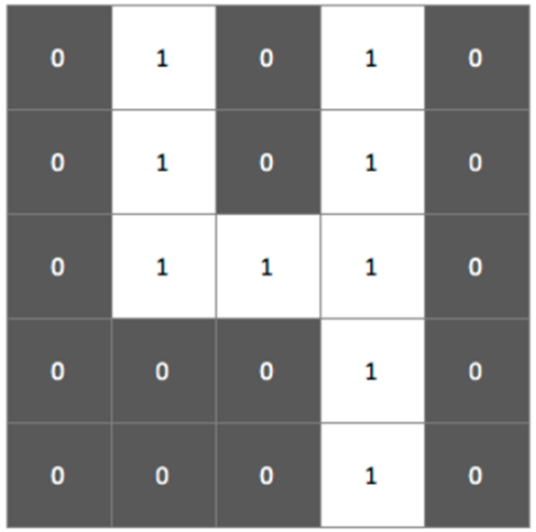


0	1	0
0	1	0
0	1	0

Convolution layers

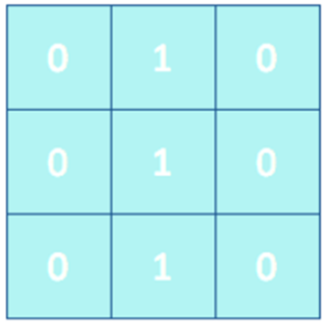
- Overlay this filter onto the image and linearly combine the pixel and filter values.
- Ex: (0*0)+(1*1)+(0*0)+(0*0)+(1*1)+(0*0)+(0*0)+(1*1)+(0*1) =3
- The output, shown on the right, identifies regions of the image in which a vertical line was present.
- We could do a similar process using a filter designed to find horizontal edges to properly characterize all of the features of a "4"

5 by 5 pixel image



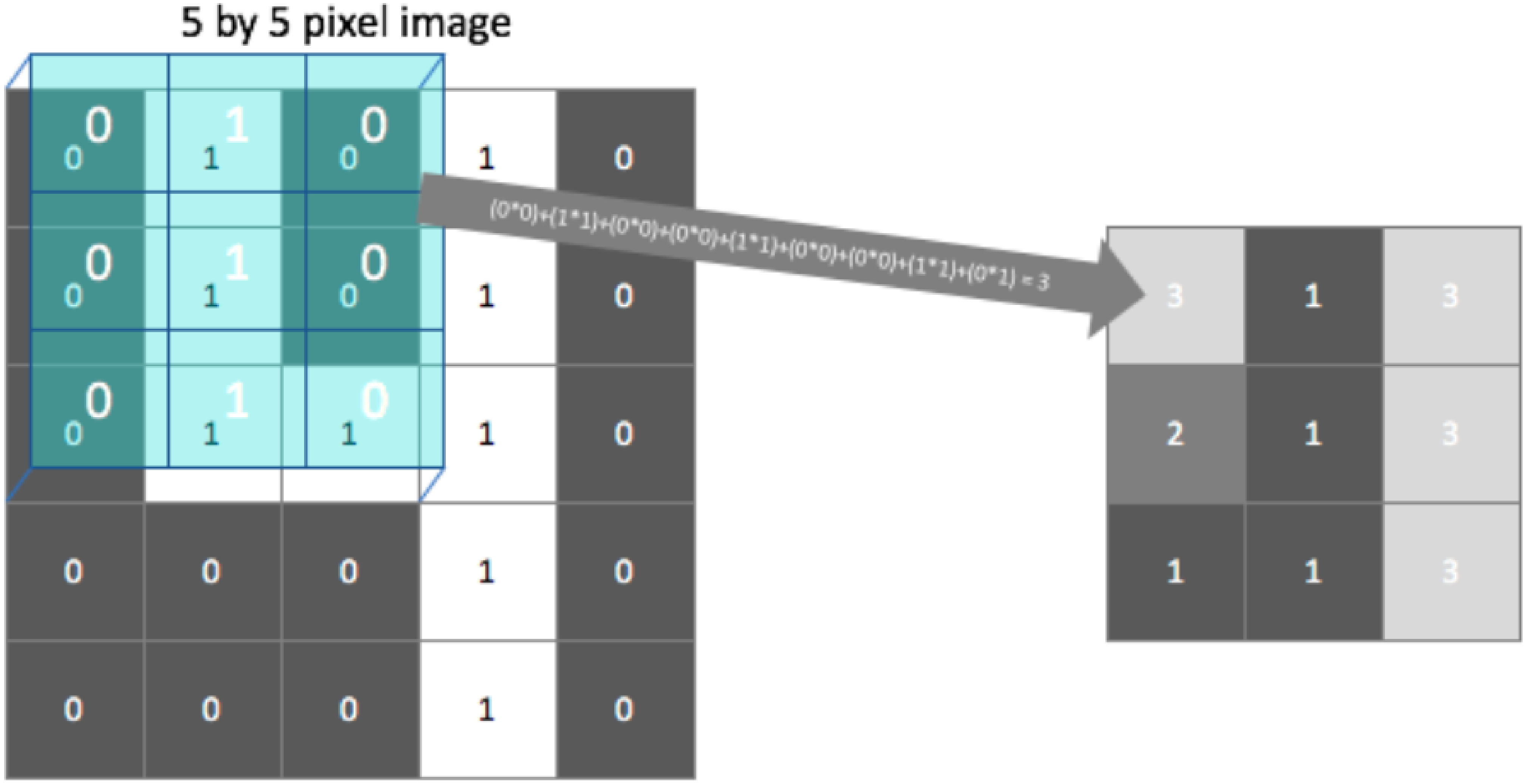
0	1	0	1	0
0	1	0	1	0
0	1	1	1	0
0	0	0	1	0
0	0	0	1	0

3 by 3 window



0	1	0
0	1	0
0	1	0

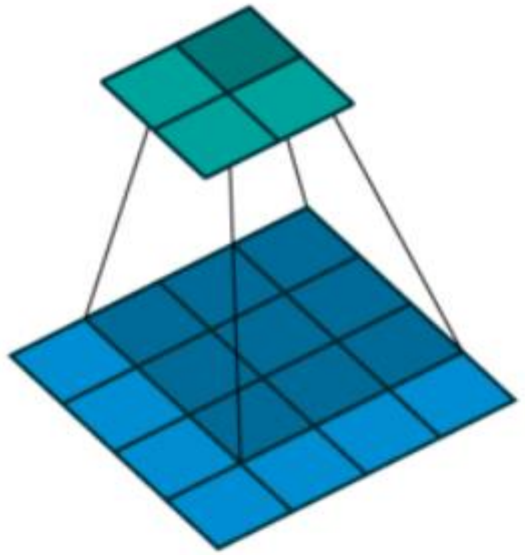
Convolution layers



(0*0)+(1*1)+(0*0)+(0*0)+(1*1)+(0*0)+(0*0)+(1*1)+(0*1) =3

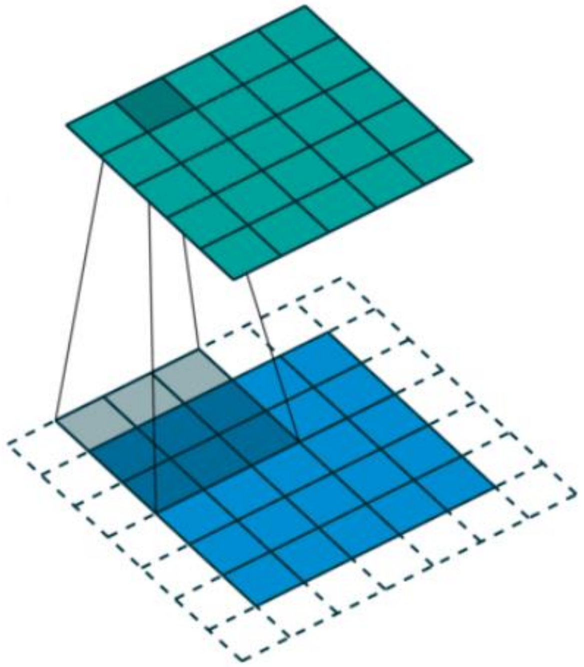
Convolution layers

- Scan each filter across the image calculating the linear combination at each step.
- In the figure, an input image is represented in blue and the convolved image is represented in green.
- We can imagine each pixel in the convolved layer as a neuron which takes all of the pixel values currently in the window as inputs, linearly combined with the corresponding weights in our filter.



Convolution layers

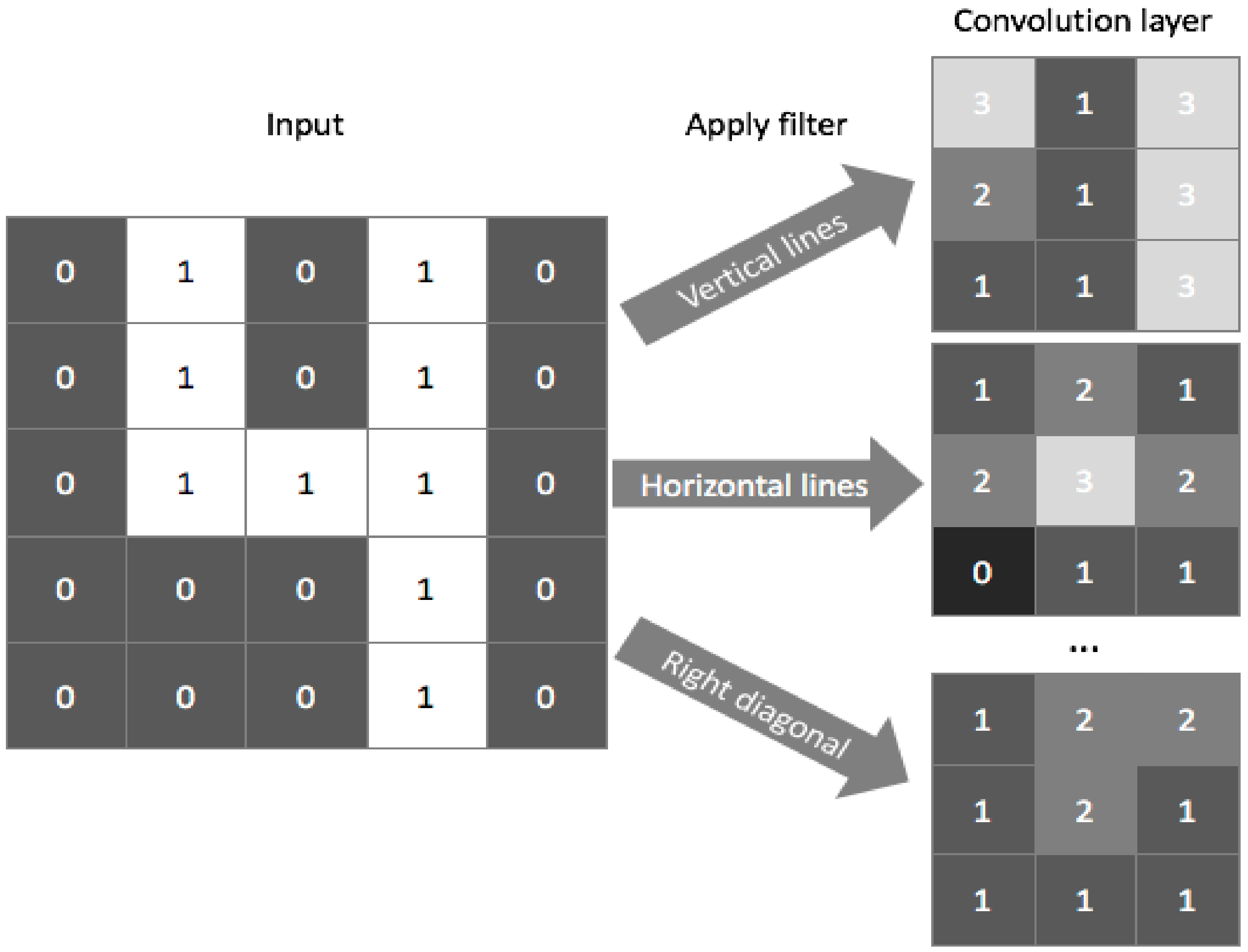
- We can also pad the edges of your images with 0-valued pixels as to fully scan the original image and preserve its complete dimensions



Multiple filters

- Note: Although we have hard-coded the filter values for this example, the filter values in a convolutional layer are learned
- A typical convolutional layer will apply multiple filters, each of which output a feature mapping of the input signifying the (spatial) locations where a feature is present.
- Notice how some feature mappings are more useful than others
- in the example next, the "right diagonal" feature mapping, which searches for right diagonals in the image, is essentially a dark image which signifies that the feature is not present

Multiple filters



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

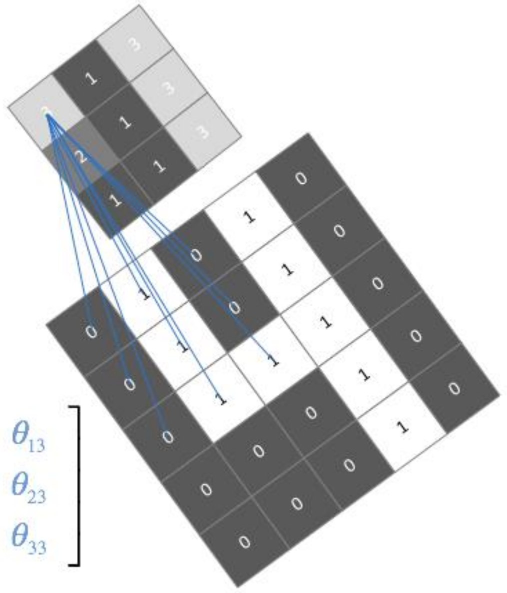
3	1	3
2	1	3
1	1	3

1	2	1
2	3	2
0	1	1

1	2	2
1	2	1
1	1	1

Multiple Filters

- In practice, we do not explicitly define the filters that our convolutional layer will use;
- we instead parameterize the filters and let the network learn the best filters to use during training.
- We do define how many filters we will use at each layer



$$\begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{21} & \theta_{22} & \theta_{23} \\ \theta_{31} & \theta_{32} & \theta_{33} \end{bmatrix}$$

Multiple Filters

- We can stack layers of convolutions together (ie. perform convolutions on convolutions) to learn more intricate patterns within the features mapped in the previous layer.
- This allows our neural network to identify general patterns in early layers, then focus on the patterns within patterns in later layers.

Pooling layers

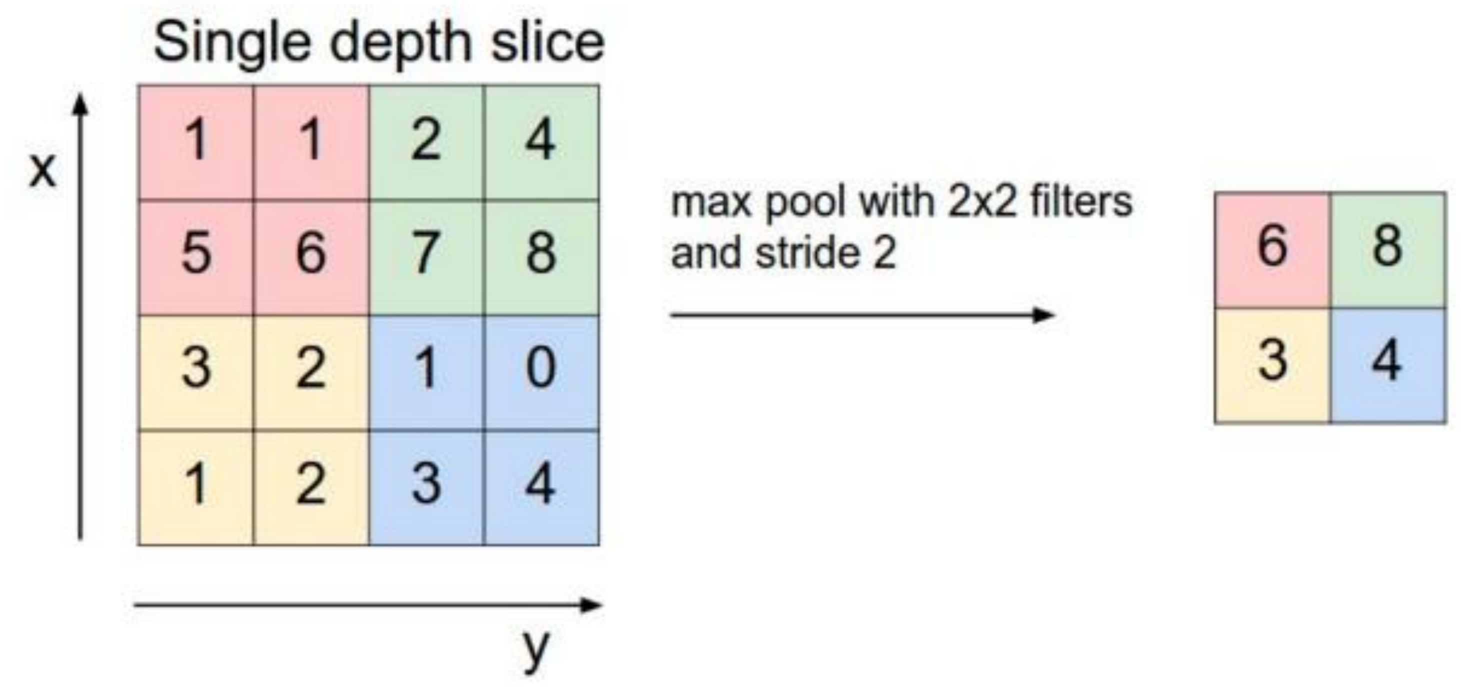
- A pooling layer can be used to compress spatial information of our feature mappings.
- Here, we still scan across the image using a window, but our goal is to compress information rather than extract certain features.
- Similar to convolutional layers, we define the window size, and stride.
- Padding is not commonly used for pooling layers.

How Max Pooling Works

- Max pooling is performed on the convolutional layers of a CNN.
- It involves sliding a window (a filter or kernel) across the input data
- Similar to the convolution step, but instead of performing a matrix multiplication, max pooling takes the maximum value within the window.
- This maximum value becomes a single pixel in the new, pooled output.
- The window is then slid across the input data by a stride of a certain number of pixels, and the process is repeated until the entire input image has been processed.

- Typically, the size of the pooling window is 2x2
- The stride with which the window is moved is also 2 pixels.
- This setup reduces the size of the input by half, both in height and width, effectively reducing the total number of pixels by 50% with a slide of 2.
- Ex: Stride=1: means that the filter will move 1 pixel at a time as we slide around

How Max Pooling Works



Here, the input image is 4×4 and the Max-pooling operation is performed using a 2×2 pooling kernel and with stride 2X2.
Stride defines how many numbers of pixels will shift over the input image.

How Max Pooling Works

- Max pooling is a downsampling technique to reduce the spatial dimensions of an input volume.
- It is a form of non-linear down-sampling to reduce the number of parameters and computation in the network.
- The pooling layer is used to reduce the spatial dimensions (i.e., the width and height) of the feature maps, while preserving the depth (i.e., the number of channels)

Advantages of Max Pooling

- Feature Invariance or Local Translation invariance: This means that the position of an object in the image does not affect the classification result, as the same features are detected regardless of the position of the object.
- Max pooling helps the model to become invariant to the location and orientation of features. This means that the network can recognize an object in an image no matter where it is located.
- Dimensionality Reduction: By downsampling the input, max pooling significantly reduces the number of parameters and computations in the network, thus speeding up the learning process and reducing the risk of overfitting.
- Noise Suppression: Max pooling helps to suppress noise in the input data. By taking the maximum value within the window, it emphasizes the presence of strong features and diminishes the weaker ones.

Challenges with Max Pooling

- Information loss: One of the main disadvantages of pooling layers is that they discard some information from the input feature maps, which can be important for the final classification or regression task.
- Over-smoothing: Pooling layers can also cause over-smoothing of the feature maps, which can result in the loss of some fine-grained details that are important for the final classification or regression task.
- Hyperparameter tuning: Pooling layers also introduce hyperparameters such as the size of the pooling regions and the stride, which need to be tuned in order to achieve optimal performance. This can be time-consuming and requires some expertise in model building.
- Max pooling is a fixed operation and does not learn from the data, unlike convolutional layers that have learnable parameters.
- Note: Modern CNN architectures have started to move away from traditional max pooling layers, using alternatives like strided convolutions for downsampling or incorporating learnable pooling operations that can adapt to the data

CNN Tensor Input Shape And Feature Maps

- To build a convolutional neural network (CNN), look at a tensor input for a CNN
- Consider an image input as a tensor to a CNN.
- Shape of A CNN Input - The shape of a CNN input typically has a length of four.
- This means that we have a rank-4 tensor with four axes
- [Batch, Channels, Height, Width]
- It is common to see the B replaced by an N - N standing for number of samples in a batch
- PyTorch uses NCHW convention

CNN Tensor Input Shape And Feature Maps

- Ex: Interpret the Shape [3, 1, 28, 28] of a given tensor
- A batch of three images, each image with a single color channel, and the image height and width are 28 x 28 respectively.

Batch size

Color channels

Height

Width

- This gives us a single rank-4 tensor that will ultimately flow through our convolutional neural network
- Given a tensor of images like this, we can navigate to a specific pixel in a specific color channel of a specific image in the batch using four indexes

Batch size

- Batch size is the number of samples that will be passed through to the network at one time.
- Epoch is one single pass over the entire training set to the network.
- Batches in Epoch
- Suppose we have 1000 images that we want to train our network to identify different breeds.
- If we specify our batch size as 10. This means that 10 images will be passed as a group, or as a batch, at one time to the network.
- A single epoch is one single pass of all the data through the network
- It will take 100 batches to make up full epoch.
- We have 1000 images divided by a batch size of 10, which equals 100 total batches.
- batches in epoch = training set size / batch_size

Why Use Batches?

- batch size is one of the hyperparameters that we must test and tune based on how our specific model is performing during training.
- Batch size has to be tested to check how our machine is performing in terms of its resource utilization when using different batch sizes.
- Ex: if we set batch size to a relatively high number, say 100, then our machine may not have enough computational power to process all 100 images in parallel, and this would suggest that we need to lower our batch size.

Image Height And Width

- To represent two dimensions, we need two axes.
- The image height and width are represented on the last two axes.
- Possible values here are 28 x 28, for image data in the MNIST or fashion-MNIST dataset

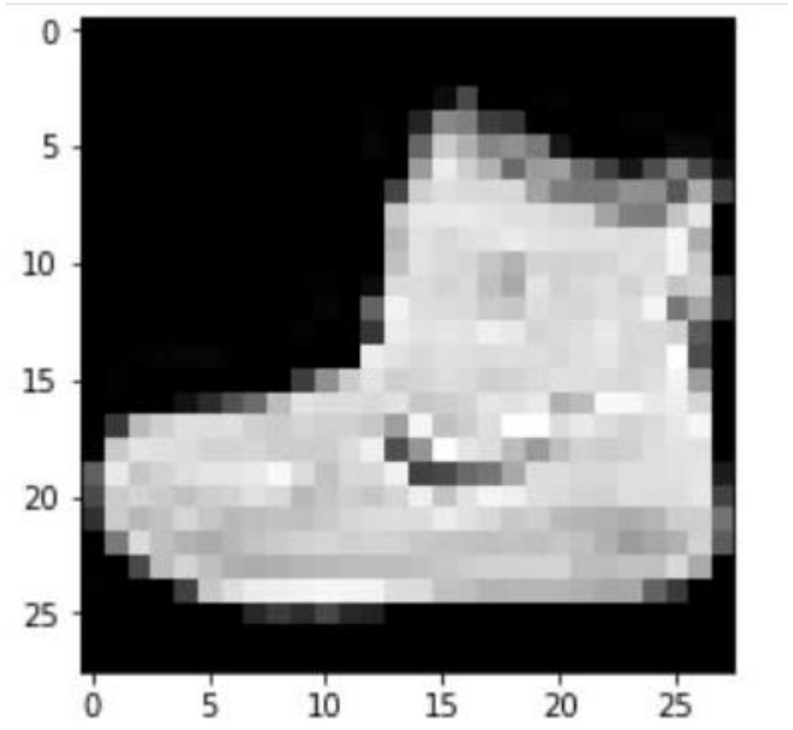


Image Color Channels

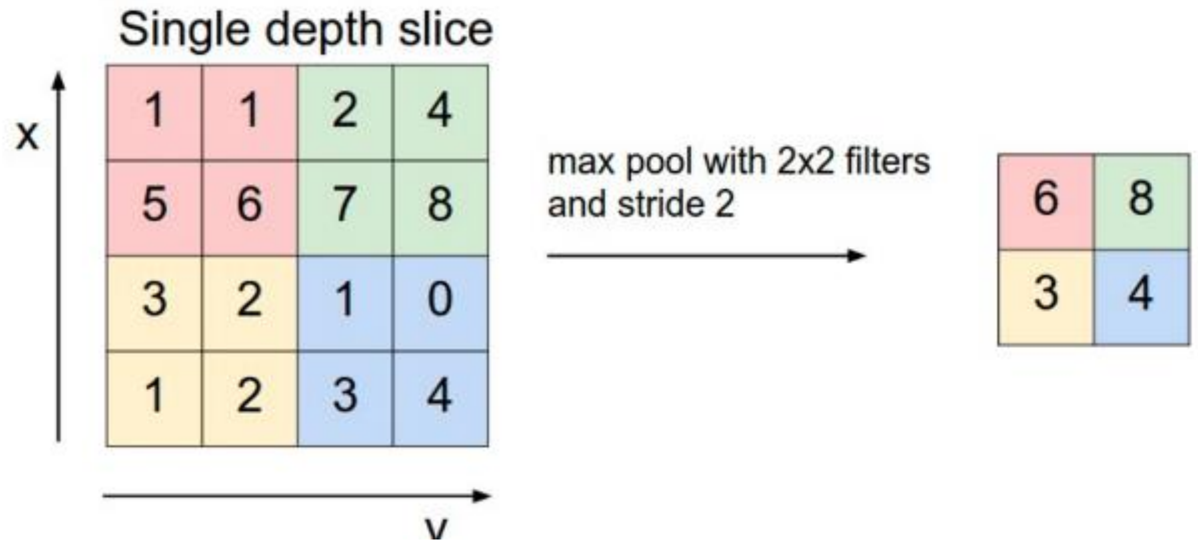
- The 2nd axis represents the color channels.
- Typical values here are 3 for RGB images, 1 for grayscale images.
- This color channel interpretation only applies to the input tensor.
- Interpretation of this axis changes after the tensor passes through a convolutional layer.

Image Batches

- The first axis of the four represents the batch size.
- In neural networks, we usually work with batches of samples opposed to single samples
- so the length of this axis tells us how many samples are in our batch

Problem Statement 1 - Implement maxpooling

- Ex: A tensor that contains data from a single 28 x 28 grayscale image.
- So the tensor shape: [1, 1, 28, 28]
- Implement maxpooling for the following input image 4×4 and the Max-pooling operation is performed using a 2×2 pooling kernel and stride 2X2

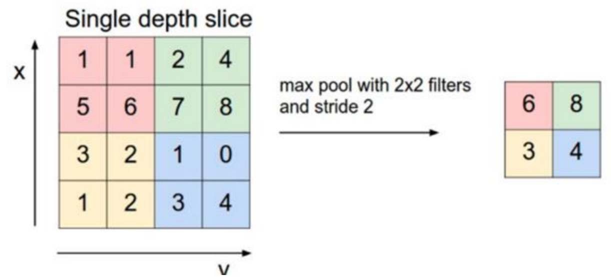


How to compute the output shape

- Let the input tensor be of a shape (n, c, h_{in}, w_{in})
- Let kernel size is (k_h, k_w)
- Output shape is (n, c, h_out, w_out)
- The output height and width is

$$h_{out} = \left[\frac{h_{in} - \text{kernel\_size}[0] + 2 * \text{padding}[0]}{\text{stride}[0]} + 1\right]$$

$$w_{out} = \left[\frac{w_{in} - \text{kernel\_size}[1] + 2 * \text{padding}[1]}{\text{stride}[1]} + 1\right]$$



Apply to the Problem Statement 1 and get the output shape

MaxPool2d Operation for max pooling

- Syntax :

torch.nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False)

- kernel_size : the filtered kernel shape,
- stride : Number of pixel shift over the input feature,(default is kernel_size)
- padding : Extra 0 padding layers around both side the feature, (default is 0)
- dialation: Control the stride (default is 1)
- return_indices : True or False (default is False) return the max indices.
- ceil_mode : True or False (default is False) when True it will use ceil instead of floor.

Problem Statement 1 - Implement maxpooling

```
import torch
import torch.nn as nn

# Define the input tensor
input_tensor = torch.tensor(
            [
                        [1, 1, 2, 4],
                        [5, 6, 7, 8],
                        [3, 2, 1, 0],
                        [1, 2, 3, 4]
            ], dtype = torch.float32)
print("input_tensor shape=", input_tensor.shape)
# Reshape the input_tensor
input_tensor = input_tensor.reshape(1, 1, 4, 4)
print("Reshaped input_tensor shape=", input_tensor.shape)
print("Reshaped input_tensor =", input_tensor)
```

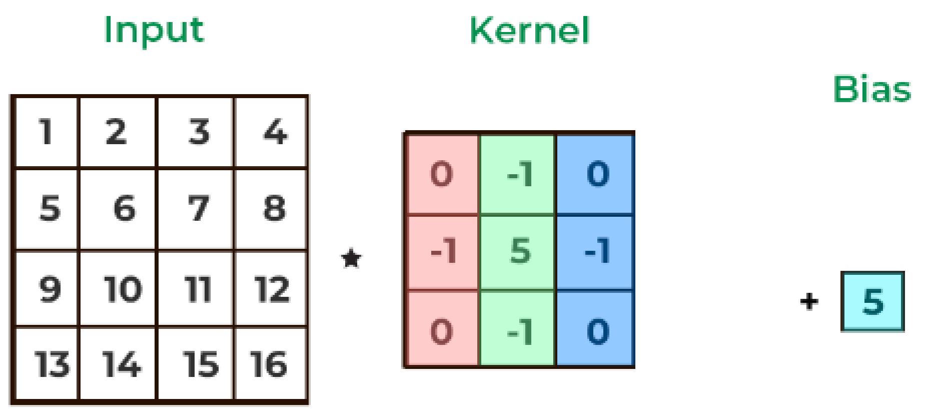
```
# Initialize the Max-pooling layer with kernel 2X2 and stride 2
pool = nn.MaxPool2d(kernel_size=2, stride=2)

# Apply the Max-pooling layer to the input tensor
output = pool(input_tensor)

# Print the output tensor
print(output)
print("output shape=", output.shape)
```

Output

```
input_tensor shape= torch.Size([4, 4])
Reshaped input_tensor shape= torch.Size([1, 1, 4, 4])
Reshaped input_tensor = tensor([[[[1., 1., 2., 4.],
          [5., 6., 7., 8.],
          [3., 2., 1., 0.],
          [1., 2., 3., 4.]]]])
tensor([[[[6., 8.],
          [3., 4.]]]])
output shape= torch.Size([1, 1, 2, 2])
```

Problem Statement 2- Implement Convolution using torch.nn

- Define a custom image of shape 4X4 and kernel 3X3 and bias 1X1



Conv2D function

torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode=’zeros’, device=None, dtype=None)

- in_channels (int) – Number of channels in the input image. Refers to depth of input image, for a grayscale image the depth = 1
- out_channels (int) – Number of channels produced by the convolution. Refers to the desired depth of output
- kernel_size (int or tuple) – Size of the convolving kernel.
- bias (bool, optional) – If True, adds a learnable bias to the output. Default: True.
- stride : controls the stride for the cross-correlation, a single number or a tuple.
- padding : controls the amount of padding applied to the input. It can be either a string {‘valid’, ‘same’} or a tuple of ints giving the amount of implicit padding applied on both sides.
- dilation : controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what dilation does.
- groups : controls the connections between inputs and outputs. in_channels and out_channels must both be divisible by groups.

nn. Parameter

- The `torch.nn. Parameter` class in Python's PyTorch library is a subclass of the `torch.Tensor` class.
- It represents a learnable parameter in a neural network model.
- Instances of this class are used to define trainable model parameters such as weights and biases.
- By default, these parameters are automatically registered as model parameters when assigned as attributes to a `torch.nn.Module`.
- Parameters can be accessed as attributes of the module and are optimized during the model's training process to minimize a given loss function.

```
conv = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=3, bias=True)
conv.weight = nn.Parameter(kernel)
conv.bias = nn.Parameter(bias)
output = conv(image)
```

Problem Statement 2- Implement Convolution using torch.nn

```
import torch
import torch.nn as nn
import torch.nn.functional as F
# Define the filter
kernel = torch.tensor(
                [[0, -1, 0],
                [-1, 5, -1],
                [0, -1, 0]], dtype=torch.float32)
kernel = kernel.reshape(1, 1, 3, 3)
# Define the bias
bias = torch.tensor([5], dtype=torch.float32)
# Define the input image
image = torch.tensor(
                [[1, 2, 3, 4],
                [5, 6, 7, 8],
                [9, 10, 11, 12],
                [13, 14, 15, 16]], dtype=torch.float32)
```

```
# Define the convolution operation
conv = nn.Conv2d(in_channels=1,
out_channels=1, kernel_size=3, bias=True)
# Set the filter for the convolution
conv.weight = nn.Parameter(kernel)
conv.bias = nn.Parameter(bias)
# Apply the convolution operation
output = conv(image)
print('Output Shape :',output.shape)
print('Output \n',output)
```

Output – Verify the result

```
Output Shape : torch.Size([1, 1, 2, 2])
Output
 tensor([[[[11., 12.],
          [15., 16.]]]], grad_fn=<ConvolutionBackward0>)
```

conv2d from nn.functional and Conv2d from nn

```
import torch.nn.functional as F
F.conv2d(image, kernel, stride=1, padding=0)


import torch.nn as nn
conv = nn.Conv2d(in_channels=1, out_channels=1, kernel_size=3,
bias=True)
```

Problem Statement 3- Implement Convolution using torch.nn.functional

```
import torch
import torch.nn.functional as F

image = torch.rand(6,6)
print("image=", image)

#Add a new dimension along 0th dimension
#i.e. (6,6) becomes (1,6,6). This is because
#pytorch expects the input to conv2D as 4d tensor

image = image.unsqueeze(dim=0)
print("image.shape=", image.shape)
image = image.unsqueeze(dim=0)
print("image.shape=", image.shape)
print("image=", image)
```

```
kernel = torch.ones(3,3)
#kernel = torch.rand(3,3)
print("kernel=", kernel)
kernel = kernel.unsqueeze(dim=0)
kernel = kernel.unsqueeze(dim=0)
#Perform the convolution
outimage = F.conv2d(image, kernel,
stride=1, padding=0)
print("outimage=", outimage)
```

Problem Statement 3- Implement Convolution using torch.nn.functional – Verify the Output

```
image= tensor([[0.6868, 0.2533, 0.0259, 0.4412, 0.4339, 0.9242],
        [0.9915, 0.4162, 0.6781, 0.2680, 0.1099, 0.2549],
        [0.8064, 0.5468, 0.0398, 0.8435, 0.1475, 0.5481],
        [0.8009, 0.8932, 0.8112, 0.8710, 0.2113, 0.3962],
        [0.3849, 0.0399, 0.6809, 0.3007, 0.6260, 0.3822],
        [0.4638, 0.7756, 0.4571, 0.7085, 0.6496, 0.9681]])
image.shape= torch.Size([1, 6, 6])
image.shape= torch.Size([1, 1, 6, 6])
image= tensor([[[[0.6868, 0.2533, 0.0259, 0.4412, 0.4339, 0.9242],
          [0.9915, 0.4162, 0.6781, 0.2680, 0.1099, 0.2549],
          [0.8064, 0.5468, 0.0398, 0.8435, 0.1475, 0.5481],
          [0.8009, 0.8932, 0.8112, 0.8710, 0.2113, 0.3962],
          [0.3849, 0.0399, 0.6809, 0.3007, 0.6260, 0.3822],
          [0.4638, 0.7756, 0.4571, 0.7085, 0.6496, 0.9681]]]])
```

```
kernel= tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
outimage= tensor([[[[4.4446, 3.5127,
2.9878, 3.9712],
          [5.9840, 5.3677, 3.9803, 3.6504],
          [5.0041, 5.0271, 4.5320, 4.3265],
          [5.3076, 5.5382, 5.3163, 5.1136]]]])
```

Problem Statement 4- Using nn. Conv2d and F. conv2d – verify the output

```
import torch
import torch.nn.functional as F
import torch.nn as nn
filter = torch.tensor(
        [[0, -1, 0],
        [-1, 5, -1],
        [0, -1, 0]], dtype=torch.float32)
filter = filter.reshape(1, 1, 3, 3)
inputs = torch.tensor(
        [[1, 2, 3, 4],
        [5, 6, 7, 8],
        [9, 10, 11, 12],
        [13, 14, 15, 16]], dtype=torch.float32)
inputs = inputs.reshape(1, 1, 4, 4)
print("filter=", filter)
print("inputs=", inputs)
```

```
o1=F.conv2d(inputs, filter, padding=0, bias=None)
print("F.conv2d output=", o1)

# using nn.Conv2d
conv = nn.Conv2d(1,1,kernel_size=3,padding=0,
bias=False)
conv.weight = nn.Parameter(filter)
o2=conv(inputs)
print("nn.Conv2d output=", o2)
```

Problem Statement 4- Using nn. Conv2d and F. conv2d – verify the output

```
filter= tensor([[[[ 0., -1.,  0.],
          [-1.,  5., -1.],
          [ 0., -1.,  0.]]]])
inputs= tensor([[[[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
          [ 9., 10., 11., 12.],
          [13., 14., 15., 16.]]]])
F.conv2d output= tensor([[[[ 6.,  7.],
          [10., 11.]]]])
nn.Conv2d output= tensor([[[[ 6.,  7.],
          [10., 11.]]]], grad_fn=<ConvolutionBackward0>)
```

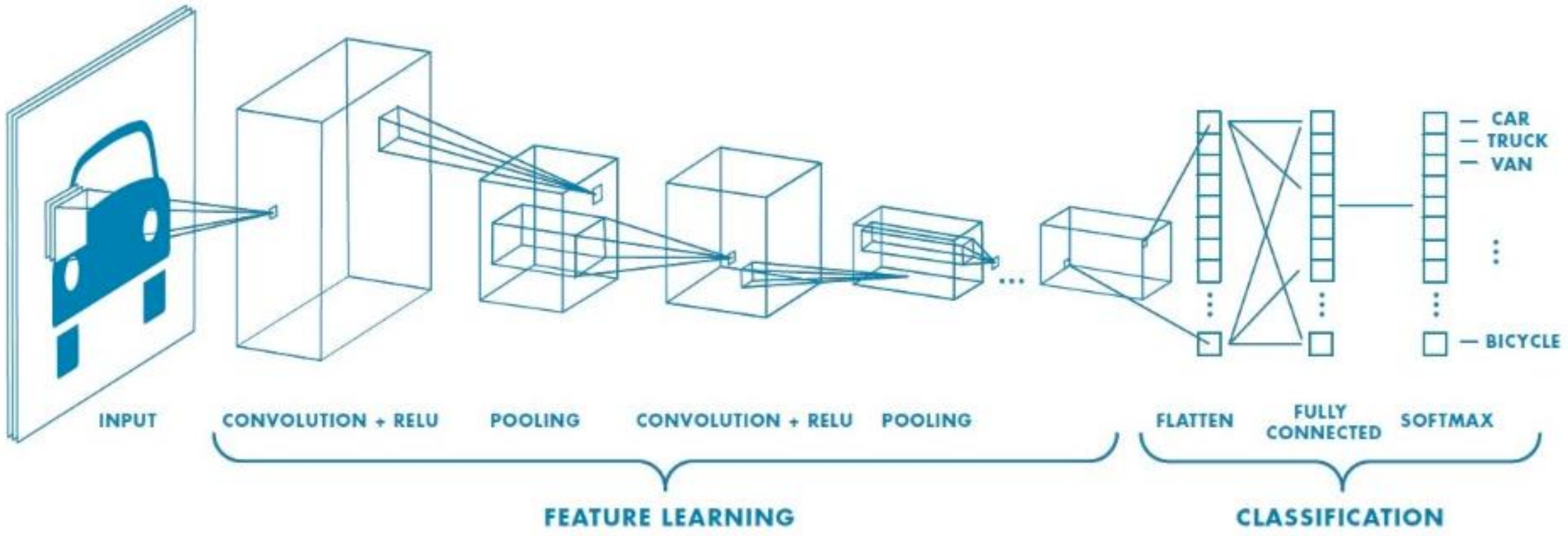
Architecture of a Convolutional Neural Network

- A typical CNN architecture consists of multiple layers, each serving a specific purpose in the image classification task.

1. The architecture begins with an input layer that takes in the image data.
2. This is followed by convolutional layers, which apply convolutional filters to the input image to extract features.

- Activation functions are then applied to introduce non-linearity into the features.
- Pooling layers are used to downsample the feature maps and reduce computational costs.
- Convolutional Neural Network consists of multiple layers like the input layer, Convolutional layer, Activation functions, Pooling layer, and followed by fully connected layers and output layer

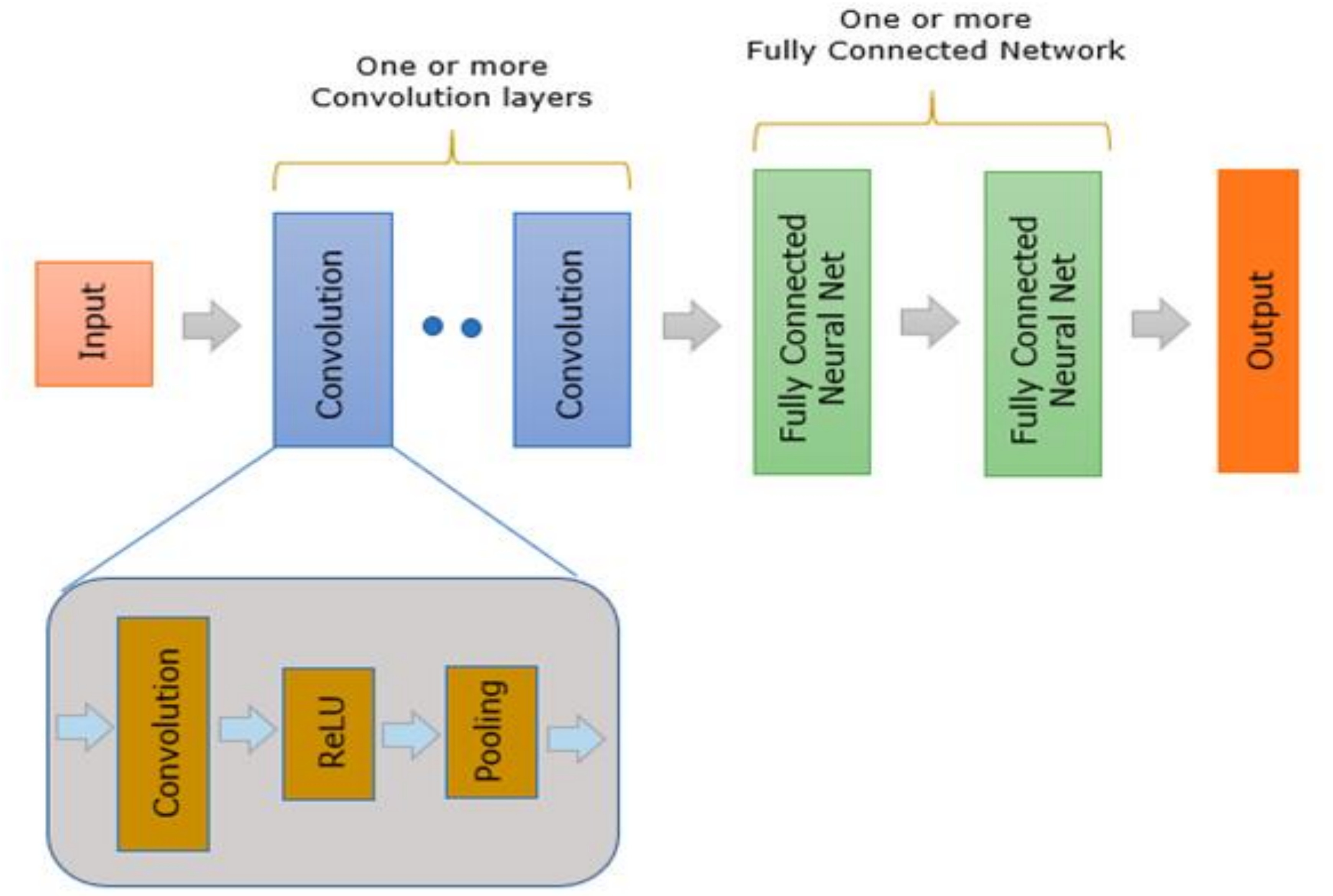
Architecture of a Convolutional Neural Network



CNN Basic Architecture

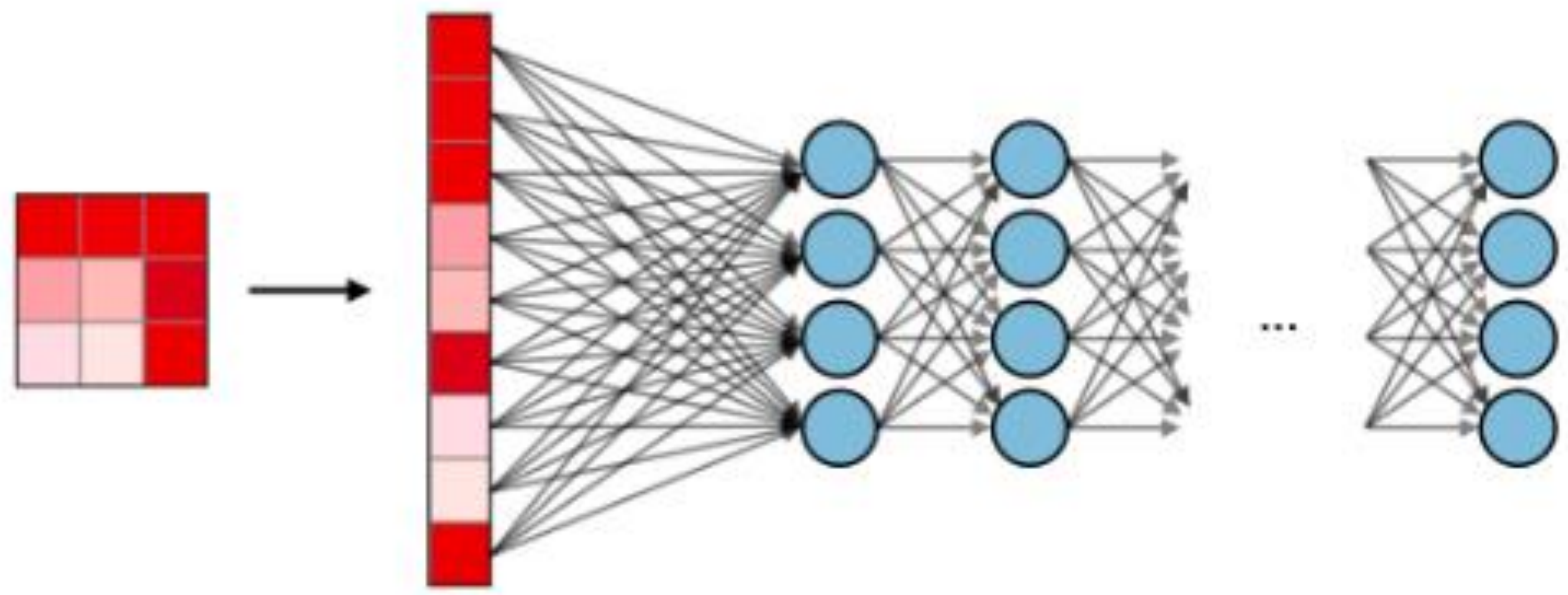
- There are two main parts to a CNN architecture
- A convolution layer that separates and identifies the various features of the image for analysis in a process called as Feature Extraction.
- The network of feature extraction consists of many pairs of convolutional layers.
- A fully connected layer that utilizes the output from the convolution process and predicts the class of the image based on the features extracted in previous stages.
- This CNN model of feature extraction aims to reduce the number of features present in a dataset.
- It creates new features which summarises the existing features contained in an original set of features.

Architecture of a Convolutional Neural Network


One or more Convolution layers
One or more Fully Connected Network
Input
Convolution
Convolution
Fully Connected Neural Net
Fully Connected Neural Net
Output
Convolution
ReLU
Pooling

A fully connected layer

- Fully Connected (FC): A fully connected layer (FC) operates on a flattened input where each input is connected to all neurons.
- FC layers are towards the end of CNN architectures and can be used to optimize objectives such as class scores.



Implementing a Convolutional Neural Network in PyTorch

- Load MNIST dataset using the PyTorch datasets module.
- Define the necessary parameters, including the number of epochs and learning rate.
- Next, create the architecture of our CNN model by defining the layers, such as convolutional layers, pooling layers, and fully connected layers.
- Finally, train the CNN model using a training loop, optimizing the model parameters using stochastic gradient descent.
- After training the CNN model, evaluate its performance on the test data.
- Calculate the accuracy of the total network as well as the accuracy for each individual class.

CALCULATING THE NUMBER OF PARAMETERS IN CNN

- We have the same general setup for the number of learnable parameters in CNN similar to FFNN
- Calculate the number of parameters per layer, and then sum up the parameters in each layer to get the total amount of learnable parameters in the entire network

For one layer:

- i - no. of input channels
- f - filter size (just the length)
- o - no. of output channels. this is also defined by how many filters are used. One filter is applied to every input channel
- num_params = weights + biases

  = size of kernel *(in_channels * out_channels)+out_channels

  = (f*f)*(i * o) + o

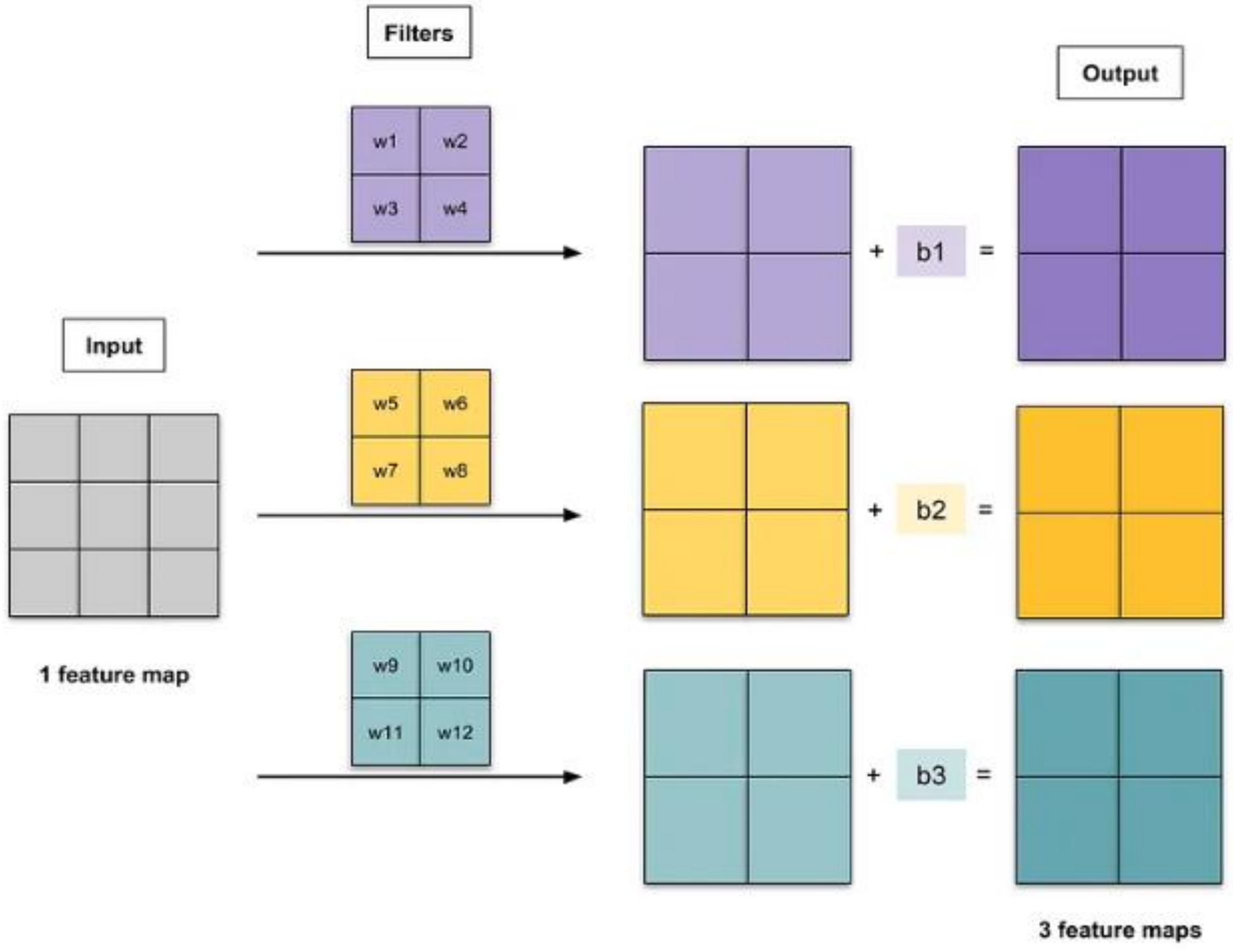
Note: In CNN, total number of kernels is unknown.

CALCULATING THE NUMBER OF PARAMETERS IN CNN

- Ex 1: Greyscale image with 2×2 filter, output 3 channels. Find weights and biases.

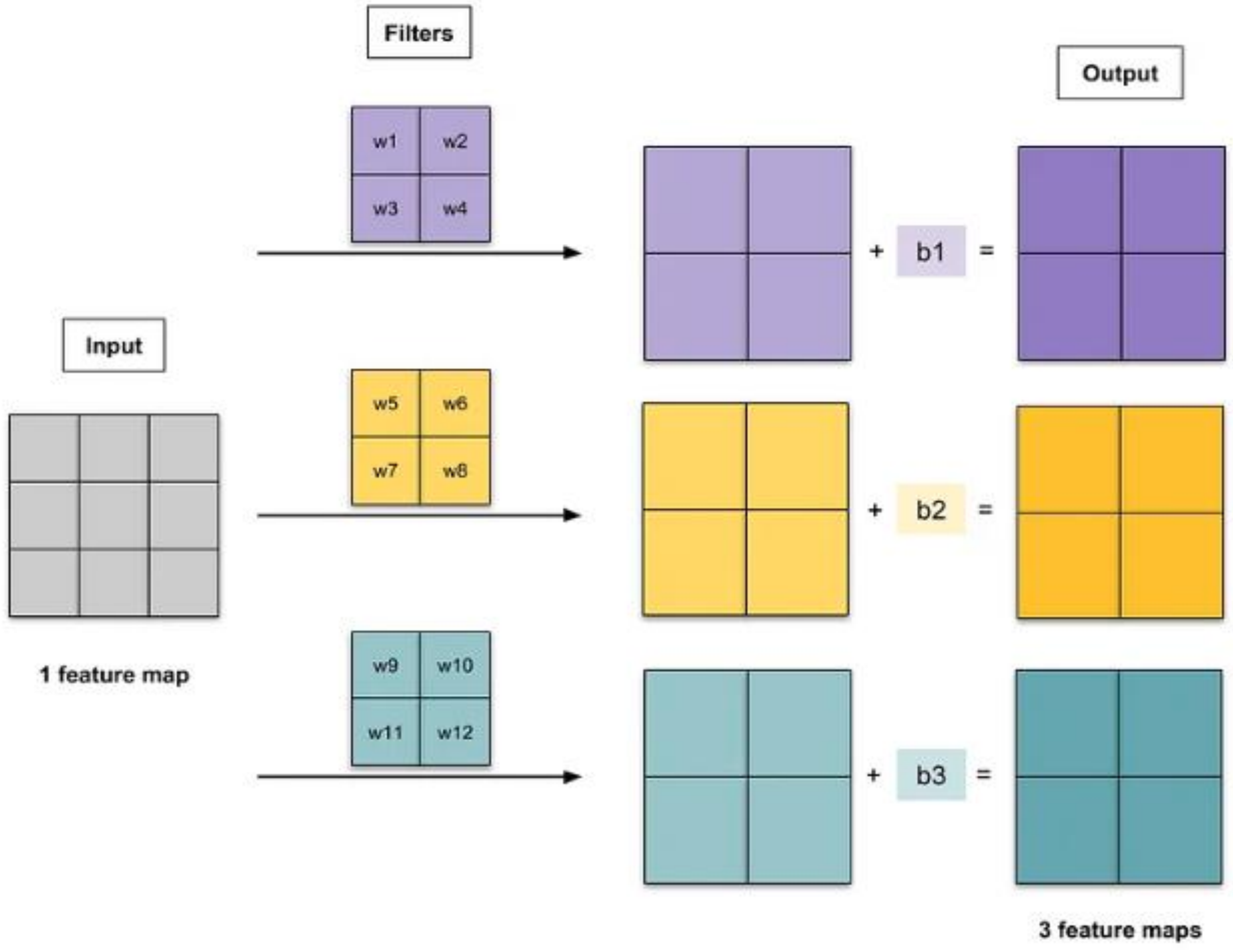
CALCULATING THE NUMBER OF PARAMETERS IN CNN

- Ex 1: Greyscale image with 2×2 filter, output 3 channels. Find weights and biases.



CALCULATING THE NUMBER OF PARAMETERS IN CNN

- Ex 1: Greyscale image with 2×2 filter, output 3 channels. Find weights and biases.
- i = 1 (greyscale has only 1 channel), f = 2, o = 3
- num_params = (f*f)*(i * o) + o = (2×2)* (1× 3) + 3= 12+3=15
- 12 weights and 3 biases

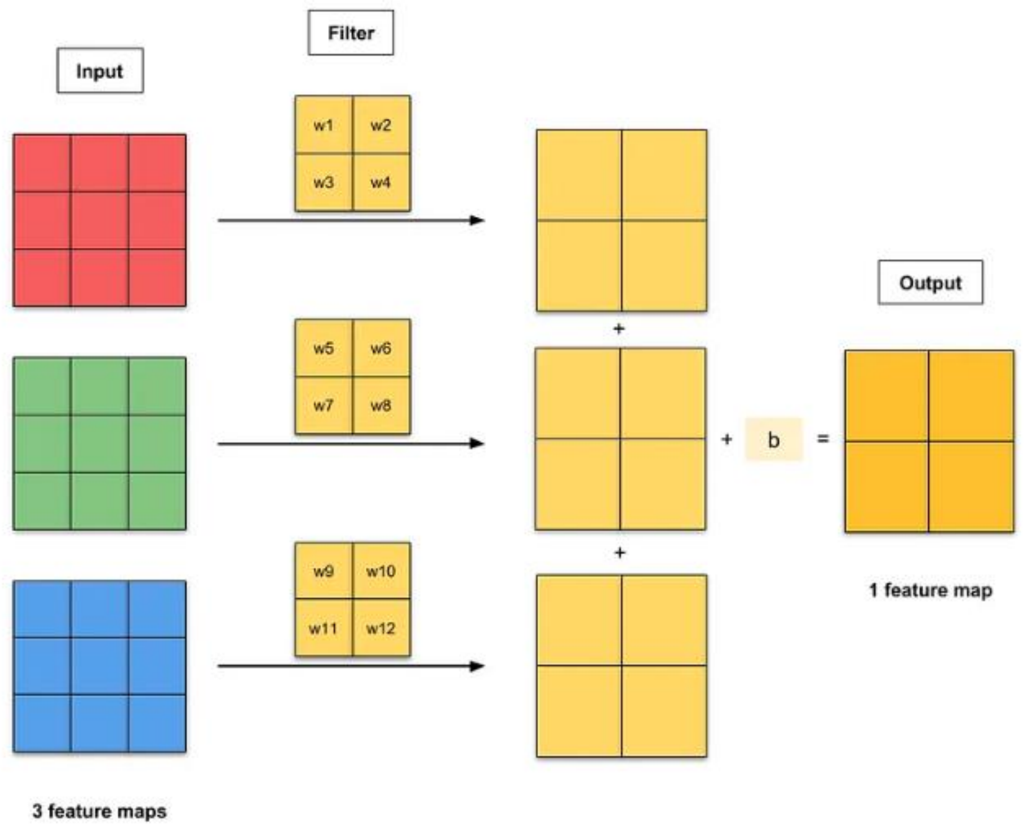


CALCULATING THE NUMBER OF PARAMETERS IN CNN

- Ex 2: RGB image with 2×2 filter, output of 1 channel. Find weights and biases.

CALCULATING THE NUMBER OF PARAMETERS IN CNN

- Ex 2: RGB image with 2×2 filter, output of 1 channel. Find weights and biases.
- There is 1 filter for each input feature map.
- The resulting convolutions are added element-wise, and a bias term is added to each element.
- This gives an output with 1 feature map



CALCULATING THE NUMBER OF PARAMETERS IN CNN

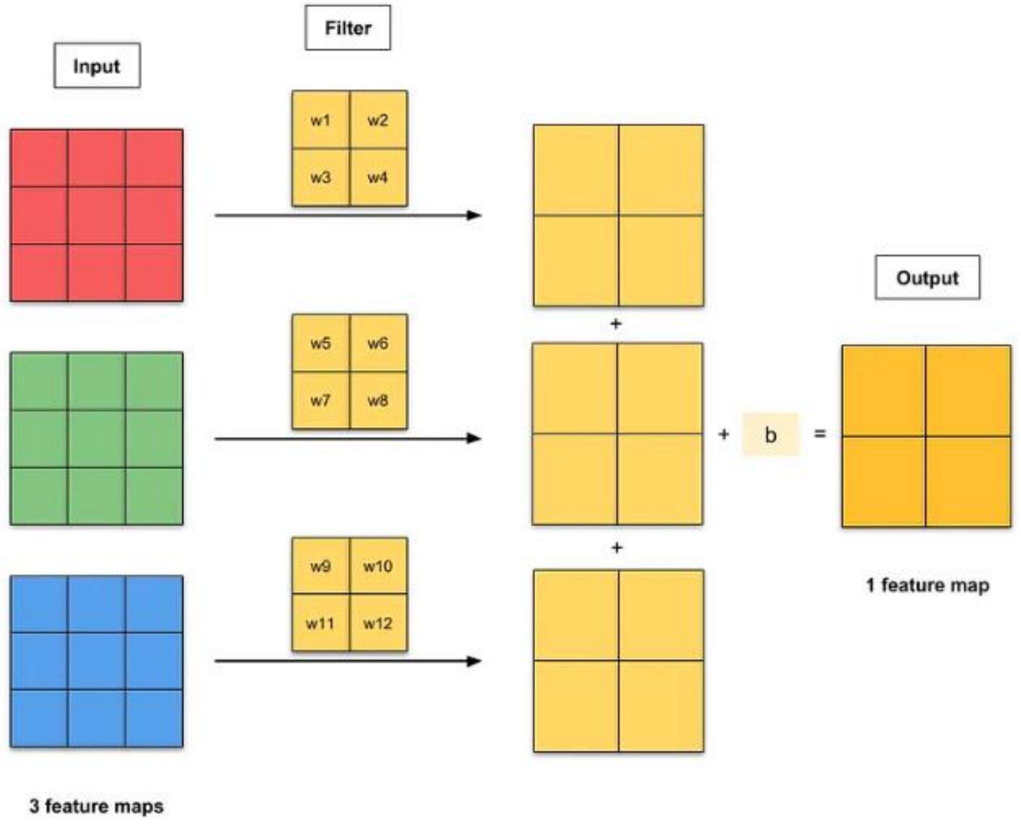
- Ex 2: RGB image with 2×2 filter, output of 1 channel. Find weights and biases.
- i = 3 (RGB image has 3 channels)
- f = 2
- o = 1
- num_params

= (f*f)*(i * o) + o

= (2×2) × (3 × 1) + 1

= 12+1

12 weights and 1 bias

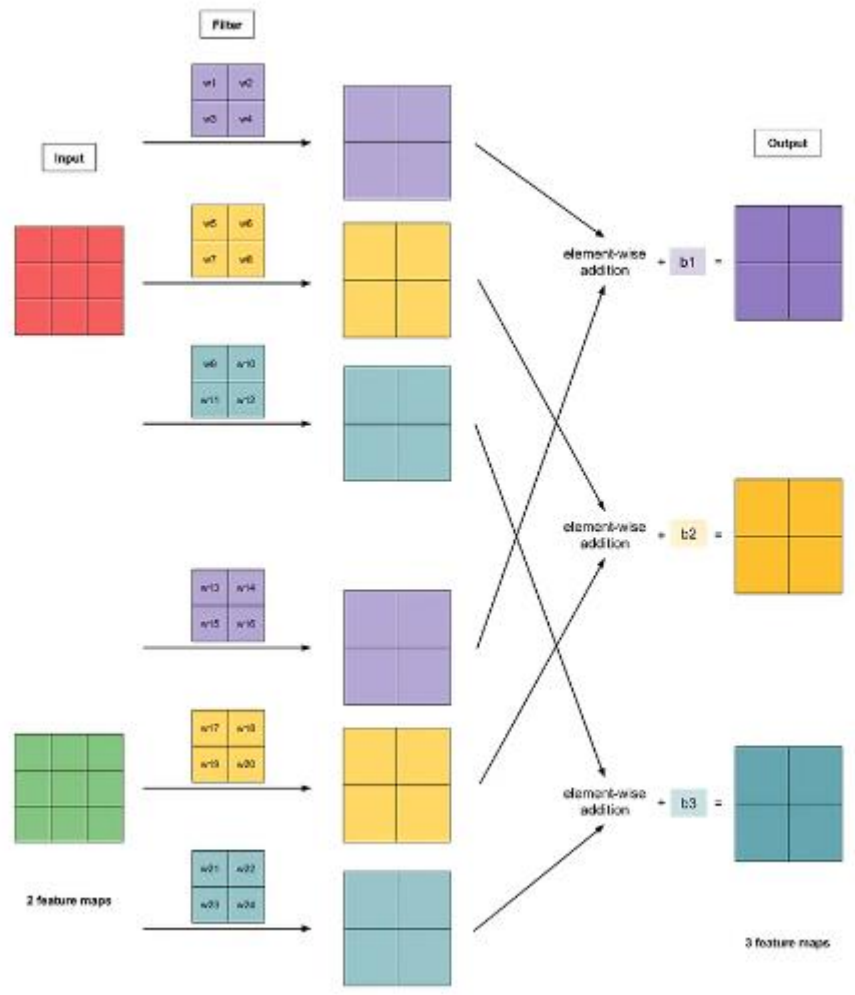


CALCULATING THE NUMBER OF PARAMETERS IN CNN

- Ex 3: Image with 2 channels, with 2×2 filter, and output of 3 channels. Find weights and biases

CALCULATING THE NUMBER OF PARAMETERS IN CNN

- Ex 3: Image with 2 channels, with 2×2 filter, and output of 3 channels. Find weights and biases
- There are 3 filters for each input feature map.
- The resulting convolutions are added element-wise, and a bias term is added to each element.
- This gives an output with 3 feature maps.



CALCULATING THE NUMBER OF PARAMETERS IN CNN

- Ex 3: Image with 2 channels, with 2×2 filter, and output of 3 channels. Find weights and biases
- i = 2
- f = 2
- o = 3
- num_params

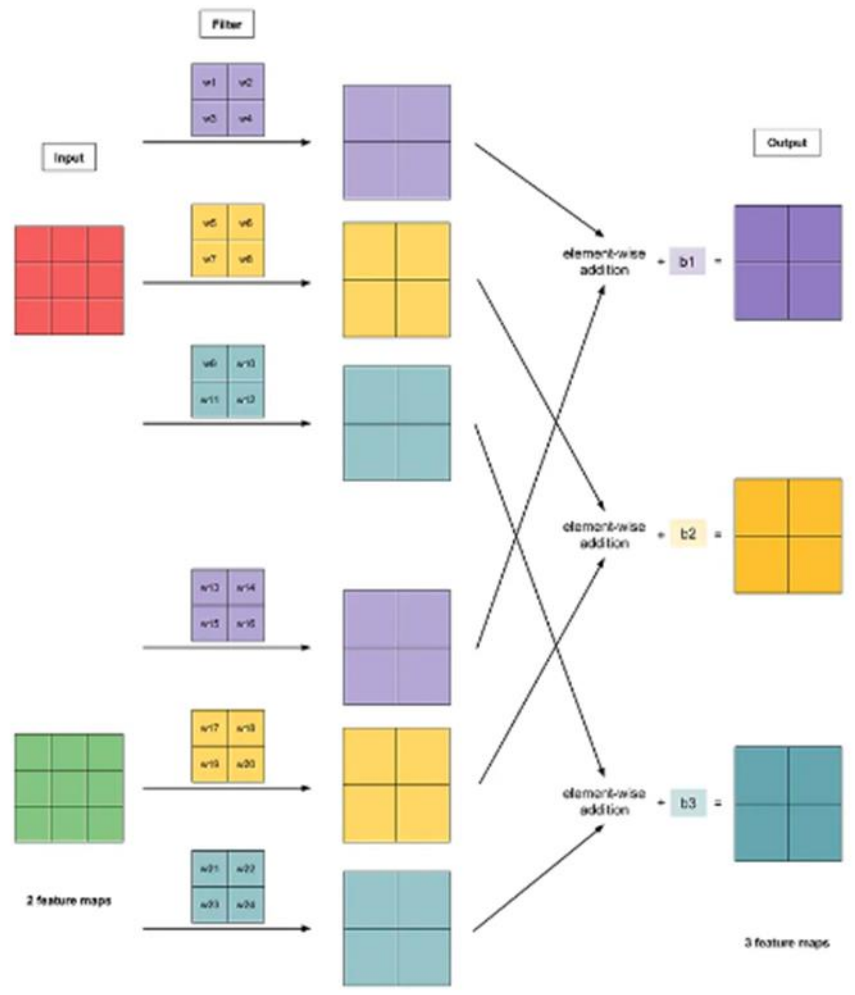
= (f*f)*(i * o) + o

= (2×2) × (2 × 3) + 3

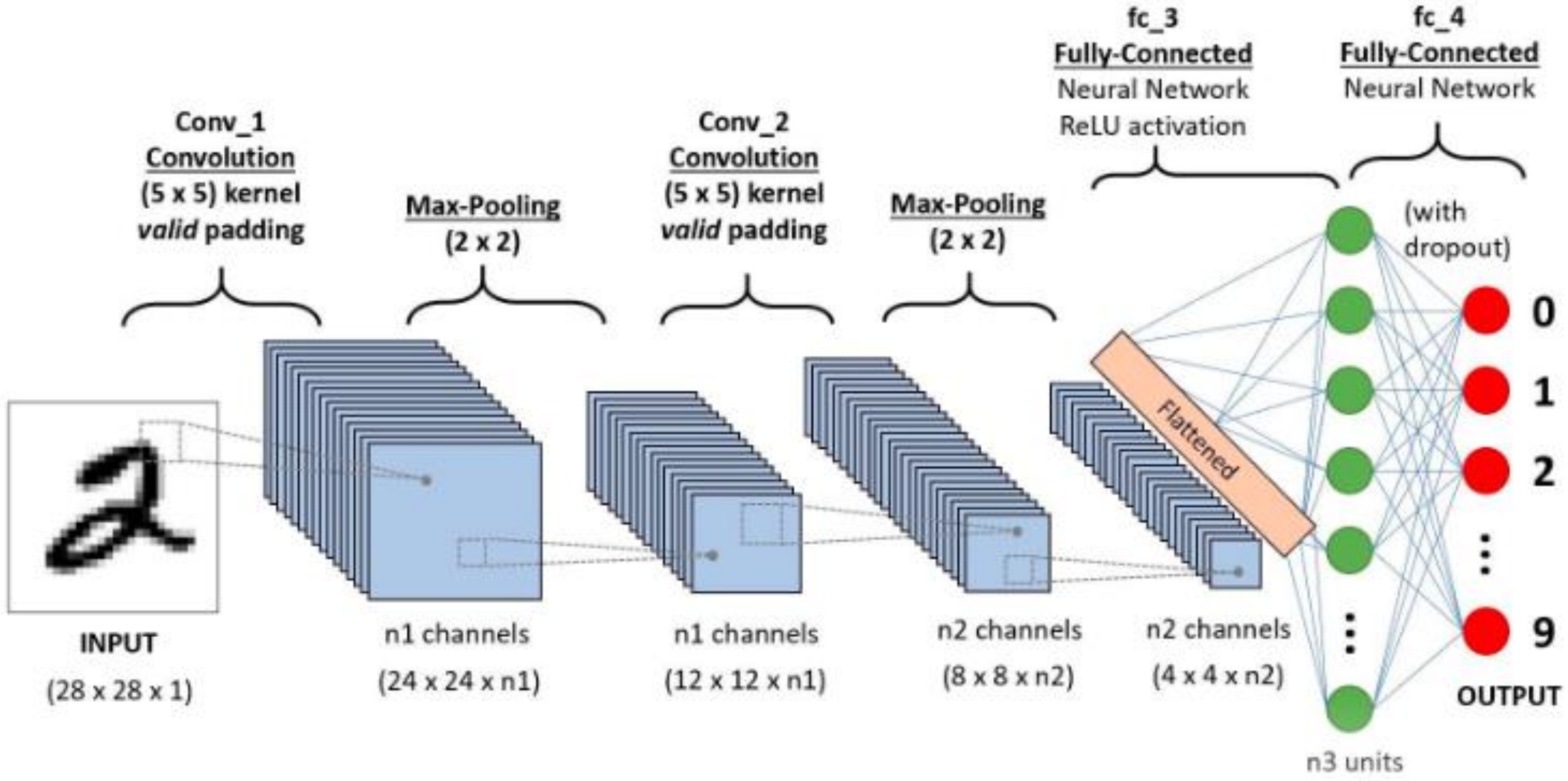
= 24+ 3

= 27

24 weights and 3 biases

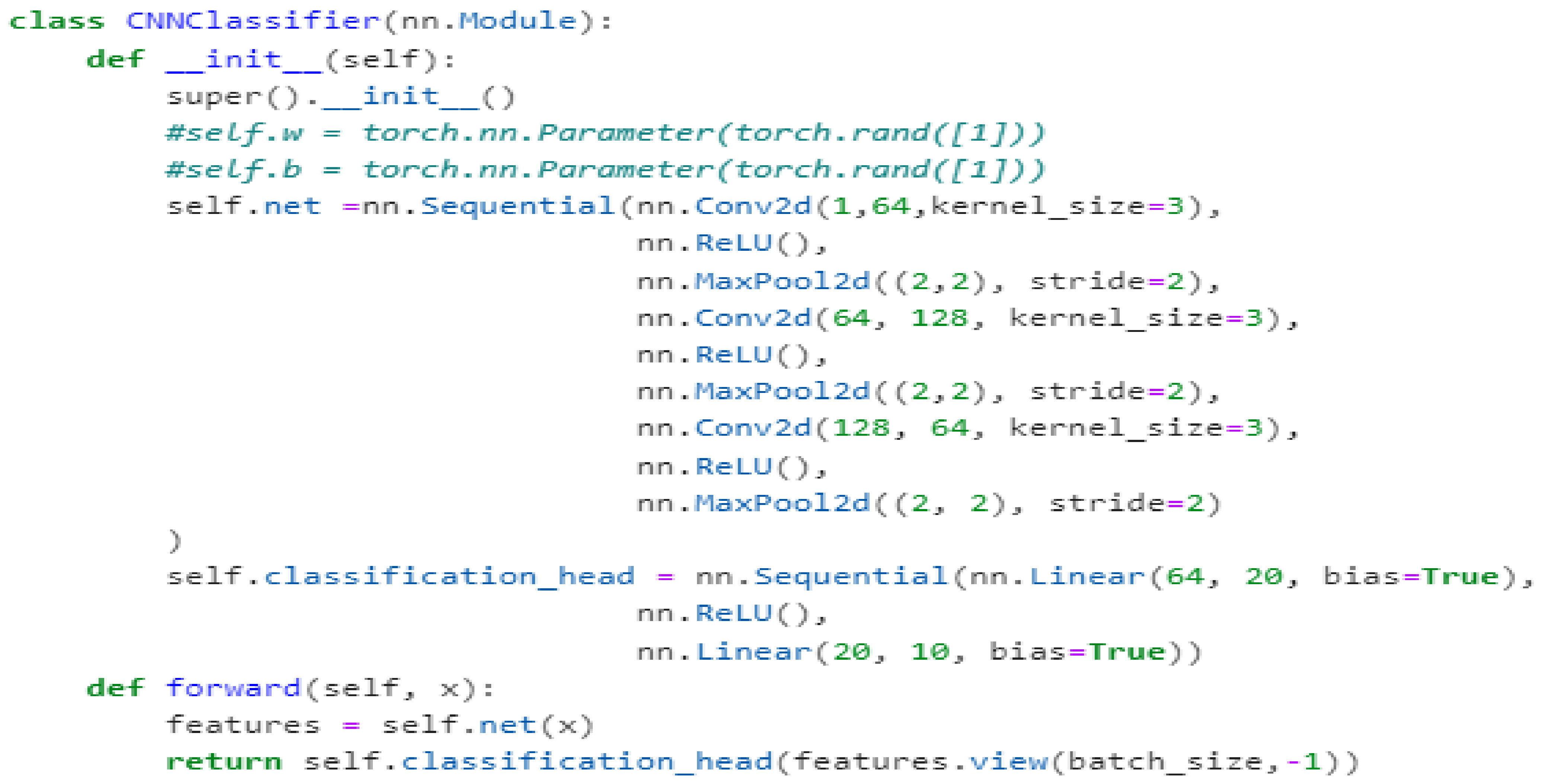


Skeleton Architecture - MNIST Digit Classification using CNN



Note:When defining CNN architecture, activation layer could be omitted.
However, the activation layers are implicitly assumed to be part of the architecture.

MNIST Digit Classification using CNN – Verify total number of parameters



```python
class CNNClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        #self.w = torch.nn.Parameter(torch.rand([1]))
        #self.b = torch.nn.Parameter(torch.rand([1]))
        self.net =nn.Sequential(nn.Conv2d(1,64,kernel_size=3),
                                nn.ReLU(),
                                nn.MaxPool2d((2,2), stride=2),
                                nn.Conv2d(64, 128, kernel_size=3),
                                nn.ReLU(),
                                nn.MaxPool2d((2,2), stride=2),
                                nn.Conv2d(128, 64, kernel_size=3),
                                nn.ReLU(),
                                nn.MaxPool2d((2, 2), stride=2)
        )
        self.classification_head = nn.Sequential(nn.Linear(64, 20, bias=True),
                                nn.ReLU(),
                                nn.Linear(20, 10, bias=True))
    def forward(self, x):
        features = self.net(x)
        return self.classification_head(features.view(batch_size,-1))
```

MNIST Digit Classification using CNN

- Note: In the forward method, define the sequence.
- Before the fully connected layers, reshape the output to match the input to a fully connected layer

MNIST Digit Classification using CNN - Output

```
CNNClassifier(
  (net): Sequential(
    (0): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=(2, 2), stride=2, padding=0, dilation=1, 
ceil_mode=False)
    (3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=(2, 2), stride=2, padding=0, dilation=1, 
ceil_mode=False)
    (6): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1))
    (7): ReLU()
    (8): MaxPool2d(kernel_size=(2, 2), stride=2, padding=0, dilation=1, 
ceil_mode=False)
  )
  (classification_head): Sequential(
    (0): Linear(in_features=64, out_features=20, bias=True)
    (1): ReLU()
    (2): Linear(in_features=20, out_features=10, bias=True)
  )
)
```

MNIST Digit Classification using CNN - Output

```
CNNClassifier(
  (net): Sequential(
    (0): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=(2, 2), stride=2, padding=0, dilation=1,
ceil_mode=False)
    (3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=(2, 2), stride=2, padding=0, dilation=1,
ceil_mode=False)
    (6): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1))
    (7): ReLU()
    (8): MaxPool2d(kernel_size=(2, 2), stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (classification_head): Sequential(
    (0): Linear(in_features=64, out_features=20, bias=True)
    (1): ReLU()
    (2): Linear(in_features=20, out_features=10, bias=True)
  )
)
```

```
net.0.weight 576
net.0.bias 64

net.3.weight 73728
net.3.bias 128

net.6.weight 73728
net.6.bias 64

classification_head.0.weight 1280
classification_head.0.bias 20

classification_head.2.weight 200
classification_head.2.bias 10

Total Parameters:149798
```

CALCULATING THE NUMBER OF PARAMETERS IN CNN

- In CNN, the exact type of features to be detected is determined by the network itself during the learning process.
- The element values for each of the kernel is not fixed/predefined and is set randomly and gets updated (changes) during learning process.
- So, theoretically we can find total number of parameters and number of output channels as shown next examples in Ex1-Ex3.
- number of kernels and the kernel size is given,
- number of input channels is given,
- number of output channels is not given
- Note: Number of kernels: This parameter is responsible for defining the depth of the output. If we have three distinct filters, we have three different feature maps, creating a depth of three. Here number of output channels is assumed same as number of kernels

CALCULATING THE NUMBER OF PARAMETERS IN CNN

1. input_shape

- input_shape = (batch_size, depth, height, width)
- batch_size= number of training examples in one forward/backward pass

2. output_shape

- output_shape = (batch_size, depth, height, width)

3. filter

- In a convolution neural network, input data is convolved over with a filter which is used to extract features.
- Filter/kernel is a matrix that will move over the image pixel data (input) and will perform a dot product with that particular region of that input data and the output will be the matrix of the dot product.

CALCULATING THE NUMBER OF PARAMETERS IN CNN – Ex1

Ex 1:Calculate the number of parameters and output shape

Input:

- filters = 1
- kernel_size=(3,3)
- input_shape=(1, 10,10)

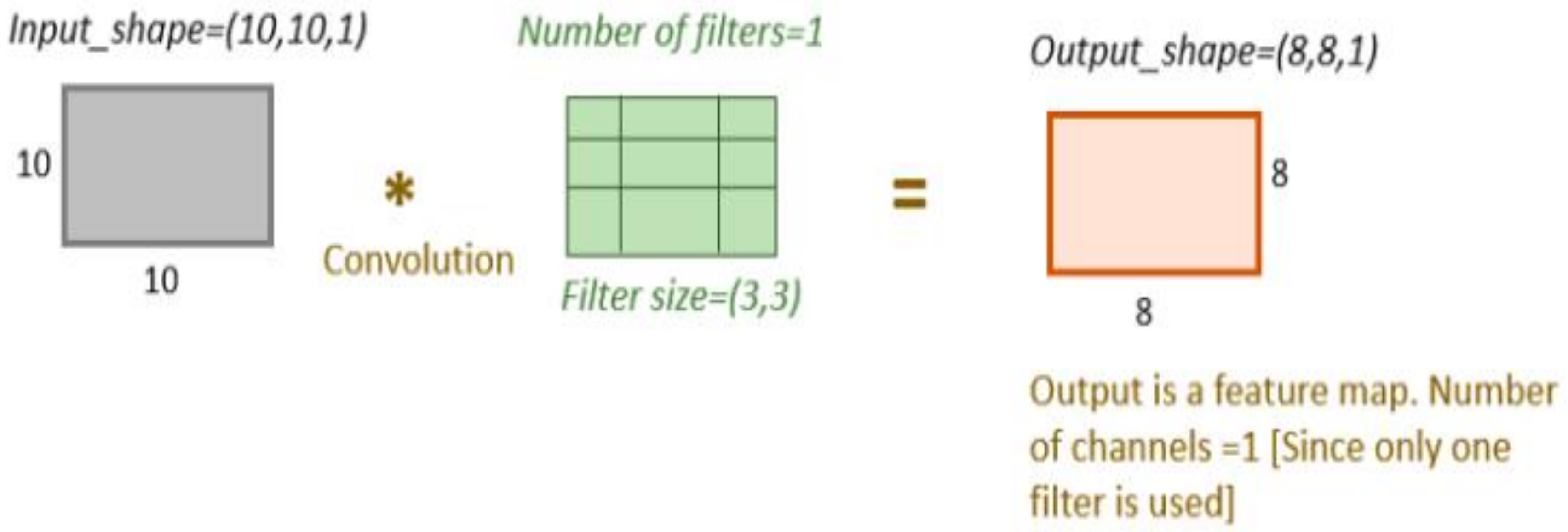
CALCULATING THE NUMBER OF PARAMETERS IN CNN – Ex1

Ex 1: Calculate the number of parameters and output shape

filters = 1

kernel_size=(3,3)

input_shape=(1, 10,10)



Note: Image Processing convention of NHWC (10, 10, 1) and (8, 8, 1)is shown in the diagram.
PyTorch uses NCHW convention of(1, 10, 10), (1, 8, 8)

CALCULATING THE NUMBER OF PARAMETERS IN CNN – Ex1

Ex 1: Calculate the number of parameters and output shape

- filters = 1
- kernel_size=(3,3)
- input_shape=(1, 10,10)
- Weights in one filter of size(3,3)= 3*3 =9
- Bias =1 [One bias will be added to each filter. Since only one filter kernel is used, bias =1]
- Total parameters for one filter kernel of size (3,3) = 9+1 =10 = 3*3*(1*1)+1
- Output shape = n-f+1 = 10–3+1 =8
- The number of channels in the feature map depends on the number of filters used. Here, in this example, only one filter is used. So, the number of channels in the feature map is 1.
- So, Output_shape of feature map= (1, 8,8)

CALCULATING THE NUMBER OF PARAMETERS IN CNN – Ex2

Ex 2:Calculate the number of parameters and output shape

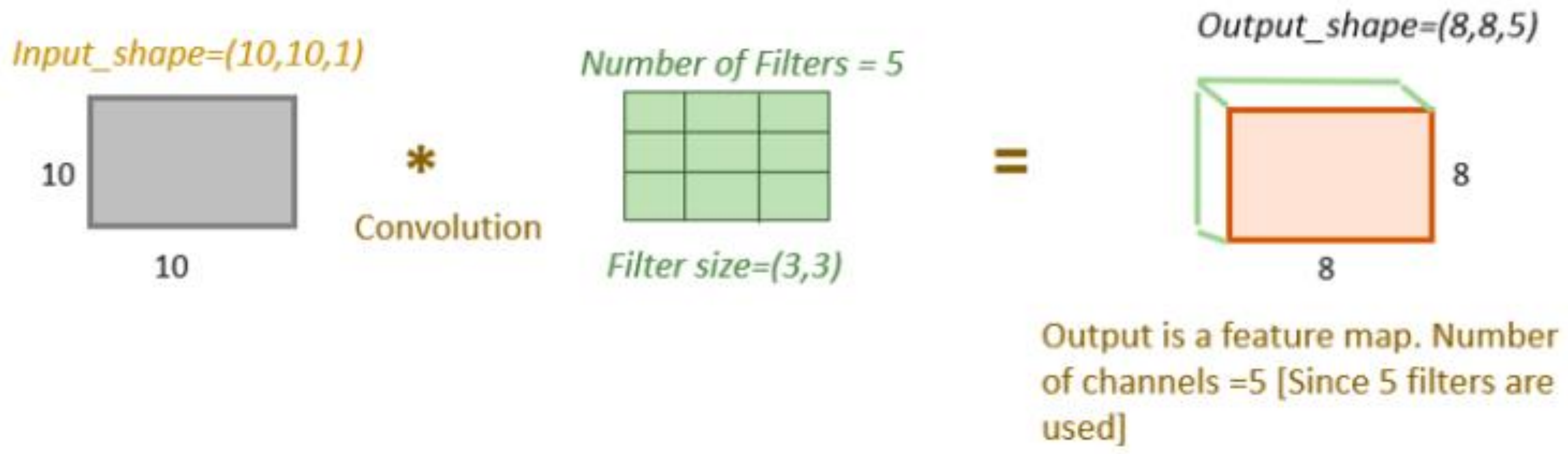
Input:

- filters = 5
- kernel_size=(3,3)
- input_shape=(1, 10,10)

CALCULATING THE NUMBER OF PARAMETERS IN CNN – Ex2

Ex 2: Calculate the number of parameters and output shape

filters = 5
kernel_size=(3,3)
input_shape=(1, 10,10)



CALCULATING THE NUMBER OF PARAMETERS IN CNN – Ex2

Ex 2: Calculate the number of parameters and output shape

filters = 5

kernel_size=(3,3)

input_shape=(1, 10,10)

- Parameters in one filter of size(3,3)= 3*3 =9
- Bias =1 [One bias will be added to each filter]
- Total parameters for filter kernel of size (3,3) = 9+1 =10
- The total number of filters= 5.
- Total parameters for 5 filter kernel of size (3,3) = 10*5=50 = 3*3*(1*5)+5
- Output shape = n-f+1 = 10–3+1 =8
- The number of channels in the feature map depends on the number of filters used. Here, in this example, 5 filters are used. So, the number of channels in the feature map is 5.
- So, Output_shape of feature map= (5, 8,8)

CALCULATING THE NUMBER OF PARAMETERS IN CNN – Ex3

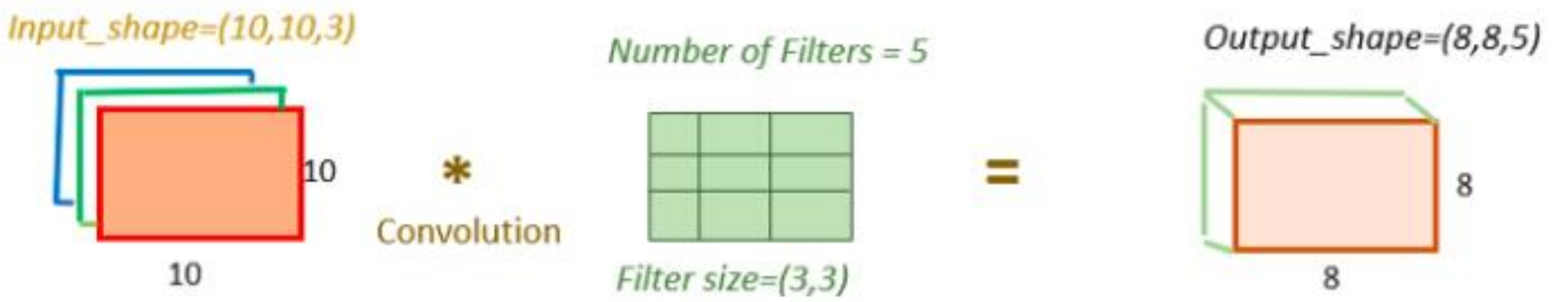
Ex 3:Calculate the number of parameters and output shape

Input:

- filters = 5
- kernel_size=(3,3)
- input_shape=(3, 10,10)

CALCULATING THE NUMBER OF PARAMETERS IN CNN – Ex3

Ex 3: Calculate the number of parameters and output shape
filters = 5
kernel_size=(3,3)
input_shape=(3, 10,10)



Output is a feature map. Number of channels =5 [Since 5 filters are used]

CALCULATING THE NUMBER OF PARAMETERS IN CNN – Ex3

Ex 3:Calculate the number of parameters and output shape

filters = 5

kernel_size=(3,3)

input_shape=(3,10,10)

Parameters in one filter of size(3,3)= 3*3 =9

The filter will convolve over all three channels concurrently(input_image depth=3). So parameters in one filter will be 3*3*3=27 [filter size * input_data depth]

Bias =1 [One bias will be added to each filter]

Total parameters for one filter of size (3,3) for depth 3=(3*3*3)+1=28

The total number of filters= 5.

Total parameters for 5 kernel of size (3,3) , input_image depth(3)= 28*5=140 = (3*3)*(3*5)+5

Output shape = n-f+1 = 10–3+1 =8

- The number of channels in the feature map depends on the number of filters used. Here, in this example, 5 filters are used. So, the number of channels in the feature map is 5.
- So, Output_shape of feature map= (5, 8,8)

CNN Summary

1. Input Layer: Think of it as the ground floor, where the raw image data enters the CNN.

2. Convolutional Layer: This is where the magic happens! Like skilled workers constructing walls, convolutional filters slide across the image, detecting patterns and extracting features.

3. Pooling Layer: This is like a foreman optimizing the construction process. Pooling reduces the image size, making computations faster and reducing memory usage.

4. Activation Layer: This is like adding color and vibrancy to the building. Activation functions introduce non-linearity, allowing the network to learn complex relationships between features.

5. Fully Connected Layer: This is like the top floor, where all the features come together for the final decision. The network takes in the extracted features and classifies the image.

6. Output Layer: This is like the roof, where the final classification result is displayed