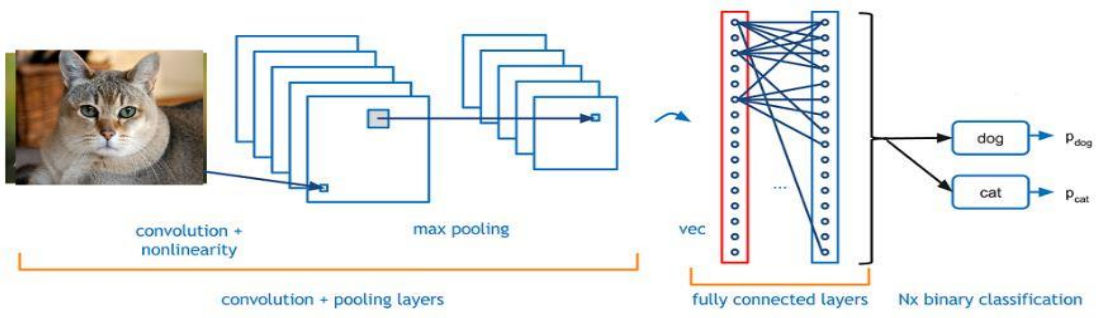
Convolution Neural Network: Convolution operation, Building Blocks of CNN, Pooling, Variants of basic convolution function, and building a CNN for image classification.

Introduction to Convolutional Neural Networks (ConvNets or CNNs):

- A type of deep learning algorithm specifically designed for image and grid-structured data processing. They are inspired by the structure and function of the visual cortex in the human brain.
- In a ConvNet, the input data is processed through multiple convolutional and pooling layers, which extract and reduce the dimensionality of the important features in the data. The final layer of the network is typically a fully connected layer, which combines the features from the previous layers to make a prediction or classification.
- *Applications* are image classification, object detection, Facial Recognition, Autonomous Vehicles, Medical Image Analysis, Natural Language Processing and segmentation. They have also been applied to other types of grid-structured data, such as speech signals and protein sequences.
- Convolutional neural networks also *minimize computation* in comparison with a regular neural network.



Convolution: -

Convolution is a mathematical operation on two functions (f and g) that produces a third function (f * g) that expresses how the shape of one is modified by the other. In image processing, convolution is the process in which each element of the image is added to its local neighbors and then it is weighted by the kernel. In convolution, the matrix doesn't perform the traditional matrix multiplication but is denoted by *.

Suppose, there are two 3x3 matrices, one is kernel and another one is an image piece. In convolution, rows and columns of the kernel are flipped and then they are multiplied and then summing is performed. Elements which are present in the centre of matrix i.e. in [2,2] of the image will be weighted combination of the image matrix and the weights will be given by the kernel. Similarly, all the other elements of the matrix will be weighted and then weights will be computed.

Another example,

I_{11}	I_{12}	I_{13}	I_{14}	I_{15}	I_{16}	I_{17}	I_{18}	I_{19}
I_{21}	I_{22}	I_{23}	I_{24}	I_{25}	I_{26}	I_{27}	I_{28}	I_{29}
I_{31}	I_{32}	I_{33}	I_{34}	I_{35}	I_{36}	I_{37}	I_{38}	I_{39}
I_{41}	I_{42}	I_{43}	I_{44}	I_{45}	I_{46}	I_{47}	I_{48}	I_{49}
I_{51}	I_{52}	I_{53}	I_{54}	I_{55}	I_{56}	I_{57}	I_{58}	I_{59}
I_{61}	I_{62}	I_{63}	I_{64}	I_{65}	I_{66}	I_{67}	I_{68}	I_{69}

K_{11}	K_{12}	K_{13}
K_{21}	K_{22}	K_{23}

In the above example, the convolution is performed by sliding the kernel over the image, generally starting at the top left corner. Each kernel position corresponds to a single output pixel, which is calculated by multiplying together the kernel value and the underlying image pixel value for each of the cells in the kernel

and then adding all these numbers together.
For example, the value of bottom right pixel in the output image will be given by

$$O_{57} = I_{57}K_{11} + I_{58}K_{12} + I_{57}K_{11} + I_{59}K_{13} + I_{67}K_{21} + I_{68}K_{22} + I_{69}K_{23}$$

Mathematically we can write the convolution as

$$O(i, j) = \sum_{k=1}^{m} \sum_{l=1}^{n} I(i+k-1, j+l-1)K(k,l)^{m}$$

Convolution can be computed using multiple for loops. But using loops causes a lot of repeated calculation and also the size of image and kernel increases. Using Discrete Fourier Transform technique calculating convolution can be done rapidly. In this technique, the entire convolution operation is converted into a simple multiplication.

In convolution the problem occurs when the kernel is near the corners.
This problem can be solved by:

i. Ones can be ignored
ii. Extra pixels can be created near the edges

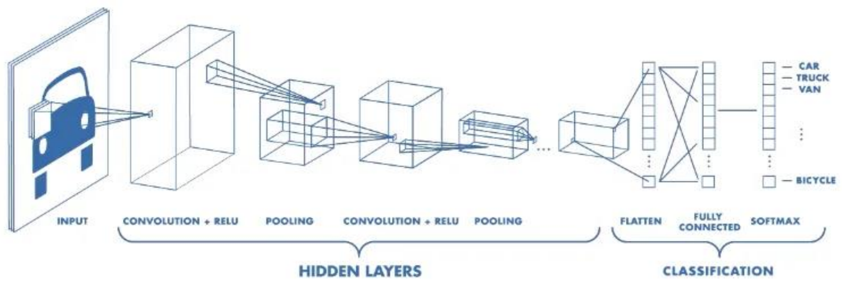
Extra pixels can be created by duplicating the edge pixels or reflecting the edges.

Basic Building blocks of Convolutional Neural Network :

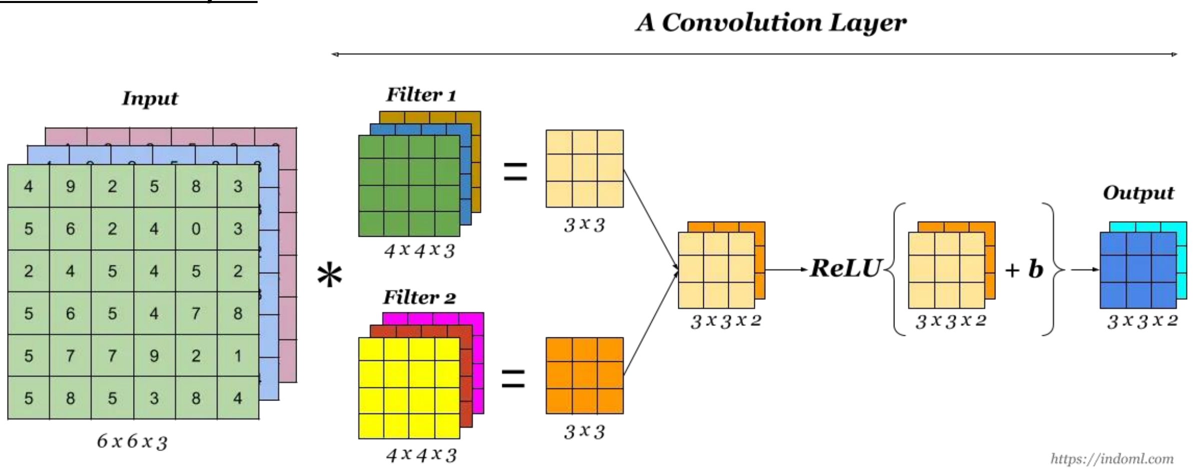
A Convolutional Neural Network (CNN) typically consists of several building blocks, which are used to process and classify images. The main building blocks of a CNN are:

- Convolutional Layer
- Kernel
- Padding
- Stride

Architecture of a Convolutional Neural Network:



<u>Convolutional Layer:</u>



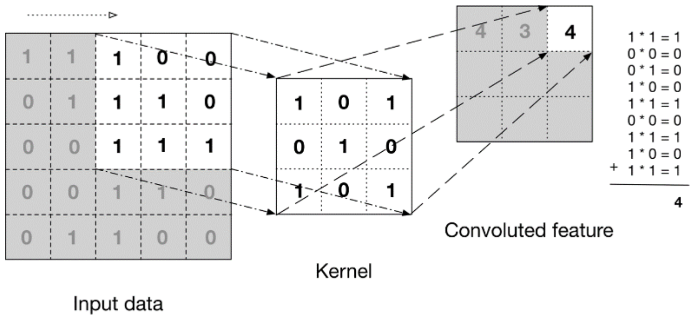
The convolution layer is the core building block of the CNN. It carries the main portion of the network's computational load. This layer applies filters to the input image to extract features. The filters slide over the input image, computing dot products at each location, and the resulting feature maps highlight different aspects of the input image.
This layer performs a dot product between two matrices, where one matrix is the set of learnable parameters otherwise known as a kernel, and the other matrix is the restricted portion of the receptive field. This means that, if the image is composed of three (RGB) channels, the kernel height and width will be spatially small, but the depth extends up to all three channels.
During the forward pass, the kernel slides across the height and width of the image-producing the image representation of that receptive region. This produces a two-dimensional representation of the image known as an activation map that gives the response of the kernel at each spatial position of the image. The sliding size of the kernel is called a stride.

If we have an input of size W x W x D and D_{out} number of kernels with a spatial size of F with stride S and amount of padding P, then the size of output volume can be determined by the following formula:

$$W_{out} = (W-F+2P/S) +1$$

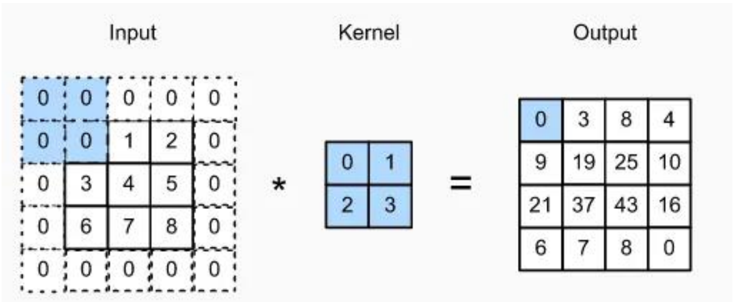
Kernel

- In Convolutional neural network, the kernel is nothing but a filter that is used to extract the features from the images.
- The kernel is a matrix that moves over the input data, performs the dot product with the sub-region of input data, and gets the output as the matrix of dot products.
- Kernel moves on the input data by the stride value. If the stride value is 2, then kernel moves by 2 columns of pixels in the input matrix.
- In short, the kernel is used to extract high-level features like edges from the image.
- By applying multiple kernels to the input data, a convolutional layer can learn to extract more complex features, such as shapes and objects.
- The weights in the kernel are learned during training using backpropagation. By adjusting the weights, the model can learn to identify the important features of the input data for the given task.
- In general, larger kernels are used to capture more complex features, while smaller kernels are used to capture finer details.



Padding

- Padding is a technique used in Convolutional Neural Networks (CNNs) to preserve the spatial dimensions of the input data during convolution. Convolution involves sliding a filter over the input data and performing element-wise multiplication between the filter and the input data. The result of this operation is a feature map.
- Without padding, the size of the feature map is reduced as we move from one convolutional layer to the next. This reduction in size can be problematic because it can lead to a loss of information at the edges of the image, and it can make it difficult to construct

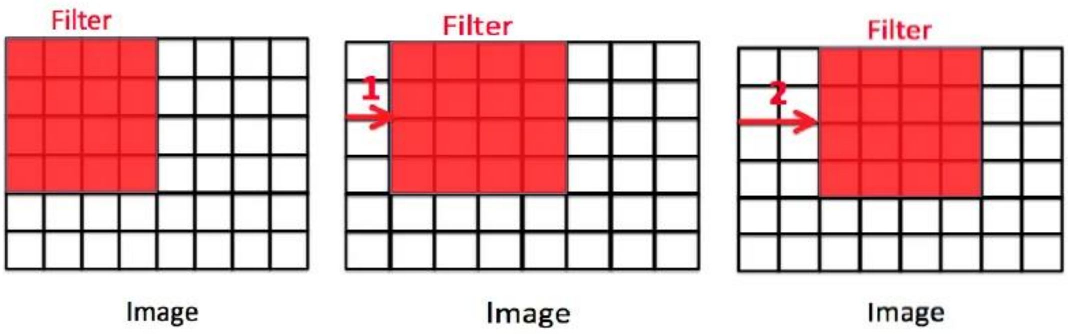


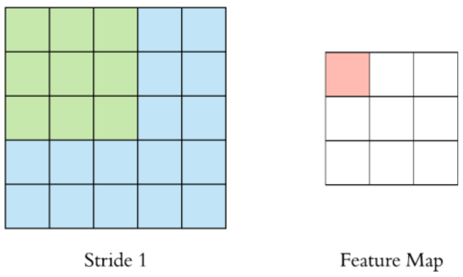
deeper networks with many layers.

- Padding solves this problem by adding extra pixels around the edge of the input data before performing convolution. The added pixels can be either zeros (zero-padding) or copies of the edge pixels (symmetric padding or reflective padding). By adding these extra pixels, we can ensure that the output feature map has the same size as the input data.
- Padding is often specified using the parameter "padding" in the convolutional layer. Common values for this parameter are "valid" (no padding) and "same" (padding so that the output size is the same as the input size).

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





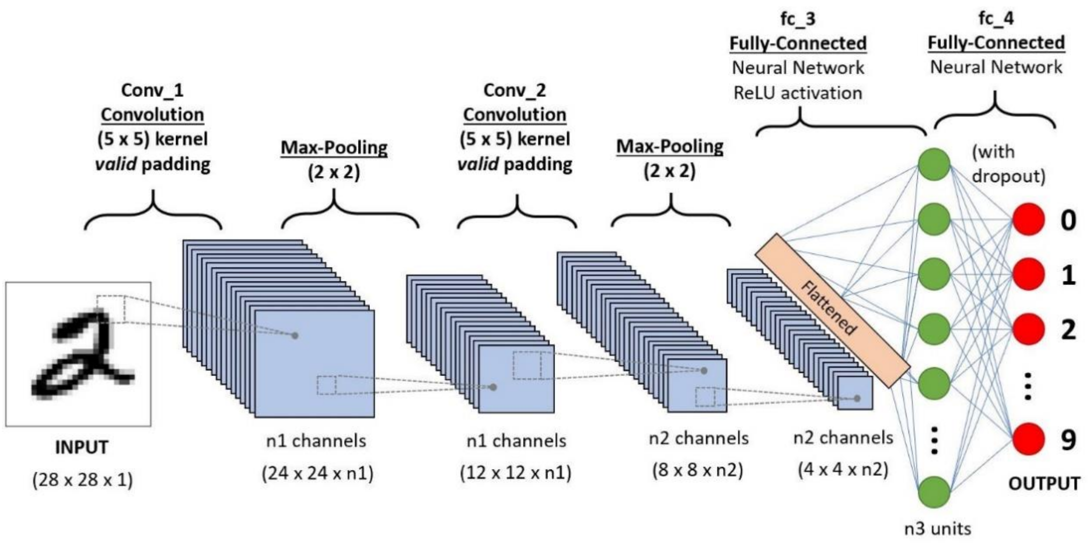
- In Convolutional Neural Networks (CNNs), stride refers to the number of pixels the convolutional kernel moves at each step during the convolution operation.
- Stride denotes how many steps we are moving in each step-in convolution. By default, it is one.
- A stride of 1 means that the kernel moves one pixel at a time, so the output feature map will have the same size as the input feature map. A stride of 2, on the other hand, means that the kernel moves two pixels at a time, resulting in an output feature map that is half the size of the input feature map.
- Using larger strides can result in a smaller output feature map, which can be useful for reducing the computational complexity of the network and increasing the receptive field (i.e., the area of the input that influences a single neuron). However, it can also lead to a loss of information, especially at the edges of the input feature map.
- Strides are often specified as a parameter in the convolutional layer of a CNN.
- For example, larger strides may be used in the early layers of a CNN to reduce the size of the feature map and capture coarse features, while smaller strides may be used in later layers to capture finer details.

The architecture of Convolution Neural Networks:

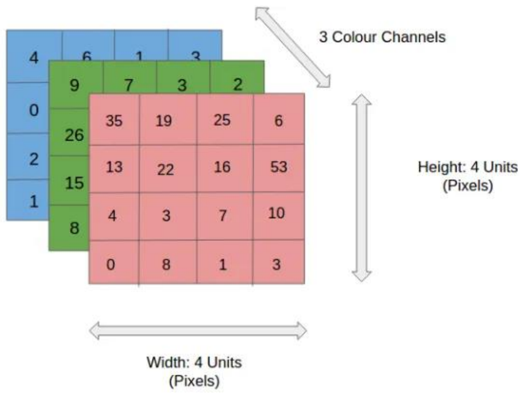
The job of a convolution neural network is to compress the images into a format that is easier to process while preserving the important elements for obtaining a decent prediction.

CNN Architecture is composed of 4 main blocks:

1. **Convolution layer:** A convolution layer that separates and identifies the various image features for analysis is called Feature Extraction. The network of feature extraction consists of many pairs of convolution and pooling layers.
2. **Fully connected layer :** A fully connected layer that utilizes the output from the convolution process and predicts the class of the image based on the extracted image features in previous stages.
3. **Pooling Layer :** This layer downsamples the feature maps generated by the convolutional layer, reducing the spatial dimensions of the feature maps and making the model more efficient.
4. **Dropout layer:** This layer randomly drops out some neurons during training to prevent overfitting.



Input Image



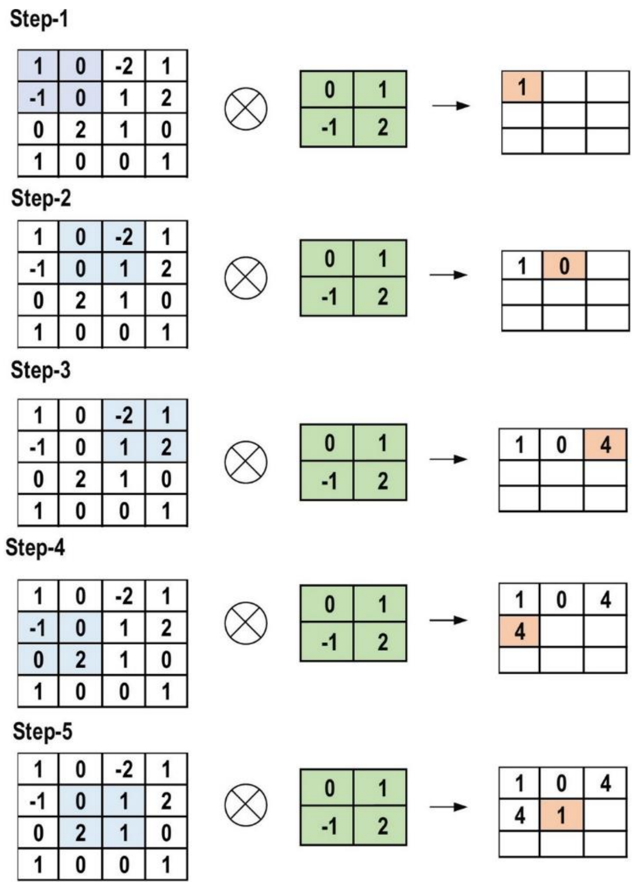
In the above figure, an RGB image has been separated by its three colors planes-red, green, and blue. There are several such color spaces like Grayscale, HSV, CMYK, etc.

Convolutional Layers

Three types of layers make up CNN which are convolutional layers, pooling layers, and fully-connected layers. When these layers are stacked, a CNN architecture is formed. There are two more important parameters which are the dropout layer and the activation function

Convolution layer: The first layer is used to extract the various features from input images. The mathematical operation of convolution is performed between the input image and the filter(kernel) of a particular size (n*n)

Consider an example of a 4x4 gray-scale image with a 2x2 kernel and the dot product as follows:



To generalize this if an m*m image is convoluted with an n*n kernel, the output image is of size (m-n+1) * (m+n-1)

There are two problems arise in convolution:

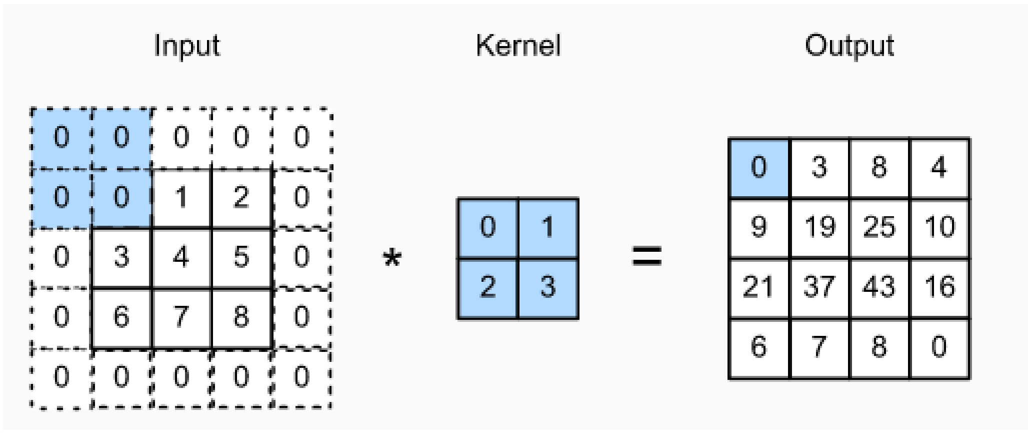
1. Every time after the convolution operation, the original image getting shrinks, and the original image will really get small. After multiple convolution operations, our original image will really get small but don't want the image to shrink every time.

2. Kernal mover original images, it touches the edge of the image less number of times and touches the middle of the image more times and overlaps in middle. So, the corner features of any image or edges aren't used much in output.

In order to solve those 2 issues, padding is introduced.

Padding

Padding preserves the size of the original image and also increases the size of the input image



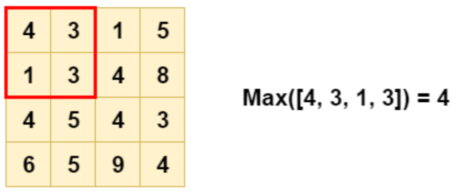
If an n*n matrix is convoluted with an f*f matrix with the padding p then the size of the output image will be (n+2p-f+1) * (n+2p-f+1)

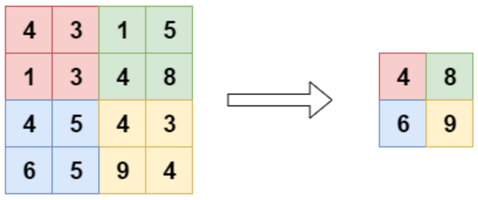
Pooling

A pooling layer is another building block of a CNN. Pooling reduces the spatial size of representation to reduce the network complexity and computational cost. It maintains the majority of the dominant information in every step of the pooling stage.
The main task of pooling is the sub-sampling of the output feature maps. This approach shrinks large-size feature maps to create smaller feature maps.
There are two types of pooling used in the CNN layer:

1. Max Pooling
2. Average Pooling

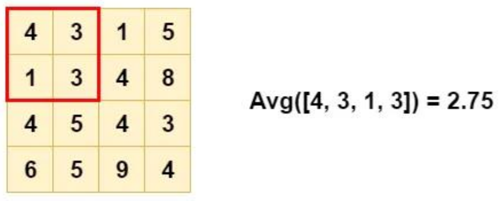
Max Pooling is to take the maximum of a region and it helps to proceed with the most important features of the image. Max pooling selects the brighter pixels from the image. It is useful when the background of the image is dark and we are interested in lighter pixels of the image. It also performs as a noise suppressant.

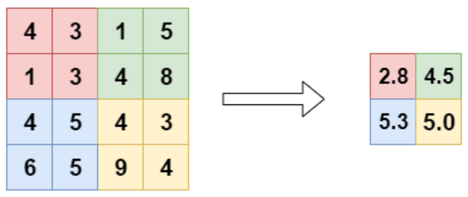




Average Pooling

It is different from max pooling that it retains much information about "less important " elements of a block or pool. Average pooling blends every value in it by calculating the average.





For every weight, the activation function is applied. Here are some activation functions as follows

Activation Function

Mapping the input to the output is the core function of all types of activation functions in all types of neural networks. The activation function makes the decision to include or exclude a neuron with reference to a particular input.

The activation functions which are most commonly used in CNN are:

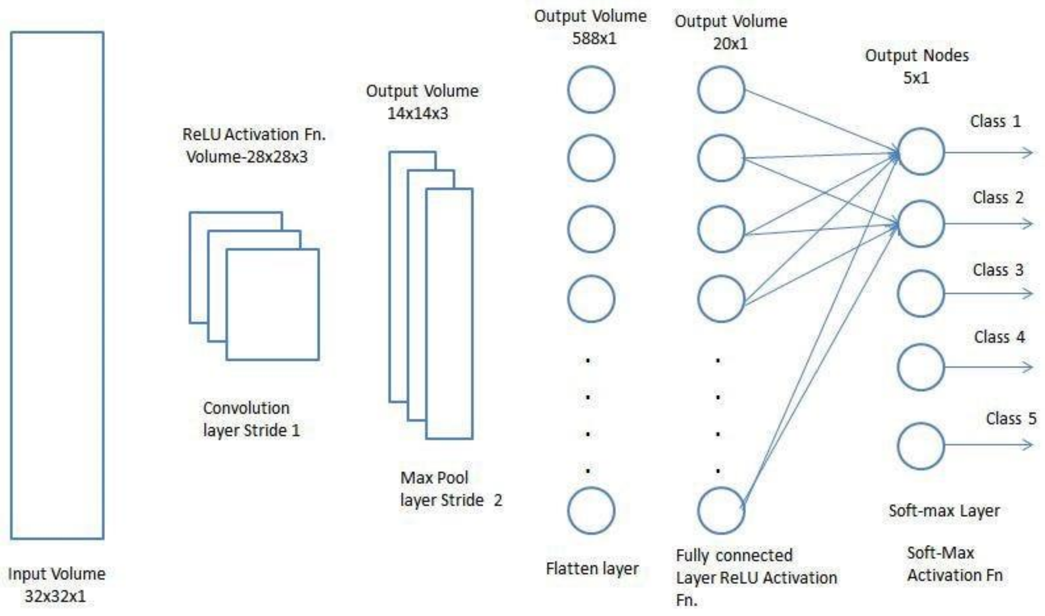
ReLU: The most commonly used function in CNN which converts the whole values of input to positive numbers.

$$f(x) = \max(0, x)$$

Though there are different activation functions, there are some least used activation functions are Sigmoid, Tanh, ReLU, Leaky ReLU

Fully Connected layer

This layer is located at the end of each CNN architecture. Inside this layer, each neuron is connected to all neurons of the previous layer. It is utilized as a CNN classifier and follows the basic method of a conventional multiple-layer perceptron neural network. The input of the FC layer comes from the last pooling or convolutional layer. This is in the form of a vector, which is created from the feature maps after flattening.



The flattened output is fed to a feed-forward neural network and backpropagation is applied to every iteration of training. After a series of epochs the model is suitable to distinguish between dominating and low-level features in images and classify them using the softmax classification technique.

There are various architectures of CNN available which have been key in building algorithms. Some of them including LeNet, AlexNet, VGGNet, GoogleNet, ResNet, ZFNet etc..

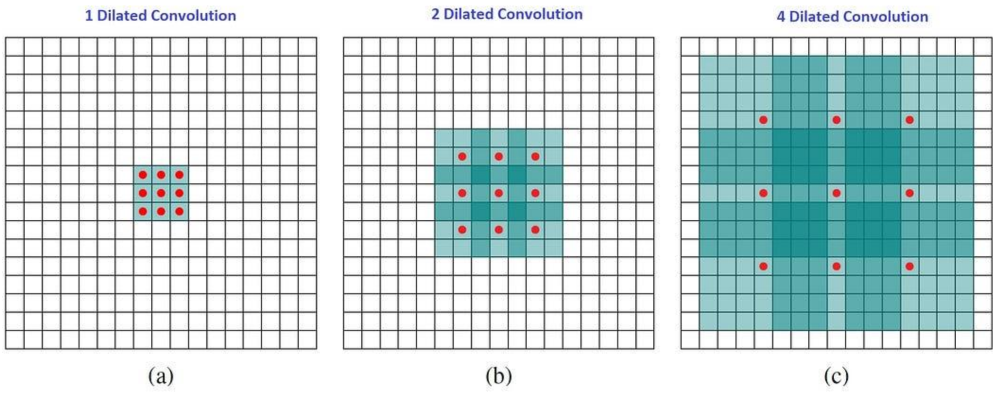
Variants of the basic convolution function:

The basic convolution function in a CNN is the operation of sliding a small filter over an input image or feature map to produce a new feature map. However, there are several variants of the basic convolution function that are commonly used in modern CNNs. Here are some examples:

- Dilated Convolution
- Transposed Convolution
- Depth-wise convolution
- Separable convolution
- Grouped convolution

Dilated convolution:

- A Dilated convolution is a convolution that operates on an input signal with gaps between the values of the signal. It is also known as an atrous convolution.
- In a dilated convolution, the filter is applied with spaces between the values, called dilation rate.
- The dilation rate determines the size of the gaps, and it is a hyperparameter that can be adjusted. When the dilation rate is 1, the dilated convolution reduces to a regular convolution.
- This results in a larger receptive field, allowing the network to capture larger patterns in the input image without increasing the number of parameters.
- Dilated convolution is often used in tasks where the input images have large spatial extents, such as semantic segmentation.
- In the image, the 3 x 3 red dots indicate that after the convolution, the output image is with 3 x 3 pixels. Although all three dilated convolutions provide the output with the same dimension, the receptive field observed by the model is dramatically different. The receptive filed is 3 x 3 for l =1. It is 7 x 7 for l =2. The receptive filed increases to 15 x 15 for l = 3. Interestingly, the numbers of parameters associated with these operations are essentially identical. We "observe" a large receptive filed without adding additional costs.



- The dilation rate effectively increases the receptive field of the filter without increasing the number of parameters, because the filter is still the same size, but with gaps between the values. This can be useful in situations where a larger receptive field is needed, but increasing the size of the filter would lead to an increase in the number of parameters and computational complexity.
- Formula for dilated Convolution:

$$(F *_l k)(\mathbf{p}) = \sum_{\mathbf{s}+l\mathbf{t}=\mathbf{p}} F(\mathbf{s})k(\mathbf{t})$$

where,
F(s) = Input
k(t) = Applied Filter
*l = l-dilated convolution
$(F*_l k)(p)$ = Output

- Here's an example of a 1D dilated convolution function with a dilation rate of 2:

```
import tensorflow as tf

# Input tensor of shape [batch_size, length, channels]
input_tensor = tf.random.normal([1, 10, 32])
```

```
# Dilated convolution layer with a filter size of 3 and dilation rate of 2
di_conv= tf.keras.layers.Conv1D(filters=64, kernel_size=3, dilation_rate=2) (input_tensor)

# Output tensor of shape [batch_size, length, filters]
print(di_conv.shape)
```

In this example, the dilated convolution operation applies a filter of size 3 to the input tensor, but with a dilation rate of 2, so there are gaps of size 2 between the values of the input tensor that the filter "sees". The resulting output tensor has a shape of [batch_size, length, filters], where "length" is the length of the input tensor, and "filters" is the number of filters applied to the input tensor.

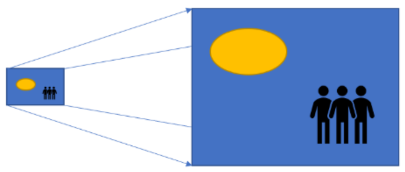
Some advantages of dilated convolutions are:

- Increased receptive field without increasing parameters
- Can capture features at multiple scales
- Reduced spatial resolution loss compared to regular convolutions with larger filters

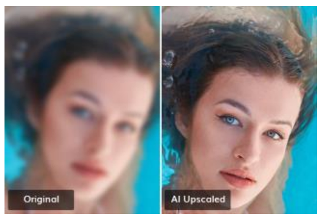
Some disadvantages of dilated convolutions are:

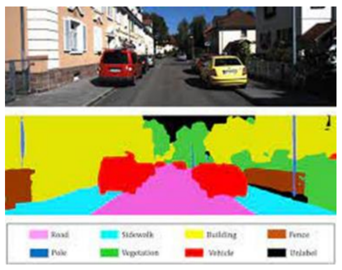
- Reduced spatial resolution in the output feature map compared to the input feature map
- Increased computational cost compared to regular convolutions with the same filter size and stride

Transposed convolution:

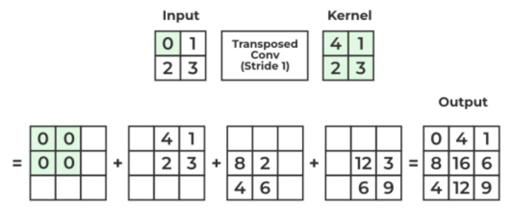


- A transposed convolution is used for tasks that involve upsampling the input data, such as converting a low-resolution image to a high-resolution one or generating an image from a set of noise vectors.
- It works by performing a convolution operation with a filter that has a larger stride than 1.
- While a standard convolution operation reduces the spatial resolution of an input signal, a transposed convolution operation increases the spatial resolution of an input signal by inserting zeros between the input values before applying the convolution operation.
- Transposed convolutional layers are used in a variety of tasks, including image generation, image super-resolution, and image segmentation.

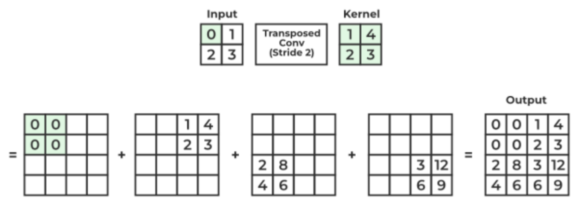




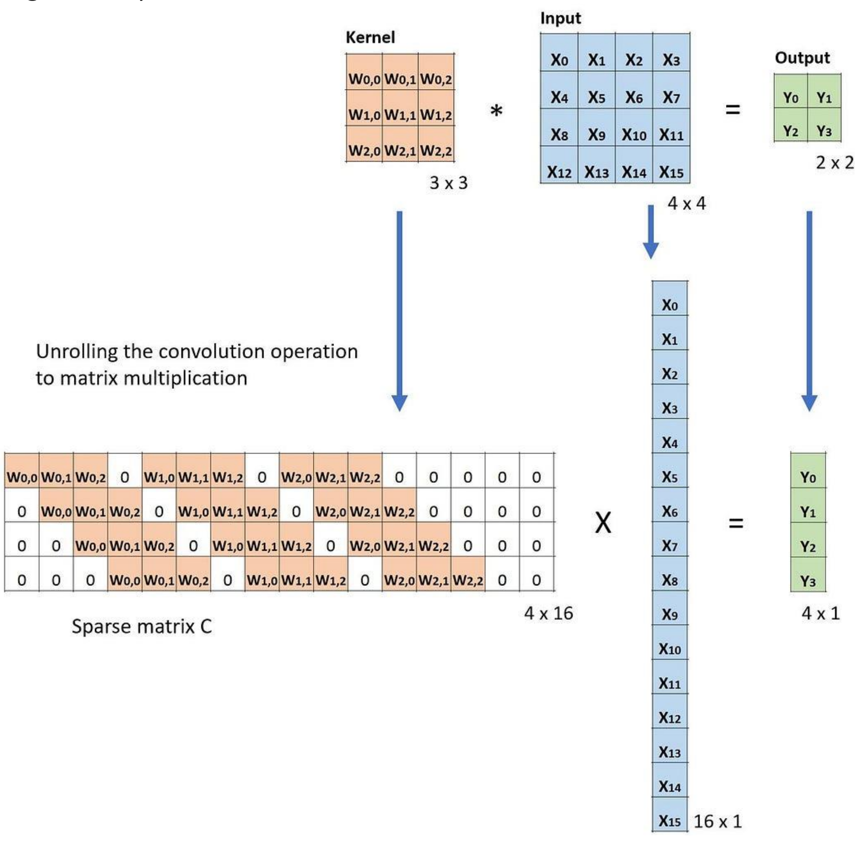
Example: Transposed convolution stride-1



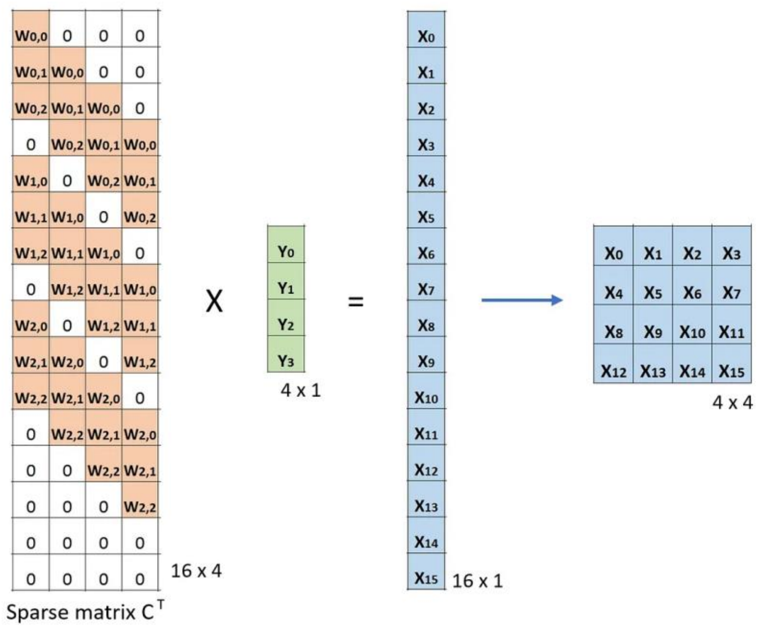
Example: Transposed convolution stride-2



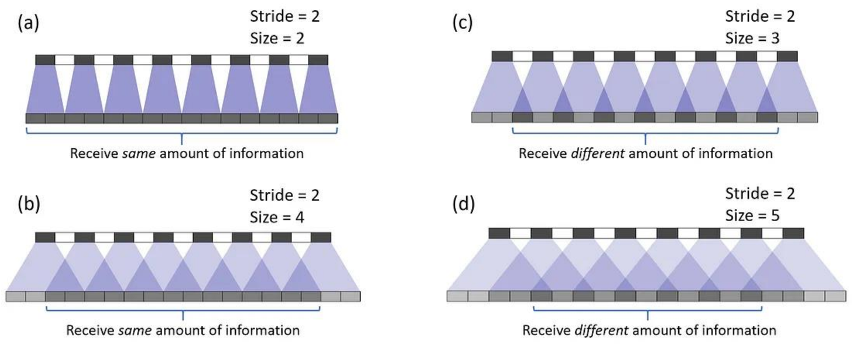
- Mathematical working of transpose convolution



- Apply transpose



- **Checkerboard artifacts**
  - The transposed convolution has uneven overlap when the filter size is not divisible by the stride.
  - This "uneven overlap" puts more of the paint in some places than others, thus creates the checkerboard effects.
  - In fact, the unevenly overlapped region tends to be more extreme in two dimensions.


(a)
Stride = 2
Size = 2
Receive same amount of information
(c)
Stride = 2
Size = 3
Receive different amount of information
(b)
Stride = 2
Size = 4
Receive same amount of information
(d)
Stride = 2
Size = 5
Receive different amount of information

- Here's an example of a 2D transposed convolution function with a stride of 2:

```
    import tensorflow as tf
# Input tensor of shape [batch_size, height, width, channels]
    input_tensor = tf.random.normal([1, 16, 16, 32])

# Transposed convolution layer with a filter size of 3, stride of 2, and "same" padding
    transposed_conv = tf.keras.layers.Conv2DTranspose(filters=64, kernel_size=3, strides=2,
    padding="same") (input_tensor)

# Output tensor of shape [batch_size, height * 2, width * 2, filters]
    print(transposed_conv.shape)
```

  In this example, the transposed convolution operation applies a filter of size 3 to the input tensor, but with a stride of 2, which results in the output tensor having a height and width twice that of the input tensor. The "same" padding ensures that the output size matches the input size. The resulting output tensor has a shape of [batch_size, height * 2, width * 2, filters], where "height" and "width" are the height and width of the input tensor, and "filters" is the number of filters applied to the input tensor.

- Advantages of Transposed convolution
  - **Upsampling:** Transpose convolution can be used to increase the spatial resolution of feature maps, which can be useful in tasks such as image segmentation or generating high-resolution images.
  - **Learning feature hierarchies:** By using transpose convolution layers, neural networks can learn feature hierarchies that are similar to those learned by convolutional layers. This can help improve the overall performance of the network.
  - **Flexibility:** Transpose convolution can be used with different kernel sizes and strides, making it a flexible tool for various upsampling tasks.
- Disadvantages of Transposed convolution
  - **Overfitting:** Transpose convolution can sometimes lead to overfitting, where the network learns to reproduce the input data instead of generalizing to new data.
  - **Checkerboard artifacts:** When using small kernel sizes and strides, transpose convolution can create checkerboard artifacts in the output, which can degrade the quality of the image.
  - **Computational complexity:** Transpose convolution can be computationally expensive, especially for large feature maps or when using larger kernel sizes and strides. This can increase training time and memory requirements.

Separable convolution:

A Separable Convolution is a process in which a single convolution can be divided into two or more convolutions to produce the same output. A single process is divided into two or more sub-processes to achieve the same effect.

Mainly there are two types of Separable Convolutions

- Spatially Separable Convolutions.
- Depth-wise Separable Convolutions.

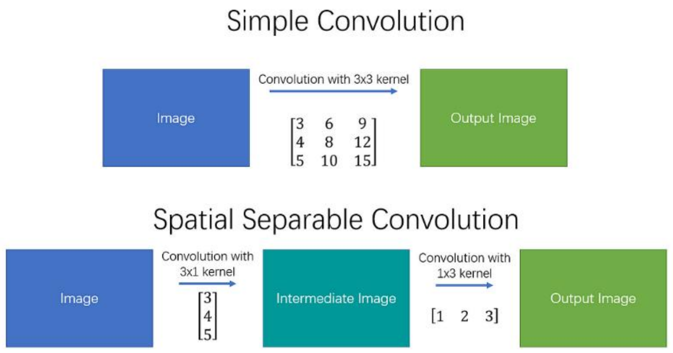
Spatially Separable Convolutions

- In images height and width are called spatial axes.
- The kernel that can be separated across spatial axes is called the spatially separable kernel.
- The kernel is broken into two smaller kernels and those kernels are multiplied sequentially with the input image to get the same effect of the full kernel.
- For an example shown below, a Sobel kernel, which is a 3x3 kernel, is divided into a 3x1 and 1x3 kernel.

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

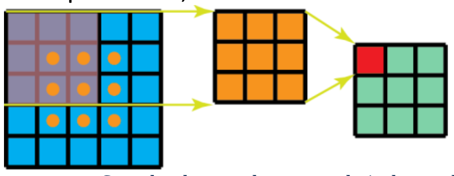
A Sobel kernel can be divided into a 3 x 1 and a 1 x 3 kernel

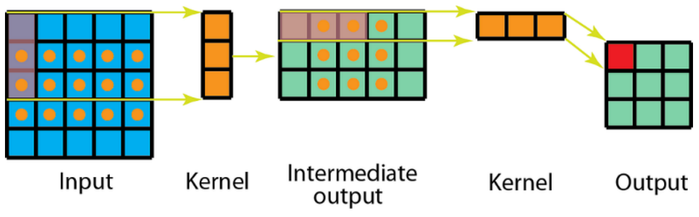
- In spatially separable convolution, the 3x1 kernel first convolves with the image. Then the 1x3 kernel is applied. This would require 6 instead of 9 parameters while doing the same operations.


Simple Convolution
Image
Convolution with 3x3 kernel
Output Image
Spatial Separable Convolution
Image
Convolution with 3x1 kernel
Intermediate Image
Convolution with 1x3 kernel
Output Image

$$\begin{bmatrix} 3 & 6 & 9 \\ 4 & 8 & 12 \\ 5 & 10 & 15 \end{bmatrix} \qquad \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

- Moreover, one need less matrix multiplications in spatially separable convolution than convolution.
- Example: First apply a 3 x 1 filter on the 5 x 5 image. We scan such kernel at 5 positions horizontally and 3 positions vertically. That's 5 x 3 = 15 positions in total, indicated as dots on the image below. At each position, 3 element-wise multiplications are applied. That is 15 x 3 = 45 multiplications. We now obtained a 3 x 5 matrix. This matrix is now convolved with a 1 x 3 kernel, which scans the matrix at 3 positions horizontally and 3 positions vertically. For each of these 9 positions, 3 element-wise multiplications are applied. This step requires 9 x 3 = 27 multiplications. Thus, overall, the spatially separable convolution takes 45 + 27 = 72 multiplications, which is less than convolution.

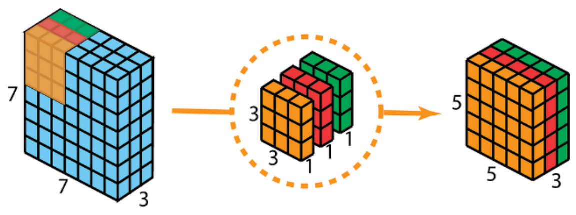

Standard convolution with 1 channel



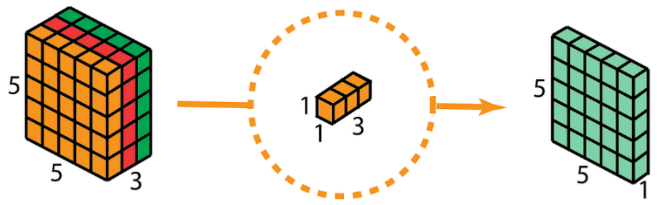
Spatially separable convolution with 1 channel

Depthwise Separable Convolutions:

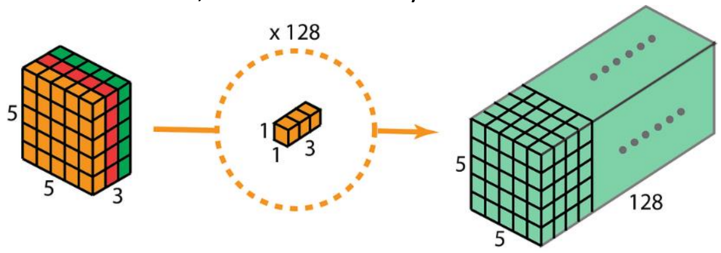
- The depth wise separable convolutions consist of two steps: depthwise convolutions and 1x1 convolutions.
- In a depthwise convolution, each filter of the convolutional layer is applied to a single input channel separately. This means that if the input has n channels and the layer has m filters, then there will be n*m convolutions. This step is called depthwise convolution because it applies a filter to the depth dimension (i.e., the channel dimension) of the input.
- After the depthwise convolution, a pointwise convolution is applied to combine the outputs of the depthwise convolution. A pointwise convolution is a regular convolution with 1x1 filters, which means that it only acts on the spatial dimensions (height and width) of the input.
- First, we apply depthwise convolution to the input layer. Each filter has size 3 x 3 x 1. Each kernel convolves with 1 channel of the input layer (1 channel only, not all channels!). Each of such convolution provides a map of size 5 x 5 x 1. We then stack these maps together to create a 5 x 5 x 3 image. We now shrink the spatial dimensions, but the depth is still the same as before.



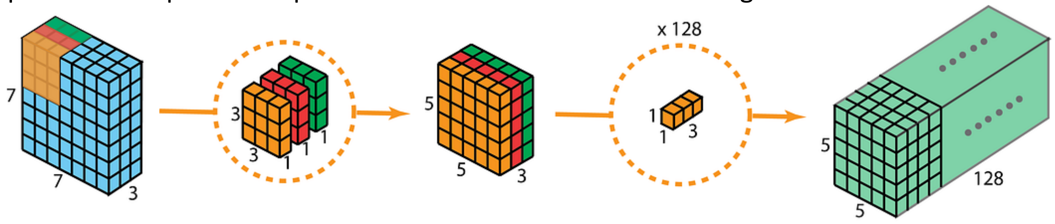
- As the second step of depthwise separable convolution, to extend the depth, we apply the 1x1 convolution with kernel size 1x1x3. Convolving the 5 x 5 x 3 input image with each 1 x 1 x 3 kernel provides a map of size 5 x 5 x 1.



- Thus, after applying 128 1x1 convolutions, we can have a layer with size 5 x 5 x 128.



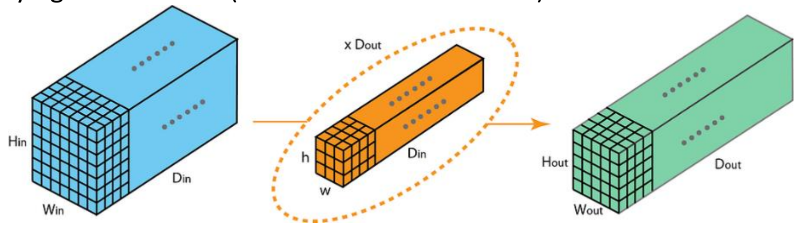
- With these two steps, depthwise separable convolution also transform the input layer (7 x 7 x 3) into the output layer (5 x 5 x 128).
- The overall process of depthwise separable convolution is shown in the figure below.



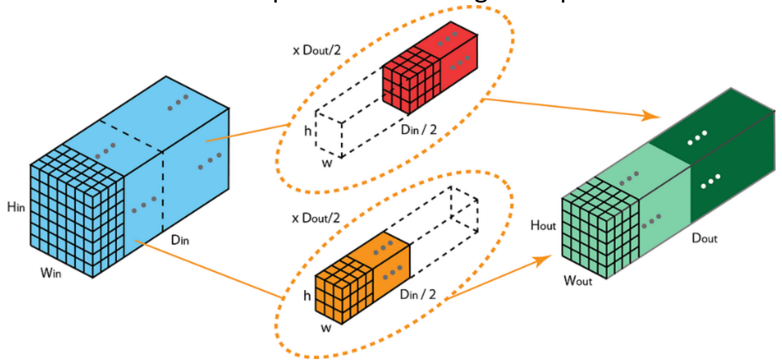
- Note: Needs much less operations for depthwise separable convolutions compared to 2D convolutions.

Grouped convolution:

- In a grouped convolution, the input channels are divided into groups, and each group is convolved with a separate filter. This allows the network to capture different types of features separately, improving performance while reducing the number of parameters.
- Grouped convolution is commonly used in large-scale image classification tasks, such as the ImageNet dataset.
- **Conventional 2D convolutions** follow the steps showing below. In this example, the input layer of size (7 x 7 x 3) is transformed into the output layer of size (5 x 5 x 128) by applying 128 filters (each filter is of size 3 x 3 x 3). Or in general case, the input layer of size (Hin x Win x Din) is transformed into the output layer of size (Hout x Wout x Dout) by applying Dout kernels (each is of size h x w x Din).



- In grouped convolution, the filters are separated into different groups. Each group is responsible for a conventional 2D convolutions with certain depth. The following examples can make this clearer.



- Above example is a grouped convolution with 2 filter groups. In each filter group, the depth of each filter is only half of the that in the nominal 2D convolutions. They are of depth Din / 2. Each filter group contains Dout /2 filters. The first filter group (red) convolves with the first half of the input layer ([:, :, 0:Din/2]), while the second filter group (blue) convolves with the second half of the input layer ([:, :, Din/2:Din]). As a result, each filter group creates Dout/2 channels. Overall, two groups create 2 x Dout/2 = Dout channels. We then stack these channels in the output layer with Dout channels.

Building a CNN for image classification

Example 1: (simple)

```
import tensorflow.keras as keras
import pandas as pd

# Load in our data from CSV files
train_df = pd.read_csv("asl_data/sign_mnist_train.csv")
valid_df = pd.read_csv("asl_data/sign_mnist_valid.csv")

# Separate out our target values
y_train = train_df['label']
y_valid = valid_df['label']
del train_df['label']
del valid_df['label']

# Separate out our image vectors
x_train = train_df.values
x_valid = valid_df.values

# Turn our scalar targets into binary categories
num_classes = 24
y_train = keras.utils.to_categorical(y_train, num_classes)
y_valid = keras.utils.to_categorical(y_valid, num_classes)

# Normalize our image data
x_train = x_train / 255
x_valid = x_valid / 255

# Reshaping Images for a CNN
x_train.shape, x_valid.shape
            o/p: ((27455, 784), (7172, 784))
x_train = x_train.reshape(-1,28,28,1)
x_valid = x_valid.reshape(-1,28,28,1)

#  Creating a Convolutional Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (
  Dense,
  Conv2D,
  MaxPool2D,
  Flatten,
  Dropout,
  BatchNormalization,
)

model = Sequential()
model.add(Conv2D(75, (3, 3), strides=1, padding="same", activation="relu",
          input_shape=(28, 28, 1)))
model.add(BatchNormalization())
model.add(MaxPool2D((2, 2), strides=2, padding="same"))
model.add(Conv2D(50, (3, 3), strides=1, padding="same", activation="relu"))
model.add(Dropout(0.2))
model.add(BatchNormalization())
model.add(MaxPool2D((2, 2), strides=2, padding="same"))
model.add(Conv2D(25, (3, 3), strides=1, padding="same", activation="relu"))
model.add(BatchNormalization())
model.add(MaxPool2D((2, 2), strides=2, padding="same"))
model.add(Flatten())
model.add(Dense(units=512, activation="relu"))
model.add(Dropout(0.3))
model.add(Dense(units=num_classes, activation="softmax"))

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 75)        750
_________________________________________________________________
batch_normalization (BatchNo (None, 28, 28, 75)        300
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 75)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 50)        33800
_________________________________________________________________
dropout (Dropout)            (None, 14, 14, 50)        0
_________________________________________________________________
batch_normalization_1 (Batch (None, 14, 14, 50)        200
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 50)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 7, 7, 25)          11275
_________________________________________________________________
batch_normalization_2 (Batch (None, 7, 7, 25)          100
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 25)          0
_________________________________________________________________
flatten (Flatten)            (None, 400)               0
_________________________________________________________________
dense (Dense)                (None, 512)               205312
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 24)                12312
=================================================================
Total params: 264,049
Trainable params: 263,749
Non-trainable params: 300
_________________________
```

```
# Compiling the Model
        model.compile(loss="categorical_crossentropy", metrics=["accuracy"])
# Training the Model
        model.fit(x_train, y_train, epochs=20, verbose=1, validation_data=(x_valid, y_valid))
```

Example 2:

```
# Importing packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, MaxPool2D, Flatten, Dropout, BatchNormalization
from tensorflow.keras.callbacks import EarlyStopping
  from tensorflow.keras.preprocessing.image import ImageDataGenerator

from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import classification_report, confusion_matrix
```

```
# The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There  are 50000
training images and 10000 test images
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

# Define the labels of the dataset
labels = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

#Data Preprocessing
# Scale the data
X_train = X_train / 255.0
X_test = X_test / 255.0

# Transform target variable into one-hotencoding
y_cat_train = to_categorical(y_train, 10)
y_cat_test = to_categorical(y_test, 10)

#model building
INPUT_SHAPE = (32, 32, 3)
KERNEL_SIZE = (3, 3)
model = Sequential()
# Convolutional Layer
#layer-1
model.add(Conv2D(filters=32, kernel_size=KERNEL_SIZE, input_shape=INPUT_SHAPE, activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(filters=32, kernel_size=KERNEL_SIZE, input_shape=INPUT_SHAPE, activation='relu', padding='same'))
model.add(BatchNormalization())
# Pooling layer
model.add(MaxPool2D(pool_size=(2, 2)))
# Dropout layers
model.add(Dropout(0.25))
#layer-2
model.add(Conv2D(filters=64, kernel_size=KERNEL_SIZE, input_shape=INPUT_SHAPE, activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(filters=64, kernel_size=KERNEL_SIZE, input_shape=INPUT_SHAPE, activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
#layer-3
model.add(Conv2D(filters=128, kernel_size=KERNEL_SIZE, input_shape=INPUT_SHAPE, activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(filters=128, kernel_size=KERNEL_SIZE, input_shape=INPUT_SHAPE, activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
#flatten
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(10, activation='softmax'))

METRICS = ['accuracy',tf.keras.metrics.Precision(name='precision'),tf.keras.metrics.Recall(name='recall')]
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=METRICS)
model.summary()

#Early Stopping(for reduce over fitting)
early_stop = EarlyStopping(monitor='val_loss', patience=2)

#Data Augmentations
batch_size = 32
```

```
data_generator = ImageDataGenerator(width_shift_range=0.1, height_shift_range=0.1, horizontal_flip=True)
train_generator = data_generator.flow(X_train, y_cat_train, batch_size)
steps_per_epoch = X_train.shape[0] // batch_size

r = model.fit(train_generator,
          epochs=50,
          steps_per_epoch=steps_per_epoch,
          validation_data=(X_test, y_cat_test),
          callbacks=[early_stop],
          batch_size=batch_size,
         )
evaluation = model.evaluate(X_test, y_cat_test)
print(f'Test Accuracy : {evaluation[1] * 100:.2f}%')
```

Applications of CNN:

- **Image Classification:** CNNs can be used to classify images into different categories. For example, a CNN could be trained to identify whether an image contains a cat or a dog.
- **Object Detection:** CNNs can also be used to detect objects within images and videos. This is a critical component of many applications, including self-driving cars, robotics, and security systems.
- **Face Recognition:** CNNs can be used to recognize faces within images and videos. This technology is used in security systems, as well as in social media platforms to tag people in photos.
- **Medical Image Analysis:** CNNs can be used to analyze medical images, such asCT scans and X-rays, to assist doctors in making diagnoses.
- **Natural Language Processing:** CNNs can be used to process text data, suchas sentiment analysis and language translation.
- **Video Analysis:** CNNs can be used to analyze video data, such as detecting motion, recognizing gestures, and identifying objects.
- **Image Captioning:** CNNs can be used to generate captions for images, which can be useful in applications such as accessibility technology for the visually impaired.
- **Style Transfer:** CNNs can be used to apply the artistic style of one image to another image, creating unique visual effects.
- **Autonomous Vehicles:** CNNs are an essential component of self-driving cars. They can be used for object detection, lane recognition, and to assist with decision-making while driving.
- **Image Super-Resolution:** CNNs can be used to improve the quality of low-resolutionimages by enhancing details and increasing resolution.