**DEEP LEARNING**

**Convolution Neural Network :**

Convolutional Neural Network (CNN) is the extended version of [artificial neural networks (ANN)](https://www.geeksforgeeks.org/artificial-neural-networks-and-its-applications/) which is predominantly used to extract the feature from the grid-like matrix dataset. For example visual datasets like images or videos where data patterns play an extensive role.

**Convolutional Layer:**

In a CNN, the convolutional layer is a fundamental building block that performs feature extraction by applying filters or kernels to input data. The key idea is to slide these filters across the input data to detect patterns such as edges, textures, and more complex features.

Operation:

Filter/Kernel: A small matrix that slides over the input data.

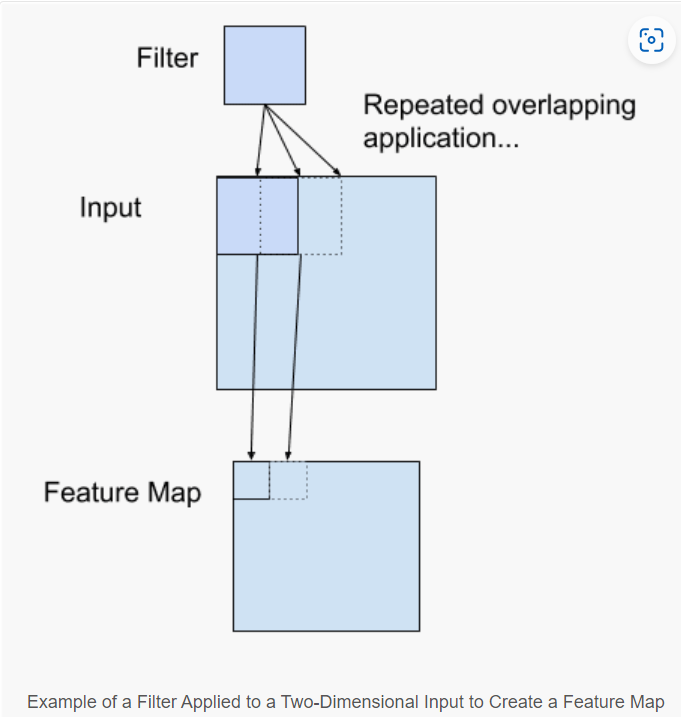
Convolution: Element-wise multiplication of the filter with the input, followed by summing the results.

Stride: The step size with which the filter moves across the input.

Feature Map: The output obtained after the convolution operation.

**Convolution in Convolutional Neural Networks**

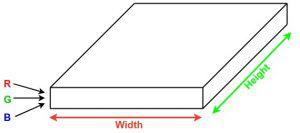
* The convolutional neural network, or CNN for short, is a specialized type of neural network model designed for working with two-dimensional image data, although they can be used with one-dimensional and three-dimensional data.
* Central to the convolutional neural network is the convolutional layer that gives the network its name. This layer performs an operation called a “*convolution*“.
* In the context of a convolutional neural network, a convolution is a linear operation that involves the multiplication of a set of weights with the input, much like a traditional neural network. Given that the technique was designed for two-dimensional input, the multiplication is performed between an array of input data and a two-dimensional array of weights, called a filter or a kernel.
* The filter is smaller than the input data and the type of multiplication applied between a filter-sized patch of the input and the filter is a dot product. A [dot product](https://en.wikipedia.org/wiki/Dot_product) is the element-wise multiplication between the filter-sized patch of the input and filter, which is then summed, always resulting in a single value. Because it results in a single value, the operation is often referred to as the “*scalar product*“.
* Using a filter smaller than the input is intentional as it allows the same filter (set of weights) to be multiplied by the input array multiple times at different points on the input. Specifically, the filter is applied systematically to each overlapping part or filter-sized patch of the input data, left to right, top to bottom.
* This systematic application of the same filter across an image is a powerful idea. If the filter is designed to detect a specific type of feature in the input, then the application of that filter systematically across the entire input image allows the filter an opportunity to discover that feature anywhere in the image. This capability is commonly referred to as translation invariance, e.g. the general interest in whether the feature is present rather than where it was present.
* *Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is. For example, when determining whether an image contains a face, we need not know the location of the eyes with pixel-perfect accuracy, we just need to know that there is an eye on the left side of the face and an eye on the right side of the face.*
* The output from multiplying the filter with the input array one time is a single value. As the filter is applied multiple times to the input array, the result is a two-dimensional array of output values that represent a filtering of the input. As such, the two-dimensional output array from this operation is called a “*feature map*“.
* Once a feature map is created, we can pass each value in the feature map through a nonlinearity, such as a ReLU, much like we do for the outputs of a fully connected layer



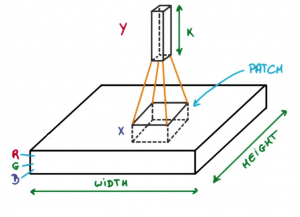
* If you come from a digital signal processing field or related area of mathematics, you may understand the convolution operation on a matrix as something different. Specifically, the filter (kernel) is flipped prior to being applied to the input. Technically, the convolution as described in the use of convolutional neural networks is actually a “*cross-correlation”*. Nevertheless, in deep learning, it is referred to as a “*convolution*” operation.
* *Many machine learning libraries implement cross-correlation but call it convolution.*In summary, we have a *input*, such as an image of pixel values, and we have a *filter*, which is a set of weights, and the filter is systematically applied to the input data to create a *feature map*

**How Convolutional Layers works :**

Convolution Neural Networks or covnets are neural networks that share their parameters. Imagine you have an image. It can be represented as a cuboid having its length, width (dimension of the image), and height (i.e the channel as images generally have red, green, and blue channels).



Now imagine taking a small patch of this image and running a small neural network, called a filter or kernel on it, with say, K outputs and representing them vertically. Now slide that neural network across the whole image, as a result, we will get another image with different widths, heights, and depths. Instead of just R, G, and B channels now we have more channels but lesser width and height. This operation is called **Convolution**. If the patch size is the same as that of the image it will be a regular neural network. Because of this small patch, we have fewer weights.



*Image source: Deep Learning Udacity*

Now let’s talk about a bit of mathematics that is involved in the whole convolution process.

* Convolution layers consist of a set of learnable filters (or kernels) having small widths and heights and the same depth as that of input volume (3 if the input layer is image input).
* For example, if we have to run convolution on an image with dimensions 34x34x3. The possible size of filters can be axax3, where ‘a’ can be anything like 3, 5, or 7 but smaller as compared to the image dimension.
* During the forward pass, we slide each filter across the whole input volume step by step where each step is called [**stride**](https://www.geeksforgeeks.org/ml-introduction-to-strided-convolutions/) (which can have a value of 2, 3, or even 4 for high-dimensional images) and compute the dot product between the kernel weights and patch from input volume.
* As we slide our filters we’ll get a 2-D output for each filter and we’ll stack them together as a result, we’ll get output volume having a depth equal to the number of filters. The network will learn all the filters.

**Example:**

[1, 0, 1, 0, 1]

[0, 1, 0, 1, 0]

[1, 0, 1, 0, 1]

[0, 1, 0, 1, 0]

[1, 0, 1, 0, 1]

**Filter:**

[1, 0, -1]

[1, 0, -1]

[1, 0, -1]

**Convolution (Stride = 1):**

[ 1\*1+0\*0-1\*1, 1\*0+0\*1-1\*0, 1\*1+0\*0-1\*1, 1\*0+0\*1-1\*0 ]

[ 0\*1+1\*0-0\*1, 0\*0+1\*1-0\*0, 0\*1+1\*0-0\*1, 0\*0+1\*1-0\*0 ]

[ 1\*1+0\*0-1\*1, 1\*0+0\*1-1\*0, 1\*1+0\*0-1\*1, 1\*0+0\*1-1\*0 ]

[ 0\*1+1\*0-0\*1, 0\*0+1\*1-0\*0, 0\*1+1\*0-0\*1, 0\*0+1\*1-0\*0 ]

**Resultant Feature Map:**

[ 0, -1, 0, -1 ]

[ 1, 0, 1, 0 ]

[ 0, -1, 0, -1 ]

[ 1, 0, 1, 0 ]

**Advantages of Convolutional Layers:\***

**1. Feature Learning**:

- Convolutional layers automatically learn hierarchical features from the input data, capturing low- level details to high-level patterns.

**2. Parameter Sharing:**

- Weight sharing in convolutional layers reduces the number of parameters, making the network easier to train and more computationally efficient.

**3. Translation Invariance:**

- Convolutional layers are able to recognize features in different spatial locations, providing a degree of translation invariance, which is beneficial for tasks like image recognition.

**4. Sparse Connectivity:**

- Neurons in convolutional layers are connected to only a small local region in the input, reducing the computational load and enhancing the ability to capture local patterns.

**5. Handling Spatial Hierarchies:**

- Convolutional layers are effective in capturing spatial hierarchies, recognizing patterns at different scales and resolutions.

**Disadvantages of Convolutional Layers:**

**1. Fixed Receptive Field:**

- The receptive field of a neuron in a convolutional layer is fixed, which might limit its ability to capture long-range dependencies in some cases.

**2. Computational Intensity:**

- Deep convolutional networks can be computationally intensive, especially with increasing model complexity and larger input sizes, making training and inference resource-demanding.

**3. Limited Understanding of Global Context:**

- Convolutional layers focus on local patterns, which might result in a limited understanding of the global context in certain tasks where a broader perspective is crucial.

**4. Pooling Information Loss:**

- Pooling layers used in conjunction with convolutional layers may lead to loss of spatial information, as they downsample the input by selecting maximum or average values.

**5. Manual Tuning of Hyperparameters:**

- Convolutional networks often require careful tuning of hyperparameters, such as kernel size, stride, and pooling size, to achieve optimal performance, which can be time-consuming.

**Pooling Layer:**

The pooling operation involves sliding a two-dimensional filter over each channel of feature map and summarising the features lying within the region covered by the filter.   
For a feature map having dimensions **nh x nw x nc**, the dimensions of output obtained after a pooling layer is 

(nh - f + 1) / s x (nw - f + 1)/s x nc

where,

-> **nh -** height of feature map

-> **nw -** width of feature map

-> **nc -** number of channels in the feature map

-> **f -** size of filter

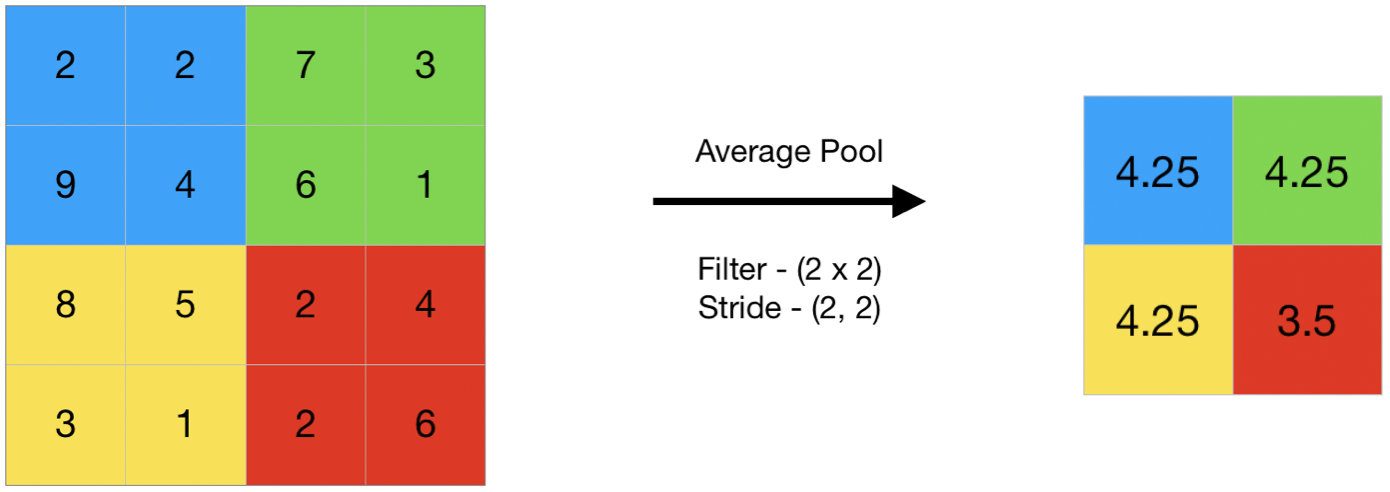
-> **s -** stride length

A common CNN model architecture is to have a number of convolution and pooling layers stacked one after the other.

**Why to use Pooling Layers?**

* Pooling layers are used to reduce the dimensions of the feature maps. Thus, it reduces the number of parameters to learn and the amount of computation performed in the network.
* The pooling layer summarises the features present in a region of the feature map generated by a convolution layer. So, further operations are performed on summarised features instead of precisely positioned features generated by the convolution layer. This makes the model more robust to variations in the position of the features in the input image.

**Types of Pooling Layers:  
   
1.Max Pooling**

1. Max pooling is a pooling operation that selects the maximum element from the region of the feature map covered by the filter. Thus, the output after max-pooling layer would be a feature map containing the most prominent features of the previous feature map.   
    
2. This can be achieved using MaxPooling2D layer in keras as follows:  
   **Code #1 : Performing Max Pooling using keras**

|  |
| --- |
| **import** numpy as np  **from** keras.models **import** Sequential  **from** keras.layers **import** MaxPooling2D    # define input image  image **=** np.array([[2, 2, 7, 3],                    [9, 4, 6, 1],                    [8, 5, 2, 4],                    [3, 1, 2, 6]])  image **=** image.reshape(1, 4, 4, 1)    # define model containing just a single max pooling layer  model **=** Sequential(      [MaxPooling2D(pool\_size **=** 2, strides **=** 2)])    # generate pooled output  output **=** model.predict(image)    # print output image  output **=** np.squeeze(output)  print(output) |

1. **Output:**

[[9. 7.]

[8. 6.]]

**2.Average Pooling :**

1. Average pooling computes the average of the elements present in the region of feature map covered by the filter. Thus, while max pooling gives the most prominent feature in a particular patch of the feature map, average pooling gives the average of features present in a patch.

**Code #2 : Performing Average Pooling using keras**

|  |
| --- |
| **import** numpy as np  **from** keras.models **import** Sequential  **from** keras.layers **import** AveragePooling2D    # define input image  image **=** np.array([[2, 2, 7, 3],                    [9, 4, 6, 1],                    [8, 5, 2, 4],                    [3, 1, 2, 6]])  image **=** image.reshape(1, 4, 4, 1)    # define model containing just a single average pooling layer  model **=** Sequential(      [AveragePooling2D(pool\_size **=** 2, strides **=** 2)])    # generate pooled output  output **=** model.predict(image)    # print output image  output **=** np.squeeze(output)  print(output) |

1. **Output:**

[[4.25 4.25]

[4.25 3.5 ]]

**3.Global Pooling :**

1. Global pooling reduces each channel in the feature map to a single value. Thus, an **nh x nw x nc** feature map is reduced to **1 x 1 x nc** feature map. This is equivalent to using a filter of dimensions **nh x nw** i.e. the dimensions of the feature map.   
   Further, it can be either global max pooling or global average pooling.

**Code #3 : Performing Global Pooling using keras**

|  |
| --- |
| **import** numpy as np  **from** keras.models **import** Sequential  **from** keras.layers **import** GlobalMaxPooling2D  **from** keras.layers **import** GlobalAveragePooling2D    # define input image  image **=** np.array([[2, 2, 7, 3],                    [9, 4, 6, 1],                    [8, 5, 2, 4],                    [3, 1, 2, 6]])  image **=** image.reshape(1, 4, 4, 1)    # define gm\_model containing just a single global-max pooling layer  gm\_model **=** Sequential(      [GlobalMaxPooling2D()])    # define ga\_model containing just a single global-average pooling layer  ga\_model **=** Sequential(      [GlobalAveragePooling2D()])    # generate pooled output  gm\_output **=** gm\_model.predict(image)  ga\_output **=** ga\_model.predict(image)    # print output image  gm\_output **=** np.squeeze(gm\_output)  ga\_output **=** np.squeeze(ga\_output)  print("gm\_output: ", gm\_output)  print("ga\_output: ", ga\_output) |

1. **Output:**

gm\_output: 9.0

ga\_output: 4.0625

In convolutional neural networks (CNNs), the pooling layer is a common type of layer that is typically added after convolutional layers. 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).

1. The pooling layer works by dividing the input feature map into a set of non-overlapping regions, called pooling regions. Each pooling region is then transformed into a single output value, which represents the presence of a particular feature in that region. The most common types of pooling operations are max pooling and average pooling.
2. In max pooling, the output value for each pooling region is simply the maximum value of the input values within that region. This has the effect of preserving the most salient features in each pooling region, while discarding less relevant information. Max pooling is often used in CNNs for object recognition tasks, as it helps to identify the most distinctive features of an object, such as its edges and corners.
3. In average pooling, the output value for each pooling region is the average of the input values within that region. This has the effect of preserving more information than max pooling, but may also dilute the most salient features. Average pooling is often used in CNNs for tasks such as image segmentation and object detection, where a more fine-grained representation of the input is required.

Pooling layers are typically used in conjunction with convolutional layers in a CNN, with each pooling layer reducing the spatial dimensions of the feature maps, while the convolutional layers extract increasingly complex features from the input. The resulting feature maps are then passed to a fully connected layer, which performs the final classification or regression task.

**Advantages of Pooling Layer:**

1. Dimensionality reduction: The main advantage of pooling layers is that they help in reducing the spatial dimensions of the feature maps. This reduces the computational cost and also helps in avoiding overfitting by reducing the number of parameters in the model.
2. Translation invariance: Pooling layers are also useful in achieving translation invariance in the feature maps. 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.
3. Feature selection: Pooling layers can also help in selecting the most important features from the input, as max pooling selects the most salient features and average pooling preserves more information.

**Disadvantages of Pooling Layer:**

1. 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.
2. 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.
3. 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.

**Code for Convolution and Pooling Layers:**

|  |
| --- |
| 1. import the necessary libraries 2. **import** numpy as np 3. **import** tensorflow as tf 4. **import** matplotlib.pyplot as plt 5. **from** itertools **import** product 7. # set the param 8. plt.rc('figure', autolayout**=**True) 9. plt.rc('image', cmap**=**'magma') 11. # define the kernel 12. kernel **=** tf.constant([[**-**1, **-**1, **-**1], 13. [**-**1,  8, **-**1], 14. [**-**1, **-**1, **-**1], 15. ]) 17. # load the image 18. image **=** tf.io.read\_file('Ganesh.jpg') 19. image **=** tf.io.decode\_jpeg(image, channels**=**1) 20. image **=** tf.image.resize(image, size**=**[300, 300]) 22. # plot the image 23. img **=** tf.squeeze(image).numpy() 24. plt.figure(figsize**=**(5, 5)) 25. plt.imshow(img, cmap**=**'gray') 26. plt.axis('off') 27. plt.title('Original Gray Scale image') 28. plt.show();  31. # Reformat 32. image **=** tf.image.convert\_image\_dtype(image, dtype**=**tf.float32) 33. image **=** tf.expand\_dims(image, axis**=**0) 34. kernel **=** tf.reshape(kernel, [**\***kernel.shape, 1, 1]) 35. kernel **=** tf.cast(kernel, dtype**=**tf.float32) 37. # convolution layer 38. conv\_fn **=** tf.nn.conv2d 40. image\_filter **=** conv\_fn( 41. input**=**image, 42. filters**=**kernel, 43. strides**=**1, # or (1, 1) 44. padding**=**'SAME', 45. ) 47. plt.figure(figsize**=**(15, 5)) 49. # Plot the convolved image 50. plt.subplot(1, 3, 1) 52. plt.imshow( 53. tf.squeeze(image\_filter) 54. ) 55. plt.axis('off') 56. plt.title('Convolution') 58. # activation layer 59. relu\_fn **=** tf.nn.relu 60. # Image detection 61. image\_detect **=** relu\_fn(image\_filter) 63. plt.subplot(1, 3, 2) 64. plt.imshow( 65. # Reformat for plotting 66. tf.squeeze(image\_detect) 67. ) 69. plt.axis('off') 70. plt.title('Activation') 72. # Pooling layer 73. pool **=** tf.nn.pool 74. image\_condense **=** pool(input**=**image\_detect, 75. window\_shape**=**(2, 2), 76. pooling\_type**=**'MAX', 77. strides**=**(2, 2), 78. padding**=**'SAME', 79. ) 81. plt.subplot(1, 3, 3) 82. plt.imshow(tf.squeeze(image\_condense)) 83. plt.axis('off') 84. plt.title('Pooling') 85. plt.show() |
|  |
|  |

**Output**:



*Original Grayscale image*



*Output*