6CS012 - Artificial Intelligence and Machine Learning. Understanding Convolutional Neural Network.

Prepared By: Siman Giri {Module Leader - 6CS012}

March 23, 2025

Tutorial - 5.

1 Instructions

This sheet contains Exercise for various Operations we perform in Convolutional Neural Network.

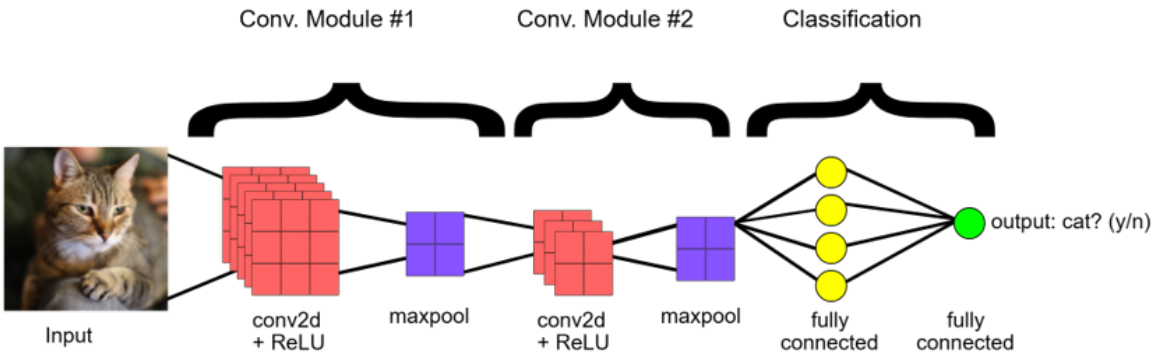

Figure 1: An example of Convolutional Neural Network.

2 Image Classification with Convolutional Neural Network:

1. **Feature Learning via Convolutional Operation:**
   - Learn Features in input image through convolution.
   - Introduce non - linearity through activation function.
   - Reduce dimensionality and preserve spatial invariance with pooling.

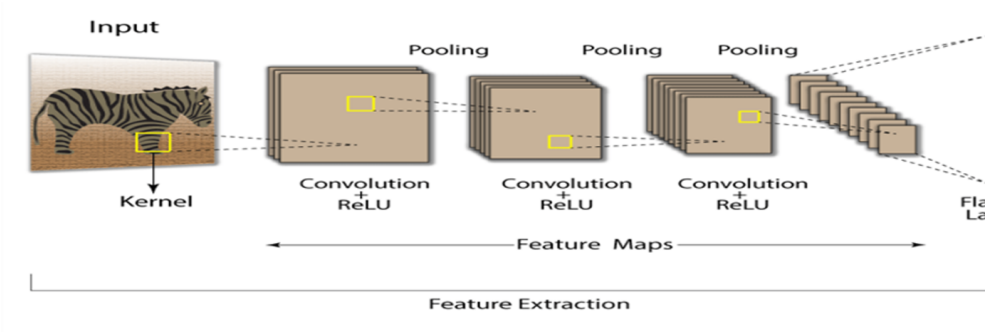

Figure 2: Feature Extraction with series of Convolution Operation.

2. **Learning Class Probabilities:**
   - CONV and POOL Layers output high - level features of input.
   - Fully connected layer uses these feature for classifying input image.
   - Express output as probability of image belonging to a particular class.

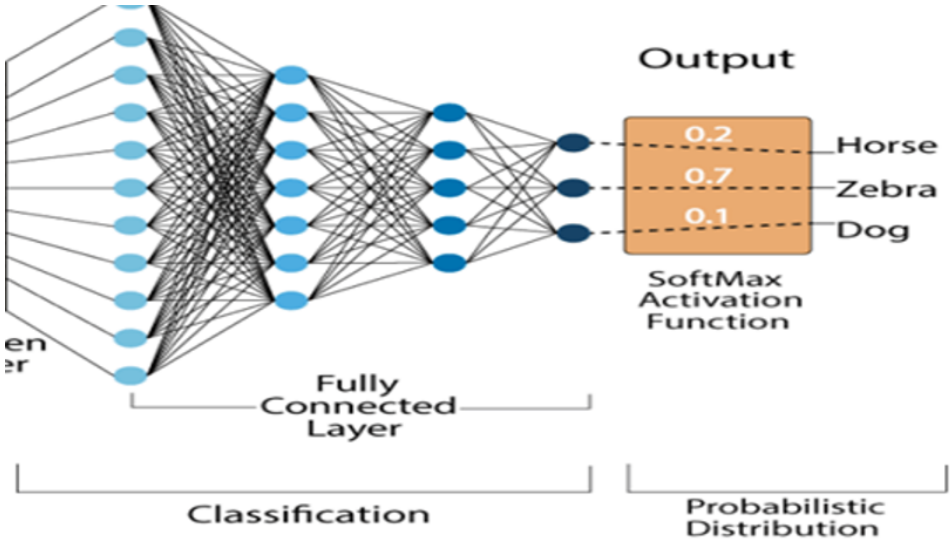

Figure 3: Assigning Class with Fully Connected Network.

3 An End to End CNN Model for Image Classification:

A typical CNN architecture has a canonical structure inspired by human vision cortex looks like:

INPUT → Convolution → Activation → Pooling → FCN.

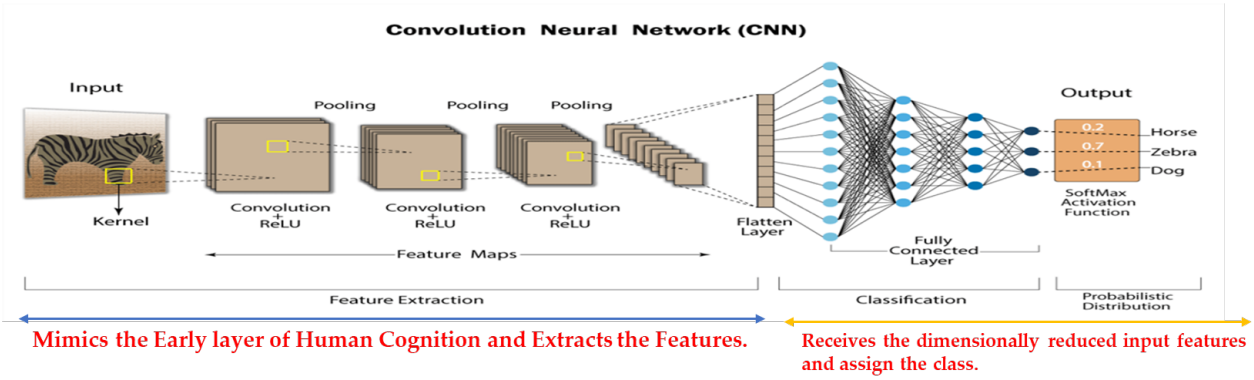

Figure 4: A basic CNN Architecture with 2 Convolutional Layer.

4 Understanding Convolution Layer:

4.1 Introduction to Convolution:

Convolution is a fundamental mathematical operation that combines two functions or signals to produce a third function. In image processing and deep learning, convolution is widely used to apply a filter also known as kernel to an image to extract important features such as edges, textures and patterns.

1. Key Concept:

Convolution allows a model to detect patterns in local regions of an image by sliding a small filter (matrix) over an image and performing a series of mathematical operations.

Mathematical Definition of Convolution:

The convolution operation between an Image I and a filter (kernel) is defined as:

$$(I * K)_{(x,y)} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} I(x+i, y+j) \cdot K(i,j)$$

Where:

- (x, y) are the coordinates of the pixel in the output feature map (convolved image).
- $m \times n$ represents the size of the kernel/filter K.
- The summation represents a weighted sum of the pixel values in the image I at positions corresponding to the filter K.

2. Understanding Convolution Operation:

Following operations happens in any convolution operation:

1. **Filter Application:**
   - A small matrix (filter/kernel) is passed over an image.
   - The kernel values are multiplied with corresponding pixel values in the image.
   - The results are summed to obtain a single pixel value in the output feature map.

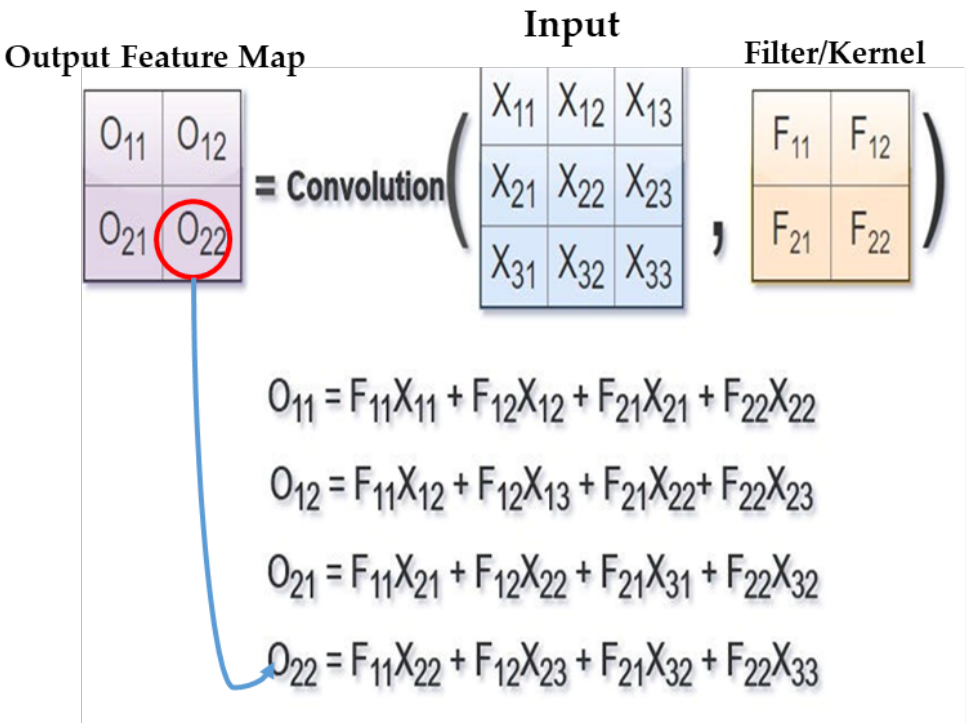

Figure 5: Mathematical Computation of Convolution Operation.

2. **Sliding Window Mechanism:**
   - The filter slides over the image from left to right, top to bottom.
   - Each step produces a new computed pixel for the output feature map.
   - The step size which determines how much the filter moves are determined by **Stride.**

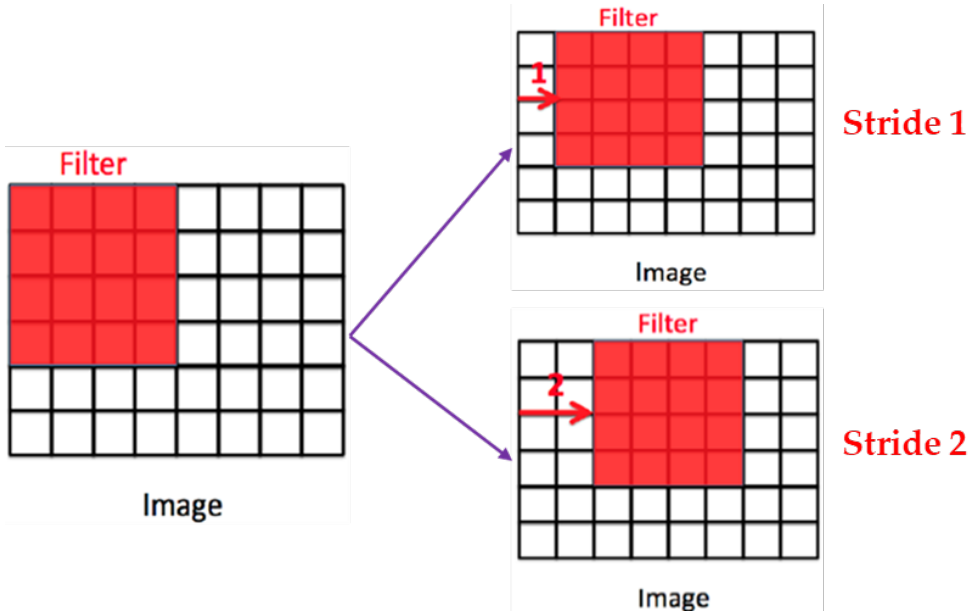

Figure 6: The Stride Mechanism.

- **Stride Mechanism:**
  (a) The stride refers to the number of pixels by which the filter shifts over the input matrix during the convolution operation.
  (b) When the stride is set to 1, the filter slides over the image one pixel at a time.
  (c) Stride is a hyperparameter that controls how much the filter moves, affecting the size of the output feature map.

3. **Padding:** When applying convolution operations with small filters, we tend to lose pixels along the boundaries of the image. While a single convolution may only discard a few pixels, this loss accumulates as we build deeper networks, potentially affecting the spatial integrity of the feature maps. To address this, we introduce padding, a technique that adds extra pixels around the image boundaries. Padding helps preserve spatial dimensions, ensuring that important edge features are not lost while allowing the network to maintain consistency across layers.

   - **Zero Padding - "YES Padding":** Extra pixels with a value of zero are added around the image boundary. This helps maintain the original spatial dimensions after convolution.
   - **Same Padding - "No Padding":** No extra pixels are added, but the padding is adjusted so that the output size remains the same as the input size (when using stride = 1). The framework automatically adds the necessary amount of padding to achieve this.
   - **Valid Padding:** No padding is applied at all, meaning the output feature map is smaller than the input after convolution.

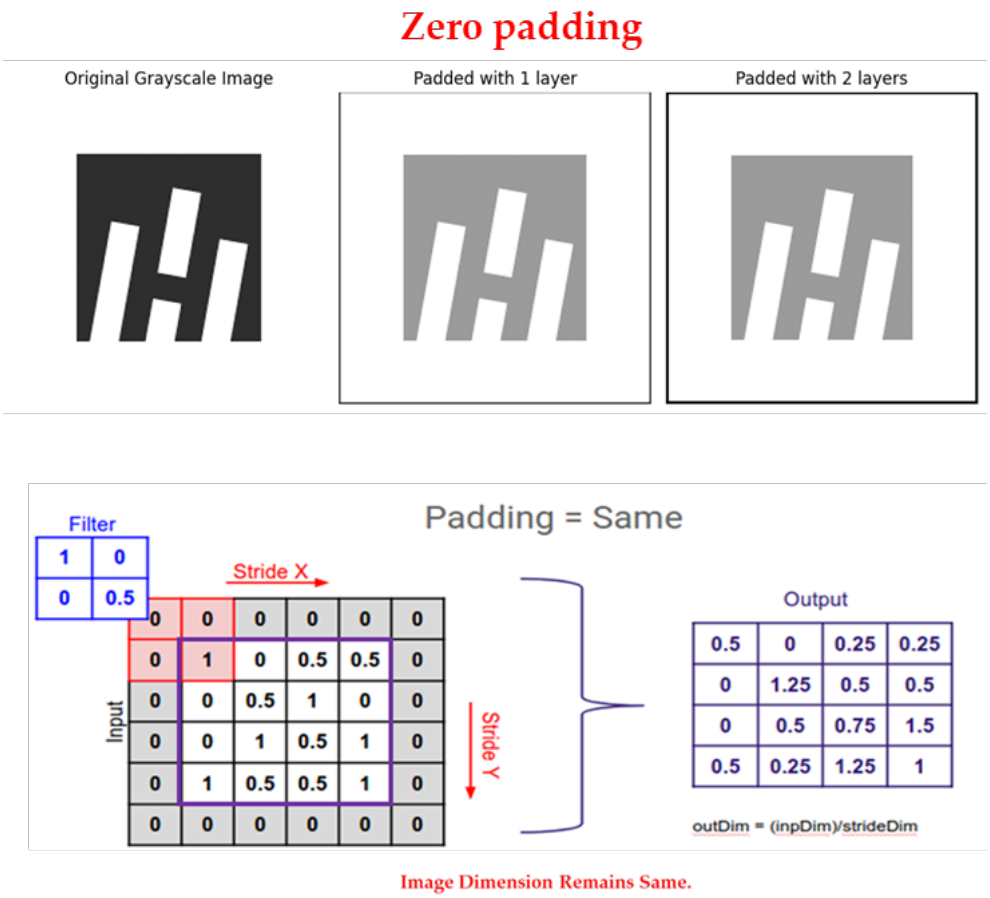

Figure 7: Example of Padding.

4. **Dimension of Output Feature Map:** The following figure illustrates the convolution operation with a stride of 1 and valid padding: A key question is: how can we determine the dimensions of

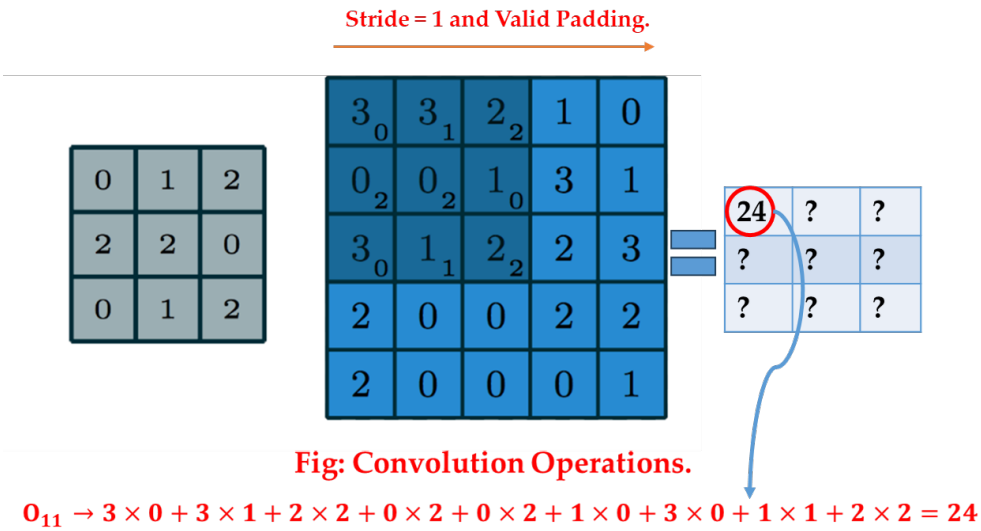

Figure 8: An Overview of Sliding Operation.

the output feature map? Understanding this is crucial from an application perspective, as without

this knowledge, we may not be able to correctly implement convolution operations in code.

- **Dimension of Output Feature Map:**

Convolutional Layer Output Shape Calculation

Hyperparameters:

- **Number of Filters (K)**: The number of feature detectors applied to the input. Determines the depth of the output volume.
- **Filter Size (F)**: The spatial size of each Filter, typically 3×3 or 5×5, but 7×7 can also be used, though larger than 7 is rare.
- **Stride (S)**: The most common choice is 1.
- **Padding (P)**:
  - `Zero Padding` (P): The number of zero pixels added around the input's border.
  - `Same Padding` ($P > 0$): Keeps the output the same size as input.
  - `Valid Padding` ($P = 0$): No extra padding, output size shrinks.

Output Shape Calculation:
Given an input of size $W_{\text{in}} \times H_{\text{in}} \times C_{\text{in}}$, the output volume dimensions are calculated as:

$$W_{\text{out}} = \frac{(W_{\text{in}} - F + 2P)}{S} + 1$$

$$H_{\text{out}} = \frac{(H_{\text{in}} - F + 2P)}{S} + 1$$

$$C_{\text{out}} = K \quad \text{(Number of filters, which determines the depth of the output)}$$

Example Calculation:
Given a filter 3×3 and input image matrix 5×5, the output dimension is calculated as: Given an input image of size 5×5, a filter of size 3×3, stride $S = 1$, and **valid** padding ($P = 0$), we calculate the output dimensions using the formula:

$$W_{\text{out}} = \frac{(W_{\text{in}} - F + 2P)}{S} + 1$$

$$H_{\text{out}} = \frac{(H_{\text{in}} - F + 2P)}{S} + 1$$

Substituting the values:

$$W_{\text{out}} = \frac{(5 - 3 + 2(0))}{1} + 1 = \frac{(5 - 3)}{1} + 1 = 3$$

$$H_{\text{out}} = \frac{(5 - 3 + 2(0))}{1} + 1 = \frac{(5 - 3)}{1} + 1 = 3$$

Thus, the output size is:

$$3 \times 3$$

5. **Activation Function:** Every Convolution Operation is followed by activation operation. For CNN we will use ReLU (Rectified Linear Unit) Activation Function.

   - **ReLU Activation Function:** ReLU is defined as:

   $$f(z) = max(0, z)\{z \rightarrow < W^T X > + b\}$$

   This means:

   $$\texttt{if} \quad z > 0, \texttt{the function returns} \rightarrow z.$$

   $$\texttt{if} \quad z \leq 0, \texttt{the function returns} \rightarrow 0.$$

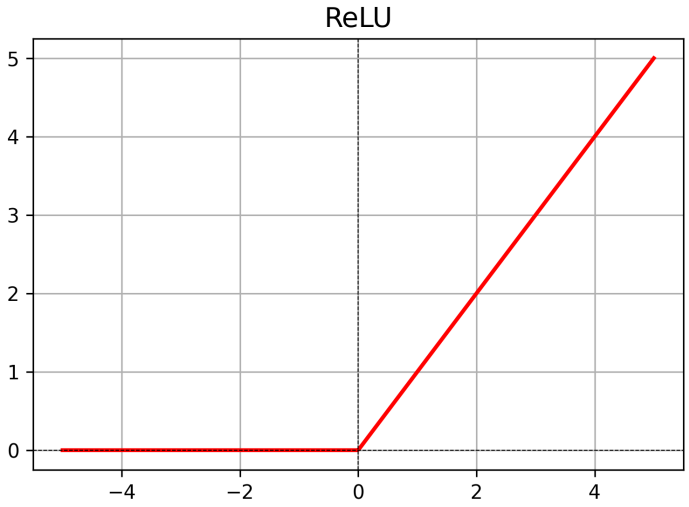

Figure 9: ReLU Activation Function.

   - **Advantages of ReLU in CNNs:**
     (a) **Non - Linearity:** Helps CNN learn complex decision boundary.
     (b) **Avoids Vanishing Gradient:** Unlike sigmoid or tanh, ReLU does not saturate for positive values.
     (c) **Computational Efficiency:** Requires only a simple threshold operation.
     (d) **Sparse Activation:** Many neurons output zero, making computations more efficient.
   - **Dying ReLU Problem in CNNs:** The dying ReLU problem occurs when a large number of neurons in a neural network output zero for all inputs, effectively becoming inactive. This happens due to the nature of ReLU activation function i.e.

   $$f(z) = max(0, z)$$

   If a neuron's weighted input is always negative, ReLU will always output zero, meaning the neuron is effectively "dead" and never "updates" during training.
   - **Effects of Dying ReLU:**
     – **Loss of network capacity:** Dead neurons do not contribute to learning.

  - **Slower Training:** Since fewer neurons are active, gradient updates become inefficient, Thus may have to train for more number of epochs.
  - **Reduced Accuracy:** Fewer active neurons mean fewer learned features, leading to poor model performance.

- **Solution to Dying ReLU:**
  - **Leaky RelU:** Instead of zeroing out negative values, Leaky ReLU introduces a small slope (α) for negative inputs:

$$\mathbf{f}(\mathbf{z}) = \mathbf{max}(\alpha(\mathbf{z}), \mathbf{z}) : \texttt{where } \alpha \texttt{ is hyperparameter and set to } 0.01 \texttt{ usually.}$$

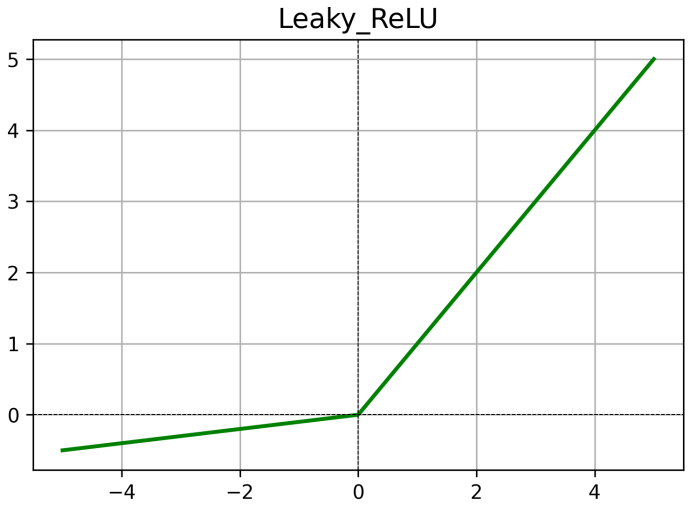

Figure 10: Leaky ReLU Activation Function

  - Since Leaky ReLU has a hyperparameter α, which can be difficult to tune, finding the optimal value may be challenging. In practice, ReLU has been observed to perform on par with Leaky ReLU. Therefore, we recommend using ReLU when designing and building CNN architectures
  - **Lower Learning Rates:** Reducing learning rates prevents drastic weight updates that might cause neurons to become inactive.

Why is the Sigmoid Activation Function Not Preferred in CNNs?

While the **Sigmoid activation function** was widely used in early networks, it is generally not recommended for **Convolutional Neural Networks** due several limitations. One of the major limitations is a Vanishing Gradient Problem, which cause training to become slow and Saturated.

Understanding Vanishing Gradient Problem:

Vanishing Gradient Problem in Sigmoid Activation

The sigmoid activation function is given by:

$$\sigma(\mathbf{z}) = \frac{\mathbf{1}}{\mathbf{1} + \mathbf{e}^{-\mathbf{z}}}$$

Its derivative, which is used during backpropagation to compute the gradients, is:

$$\sigma'(\mathbf{z}) = \sigma(\mathbf{z}) \cdot (\mathbf{1} - \sigma(\mathbf{z}))$$

where $\sigma(z)$ is the output of the sigmoid function.

Why Does the Vanishing Gradient Problem Occur?

For very high or low values of z:

- As $z \to \infty$, $\sigma(z)$ approaches $\mathbf{1}$, leading to:

  $$\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z)) \Rightarrow 1 \cdot (1 - 1) = 0$$

  Hence, the gradient becomes very small.

- As $z \to -\infty$, $\sigma(z)$ approaches $\mathbf{0}$, leading to:

  $$\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z)) \Rightarrow 0 \cdot (1 - 0) = 0$$

  Again, the gradient becomes very small.

Since the gradient is close to zero for large positive or negative z, weight updates during training become very small, causing the vanishing gradient problem.

4.1.1 What happens if gradient vanishes ?

When the gradient $\sigma'(z)$ becomes very small, the weight updates during training also become very small.The update rule for weight is:

$$\mathbf{w_{k+1}} = \mathbf{w_k} + \eta \cdot (\mathbf{\Delta w})$$

where:

$$\mathbf{w_k} \to \texttt{weight at step k.}$$

$$\mathbf{w_{k+1}} \to \texttt{weight at step } \mathbf{k+1}.$$

$$\mathbf{\Delta w} \Rightarrow \frac{\partial \mathbf{L}}{\partial \mathbf{w}} - \to \texttt{gradient.}$$

If the gradient Δw is very small as it is for large and negative values of $\mathbf{z}$, the weight update becomes very small. This slows down learning and prevents deep networks from effectively updating their parameters.

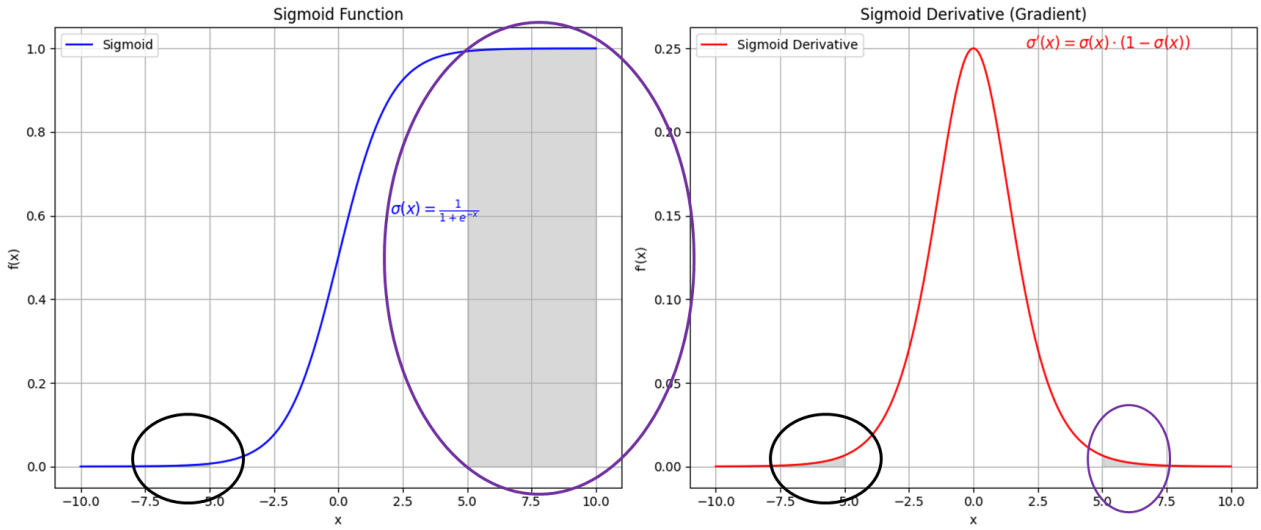

Figure 11: Illustrations of Vanishing Gradient Problem.

One Complete Convolution Operations:

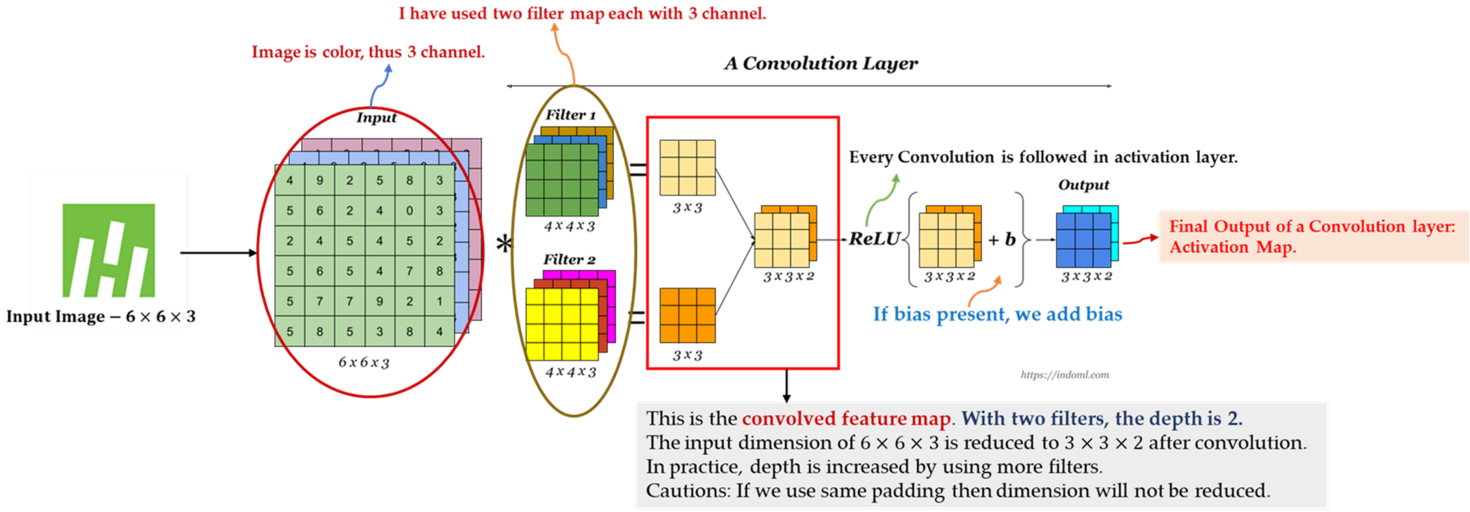

Figure 12: A Convolutional Operations.

4.2 Introduction to Pooling:

After each convolution operation, a pooling operation is typically applied. The primary objective of the convolutional layer is to extract features while reducing the spatial dimensions (**Height** $\times$ **Width**); While the convolutional layer reduces spatial dimensions (except in same padding), it also increases the depth of the output. The pooling operation further reduces the spatial dimensions and helps retain important spatial information by selecting the most relevant features from local regions.

Types of Pooling Operation:

The pooling operation is also performed suing sliding window approach with a filter and stride, typically using a $\mathbf{2} \times \mathbf{2}$ filter with stride of 2. This operation usually reduces the spatial dimensions by half. The two most common types of pooling operations are **maxpooling** and **averagepooling.**

Figure 13: Max and Average Pooling.

Max Pooling - Mathematical Formalizations:

Given an input region X of size $F \times F$, the max pooling operation is defined as:

$$Y_{i,j} = \max_{p,q} X_{i+p,j+q}, \quad \text{where } 0 \leq p, q < F$$

Explanation:

- $Y_{i,j}$ represents the output at position (i, j).
- The operation selects the **maximum** value within an $F \times F$ window.
- Typically, $F = 2$ and stride $S = 2$ are used to reduce the spatial dimensions by half.

Average Pooling - Mathematical Formalizations:

For an input region X of size $F \times F$, the average pooling operation is defined as:

$$Y_{i,j} = \frac{1}{F^2} \sum_{p=0}^{F-1} \sum_{q=0}^{F-1} X_{i+p,j+q}$$

Explanation:

- $Y_{i,j}$ represents the output at position (i, j).
- The operation computes the **average** of all values within an $F \times F$ window.

Computing Output Dimension after Pooling:

Input and Output Dimensions

A pooling layer takes an input volume:

$$W_{\text{oca}} \times H_{\text{oca}} \times C_{\text{oca}}$$

where oca represents the output after **convolution and activation**. After pooling, the output volume becomes:

$$W_{\text{ocap}} \times H_{\text{ocap}} \times C_{\text{ocap}}$$

where ocap represents the output after **convolution, activation, and pooling**.

Output Shape is given by:

$$W_{\text{ocap}} = \frac{(W_{\text{oca}} - F)}{S} + 1$$

$$H_{\text{ocap}} = \frac{(H_{\text{oca}} - F)}{S} + 1$$

$$C_{\text{ocap}} = C_{\text{oca}} \quad \text{(Depth remains unchanged)}$$

5 Exercise - 1 - Convolution and Pooling Operations.

Hint - Formula for Convolution Output Shape

The output dimensions of a convolutional layer can be computed using the formula:

$$W_{\text{out}} = \frac{(W_{\text{in}} - F + 2P)}{S} + 1$$
$$H_{\text{out}} = \frac{(H_{\text{in}} - F + 2P)}{S} + 1$$

where:

- $W_{\text{in}}, H_{\text{in}}$ = Input width and height
- F = Filter size
- S = Stride
- P = Padding

1. Compute the Following:

1. A 5×5 input feature map is convolved with a 3×3 filter using a stride of 1 and no (valid) padding. What is the output shape? {**Output Shape:** 3×3}
2. A 6×6 input feature map is processed using a 3×3 convolutional filter with a stride of 1 and "same" padding. What is the output shape? { **Output Shape:** 6×6 }
3. A 7×7 input undergoes a 5×5 convolution with a stride of 2 and padding of $P = 1$. What is the output shape? {**Output Shape:** 3×3}
4. An 8×8 input is processed with a 5×5 filter using a stride of 2 and no padding. What is the output shape? { **Output Shape:** 2×2 }
5. A 10×5 input is convolved using a 3×3 filter with a stride of 1 and "same" padding. What is the output shape? { **Output Shape:** 10×5 }

2. Compute the Following:

1. Consider the following input image:

$$I = \begin{bmatrix} 20 & 35 & 35 & 35 & 35 & 20 \\ 29 & 46 & 44 & 42 & 42 & 27 \\ 16 & 25 & 21 & 19 & 19 & 12 \\ 66 & 120 & 116 & 154 & 114 & 62 \\ 74 & 216 & 174 & 252 & 172 & 112 \\ 70 & 210 & 170 & 250 & 170 & 110 \end{bmatrix}$$

What is the output provided by a convolution layer with the following properties:

- **Stride of 1**.

- **Filter is given by:**

$$K = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

{ **Convolution Output:**

$$C = \begin{bmatrix} 225 & 258 & 250 & 209 \\ 458 & 566 & 552 & 472 \\ 708 & 981 & 887 & 802 \\ 1000 & 1488 & 1320 & 1224 \end{bmatrix}$$

}

2. **Bonus Point: Implement above with Python and NumPy.**
3. Take the output from 1. and apply a max pooling layer with the following properties:
   - Stride - $[2, 2]$.
   - Window Shape - $[2, 2]$
   - **{Hint: First Compute the Output Dimension.}**

   {**Max Pooling Output:**

$$\mathbf{M} = \begin{bmatrix} 566 & 552 \\ 1488 & 1320 \end{bmatrix}$$

   }
4. Apply an average pooling operation with a 2×2 filter and stride of 2 to the following matrix:

$$X = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

   { **Solution:**

$$M_{avg} = \begin{bmatrix} 3.5 & 5.5 \\ 11.5 & 13.5 \end{bmatrix}$$

   }

6 Training a Convolutional Neural Network.

Convolutional Neural Network is Trained using Forward and Backward Propagation.

6.1 Forward Propagation:

1. Forward Propagation in Convolutional Layer:

We already discussed how the forward computation happens in CNN i.e.

INPUT → Convolution → Activation → Pooling → FCN.

The first step is convolution followed by Pooling (Usually maxpooling).

To - DO: Complete the following Convolved Feature Map Using Stride of 1 and no padding:

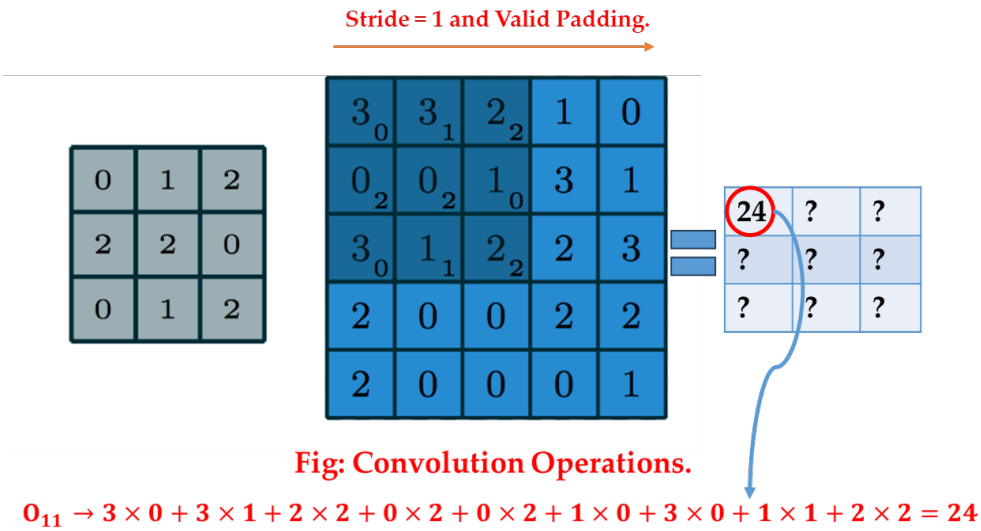

Figure 14: An Overview of Sliding Operation.

Please remember: Every convolution operation is followed by pooling operations. After the final convolution operation, the output is flattened and given to the Fully Connected Network. After all

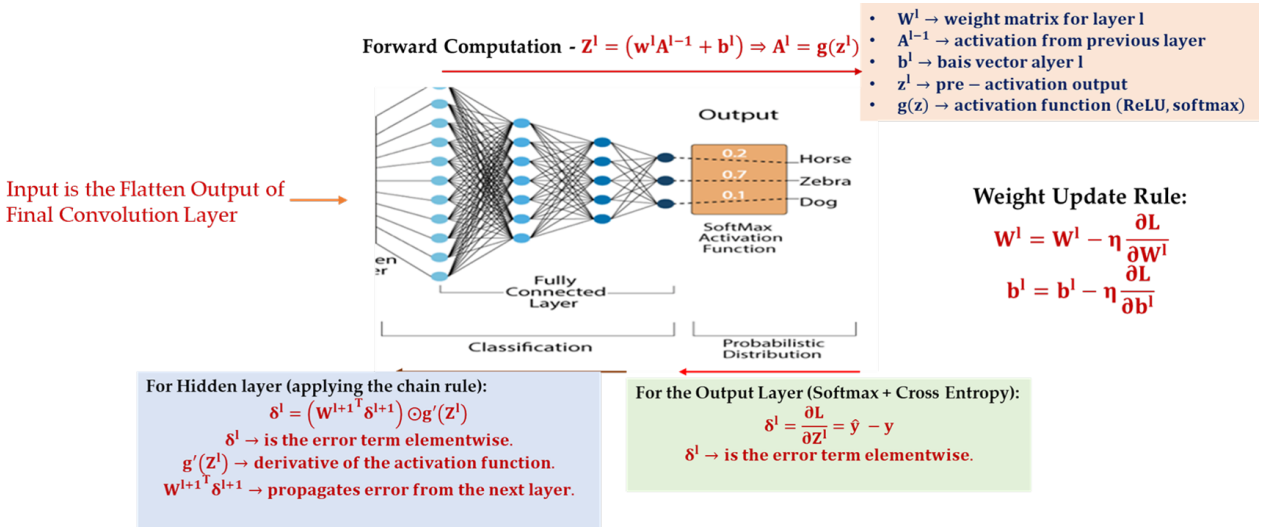

Figure 15: Forward and Backward Computation at FCN.

computations in the Fully Connected Neural Network, the loss function is computed and then back-propagated to the very first filter in the first convolutional layer.

2. Backward Propagation in Convolutional Layer:

The output of backpropagation in the Fully Connected Network (FCN) is the gradient $\frac{\partial L}{\partial W}$ .The first step is to convert this gradient back into a matrix, as the operations in the convolutional layer are matrix operations.

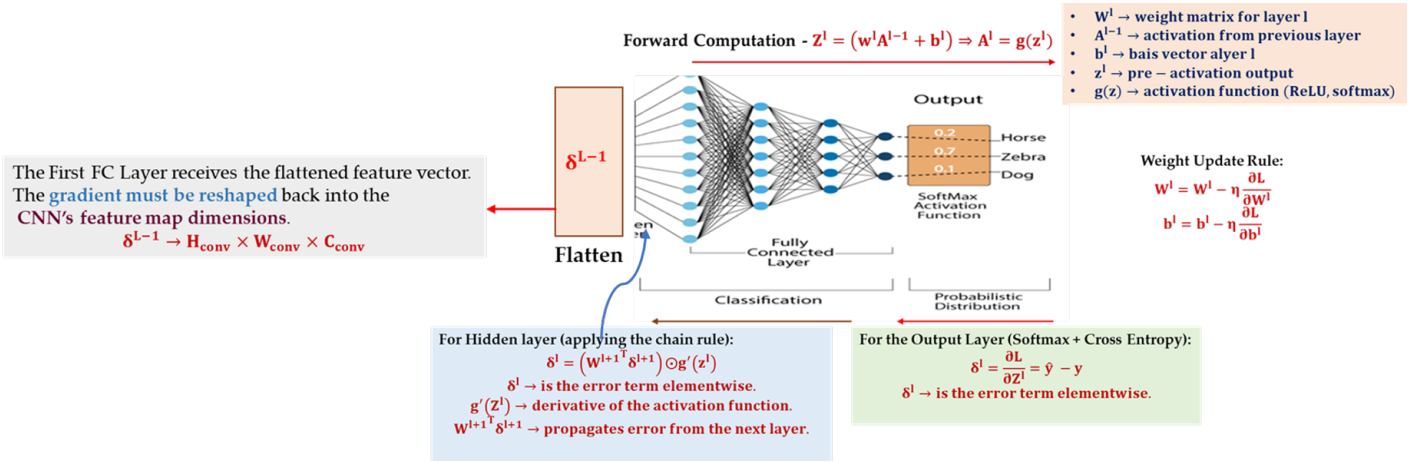

Figure 16: Getting Ready for Back - Propagation in Convolutional Layer.

In backward propagation of Convolutional Layer we perform following two operations:

1. Computing the Gradient with Respect to the Filter F

The goal of this step is to determine how much each filter F contributes to the loss. This is computed by performing a convolution between the input X and the gradient of the error $\delta = \frac{\partial L}{\partial O}$, where L is the loss and O is the output of the convolution.

The gradient with respect to the filter F is calculated as:

$$\frac{\partial L}{\partial F} = X * \delta$$

where:

- X is the input to the convolutional layer,
- δ is the gradient of the error with respect to the output, and
- $*$ denotes the convolution operation.

This tells us how much each filter element F should change in order to minimize the loss.

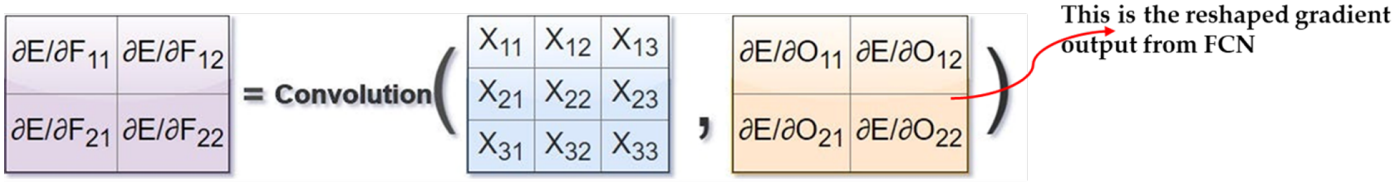

Figure 17: Computing gradient w.r.t filter F.

2. Computing the Gradient with Respect to the Input X

In this step, we calculate how much each pixel in the input X contributes to the loss. This is important because we need to propagate the error backward to the previous layers. The gradient with respect to the input X is calculated by convolving the flipped filter F^T with the gradient of the error δ. The filter F is flipped because the forward convolution used the filter in a specific orientation, and during backpropagation, we need to reverse that operation. The gradient with respect to the input X is computed as:

$$\frac{\partial L}{\partial X} = F^T * \delta$$

where:

- F^T is the flipped version of the filter F,
- δ is the gradient of the error with respect to the output, and
- $*$ represents the convolution operation.

This tells us how much each pixel in the input X should change in order to minimize the loss.

Figure 18: Computing gradient w.r.t filter input X.

7 A Bonus Problem:

Provided the sample code for Convolution Operation, Implement a Valid Convolution Operation with following filter on the lenna Image and describe what kind of Feature they extract: The Filter are:

Kernel 1: F_1

$$F_1 = \begin{bmatrix} -10 & -10 & -10 \\ 5 & 5 & 5 \\ -10 & -10 & -10 \end{bmatrix}$$

Kernel 2: F_2

$$F_2 = \begin{bmatrix} 2 & 2 & 2 \\ 2 & -12 & 2 \\ 2 & 2 & 2 \end{bmatrix}$$

Sample Code:

Valid Convolutional Operations

```
import numpy as np
# Function to apply convolution manually using NumPy
def conv2d_nopadding(image, kernel):
    # Get the dimensions of the image and kernel
    image_height, image_width = image.shape
    kernel_height, kernel_width = kernel.shape
    # No padding is applied (valid padding)
    pad_height = 0
    pad_width = 0

    # Skip padding if pad_height and pad_width are zero
    padded_image = image # No padding

    # Calculate the output dimensions without padding
    output_height = image_height - kernel_height + 1
    output_width = image_width - kernel_width + 1
    output = np.zeros((output_height, output_width))

    # Apply the convolution operation
```

```
    for i in range(output_height):
        for j in range(output_width):
            # Extract the region of interest from the image
            region = padded_image[i:i + kernel_height, j:j + kernel_width]
            # Perform element-wise multiplication and sum the result
            output[i, j] = np.sum(region * kernel)

    return output
```

Good Luck.