6CS012 – Artificial Intelligence and Machine Learning.

Lecture – 05

Getting Started with Deep Learning

Introduction Convolutional Neural Network.

Siman Giri {Module Leader – 6CS012}

Learning Objective,

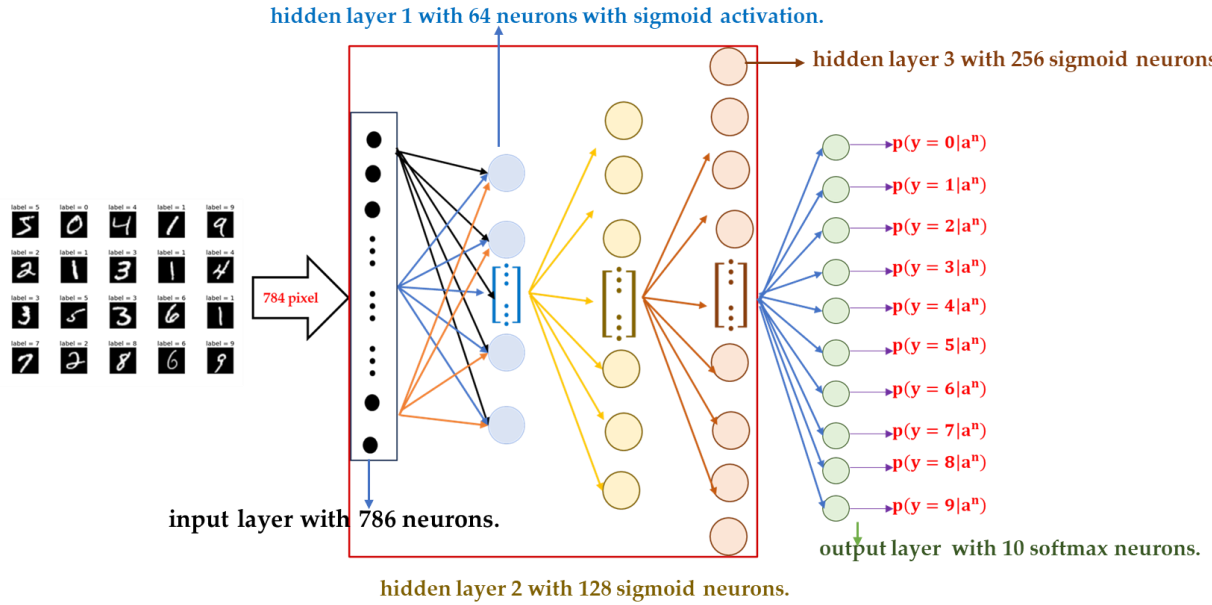
- **To solve all the challenges, we have been discussing for Image Classification.**
- Understand Image classification with CNN,
  - Also, understand various layer used to build CNN
  - Various components used to build those layer.
- Lear how to train such model.
- Appreciate how CNN share parameters locally to make them more efficient in number of parameters than FCN(relatively).

On the Quest to Solve the Challenges.

{Recap the Challenge … }

Recap the Challenges:

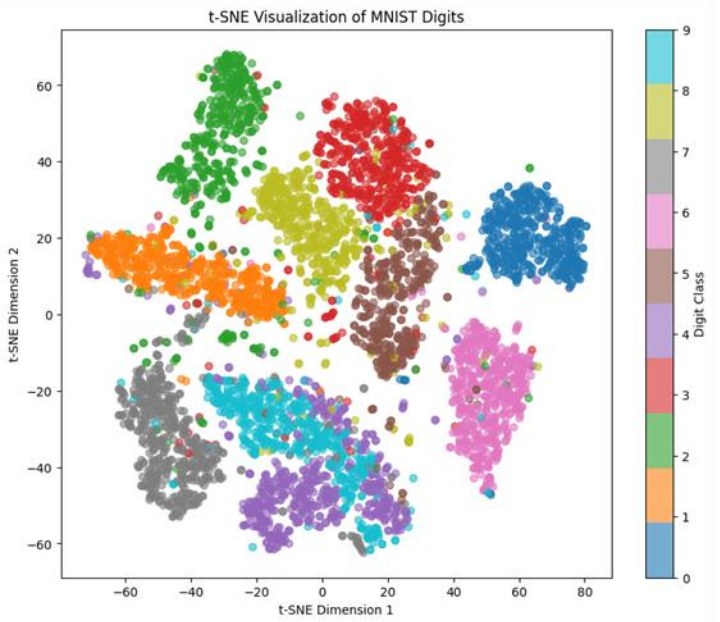
- **Challenge 2:**
  - **Non-Linear Decision Boundary:**
    - **Perceptron** was better on separating a data in classes/label those were **linearly separable**.
    - **What for data where classes/label are not linearly separable?**
    - ***"Did Fully Connected Neural Network Solved the Challenge? …"***



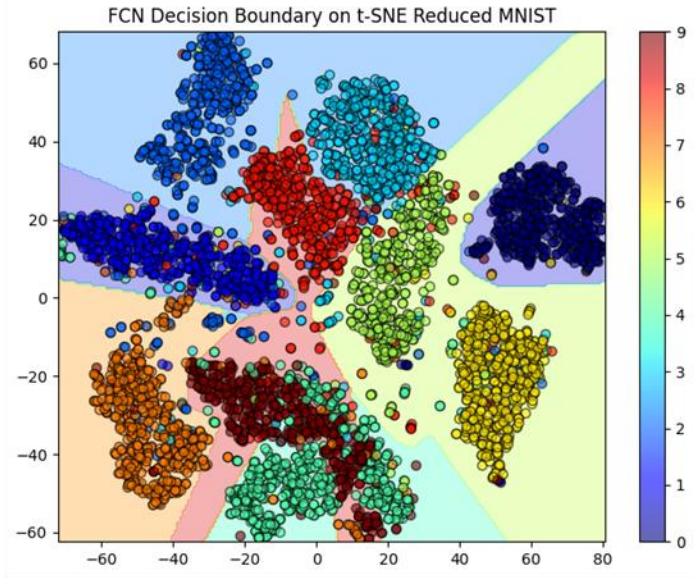
Did it Work?

Recap the Challenges:

- **Challenge 2:**
  - **Non-Linear Decision Boundary:**
    - **Perceptron** was better on separating a data in classes/label those were **linearly separable**.
    - **What for data where classes/label are not linearly separable?**
    - ***"Did Fully Connected Neural Network Solved the Challenge? …"***



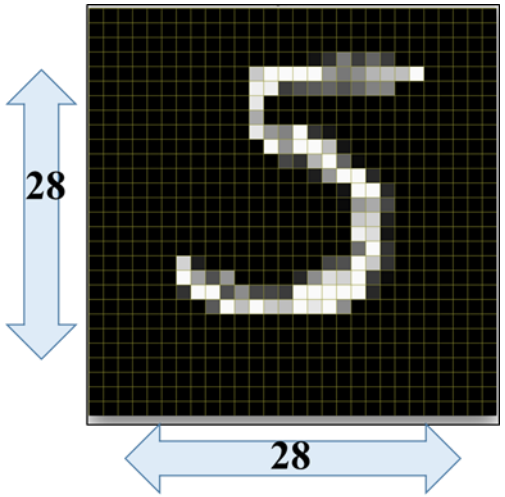
Fully Connected Network.



We can still see some misclassification but better than The Perceptron

Recap the Challenges:

- **Challenge 1:**
  - **Extraction of Features:**
    - When we extracted only pixel values we got the csv file with 784 columns i.e. very high dimensions , and our dataset was only of the size of 28 × 28.
    - Imagine for larger images, how big our dataset may be.



- *"We want to automate the process of feature extraction."*

1. Fully Connected Neural Networks.

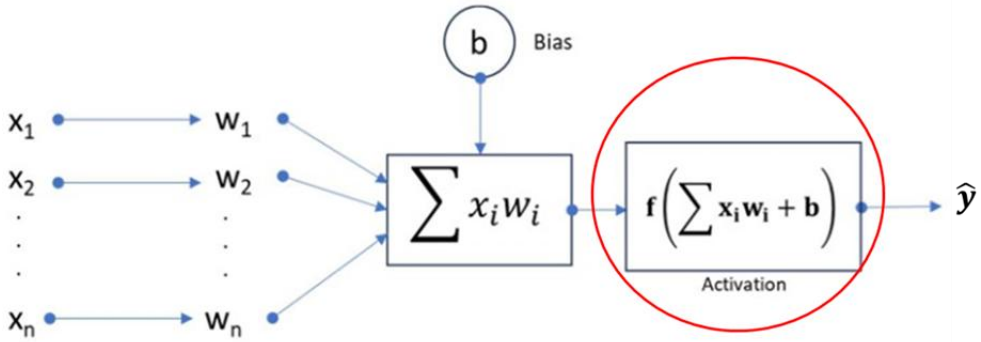
{ Good and Bad Things → Lesson Learned.}

1.1 Good Things.

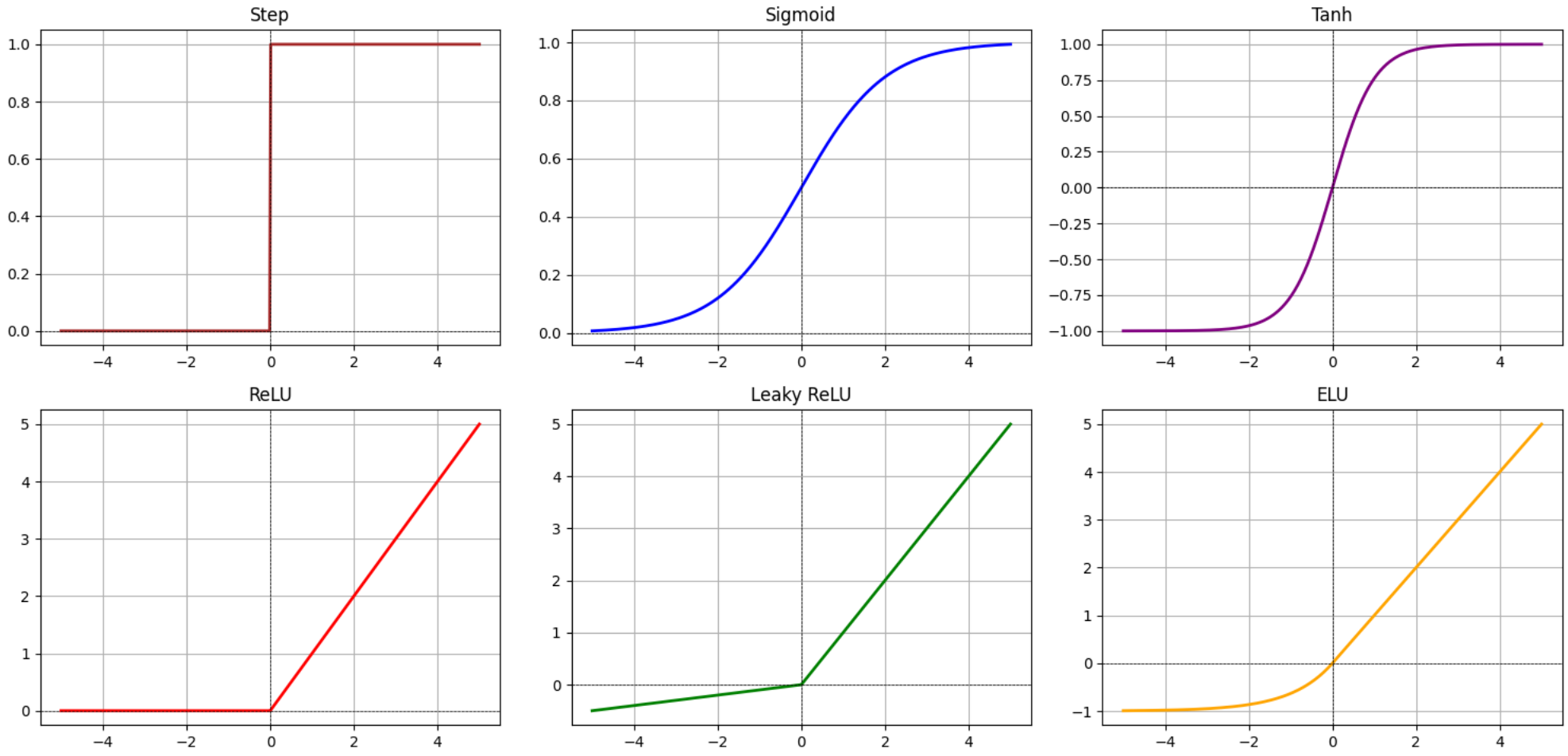
- **Hierarchical Structure of Fully Connected Networks(FCN):**
  - FCNs follows a structured layered architecture, consisting of:
    - **Input Layer** – Receives raw data or features.
    - **Hidden Layers** – Extract and refine hierarchical feature representations.
    - **Output Layer** – Produces predictions or classifications.
  - This design enables **progressive feature extraction**, where deeper layers capture **higher – level abstractions**.
- **Key Factors Enhancing Hierarchical Learning:**
  - **Layered Representations**:
    - Each hidden layers learns increasingly complex patterns, moving from low level to high level features.
  - **Non – linear Activation Functions:**
    - Unlike early **step functions**, modern activations (**ReLU, Sigmoid, Tanh**) introduce **non linearity**, allowing FCNs to model **complex decision boundaries**.
- This architecture mimics human cognition, enabling deep networks to learn **intricate relationship** in data.

1.2 The Activation Function.

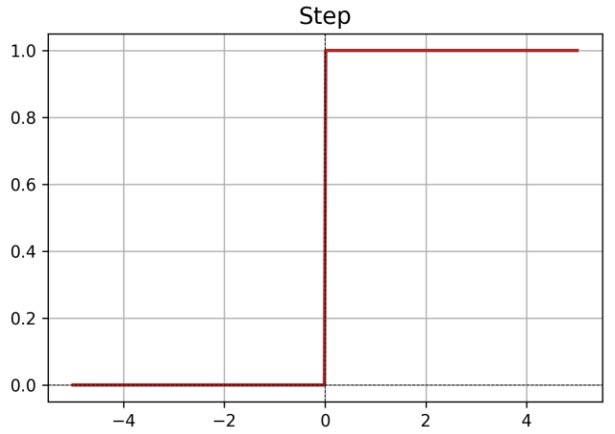
- {From Last week}
- In General, Activation Functions are group of function that introduces **non-linearity** to the **output** of neurons, making it capable to learn more **complex decision boundary**.
- The Good activation function must have following key properties:
  - **Non – Linearity**: Allows the network to learn complex patterns and decision boundaries.
  - **Differentiability**: Must be differentiable to enable gradient based optimizations i.e. gradient descent.
  - **Computationally Efficiency**: Should be fast to compute for real – time applications.
  - **Gradient Behavior**: Should avoid vanishing or exploding gradients to stabilize deeper network.{*we will discuss this further after defining deep networks*.}



1.2.1 Some Modern Activation Function.

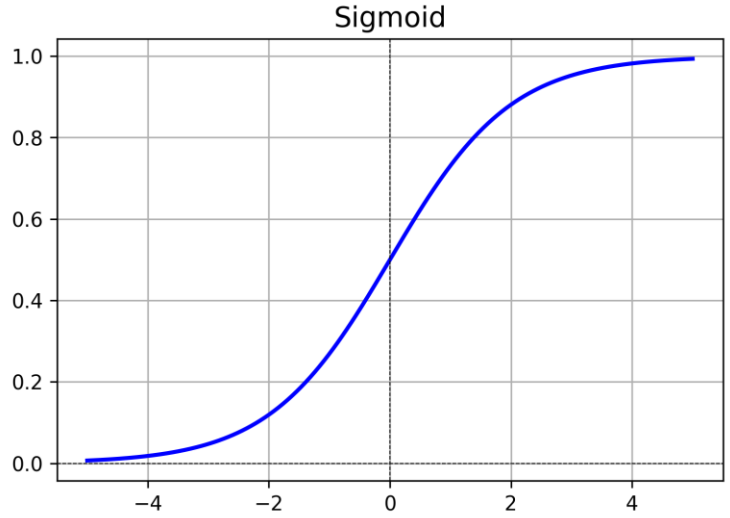


1.2.1 Step Function (Heaviside Step Function.)



- **Mathematical Representation:**
  - $f(x) = \begin{cases} 1, \text{if } x \geq 0 \\ 0, \text{if } x < 0 \end{cases}$
- **Pros:**
  - Simple and easy to compute.
  - Effective in binary classification problems.
  - Can be used in threshold – based decision making.
- **Cons:**
  - **Non – differentiable** at x = 0, which makes gradient – based optimization algorithms (e.g. backpropagation) difficult.
  - **Saturated output**: Outputs are always 0 or 1.
  - **Not smooth**: It creates a hard transition which can hinder learning in deep networks.
- **When to use?**
  - Our suggestion: **Never use for Image Classification Task**.

1.2.2 Sigmoid Function:



- **Mathematical Representation:**
  - $\mathbf{f(x) = \frac{1}{1+e^{-z}}; \{For\ Neural\ Network\ z = \langle w^T \cdot x \rangle + b\}}$
- This is equivalent of representing:
  - $\mathbf{P(y = 1|z) = f(x) = \frac{1}{1+e^{-z}}}$
- **Pros:**
  - Smooth and continuously differentiable i.e. differentiated everywhere, allowing for gradient – based optimization.
  - Maps output to a range between 0 and 1, which makes it suitable for binary classification tasks (probability like outputs).
- **Cons:**
  - **Vanishing Gradient Problem**: For very high or low inputs, the gradient becomes very small, slowing down the learning process and also can lead to a loss of information in deeper networks.
  - **Not zero centered**: The output of the sigmoid is always positive, meaning it could lead to inefficient optimization. Specifically, since the output ranges from 0 to 1, the gradients are always either positive or negative, which can slow down training.
  - **Computationally expensive**: Calculating the exponential function is computationally expensive.
- **When to use:**
  - Sigmoid is useful for binary classification, particularly when you need a probability output that represents the likelihood of an event occurring.
    - **However, because of its vanishing gradient issue, it's rarely used in deep networks or hidden layers.**

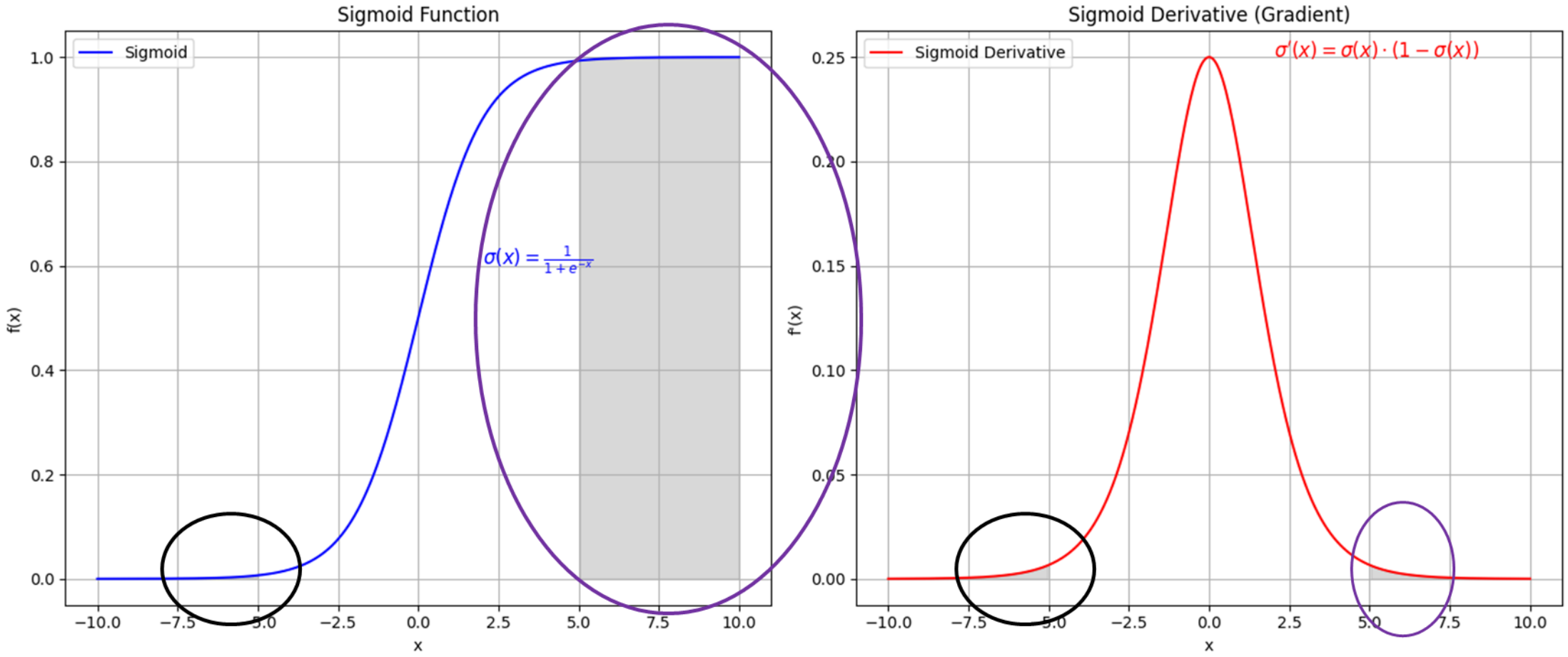
1.2.2.1 Understanding Vanishing Gradients.

- **Vanishing Gradient Problem in Sigmoid:**
  - The sigmoid activation function is given by: $\boldsymbol{\sigma(z) = \frac{1}{1+e^{-z}}}$; it's derivative, which is used during backpropagation to compute the gradients is:
    - $\boldsymbol{\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))}$, here $\boldsymbol{\sigma(z)}$ is the output of the sigmoid function.
- **Why does the vanishing gradient problem occur?**
  - For very **high or low** values of $\boldsymbol{z}$:
    - As $\boldsymbol{z \to \infty}$, $\boldsymbol{\sigma(z)}$ approaches $\boldsymbol{1}$, and hence $\boldsymbol{\sigma'(z)}$ becomes very small because for $\boldsymbol{\sigma(z) = 1}$ derivative is :
      - $\boldsymbol{\{\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z)) \Rightarrow 1 \cdot (1 - 1) = 0\}}$
    - Similarly, as $\boldsymbol{z \to -\infty}$, $\boldsymbol{\sigma(z)}$ approaches $\boldsymbol{0}$, and hence $\boldsymbol{\sigma'(z)}$ becomes very small because for $\boldsymbol{\sigma(z) = 0}$ derivative is:
      - $\boldsymbol{\{\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z)) \Rightarrow 0 \cdot (1 - 0) = 0\}}$

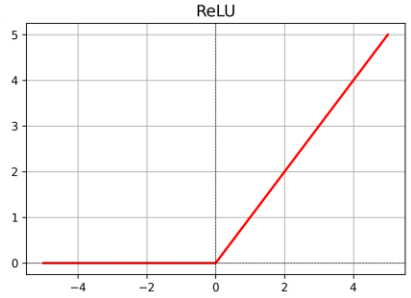
1.2.2.2 Understanding Vanishing Gradients.

- **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 weights is:
    - $w_{k+1} = w_k + \eta \Delta w$
  - Where:
    - $w_k \rightarrow$ **weight at step k**
    - $w_{k+1} \rightarrow$ **weight at step k + 1**
    - $\Delta w \Rightarrow \frac{\partial L}{\partial w} \rightarrow$ **gradient**.
- If the **gradient Δw** is very small as it is for large positive and **negative values** of **z**,
  - **the weight update becomes very small**,
- As a result, the change in weights becomes minimal, and learning slows down
  - and **for deep networks there is very high chances that gradient may be zero which may stop the learning**.

1.2.2.3 To illustrate Vanishing Gradients.

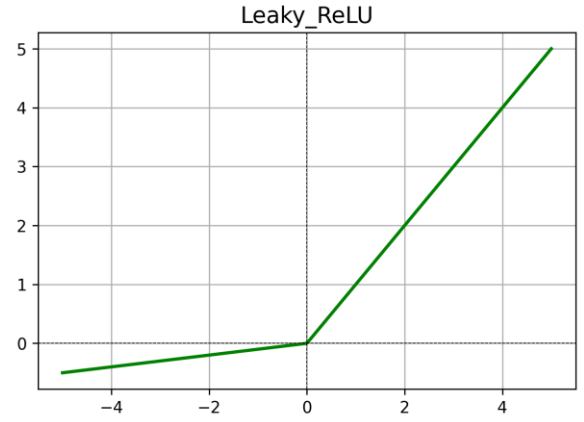


1.2.3 RELU (Rectified Linear Unit).



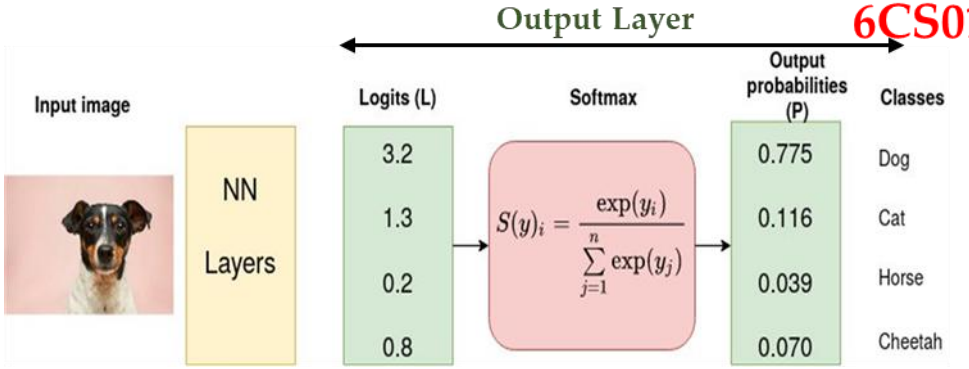
- **Mathematical Representation:**
  - $\mathbf{f(x) = max(0, x)}$
  - ReLU outputs the input directly if it's positive, otherwise, it outputs zero.
- **Pros:**
  - **Efficient Computation**: ReLU is computationally simple because it only involves a thresholding operation, which makes it faster to compute than sigmoid.
  - **No vanishing gradients:** For **positive values**, ReLU does not suffer from the vanishing gradient problem because its **derivative is 1** (for positive inputs).
  - **Sparsity:** Since negative inputs are set to zero, ReLU produces **sparse activations**, which can help with reducing overfitting and improve model efficiency.
- **Cons:**
  - **Dying ReLU problem**: When inputs are negative ReLU outputs zero, which means **the gradient is also zero**. If this happens too often, **some neurons may never activate (known as "dying ReLU"),** leading to situation where those **neurons are essentially useless during training**.
  - **Not zero – centered**: Although **ReLU provides non – zero values for positive inputs**, the negative outputs (zeros) can hinder **convergence as gradients are not zero – centered**.
- **When to use ?**
  - **Hidden layers in deep networks**: ReLU is widely **used hidden layers of deep neural networks**, despite the issue of dying ReLU problem **it is the go – to activation function for CNNs** because of its simplicity and performance.

1.2.4 Leaky ReLU.



- **Mathematical Representation:**
  - $f(x) = \begin{cases} x \text{ if } x > 0 \\ \alpha x \text{ if } x \leq 0 \end{cases}$ **{ α is a small constant typically 0.01}**
  - Unlike ReLU, Leaky ReLU allows a small, non – zero gradient for negative values, which helps avoid the "dying ReLU" problem.
- **Pros:**
  - **Avoids dying ReLU problem**: Since **Leaky ReLU allows a small slope for negative inputs** (***defined by α***), it **mitigates the issue of neurons dying**. (i.e. not activating during training).
  - **Computationally efficient**: Like ReLU, Leaky ReLU is computationally simple to implement.
  - **Better performance than ReLU**: In practice, Leaky ReLU can lead to better convergence and performance but depends on the **correct selection of hyper – parameter α**
- **Cons:**
  - **Still not zero – centered**: Although it allows for negative values in the output, the **activation function** is still not zero centered, which can cause issues during optimization.
  - **Hyper – parameter tuning**: The **α parameter needs to be set and tuned**, and choosing an inappropriate value may lead to suboptimal performance.
- **When to use ?**
  - **Can be used in Hidden layers if deep network or in CNN if Overfitting is an issue.**

1.2.5 Softmax.

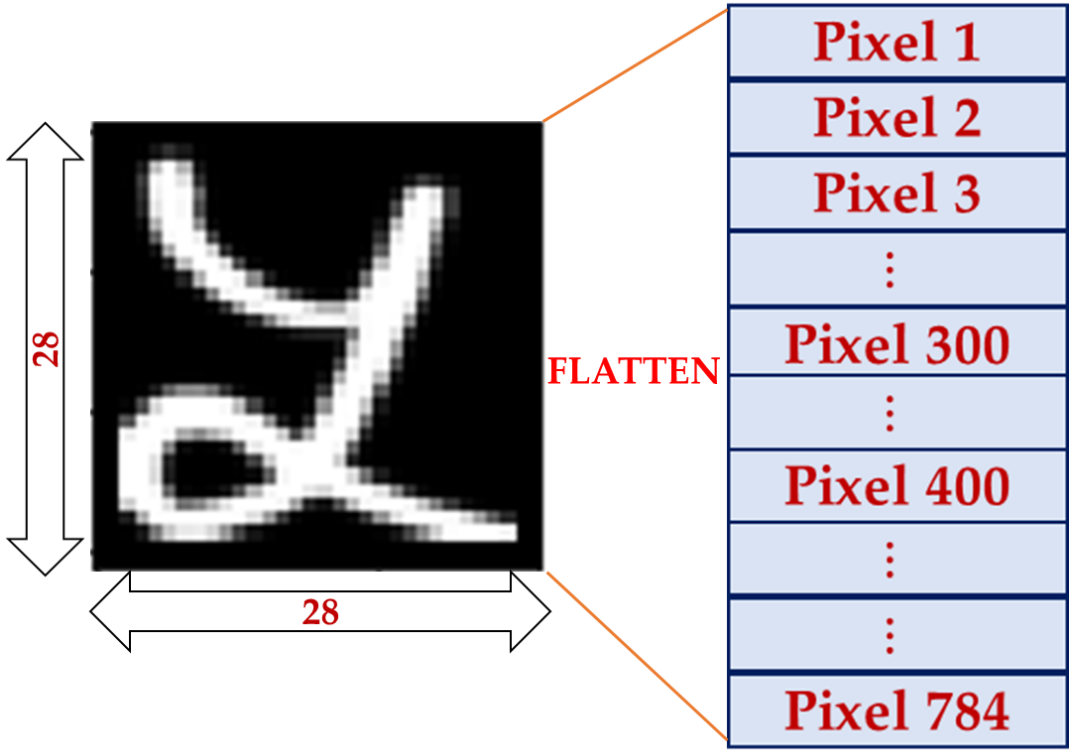


- **Mathematical Representations:**
  - The softmax function is commonly used in the output layer of a neural network for multi class classification problems. It converts raw scores (also called logit) from the network into probabilities by exponentiating each output and normalizing it.
    - $\mathbf{f(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}}$; [Here: z → raw score or logit for the i^{th} class, C → number of class]
- Softmax outputs a probability distribution, where **each output $f(z_i)$** is a value **between 0 and 1**, and the sum of all the **outputs equals 1**.
- **Practical Use:**
  - **Output Layer for Multi-class Classification**:
    - Softmax is used in the **output layer** of neural networks where the task is to classify data into multiple mutually exclusive classes.
    - Each output neuron represents a class, and Softmax is used to convert the raw scores into class probabilities.

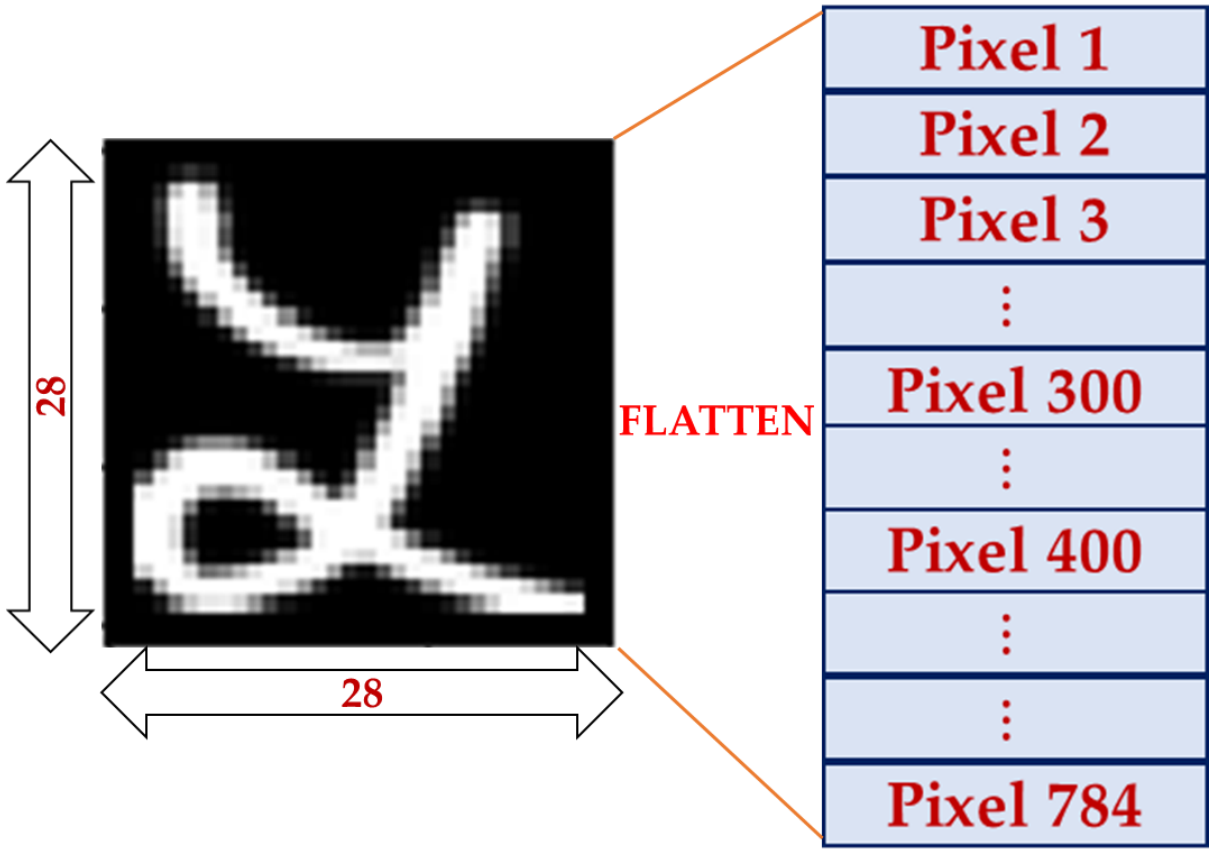
2. Fully Connected Network and Their Limitations.

2.1 Limitations of FCN – Feature Extraction.

- **How did we extracted features to build our Fully Connected Neural Network?**
  - **What we know:**
    - Our **Deep Neural models** uses **vector** as **an input**
  - **Idea:**
    - Represent **image data** as **vectors**.
  - **How did we represent our image data as vectors?**
    - We **flatten** a two (three for color) – dimensional
      - **image** into a **single stream of numbers** such
        - that we could capture it as **vector**.



2.1 Limitations of FCN – Feature Extraction.



Is this a Good Idea?

2.2 Limitation – 1 – Loss of Spatial Information.

- Loss of Spatial Information :
- **Stretching pixel values to single stream of numbers {columns}** may lose some key **relationships between the pixels** such as **distance between pixels** representing the same object.

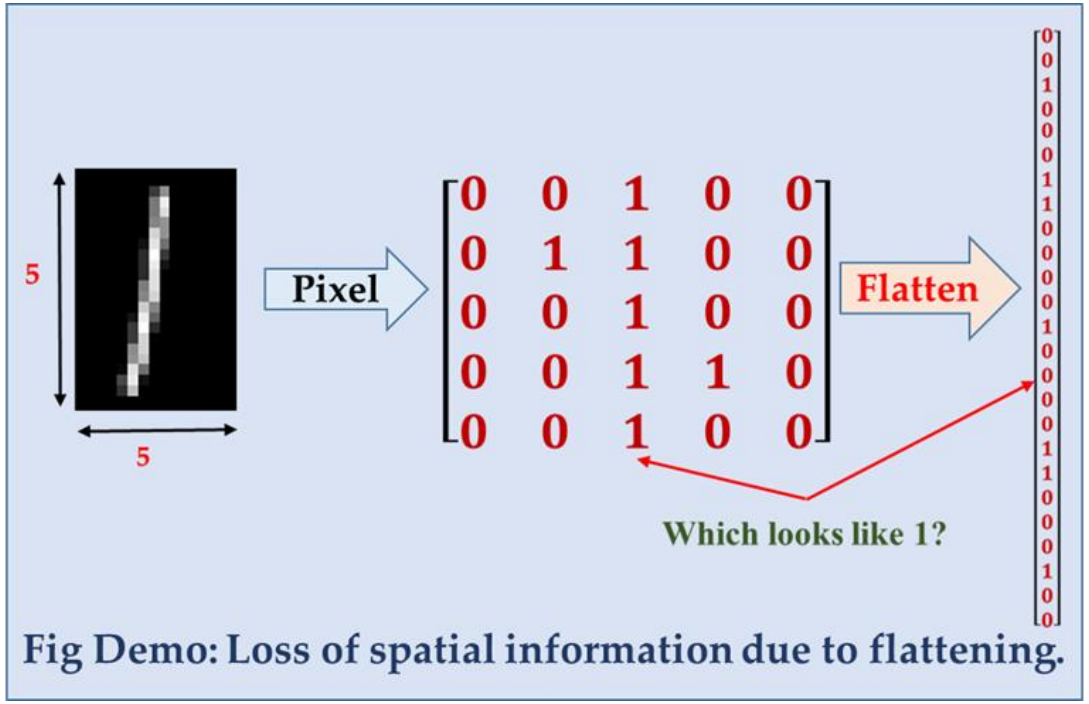
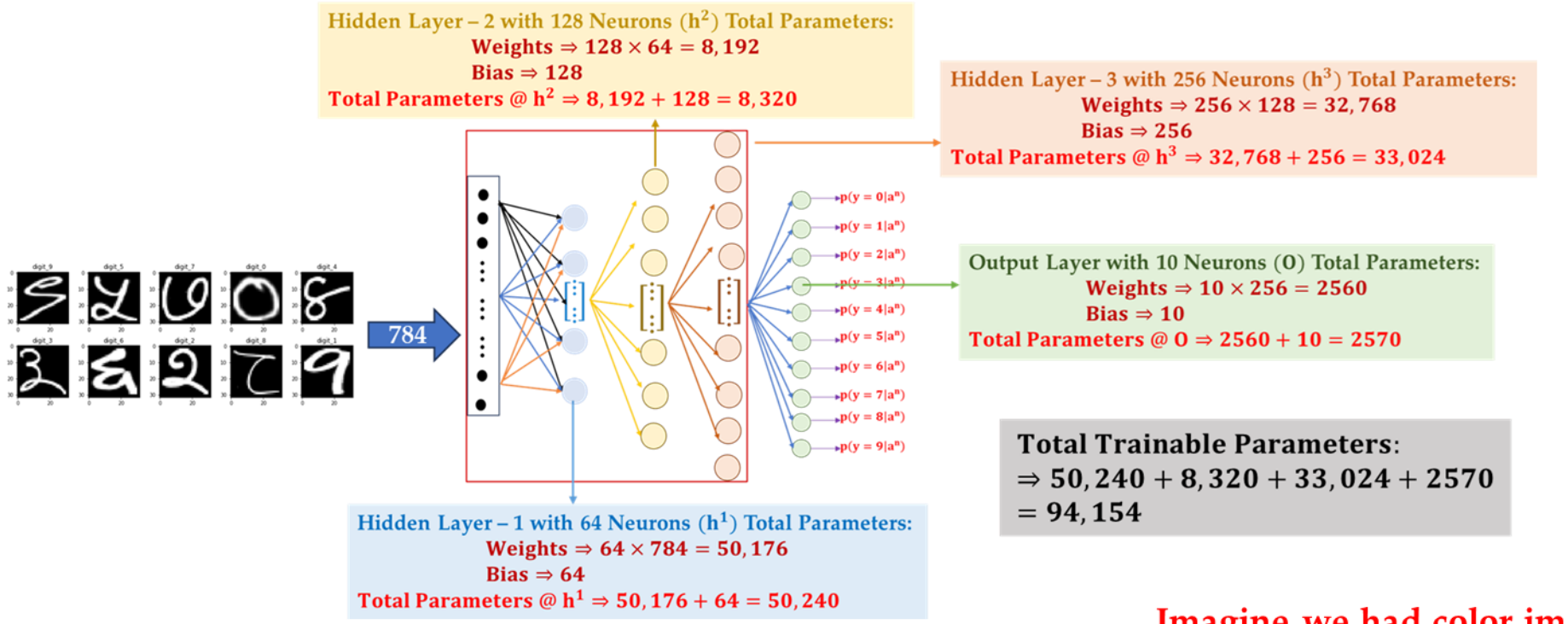


Fig Demo: Loss of spatial information due to flattening.

2.3 Limitation – 2 – Too Many Parameters.

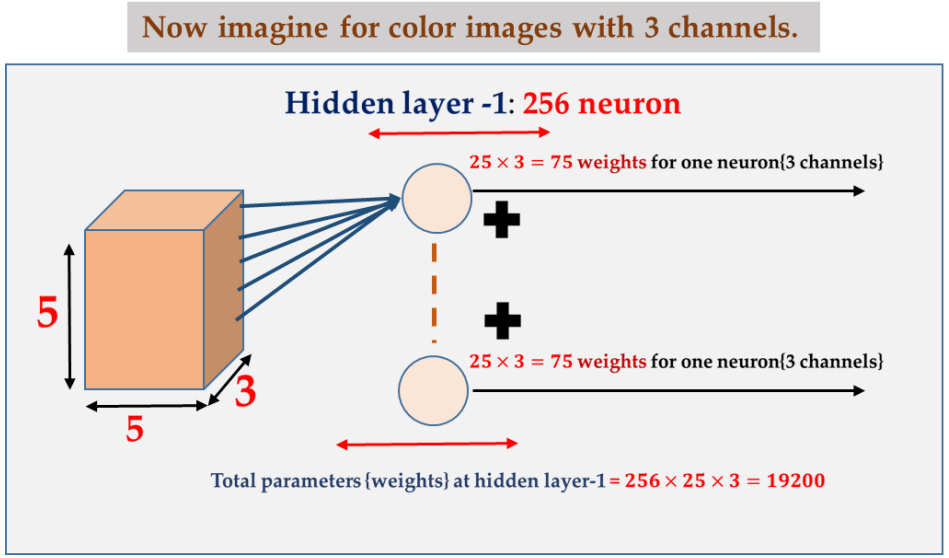
- **Our Observation:**
  - For our experiment with 3 Hidden Layers:



Imagine we had color image:

2.3.1 Limitation – 2 – Too Many Parameters.

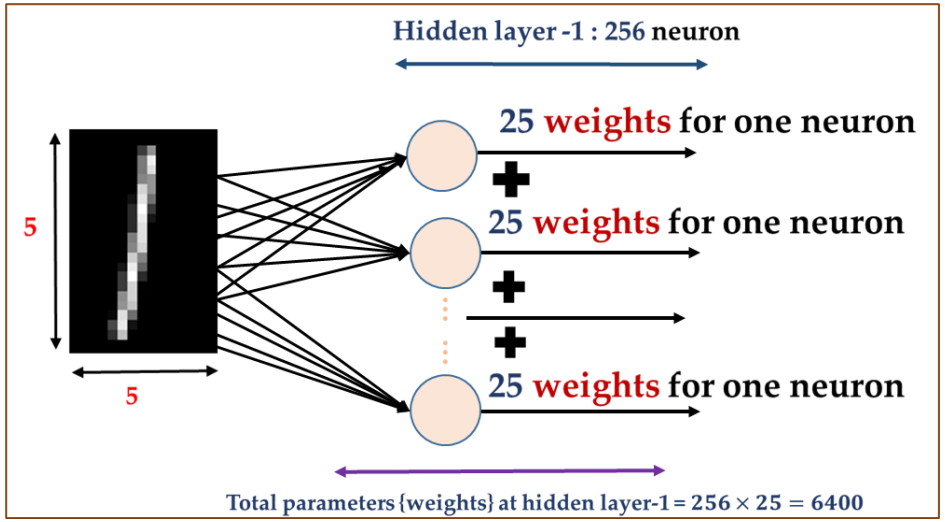
- FCN or MLPs are not known to be parametrically efficient.



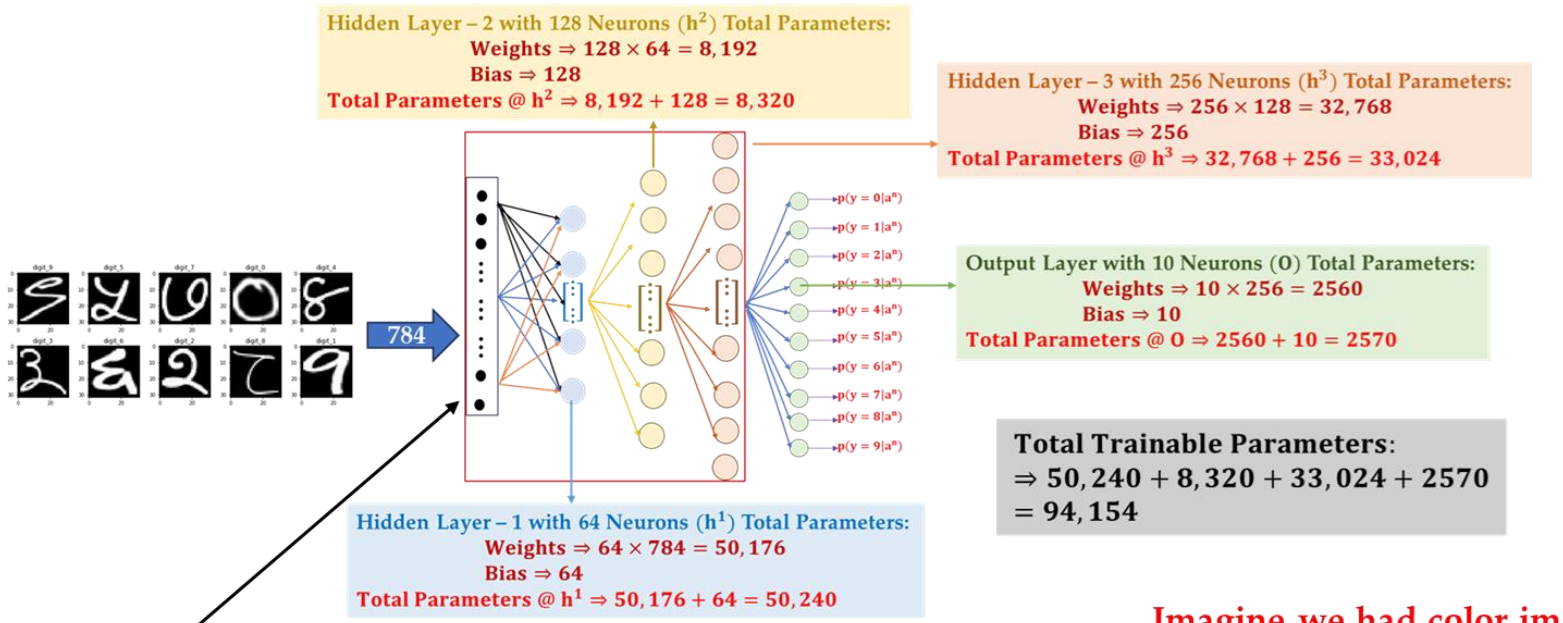
- Why are Too Many parameters a Problem?
- Do we need **so many parameters** to represent such small image?
- What happens if the image is **1024 × 1024 × 3**?
- How do we manage such parameters?
- What Could be the Solution?

2.3.2 Why are Too Many Parameters a Problem?

- When dealing with deep learning models, particularly Fully Connected Networks (FCNs), having **too many parameters** can introduce several challenges:
  - **Increased Computational Cost:**
    - More parameters mean more multiplications and additions, making **training and inference** computationally expensive.
    - Training requires more memory (RAM/VRAM) and computational power (GPUs, /TPUs).
  - **Overfitting:**
    - With **excessive parameters**, the **model can memorize** the **training data** instead of **learning general patterns**, reducing its **ability to generalize to unseen data**, leading to high train accuracy but poor test performance.
  - **Vanishing and Exploding Gradients:**
    - Deep networks with too many parameters suffer from unstable gradients, slowing down training or causing divergence.



2.3.3 What Could be the Solution?



Imagine we had color image:

If we can reduce the number of parameters at Input layer, we can significantly reduce the number of parameters.

How can we do that?

- Instead of flattening an Image, can we extract meaningful features of an Image reducing the dimension.
  - Flattening the image cause to lose the spatial relationship between pixels,
    - Instead, we can process local regions of the image extracting relevant features while keeping the spatial structure intact.

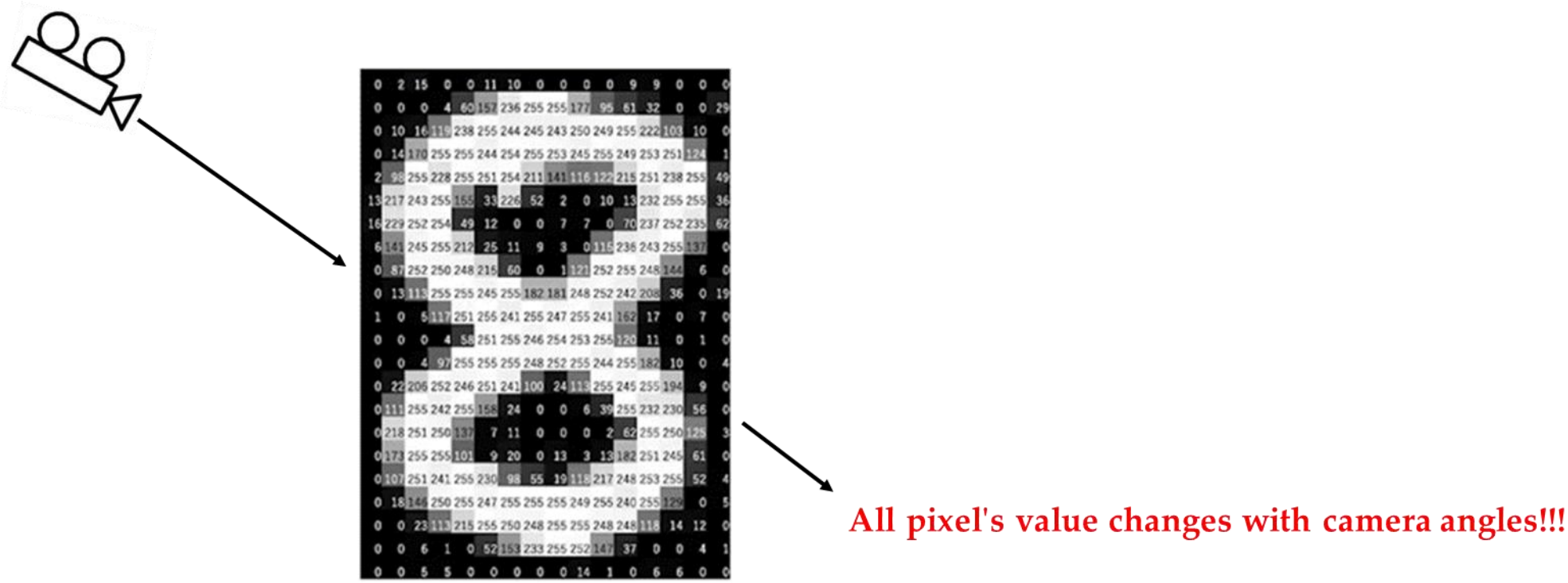
and **Why FCN can not do that**.

2.3.4 Why Can FCN do that?

- **FCN were not designed for Image Recognition Task:**
  - FCN treats each pixel as an independent feature, Thus require a massive number of parameters to capture the patterns.
  - FCNs can not efficiently extract spatial features, thus they do not leverage local structures in images.
  - They can not detect local textures, shapes, or edges effectively making them unsuitable for images.
  - Features could be extracted manually using classical computer vision approach like edge detecting, blurring etc.,
    - But handcrafted features may not generalize well and require domain expertise.
      - Thus, the need for automated feature extraction i.e. features has to be learned and extracted as we train our model to learn the parameters.
- Can we build an algorithm that leverages the local structures of an image to extract features, promotes parameter sharing locally, and preserves important spatial information while reducing dimensionality by retaining only the most significant features?
- For motivation can you think **"How might our brain recognize the Image?"**

2.4 Challenges of Processing Image Data for Feature Extraction.

- Challenge – 1 – Viewpoint Variation:



2.4 Challenges of Processing Image Data for Feature Extraction.

- **Challenge2 : Illumination:**

image copy right CC01.0 public domain

- **Challenge3: Deformation:**

image copy right CC01.0 public domain

2.4 Challenges of Processing Image Data for Feature Extraction.

- **Challenge 4: Occlusion:**

image copy right CC01.0 public domain

- **Challenge 5: Background Clutter:**

image copy right CC01.0 public domain

2.4 Challenges of Processing Image Data for Feature Extraction.

- **Challenge 6: Intra-class Variation:**

image copy right CC01.0 public domain

- **Challenge 7: And many More……**

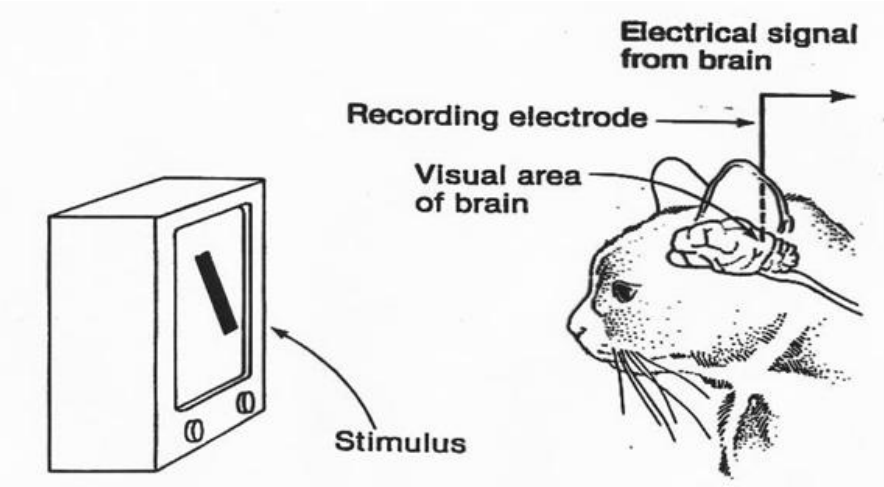
- Can we build an algorithm that leverages the **local structures of an image** to **extract features**, promotes **parameter sharing locally, and preserves important spatial information** while **reducing dimensionality** by retaining only the **most significant features**?
  - Yes → **Convolutional Neural Network**
- For motivation can you think **"How might our brain recognize the Image?"**

3. Convolutional Neural Network.

{ Inspired by Human: How do we recognize an Image?}

3.1 Motivation from Brain.

- **Hubel and Wiesel Experiment (1962):**
- Hubel and Wiesel's research demonstrated how the visual cortex process images in a hierarchical manner. In a simpler terms, our brain does not recognize objects all at once – it builds up understanding step by step through different layer of processing, where different layer respond to specific visual features:
  - **Early layers detect basic features such as edges, orientations, and colors.**
  - **Intermediate layers capture more complex structures, like textures or object parts.**
  - **Final layers integrate these features to recognize full objects.**
- The brain makes decisions by combining information from all layers.



Hubel & Wiesel Experiment

1. Primary visual area (area17)
2. Visual association area (area 18 & 19)
3. Higher visual association area (area 39)

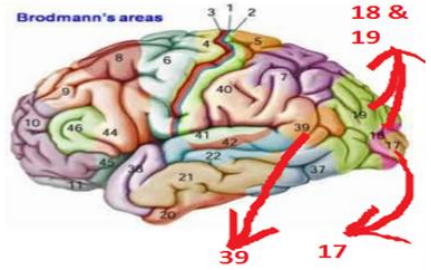
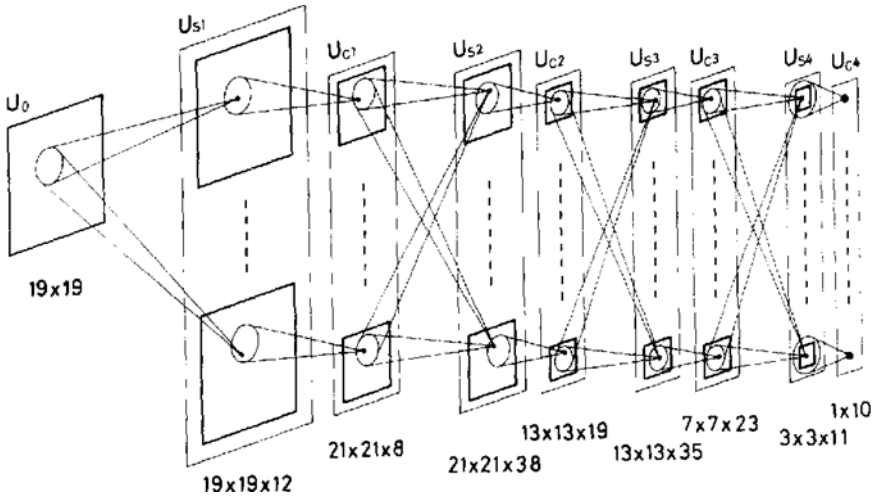


Fig: Known Visual Cortex layer which work together for image recognition.

Hubel, D. H., & Wiesel, T. N. (1962). "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex." The Journal of Physiology, 160(1), 106–154. https://doi.org/10.1113/jphysiol.1962.sp006837

3.2 Neocognitron and Further.

- In 1980 Kunihiko Fukushima introduced Neocognitron, a hierarchical neural network designed for pattern recognition, Inspired by Hubel and Wiesel's findings, it featured:
  - Multiple layers that progressively extracted features.
  - Local receptive fields to detect simple patterns in early layers.
  - Weight sharing to recognize shapes regardless of position.
  - Unlike modern deep learning models, the Neocognitron relied solely on self organizing forward structure for learning **without the use of backpropagation**.



Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological Cybernetics, 36(4), 193–202. https://doi.org/10.1007/BF00344251

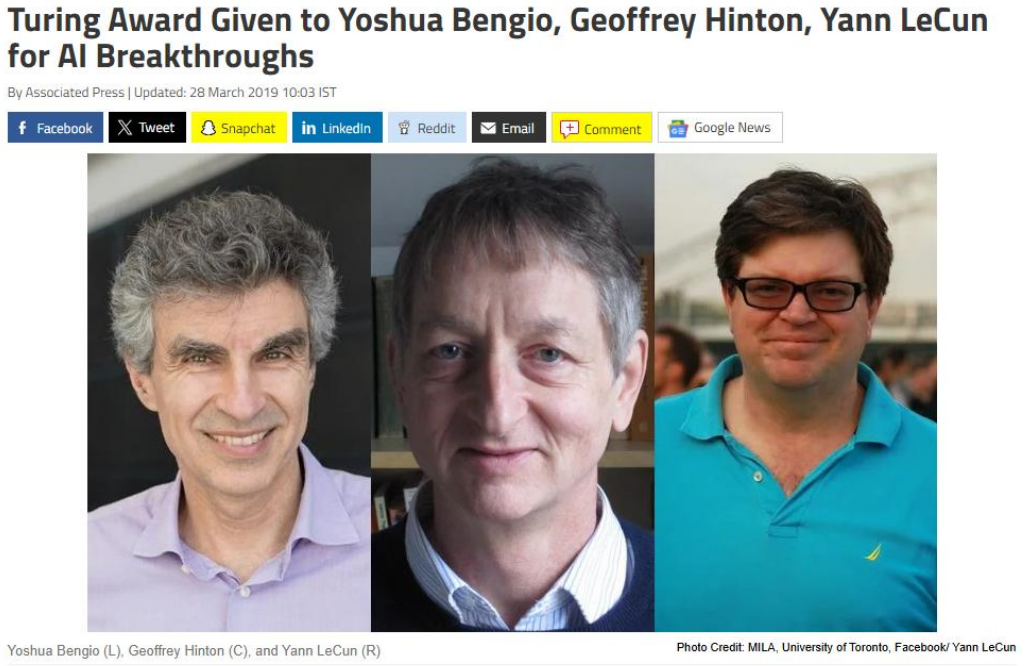
3.3 Introduction of Backpropagation and Further.

- The first use of backpropagation in neural networks can be tracked back to work of Paul Werbos in 1974. In his thesis, werbos introduced the backpropagation algorithm for training multi – layer neural networks.
- However, backpropagation became widely known and popular after the groundbreaking work by Geoffery Hinton, David Parker and Ronald J. Williams in the 1980's particularly with the publication of the 1986 paper
  - *"Learning representations by back – propagating error"*
- This work demonstrated how backpropagation could be used to efficiently train deep neural networks by computing gradients of the error with respect to each weight in the network, enabling gradient-based optimization.
- This is considered a seminal work in AI research, Thus Geoffery Hinton along with John Hopfield was awarded with Nobel Prize in Physics for 2024.
- {**John J. Hopfield** developed an associative memory model that allows for the storage and reconstruction of patterns within data, providing a framework for understanding how information can be retrieved in neural networks.}

Image source Reuters!

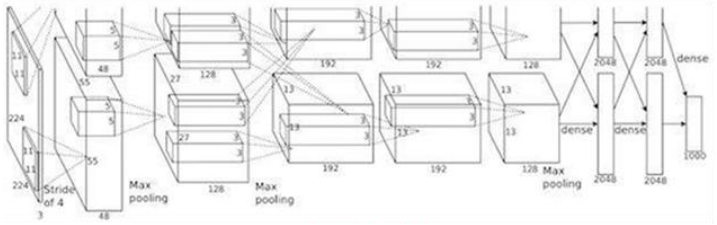
3.3.1 LeNet – 5: A very first CNN Architecture.

- In **1998**, **Yann LeCun**, **Léon Bottou**, **Yoshua Bengio**, and **Patrick Haffner** published a significant paper titled
  - "**Gradient-Based Learning Applied to Document Recognition**".
- In this work, they proposed a CNN architecture for recognizing handwritten digits(MNIST), which was a critical step in the application of CNNs to real-world image classification tasks.
- This paper introduced the **LeNet-5 architecture**, which was one of the first CNNs that successfully demonstrated the potential of deep learning for computer vision.



3.3.2 LeNet to AlexNet.

- The development of **AlexNet** in 2012, led by **Alex Krizhevsky**, **Ilya Sutskever**, and **Geoffrey Hinton**, marked a significant leap forward in the **application of CNNs** to **large-scale image classification**.
- AlexNet demonstrated the **power of deep learning for visual recognition** tasks by **winning the ImageNet competition** with a **substantial margin**, drastically reducing error rates compared to traditional computer vision methods.
- It leveraged techniques such as **ReLU activations**, **dropout**, and **data augmentation** to train a deep network with 8 layers, laying the foundation for many subsequent advancements in computer vision and deep learning.
- {we will discuss all above techniques in Next Classes.}



"AlexNet"

Ilya was a Founder and Chief Scientist in OpenAI which gave us chatgpt.

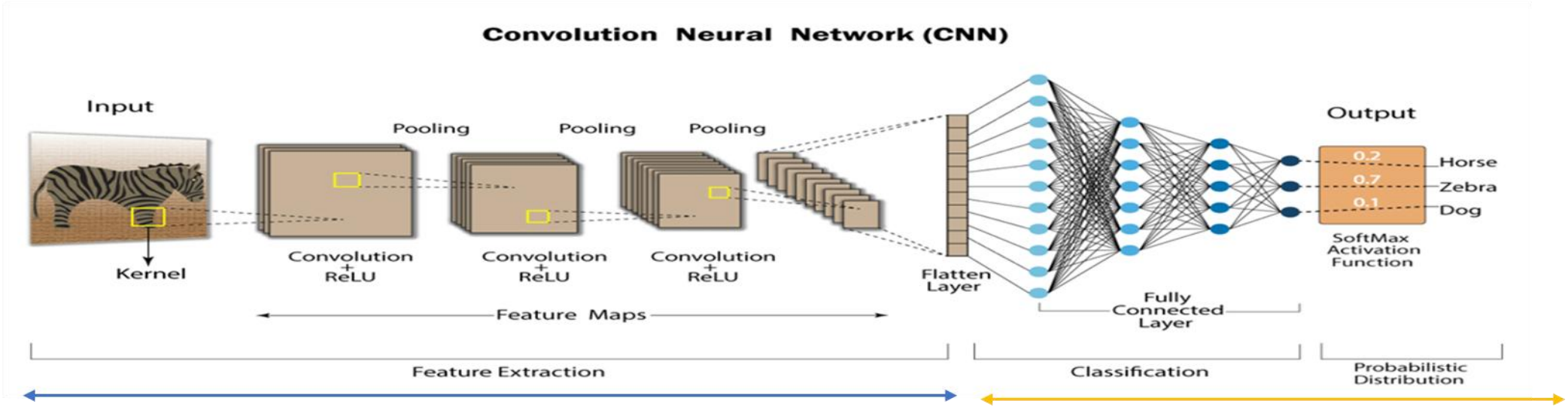
But what Exactly is CNN?

3.4 CNN: Convolutional Neural Network.

- **Convolutional Neural Networks {CNN} aka convnets** are a special case of **fully connected neural network.**
  - They are similar **to the neural network** we discussed and build **last week** as they are also made of **neurons** with **learnable weights and biases**
- The essential difference is that **CNNs are designed** with **implicit assumptions** that **inputs** are in the **image format**.
  - Which allows us **to encode certain properties** into the **architecture**.
  - They are **add-ons** on the **fully connected neural network**, used **to extract the features** while **preserving spatial information and promotes weight sharing.**
    - This is achieved by following layer of operations:
      - **Convolution**
      - **Activation → {ReLU}**
      - **Pooling**
      - **Fully Connected layers**

3.5 CNN's Architecture.

- A typical CNN architecture has a canonical structure inspired by human vision cortex and looks like:
  - [**INPUT → Convolution →Activation → Pooling → FCN**]



3.6 What is Convolution?

- Convolution is a **mathematical operation** that combines two functions or signals to produce a third function.
- In the context of **image processing and deep learning**, it's typically **used to apply a filter (also known as kernel)** to **an image** in order to extract important features like edges, textures, or patterns.
- In simple terms, convolution helps the model detect patterns in local regions of the image by **sliding a small filter(matrix)** over the image and performing a **series of mathematical operations**.
- Mathematical Definition of Convolution:
  - The convolution operation between an image I and a filter (kernel) K is defined as:
  - $(\mathbf{I} * \mathbf{K})_{x,y} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \mathbf{I}(x+i, y+j) \cdot \mathbf{K}(i,j)$
  - Where:
    - $(x, y) \rightarrow$ are the coordinates of the pixel in the output feature map (**convolved image**).
    - $m \times n \rightarrow$ size of the kernel/Filter **K**.
    - The sum represents a weighted sum of the pixel values in **the Image I** at positions that corresponds to **the filter K.**

3.6 What is Convolution?

- Convolution is a **mathematical operation** that combines two functions or signals to produce a third function.
- In the context of **image processing and deep learning**, it's typically **used to apply a filter (also known as kernel)** to **an image** in order to extract important features like edges, textures, or patterns.
- In simple terms, convolution helps the model detect patterns in local regions of the image by **sliding a small filter(matrix)** over the image and performing a **series of mathematical operations**.
- Mathematical Definition of Convolution:
  - The convolution operation between an image I and a filter (kernel) K is defined as:
  - $(\mathbf{I} * \mathbf{K})_{x,y} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \mathbf{I}(x+i, y+j) \cdot \mathbf{K}(i,j)$
  - Where:
    - $(x, y) \rightarrow$ are the coordinates of the pixel in the output feature map (**convolved image**).
    - $m \times n \rightarrow$ size of the kernel/Filter **K**.
    - The sum represents a weighted sum of the pixel values in **the Image I** at positions that corresponds to **the filter K.**

Confused Let me illustrate!!

3.6.1 Illustrations of Convolution.

$$\text{Image}_{28\times28\times1} * \begin{bmatrix} -1 & 0 & 1 \\ -2 & -0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = ?$$

$$\text{Filter}_{3\times3}$$

- Q: Can you multiply above two matrix?
  - **A: No, dimension check fails.**
- Solution: If we interpret Image and Filter as a Function i.e.
  - $I^{[0\ -255]_{(28\times28\times1)}}$
    - Meaning: The image is $28 \times 28 \times 1$ matrix (Grays scale image), where pixel values (**element of a matrix**) are in $0 - 255$.
  - **and** $F^{\mathbb{R}_{3\times3\times1}}$**:**
    - Meaning: The Filter is $3 \times 3 \times 1$ matrix where each element can be any value from $-\infty$ **to** $+\infty$.
  - we can convolve Filter upon Image using sliding window operation.

3.6.2 Illustrations of Convolution.

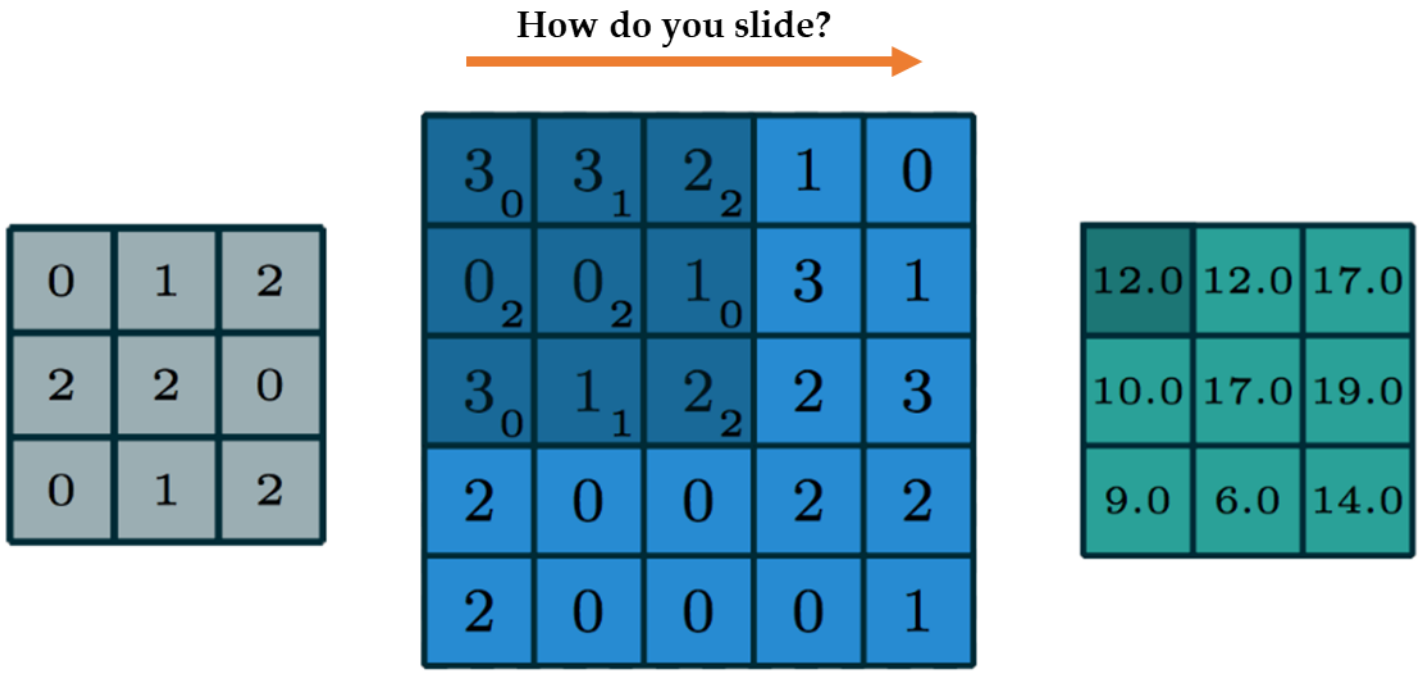


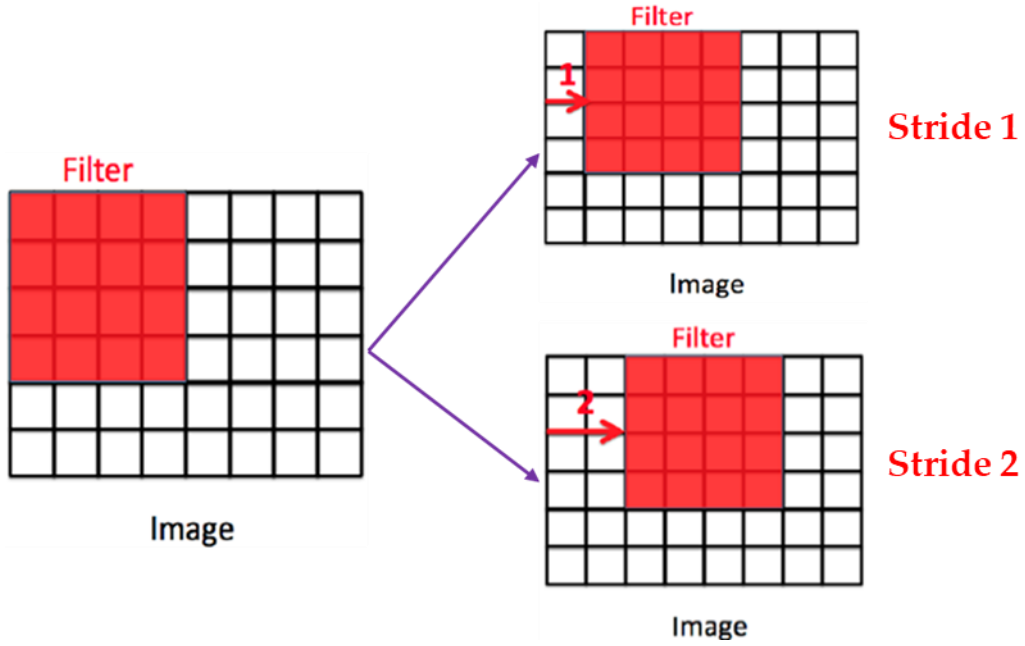
Fig: Convolution Operations.

Heads Up:

- Number of channels in Feature is same as input,
  - If image is gray scale with shape $28 \times 28 \times 1$ Filter will also be $5 \times 5 \times 1$
  - For color image of shape $28 \times 28 \times 3$ Filter will also be $5 \times 5 \times 3$.

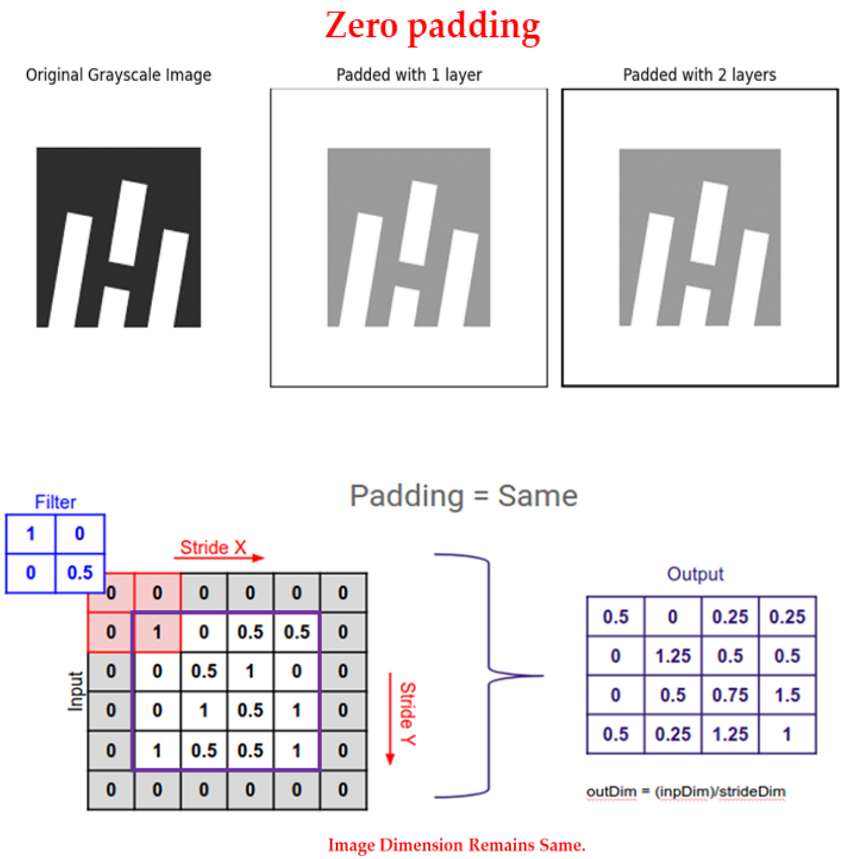
3.6.2.1 Stride.

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

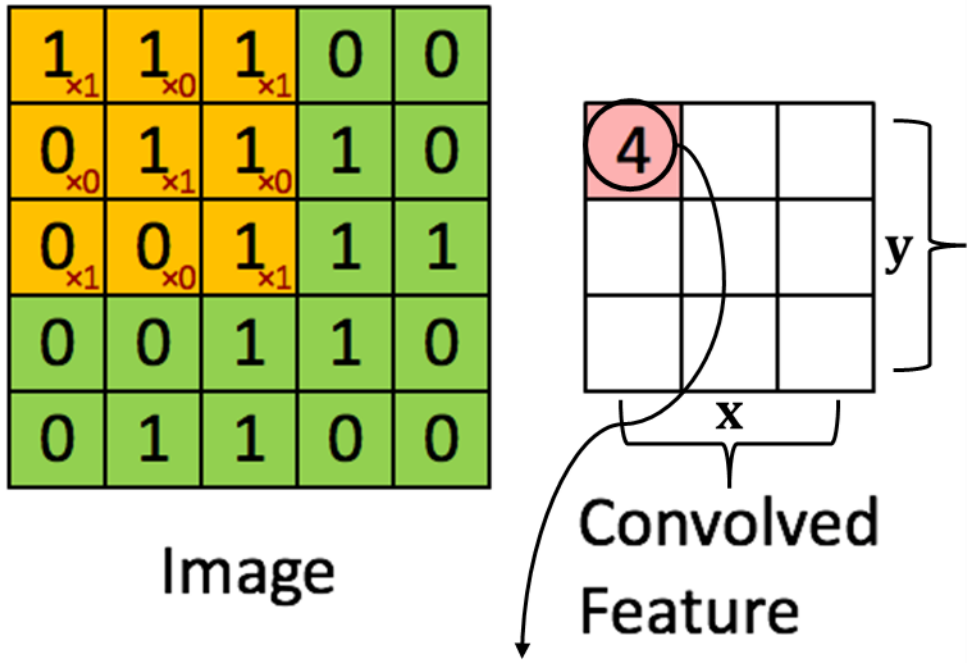


3.6.2.2 Padding.

- 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.



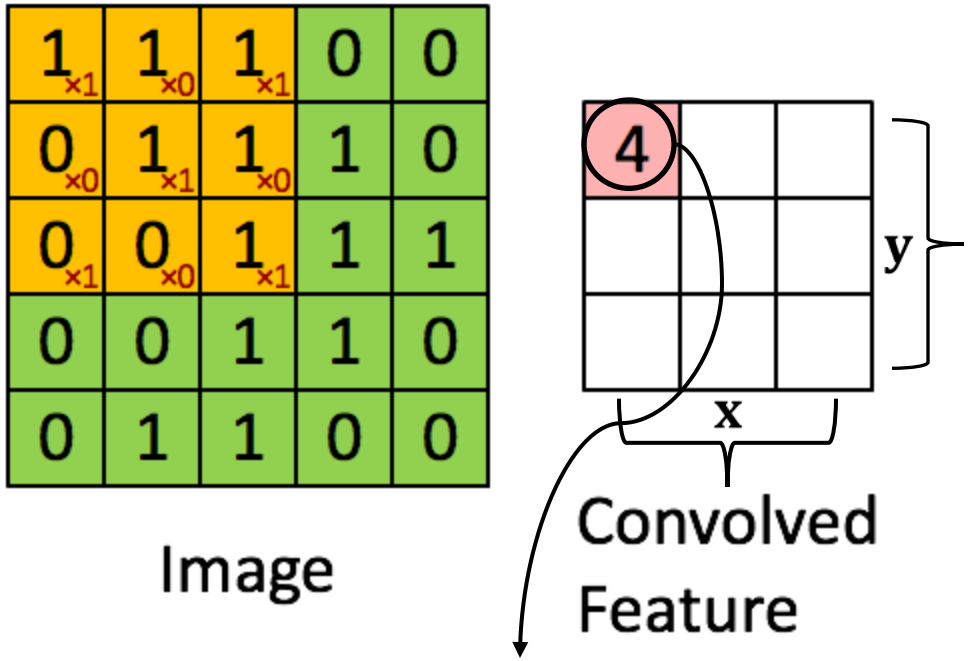
3.6.3 Complete Convolution.



$$O_{11} = 1 \times 1 + 1 \times 0 + 1 \times 1 + 0 \times 0 + 1 \times 1 + 1 \times 0 + 0 \times 1 + 0 \times 0 + 1 \times 1 = 4$$

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

3.6.3 Complete Convolution.



$$O_{11} = 1 \times 1 + 1 \times 0 + 1 \times 1 + 0 \times 0 + 1 \times 1 + 1 \times 0 + 0 \times 1 + 0 \times 0 + 1 \times 1 = 4$$

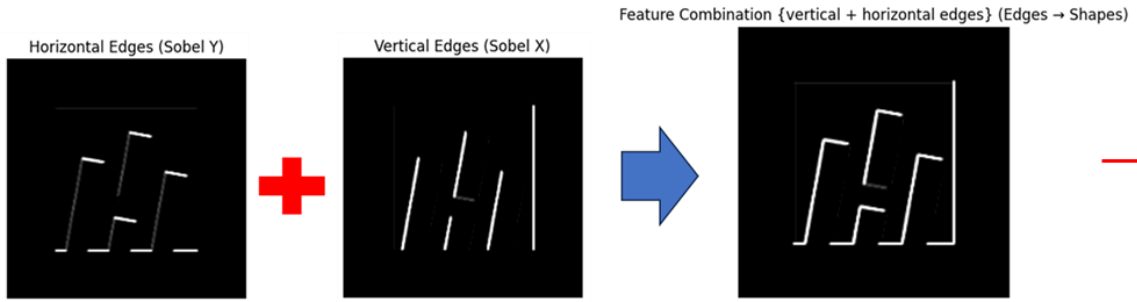
$$(I * F)_{x,y} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} I(x+i, y+j) \cdot F(i,j)$$

3.6.4 Why Convolve with Filter?

- This help us to extract various features, for example:

This set of weight learn the vertical edges.

Different set of weight learn the different features Horizontal edges.

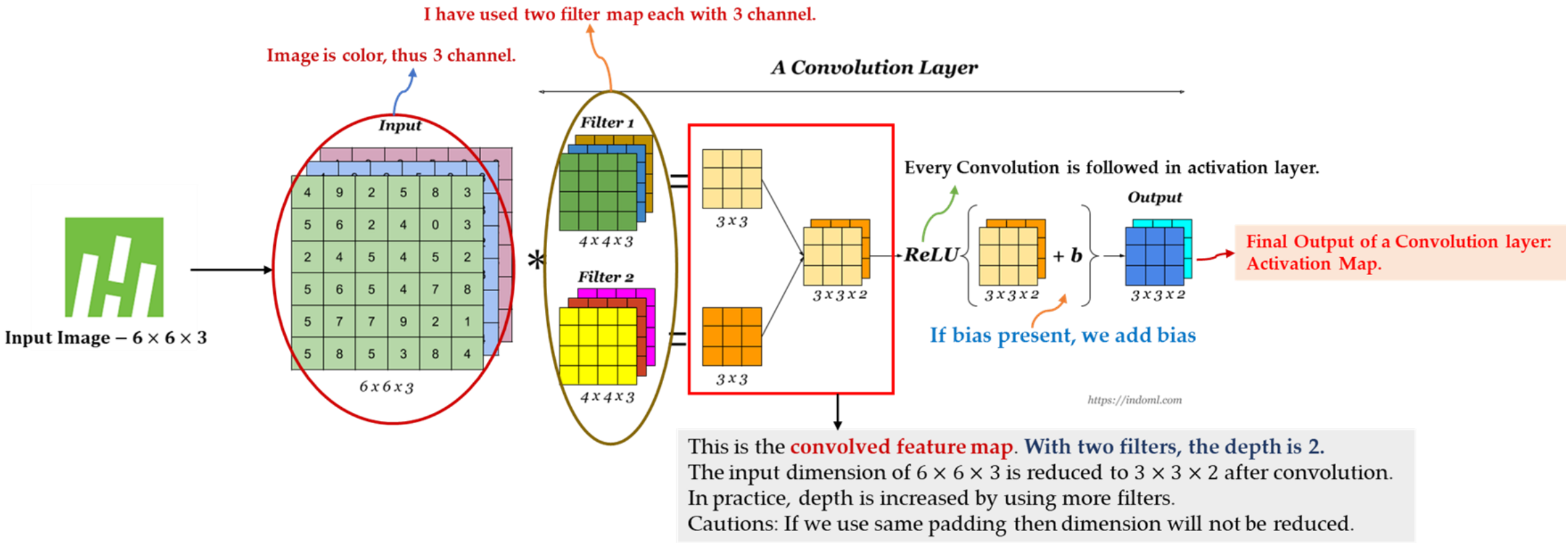


This hierarchical feature extraction, inspired by the human visual cognition ability, is a powerful approach that enables CNN to perform complex tasks such as object recognition.

Features extracted by different Filter and set of weights can be combined to Extract more advance feature.

3.6.5 One Complete Convolution Operation.

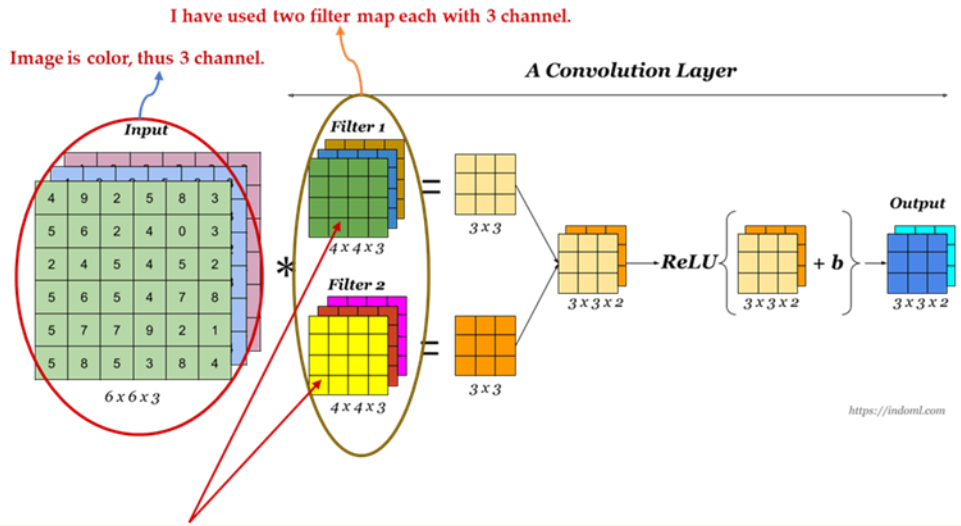
- After the convolution operation we apply an activation function, the go to choice of activation function for CNN Models are ReLU.



3.6.6 Output Dimension after Convolution Operation.

- A convolutional Layer is defined by following **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**, 7×7 are also used by some design, but above than 7 are used rarely.
  - **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 same size as input.
    - ***Valid Padding*** $(P = 0)$***:*** No extra padding, output size shrinks.
- Output Shape Calculation:
  - Given an input of size $W_{in} \times H_{in} \times C_{in}$ the output volume dimensions are:
    - $W_{oc} = \frac{(W_{in} - F + 2P)}{S} + 1$
    - $H_{oc} = \frac{H_{in} - F + 2P}{S} + 1$
    - $C_{oc} = K$ (Number of filters, which determines the depth of the output)

3.6.7 The benefits of Convolution:



- For an input of $6 \times 6 \times 3$:
- **Fully Connected Network (FCN):**
  - Each input pixel ($6 \times 6 \times 3 = 108$) requires its own weight, resulting 108 weights excluding bias.
  - Each neuron in the next Layer would need a separate set of weights, leading to rapid parameter growth.
- **Convolutional Layer:**
  - Using two filters of size 4×4, each filter has a depth of 3 to match the input channels. Requiring:

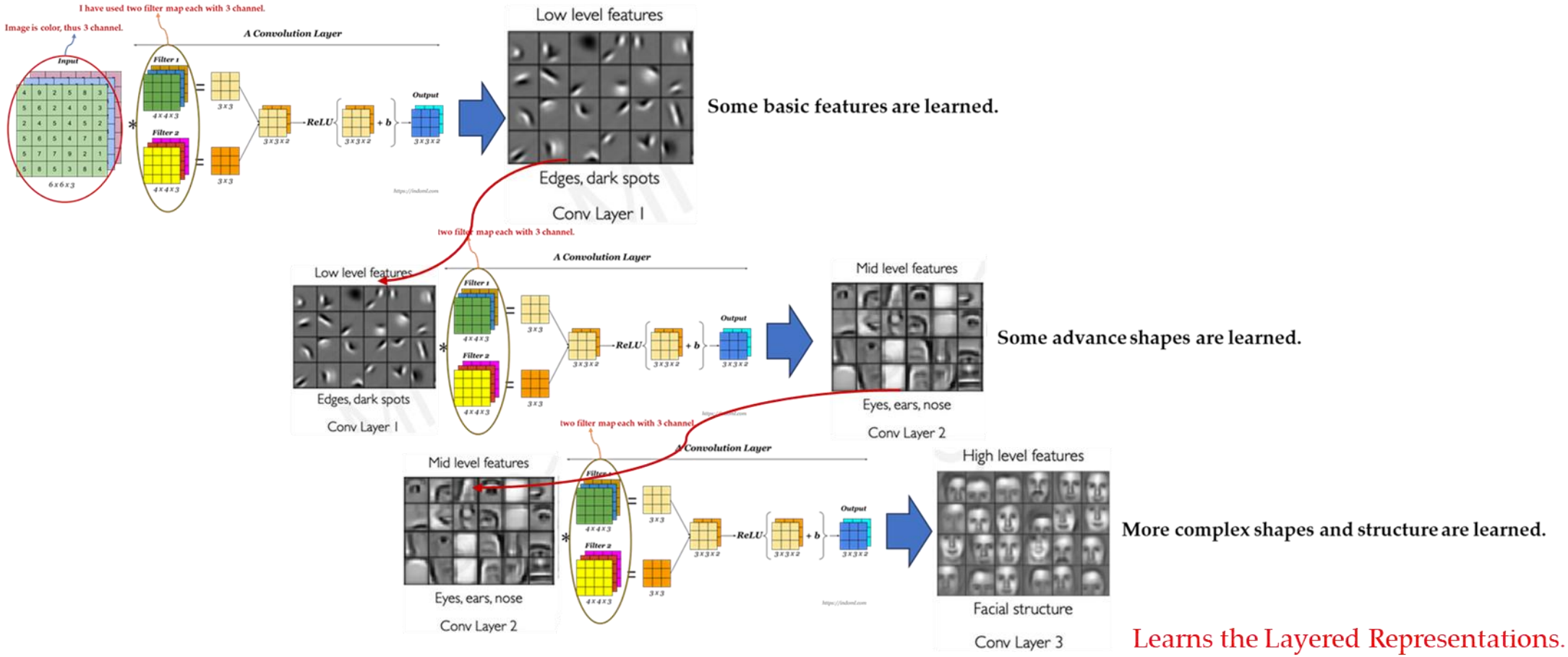
    $4 \times 4 \times 3 = 48$ (weights per filter.)
  - Since we use two filters, total parameters required:

    $48 + 48 = 96$

This significantly fewer than an FC layer, making convolution more efficient by leveraging parameter sharing and sparse connectivity.

- Key Takeaways:
- **Parameter Sharing –** The same filter is applied across spatial locations, reducing the number of unique parameters.
- **Local Connectivity –** Unlike FC layers, which connect every input to every output, convolution focuses on small local regions, preserving spatial structure.

3.6.7 The benefits of Convolution – II:



3.7 Pooling Layer.

- One of the objectives of Convolution Operation is also to reduce the spatial dimension (i.e. **height × width**) and increase the depth i.e. **channel**.
  - However, this reduction in spatial dimensions is not always guaranteed in convolution layer, especially when using **same padding**, which preserves **the spatial dimensions**.
- Therefore, a pooling layer is typically used **after convolution** to **reduce spatial dimensions** while **retaining essential information.**
- **Type of Pooling Operation:**

3.7.1 Pooling Operation : Demo.

- Pooling Operation is also achieved via window sliding operation with stride.

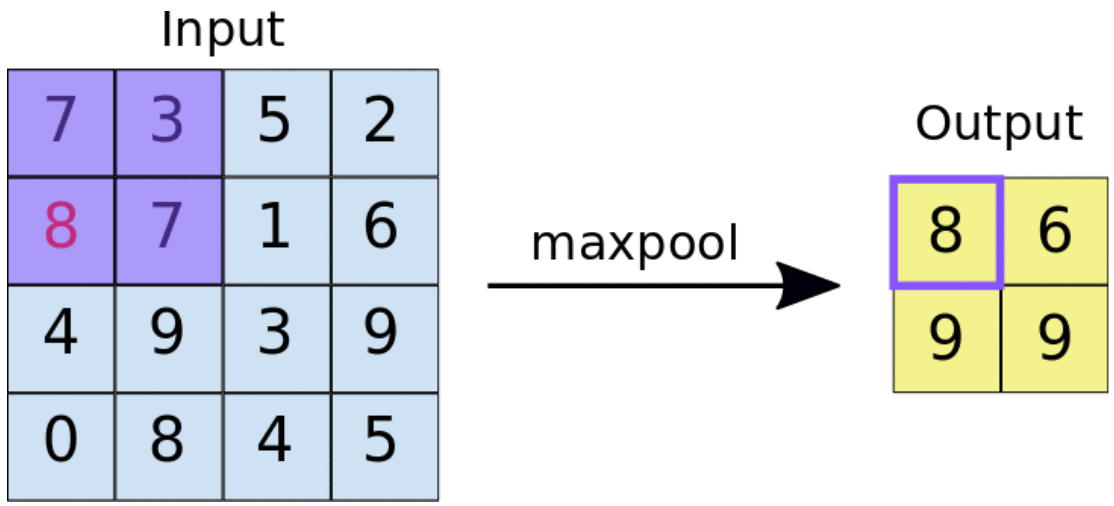


Fig: Max Pooling with Stride 1.

3.7.2 Output Dimension after Pooling.

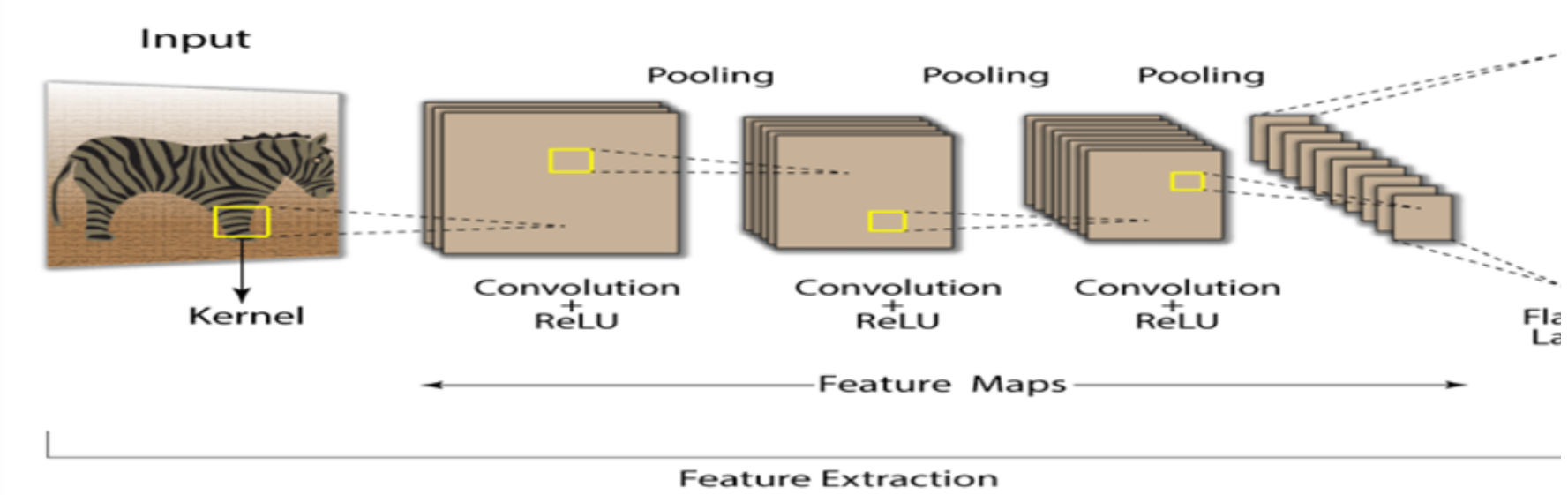
- **Hyper-parameters in the Pooling Layer:**
  - **Size of Filters (F):** For max pooling, it is typically **2×2.**
  - **Stride (S):** When used **2×2 pooling filter stride of 2** is most common, this reduces the spatial dimension by half.
- **A pooling layer takes an input of volume:**
  - **$W_{oca} \times H_{oca} \times C_{oca}$ {oca → Output after convolution and activation.}**
- And produces an output of volume:
  - **$W_{ocap} \times H_{ocap} \times C_{ocap}$ {ocap → Output after convolution, activation and pooling.}**
- Where:
  - $W_{ocap} = \frac{W_{oca} - F}{S} + 1$
  - $H_{ocap} = \frac{H_{oca} - F}{S} + 1$
  - $C_{ocap} = C_{oca}$ **(depth remains unchanaged)**
- The Pooling Layer does not have learnable parameters (weights or biases), so no weight updates are required during training.

4. CNN for Image Classification.

{ An End-to-End Model.}

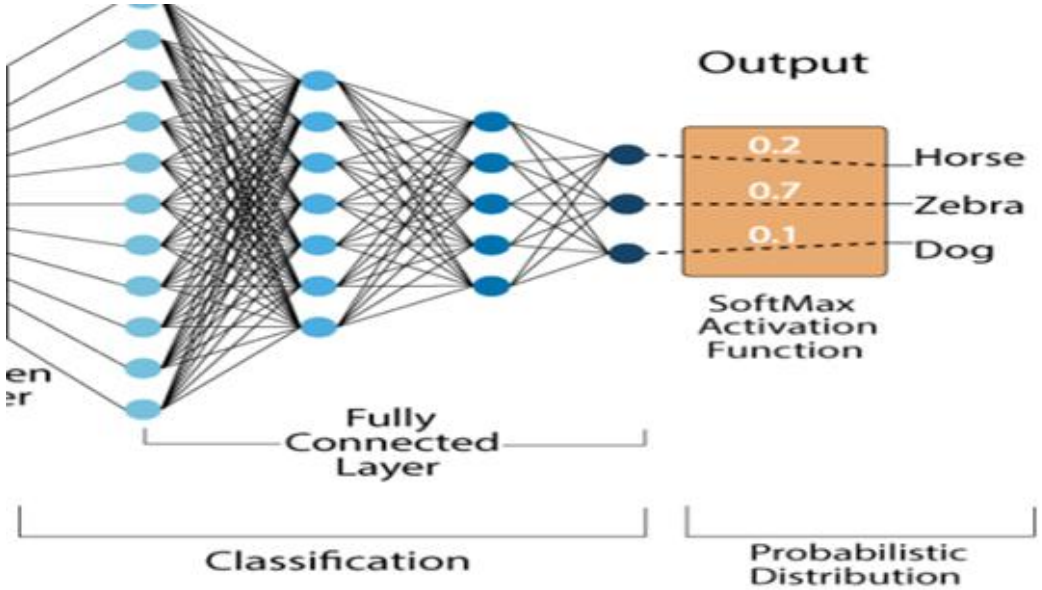
4.1 Convolution for Feature Learning.

- **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.

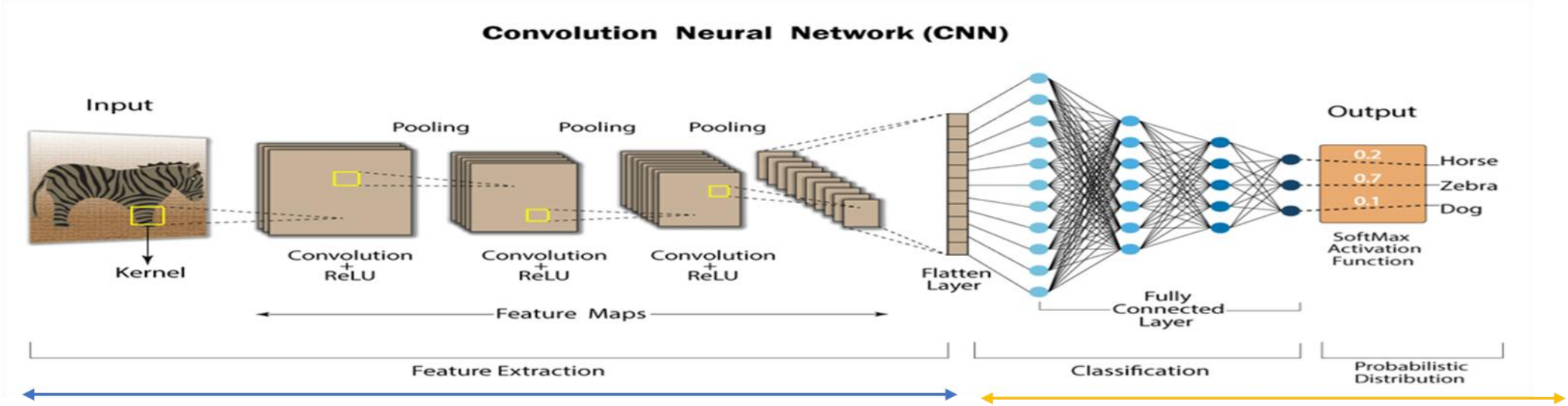


4.2 FCN for Learning Class Probabilities.

- **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.



4.3 CNNs for Image Classification.



Mimics the Early layer of Human Cognition and Extracts the Features.

Receives the dimensionally reduced input features and assign the class.

How do we Train Such Model ?

5. Training a CNN.

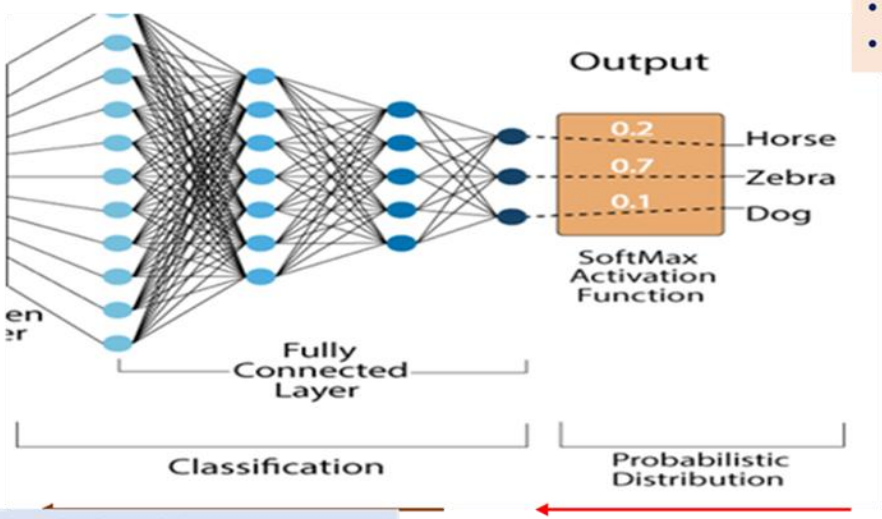
{ A forward Pass with Back Propagation of Error with Gradient Descent.}

5.1 Forward and Backward @ FCN.

- $W^l \rightarrow$ weight matrix for layer l
- $A^{l-1} \rightarrow$ activation from previous layer
- $b^l \rightarrow$ bais vector alyer l
- $z^l \rightarrow$ pre – activation output
- $g(z) \rightarrow$ activation function (ReLU, softmax)

Forward Computation - $Z^l = (w^l A^{l-1} + b^l) \Rightarrow A^l = g(z^l)$

Input is the Flatten Output of Final Convolution Layer



Weight Update Rule:

$$W^l = W^l - \eta \frac{\partial L}{\partial W^l}$$

$$b^l = b^l - \eta \frac{\partial L}{\partial b^l}$$

For Hidden layer (applying the chain rule):

$$\delta^l = \left(W^{l+1^T} \delta^{l+1}\right) \odot g'(Z^l)$$

$\delta^l \rightarrow$ is the error term elementwise.

$g'(Z^l) \rightarrow$ derivative of the activation function.

$W^{l+1^T} \delta^{l+1} \rightarrow$ propagates error from the next layer.

For the Output Layer (Softmax + Cross Entropy):

$$\delta^l = \frac{\partial L}{\partial Z^l} = \hat{y} - y$$

$\delta^l \rightarrow$ is the error term elementwise.

5.2 Forward Computation @ Convolutional Layer.

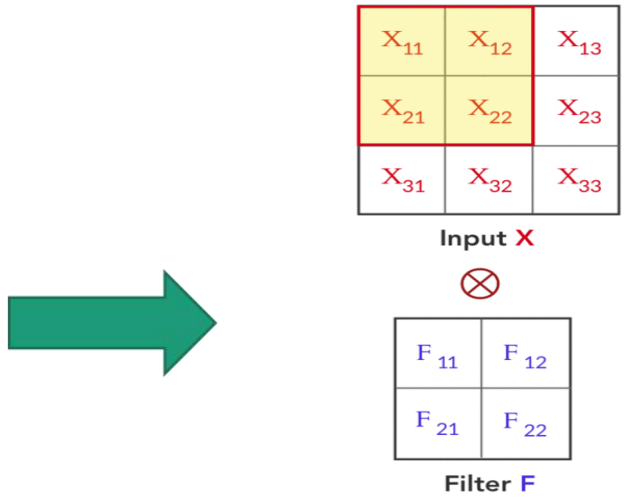
$$\begin{bmatrix} O_{11} & O_{12} \\ O_{21} & O_{22} \end{bmatrix} = \text{Convolution}\left(\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix}, \begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix} \right)$$

$$O_{11} = F_{11}X_{11} + F_{12}X_{12} + F_{21}X_{21} + F_{22}X_{22}$$

$$O_{12} = F_{11}X_{12} + F_{12}X_{13} + F_{21}X_{22} + F_{22}X_{23}$$

$$O_{21} = F_{11}X_{21} + F_{12}X_{22} + F_{21}X_{31} + F_{22}X_{32}$$

$$O_{22} = F_{11}X_{22} + F_{12}X_{23} + F_{21}X_{32} + F_{22}X_{33}$$



$X_{11}F_{11}$	$X_{12}F_{12}$	X_{13}
$X_{21}F_{21}$	$X_{22}F_{22}$	X_{23}
X_{31}	X_{32}	X_{33}

$$O_{11} = X_{11}F_{11} + X_{12}F_{12} + X_{21}F_{21} + X_{22}F_{22}$$

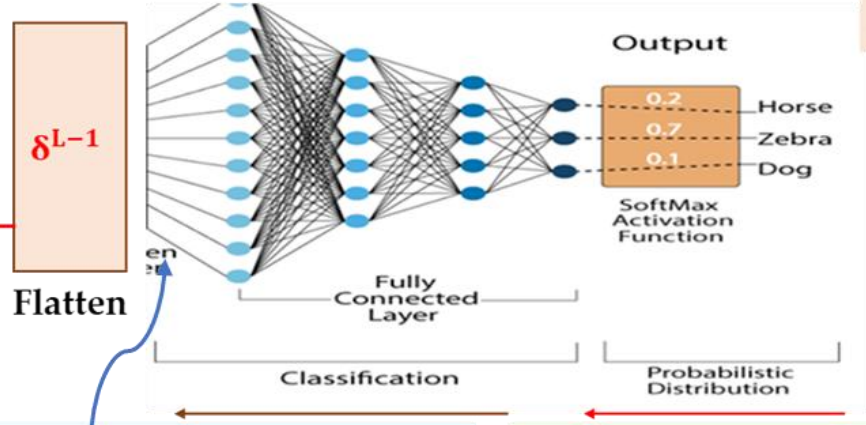
Image from Internet: Subject to copyright.

Same Operation happens at every Layer, the inputs shall change depending on the layer.

5.3 Backpropagation @ Convolutional Layer.

Forward Computation - $Z^l = (w^l A^{l-1} + b^l) \Rightarrow A^l = g(z^l)$

- $W^l \rightarrow$ weight matrix for layer l
- $A^{l-1} \rightarrow$ activation from previous layer
- $b^l \rightarrow$ bais vector alyer l
- $z^l \rightarrow$ pre – activation output
- $g(z) \rightarrow$ activation function (ReLU, softmax)



The First FC Layer receives the flattened feature vector. The gradient must be reshaped back into the CNN's feature map dimensions.

$\delta^{L-1} \rightarrow H_{conv} \times W_{conv} \times C_{conv}$

Weight Update Rule:

$$W^l = W^l - \eta \frac{\partial L}{\partial W^l}$$

$$b^l = b^l - \eta \frac{\partial L}{\partial b^l}$$

For Hidden layer (applying the chain rule):

$$\delta^l = \left(W^{l+1^T} \delta^{l+1}\right) \odot g'(z^l)$$

$\delta^l \rightarrow$ is the error term elementwise.

$g'(Z^l) \rightarrow$ derivative of the activation function.

$W^{l+1^T} \delta^{l+1} \rightarrow$ propagates error from the next layer.

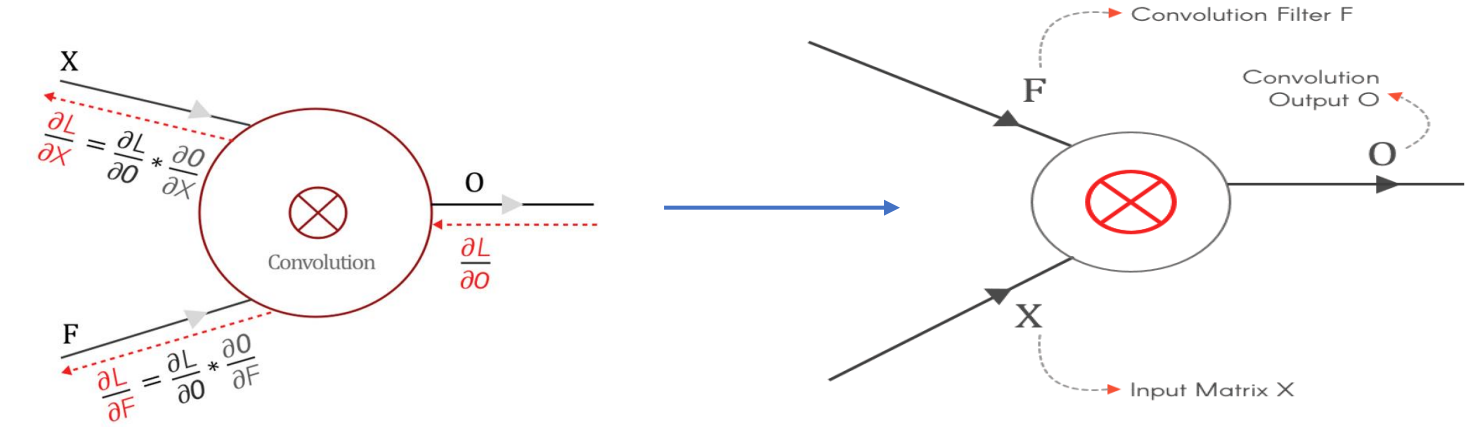
For the Output Layer (Softmax + Cross Entropy):

$$\delta^l = \frac{\partial L}{\partial Z^l} = \hat{y} - y$$

$\delta^l \rightarrow$ is the error term elementwise.

5.4 Backward Pass @ Convolution Layer.

- In the backward pass of a convolutional layer: we perform two main steps:
  - **1. Computing the gradient w.r.t the filter F:**
    - This tells us how much each filter contributes to the loss:
    - Calculated as a convolution between input X and the gradient of the error $\delta = \frac{\partial L}{\partial O}$.
  - **2. Computing the gradient w.r.t the input X:**
    - This tells us how much each pixel in the input contributes to the loss:
    - Calculated as a convolution between flipped filter F and the gradient of the error $\delta = \frac{\partial E}{\partial O}$.



5.4 Backward Pass @ Convolution Layer.

- In the backward pass of a convolutional layer: we perform two main steps:
  - **1. Computing the gradient w.r.t the filter F:**
    - This tells us how much each filter contributes to the loss:
    - Calculated as a convolution between input X and the gradient of the error $\delta = \frac{\partial E}{\partial O}$.

$$\begin{bmatrix} \partial E/\partial F_{11} & \partial E/\partial F_{12} \\ \partial E/\partial F_{21} & \partial E/\partial F_{22} \end{bmatrix} = \text{Convolution}\left(\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix}, \begin{bmatrix} \partial E/\partial O_{11} & \partial E/\partial O_{12} \\ \partial E/\partial O_{21} & \partial E/\partial O_{22} \end{bmatrix} \right)$$

This is the reshaped gradient output from FCN

$$\frac{\partial E}{\partial F_{11}} = \frac{\partial E}{\partial O_{11}} X_{11} + \frac{\partial E}{\partial O_{12}} X_{12} + \frac{\partial E}{\partial O_{21}} X_{21} + \frac{\partial E}{\partial O_{22}} X_{22}$$

$$\frac{\partial E}{\partial F_{12}} = \frac{\partial E}{\partial O_{11}} X_{12} + \frac{\partial E}{\partial O_{12}} X_{13} + \frac{\partial E}{\partial O_{21}} X_{22} + \frac{\partial E}{\partial O_{22}} X_{23}$$

$$\frac{\partial E}{\partial F_{21}} = \frac{\partial E}{\partial O_{11}} X_{21} + \frac{\partial E}{\partial O_{12}} X_{22} + \frac{\partial E}{\partial O_{21}} X_{31} + \frac{\partial E}{\partial O_{22}} X_{32}$$

$$\frac{\partial E}{\partial F_{22}} = \frac{\partial E}{\partial O_{11}} X_{22} + \frac{\partial E}{\partial O_{12}} X_{23} + \frac{\partial E}{\partial O_{21}} X_{32} + \frac{\partial E}{\partial O_{22}} X_{33}$$

5.4 Backward Pass @ Convolution Layer.

- In the backward pass of a convolutional layer: we perform two main steps:
  - **2. Computing the gradient w.r.t the input X:**
    - This tells us how much each pixel in the input contributes to the loss:
    - Calculated as a convolution between flipped filter F and the gradient of the error $\delta = \frac{\partial E}{\partial O}$.

$$\begin{bmatrix} \partial E/\partial X_{11} & \partial E/\partial X_{12} & \partial E/\partial X_{13} \\ \partial E/\partial X_{21} & \partial E/\partial X_{22} & \partial E/\partial X_{23} \\ \partial E/\partial X_{31} & \partial E/\partial X_{32} & \partial E/\partial X_{33} \end{bmatrix} = \text{Full\_Convolution} \left(\begin{bmatrix} \partial E/\partial O_{11} & \partial E/\partial O_{12} \\ \partial E/\partial O_{21} & \partial E/\partial O_{22} \end{bmatrix}, \begin{bmatrix} F_{22} & F_{21} \\ F_{12} & F_{11} \end{bmatrix} \right)$$

Flipped Filter.

Reshaped gradient from last layer of FCN.

$$\begin{aligned}
\frac{\partial E}{\partial X_{11}} &= \frac{\partial E}{\partial O_{11}} F_{11} + \frac{\partial E}{\partial O_{12}} 0 + \frac{\partial E}{\partial O_{21}} 0 + \frac{\partial E}{\partial O_{22}} 0 \\
\frac{\partial E}{\partial X_{12}} &= \frac{\partial E}{\partial O_{11}} F_{12} + \frac{\partial E}{\partial O_{12}} F_{11} + \frac{\partial E}{\partial O_{21}} 0 + \frac{\partial E}{\partial O_{22}} 0 \\
\frac{\partial E}{\partial X_{13}} &= \frac{\partial E}{\partial O_{11}} 0 + \frac{\partial E}{\partial O_{12}} F_{12} + \frac{\partial E}{\partial O_{21}} 0 + \frac{\partial E}{\partial O_{22}} 0 \\
\frac{\partial E}{\partial X_{21}} &= \frac{\partial E}{\partial O_{11}} F_{21} + \frac{\partial E}{\partial O_{12}} 0 + \frac{\partial E}{\partial O_{21}} F_{11} + \frac{\partial E}{\partial O_{22}} 0 \\
\frac{\partial E}{\partial X_{22}} &= \frac{\partial E}{\partial O_{11}} F_{22} + \frac{\partial E}{\partial O_{12}} F_{21} + \frac{\partial E}{\partial O_{21}} f_{12} + \frac{\partial E}{\partial O_{22}} F_{11} \\
\frac{\partial E}{\partial X_{23}} &= \frac{\partial E}{\partial O_{11}} 0 + \frac{\partial E}{\partial O_{12}} F_{22} + \frac{\partial E}{\partial O_{21}} 0 + \frac{\partial E}{\partial O_{22}} F_{11} \\
\frac{\partial E}{\partial X_{31}} &= \frac{\partial E}{\partial O_{11}} 0 + \frac{\partial E}{\partial O_{12}} 0 + \frac{\partial E}{\partial O_{21}} F_{21} + \frac{\partial E}{\partial O_{22}} 0 \\
\frac{\partial E}{\partial X_{32}} &= \frac{\partial E}{\partial O_{11}} 0 + \frac{\partial E}{\partial O_{12}} 0 + \frac{\partial E}{\partial O_{21}} F_{22} + \frac{\partial E}{\partial O_{22}} F_{21} \\
\frac{\partial E}{\partial X_{33}} &= \frac{\partial E}{\partial O_{11}} 0 + \frac{\partial E}{\partial O_{12}} 0 + \frac{\partial E}{\partial O_{21}} 0 + \frac{\partial E}{\partial O_{22}} F_{22}
\end{aligned}$$

5.4 Backward Pass @ Convolution Layer.

- In the backward pass of a convolutional layer: we perform two main steps:
  - **2. Computing the gradient w.r.t the input X:**
    - This tells us how much each pixel in the input contributes to the loss:
    - Calculated as a convolution between flipped filter F and the gradient of the error $\delta = \frac{\partial E}{\partial O}$.

$$\begin{bmatrix} \partial E/\partial X_{11} & \partial E/\partial X_{12} & \partial E/\partial X_{13} \\ \partial E/\partial X_{21} & \partial E/\partial X_{22} & \partial E/\partial X_{23} \\ \partial E/\partial X_{31} & \partial E/\partial X_{32} & \partial E/\partial X_{33} \end{bmatrix} = \text{Full\_Convolution} \left(\begin{bmatrix} \partial E/\partial O_{11} & \partial E/\partial O_{12} \\ \partial E/\partial O_{21} & \partial E/\partial O_{22} \end{bmatrix}, \begin{bmatrix} F_{22} & F_{21} \\ F_{12} & F_{11} \end{bmatrix} \right)$$

Flipped Filter.

Reshaped gradient from last layer of FCN.

$$\begin{aligned}
\frac{\partial E}{\partial X_{11}} &= \frac{\partial E}{\partial O_{11}} F_{11} + \frac{\partial E}{\partial O_{12}} 0 + \frac{\partial E}{\partial O_{21}} 0 + \frac{\partial E}{\partial O_{22}} 0 \\
\frac{\partial E}{\partial X_{12}} &= \frac{\partial E}{\partial O_{11}} F_{12} + \frac{\partial E}{\partial O_{12}} F_{11} + \frac{\partial E}{\partial O_{21}} 0 + \frac{\partial E}{\partial O_{22}} 0 \\
\frac{\partial E}{\partial X_{13}} &= \frac{\partial E}{\partial O_{11}} 0 + \frac{\partial E}{\partial O_{12}} F_{12} + \frac{\partial E}{\partial O_{21}} 0 + \frac{\partial E}{\partial O_{22}} 0 \\
\frac{\partial E}{\partial X_{21}} &= \frac{\partial E}{\partial O_{11}} F_{21} + \frac{\partial E}{\partial O_{12}} 0 + \frac{\partial E}{\partial O_{21}} F_{11} + \frac{\partial E}{\partial O_{22}} 0 \\
\frac{\partial E}{\partial X_{22}} &= \frac{\partial E}{\partial O_{11}} F_{22} + \frac{\partial E}{\partial O_{12}} F_{21} + \frac{\partial E}{\partial O_{21}} f_{12} + \frac{\partial E}{\partial O_{22}} F_{11} \\
\frac{\partial E}{\partial X_{23}} &= \frac{\partial E}{\partial O_{11}} 0 + \frac{\partial E}{\partial O_{12}} F_{22} + \frac{\partial E}{\partial O_{21}} 0 + \frac{\partial E}{\partial O_{22}} F_{11} \\
\frac{\partial E}{\partial X_{31}} &= \frac{\partial E}{\partial O_{11}} 0 + \frac{\partial E}{\partial O_{12}} 0 + \frac{\partial E}{\partial O_{21}} F_{21} + \frac{\partial E}{\partial O_{22}} 0 \\
\frac{\partial E}{\partial X_{32}} &= \frac{\partial E}{\partial O_{11}} 0 + \frac{\partial E}{\partial O_{12}} 0 + \frac{\partial E}{\partial O_{21}} F_{22} + \frac{\partial E}{\partial O_{22}} F_{21} \\
\frac{\partial E}{\partial X_{33}} &= \frac{\partial E}{\partial O_{11}} 0 + \frac{\partial E}{\partial O_{12}} 0 + \frac{\partial E}{\partial O_{21}} 0 + \frac{\partial E}{\partial O_{22}} F_{22}
\end{aligned}$$

Final Gradient Computing Formula:

$$\frac{\partial E}{\partial X} = \delta \otimes \tilde{F}$$

$\tilde{F} \rightarrow$ **Flipped filter**

Looks like both Forward and Backward Computation are Convolution Operation.

Thought for a Doughnut!!!

- You are given a training set with 1M labeled points. When you train a shallow neural net with one **fully-connected feed-forward hidden layer on this data you obtain 86% accuracy on test data**. When you train a deeper neural net as in which consist of a **convolutional layer, pooling layer, and three fully-connected feed-forward layers on the same data you obtain 91% accuracy** on the same test set.
- What might be the source of this improvement?

Thank You