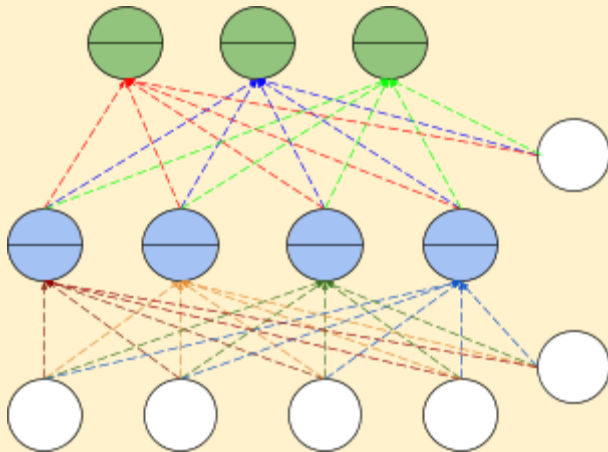


## The Convolution Operation

### Setting the Context

Making sense of everything we have seen so far

1. So far, all the Neural Networks that we have studied are called fully connected networks



2. **Fully Connected Neural Networks:** Any neuron in a given layer is fully connected to all the neurons in the previous layer
3. Let's look at some of the pros and cons of Fully Connected Neural Networks (DNNs)

Pros	Cons
The Universal Approximation Theorem says that DNNs are power function approximators	DNNs are prone to overfitting (too many parameters)
We can come up with a neural network of output $\hat{f}(x)$ which is very close to the true output $f(x)$	Even a slight change in the training set could cause the model to arrive at very different weight configurations
Can be trained using backpropagation	Gradients can vanish due to long chains
In PyTorch, backpropagation is automated.	Vanishing gradient problem could occur in the case of saturated neurons

4. We aimed to mitigate these issues using
  - a. Better Optimization Algorithms
  - b. Better Activation Functions
  - c. Better Initialisation Methods
  - d. Better Regularization
5. Can we have DNNs which are complex (many non-linearities) but have fewer parameters and hence less prone to overfitting?

# PadhAI: The Convolution Operation

## One Fourth Labs

### The 1D convolution operation

What does the convolution operation do?

- Let's approach this with a real world example
- Consider a flight from Chennai to Delhi
  - We measure the distance of the flight from Chennai at regular intervals,
    - $x_0$  at  $t_0$
    - $x_1$  at  $t_1$
    - $x_2$  at  $t_2$
  - In general, to calculate the overall speed, we would take the average speed at these measured points i.e  $\frac{1}{3}(x_0 + x_1 + x_2)$ .
  - However, let us try giving the most importance to the current reading, and a progressively decreasing level of importance to every reading preceding the current one.
  - Let's assign different weights to each of these reading points
    - $x_0 \rightarrow w_0$  (0 indicates current reference point)
    - $x_1 \rightarrow w_{-1}$  (1 reading before reference point)
    - $x_2 \rightarrow w_{-2}$  (1 readings before reference point)
  - So the new overall speed would be calculated by  $w_{-2}x_0 + w_{-1}x_1 + w_0x_2$  where the weights are decreasing from  $w_0$
- The formula could be written as follows
  - $$s_t = \sum_{a=0}^{\infty} w_{-a}x_{t-a} = (x * w)_t$$
  - Where t refers to reference point
  - a is the index of the weight, ranging from 0 for reference point to  $\infty$
- In practice, we wouldn't want to take the reading up till  $-\infty$ , thus we can simply say that those unwanted weights are all 0.
- Consider the following table

	$w_{-6}$	$w_{-5}$	$w_{-4}$	$w_{-3}$	$w_{-2}$	$w_{-1}$	$w_0$					
W	0.01	0.01	0.02	0.02	0.04	0.04	0.05					
X	1.00	1.10	1.20	1.40	1.70	1.80	1.90	2.10	2.20	2.40	2.50	2.70
S							1.80					

- In the above table,  $w_{-7}$  to  $w_{-\infty}$  are all consider to be 0
- Here,  $s_6 = x_6w_0 + x_5w_{-1} + x_4w_{-2} + x_3w_{-3} + x_2w_{-4} + x_1w_{-5} + x_0w_{-6}$

# PadhAI: The Convolution Operation

## One Fourth Labs

### The 2D convolution operation

What about 2D inputs? What are the neighbors that we consider?

1. In a nutshell, the convolution operation boils down to taking a given input and re-estimating it as a weighted average of all the inputs around it.
2. The above definition is easy to visualise in 1D, but what about 2D?
3. In 2D, we would consider neighbors along the rows and columns, using the following formula

$$s_{ij} = (I * K)_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{n-1} I_{i+a, j+b} * K_{a,b}$$

- a. K refers to kernel or weights and I refers to the input. And \* refers to the convolution operation

a = rows

K	$K_{00}$	$K_{01}$
	$K_{10}$	$K_{11}$

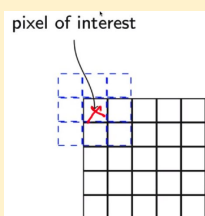
b = cols

- b. Let a be the number of rows and b be the number of columns
- c. m & n specify the size of the matrix, in this case we consider them to be 2 each. So it's a 2x2 matrix. Therefore a & b range from 0-1 each.
- d. Now, to calculate the new value at a particular pixel  $I_{ij}$ , we simply need to fill in the values into the formula.
- e.  $s_{ij} = I_{i+0, j+0}K_{0,0} + I_{i+0, j+1}K_{0,1} + I_{i+1, j+0}K_{1,0} + I_{i+1, j+1}K_{1,1}$
- f. Here is a pictorial representation

a	b	c	d	Convolution ->	aw+bx+ey+fz	bw+cx+fy+gz	cw+dx+gy+hz
e	f	g	h		ew+fx+iy+jz	fw+gx+jy+kz	gw+hx+ky+lz
i	j	k	l				
Input				Kernel	Output		

- g. This is how the convolutional operation looks like in 2D
- h. Instead of only choosing successive points, we must also consider previous points, on both

sides of the reference pixel.  $s_{ij} = (I * K)_{ij} = \sum_{a=-\frac{m}{2}}^{\frac{m}{2}} \sum_{b=-\frac{n}{2}}^{\frac{n}{2}} I_{i-a, j-b} * K_{\frac{m}{2}+a, \frac{n}{2}+b}$





# PadhAI: The Convolution Operation

## One Fourth Labs



### Examples of 2D convolution

How is the convolution operation used in practice?



1. Let us consider a 3x3 kernel and run it over an image, pixel-by-pixel.
2. This is done to re-estimate every pixel in that 3x3 neighborhood

Input 30 x 30	conv	Kernel 3x3	Output 30x30 (blur)									
	*	<table><tr><td>1/9</td><td>1/9</td><td>1/9</td></tr><tr><td>1/9</td><td>1/9</td><td>1/9</td></tr><tr><td>1/9</td><td>1/9</td><td>1/9</td></tr></table>	1/9	1/9	1/9	1/9	1/9	1/9	1/9	1/9	1/9	
1/9	1/9	1/9										
1/9	1/9	1/9										
1/9	1/9	1/9										

- a. Here, we can see that the kernel is essentially an average operation, so what it does is it converts the value of every pixel to  $\frac{1}{9}^{th}$  of its original value.
  - b. In any photo editing tool like GIMP or Photoshop, when we select an image blur, we are essentially performing a convolution operation using an average valued kernel.
3. Let's look at another convolution operation

Input 30 x 30	conv	Kernel 3x3	Output 30x30 (sharpens)									
	*	<table><tr><td>0</td><td>-1</td><td>0</td></tr><tr><td>-1</td><td>5</td><td>-1</td></tr><tr><td>0</td><td>-1</td><td>0</td></tr></table>	0	-1	0	-1	5	-1	0	-1	0	
0	-1	0										
-1	5	-1										
0	-1	0										

- a. Here, the selected pixel is magnified by multiplying by 5 and then we subtract the 4 neighbors from it. This results in a sharper image, as it boosts the current pixel, thereby making it appear more prominent when compared to its neighbors.
4. Let's look at one more example

Input 30 x 30	conv	Kernel 3x3	Output 30x30 (Edge detection)									
	*	<table><tr><td>1</td><td>1</td><td>1</td></tr><tr><td>1</td><td>-8</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1	1	-8	1	1	1	1	
1	1	1										
1	-8	1										
1	1	1										

# PadhAI: The Convolution Operation

## One Fourth Labs

---

- a. Here, pixels near others pixels of the same value are reduced to 0, leaving only the edges.

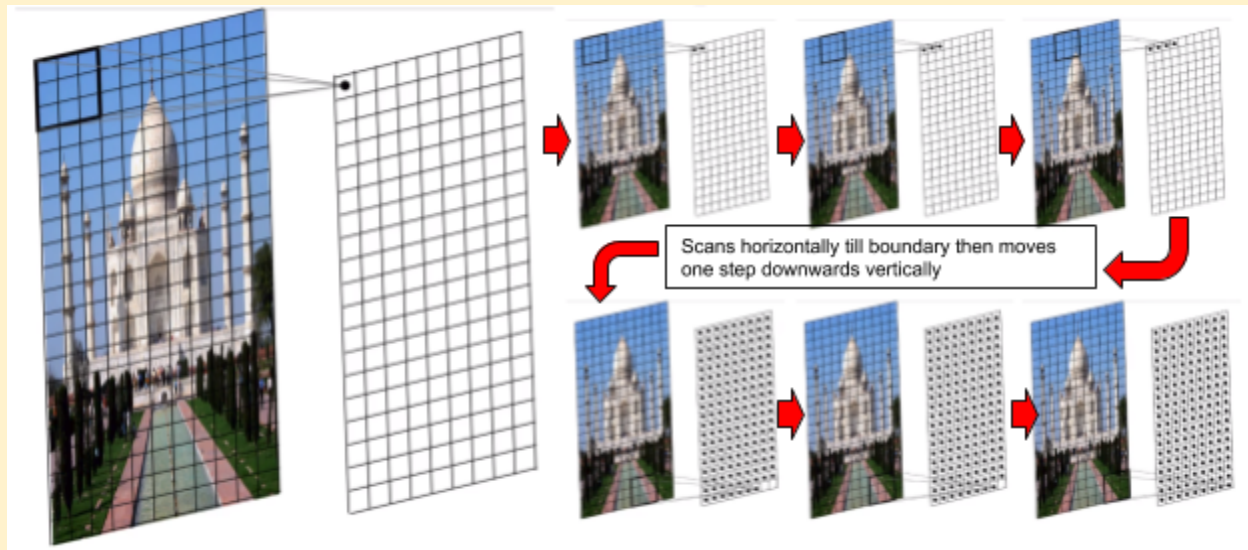
# PadhAI: The Convolution Operation

## One Fourth Labs

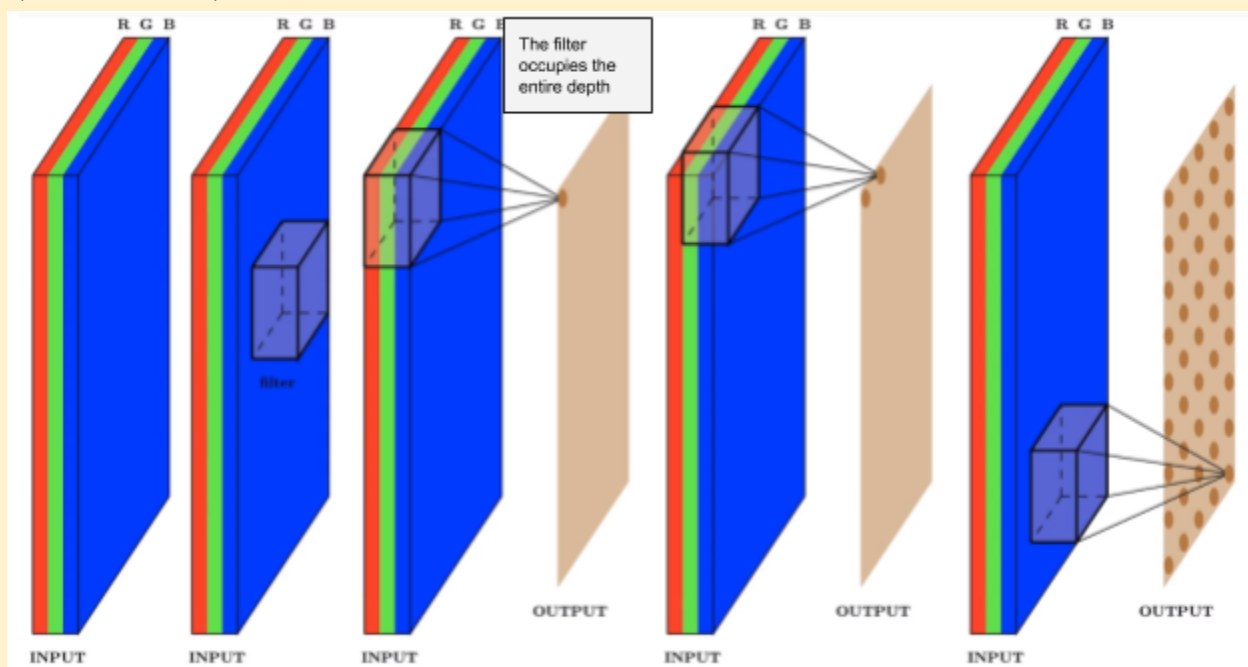
### 2D convolution with a 3D filter

How is this operation performed over the entire image?

1. The following diagram illustrates how the kernel scans through the entire image and applies the transformation



2. How do we do this in the case of a 3D input?
3. The fact is, all the images that we have been considering till now are all 3D inputs, i.e. each pixel is associated with 3 values (Red, Green and Blue). So let's take a look at how the convolution operation takes place in 3D



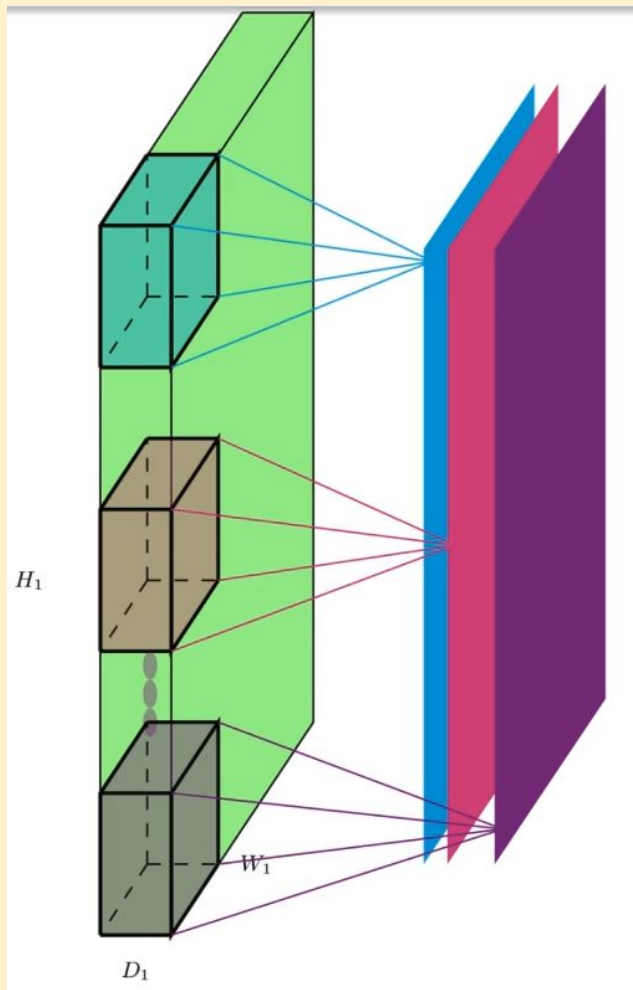
4. Here, since the filter is of the same depth as the image, there is no movement along the depth axis, essentially it moves along the horizontal & vertical axes just as we have seen before.

# PadhAI: The Convolution Operation

## One Fourth Labs

---

5. Some important points about the 3D convolution Operation are:
  - a. The input is 3D
  - b. The filter is also 3D
  - c. The Convolution operation that we perform is 2D
  - d. We only slide vertically & horizontally and not along the depth
  - e. This is because the depth of the filter is the same as the depth of the input.
6. We can also apply multiple filters to the same image



- a. Here, we can see how multiple filters are applied to the input volume(3D) to get an output area(2D).
- b. This is how it is commonly done in practice.
- c. It is called an input volume because it has 3 dimensions (width, height and depth)
- d. The output areas contain 2 dimensions (width and height)
- e. They can be stacked together to get an output volume,
- f. In this case, the depth of the output volume is 3, as we are stacking 3 output areas.



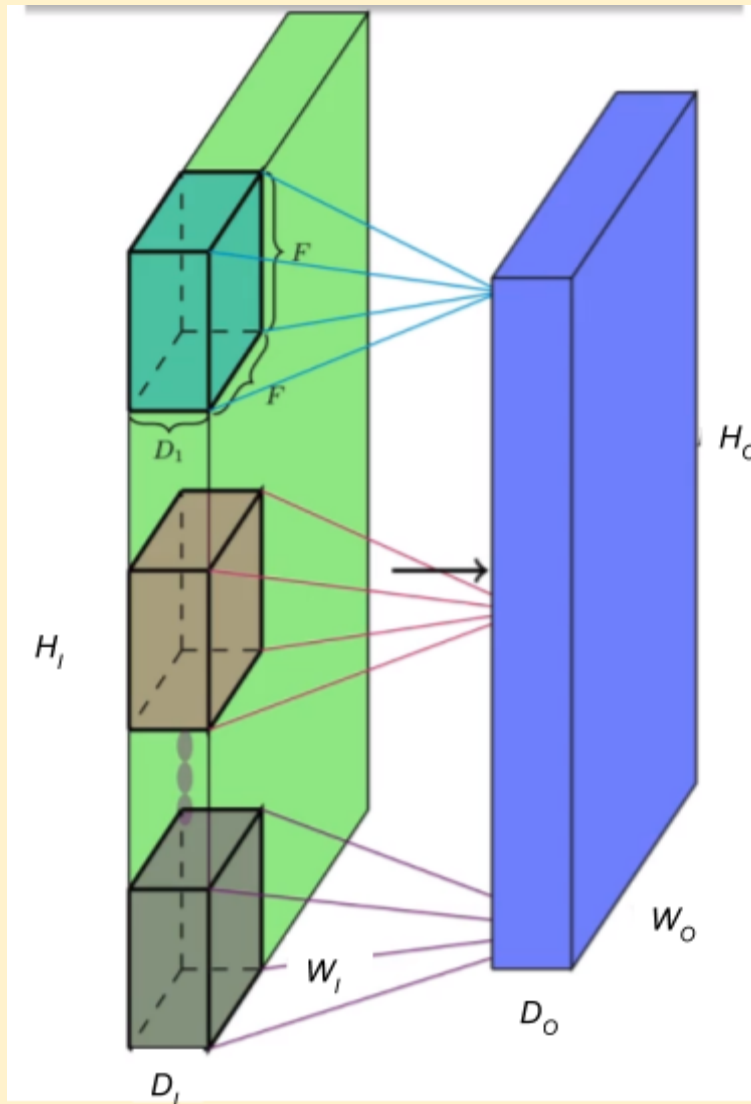
# PadhAI: The Convolution Operation

## One Fourth Labs

### Terminology

Let's look at some terminology

1. Consider the following 3D convolution operation and look at the terminology associated with it



### 2. Terminology:

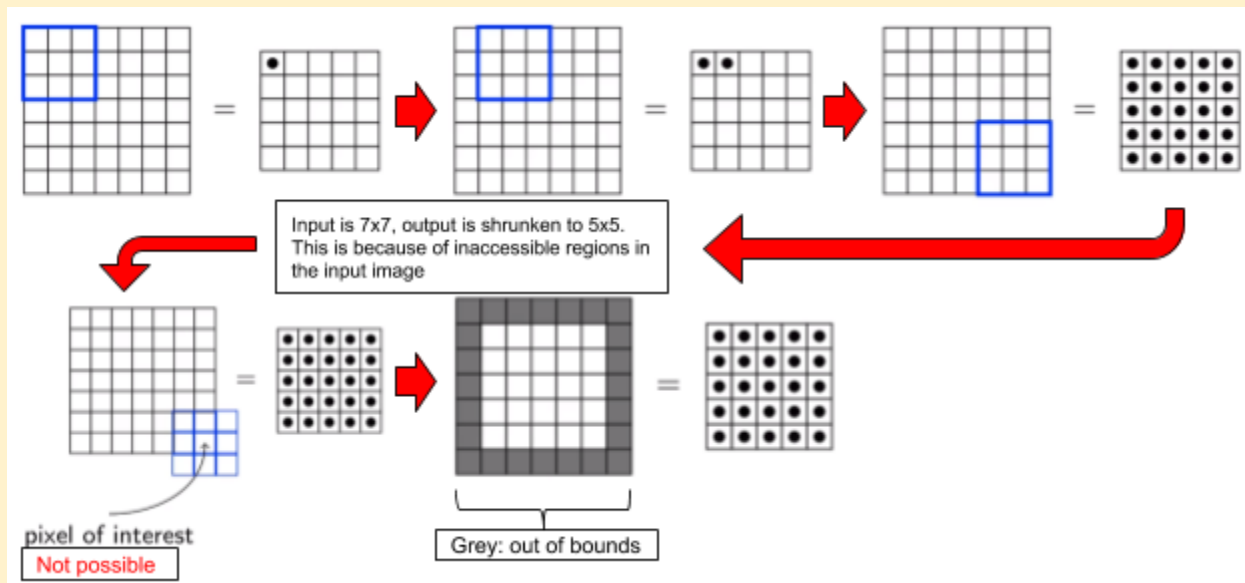
- a. Input Width ( $W_i$ ), Height ( $H_i$ ) and Depth ( $D_i$ )
  - b. Output Width ( $W_o$ ), Height ( $H_o$ ) and Depth ( $D_o$ )
  - c. The spatial extent of a filter ( $F$ ), a single number to denote width and height as they are equal
  - d. Filter depth is always the same as the Input Depth ( $D_i$ )
  - e. The number of filters ( $K$ )
  - f. Padding ( $P$ ) and Stride ( $S$ )
3. **Question:** Given  $W_i$ ,  $H_i$ ,  $D_i$ ,  $F$ ,  $K$ ,  $S$  and  $P$  how do you compute  $W_o$ ,  $H_o$ , and  $D_o$ ?



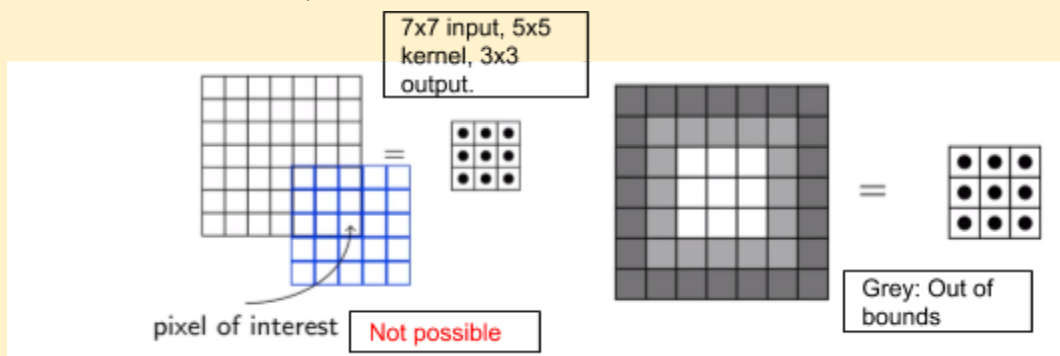
# PadhAI: The Convolution Operation

## One Fourth Labs

4. To answer that, let us look at a sample 3x3 kernel passing over a 7x7 image



- Here, we can see that by running the 3x3 kernel over a 7x7 image, we get a smaller 5x5 image.
  - This is because we can't place the kernel at the corners as it will cross the input boundaries
  - This is true of all the shaded points.
  - Hence the size of the output will be smaller than that of the input
5. Let's see another example with a 5x5 kernel



- Here, we can see that by running a 5x5 kernel over a 7x7 input, we get a smaller 3x3 image
  - Here, the out-of-bounds regions are larger.
  - Thus the output is much smaller.
6. We can see that the reduction in size can be given by the following equations
- $W_O = W_I - F + 1$
  - $H = H_I - F + 1$
7. However in practice, we could still place the kernel on the boundary and take only the valid neighbors. This is roughly what is being done.

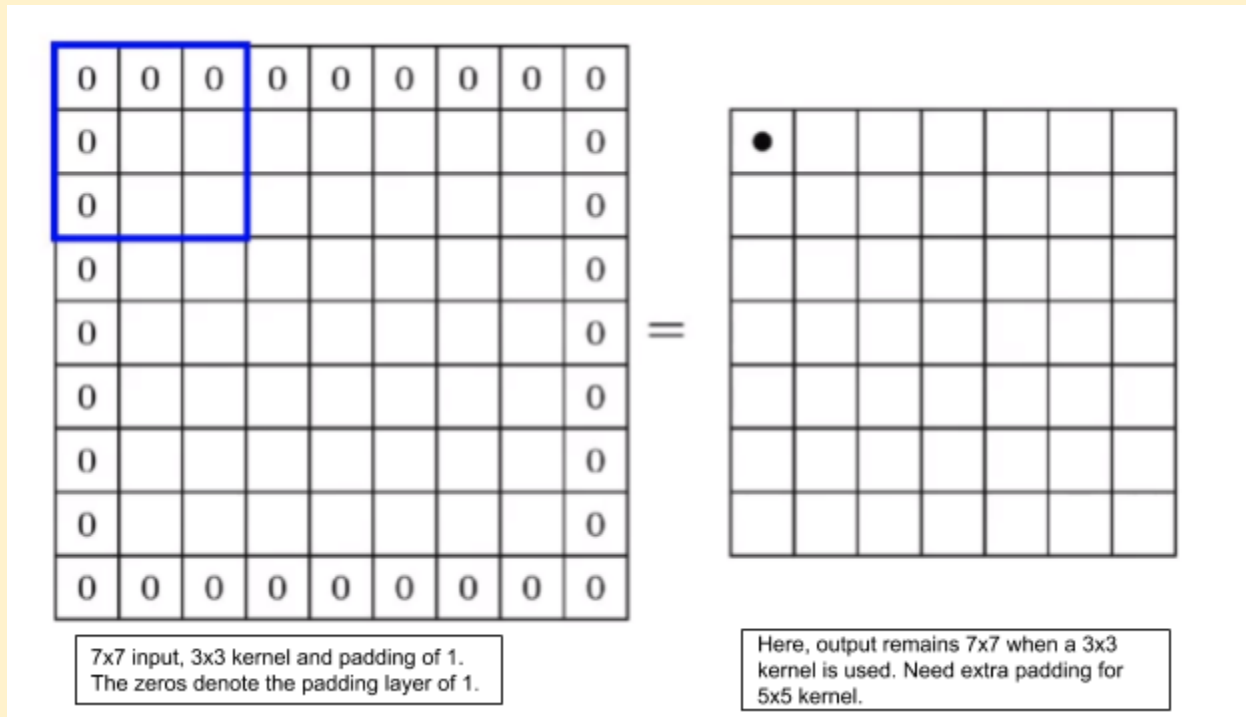
# PadhAI: The Convolution Operation

## One Fourth Labs

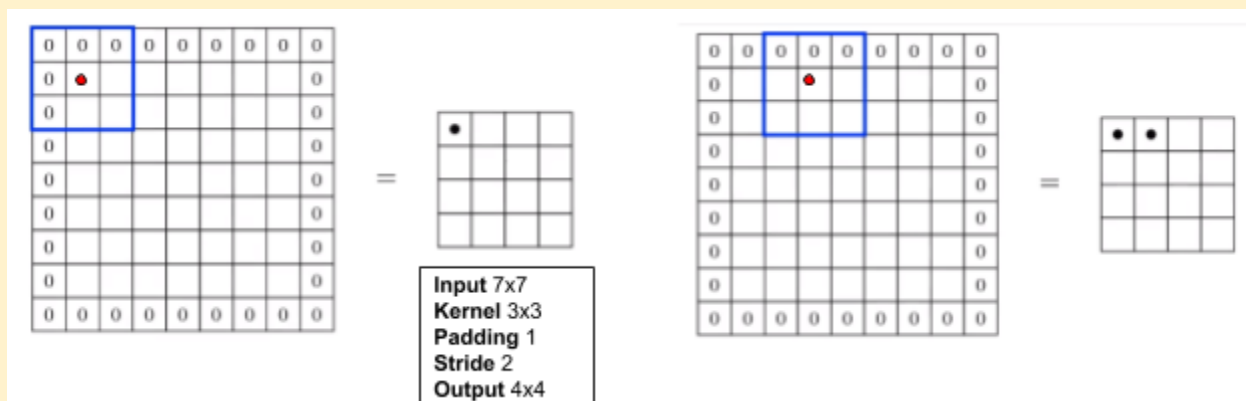
### Padding and Stride

What if we want the output to be the same size as the input?

1. Let us consider adding extra rows+columns of zeros so that we can access all the image pixels



- a. We can see that we must apply padding to preserve the output size
  - b. The bigger the kernel size, the larger the padding required.
2. Thus, the formulae from the last section can be updates as follows
    - a.  $W_O = W_I - F + 2P + 1$
    - b.  $H = H_I - F + 2P + 1$
  3. Another term that we use is called stride (S). It also affects the size of the output image.



- a. Stride defines the interval at which the filter is applied
- b. Higher the stride, the smaller the size of the output

# PadhAI: The Convolution Operation

## One Fourth Labs

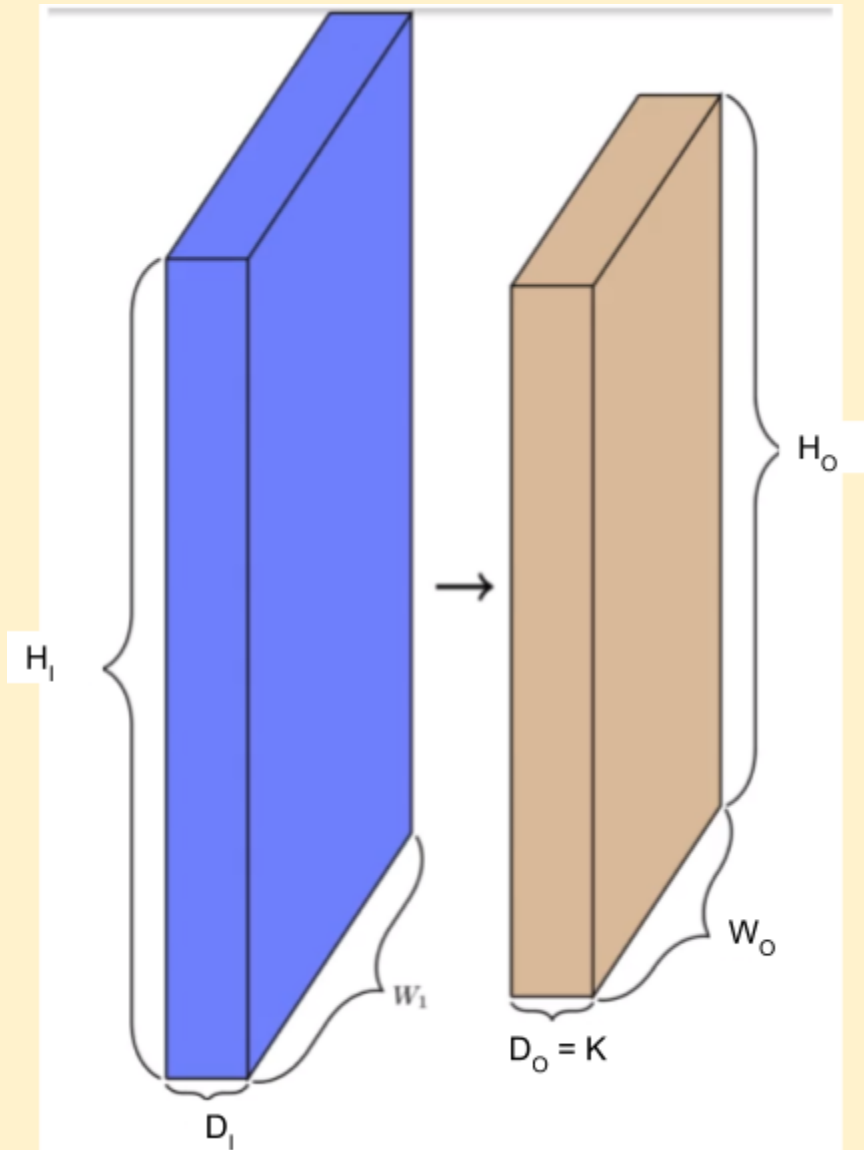
4. We can see that the reduction in size can be given by the following equations

a.  $W_O = \frac{W_I - F + 2P}{S} + 1$

b.  $H_O = \frac{H_I - F + 2P}{S} + 1$

5. How do we compute the depth D of the output?

6. Consider the following image of a convolution operation



7. Each filter gives on 2D output

8. K filters will give K such 2D outputs

9. The depths of the output is the same as the number of filters

10. Thus, our final set of formulae are

a.  $W_O = \frac{W_I - F + 2P}{S} + 1$

b.  $H_O = \frac{H_I - F + 2P}{S} + 1$

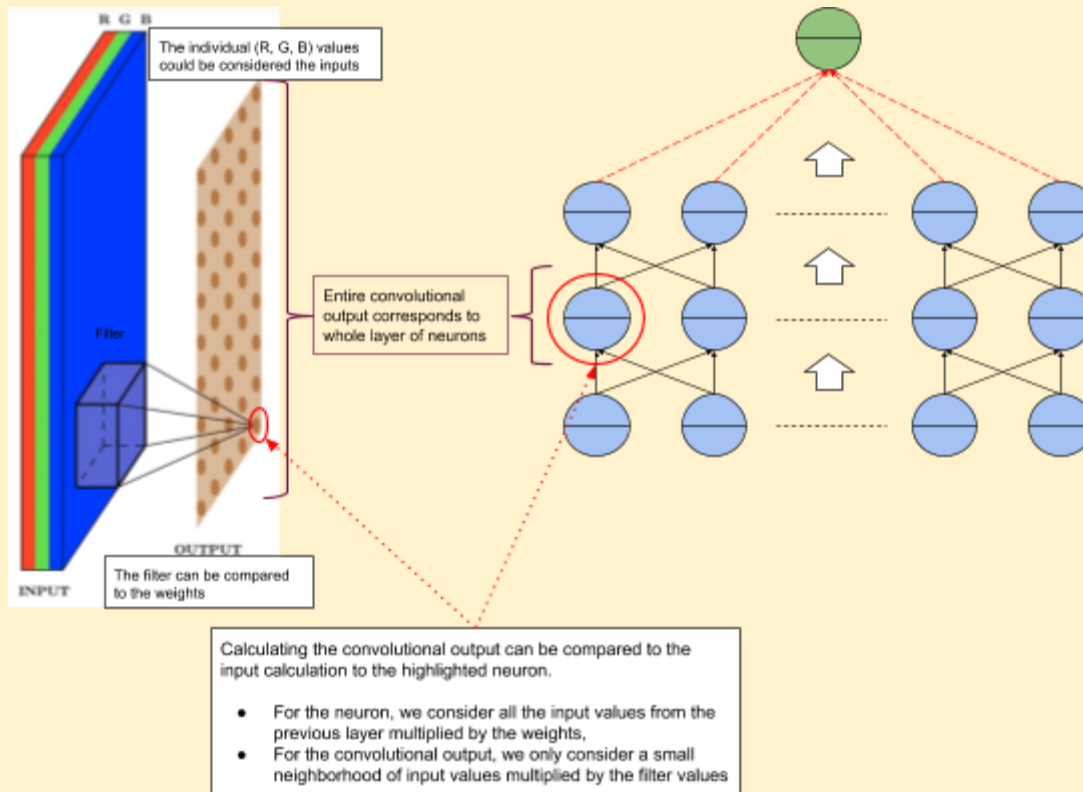
c.  $D_O = K$

### The convolution operation and Neural Networks

#### Part 1

What is the relation between the convolution operation and neural networks?

1. The following diagram illustrates the similarities between the convolutional operation and DNNs



2. As we can see from the diagram, both the highlighted output neuron and the highlighted convolutional output are essentially weighted sums of the inputs provided to them
3. Let's look at a direct comparison

	Neural Network	Convolution Operation on image
<b>Input</b>	Numerical input values.	The RGB values for each pixel in the image
<b>Output</b>	Neuron which takes weighted sum of inputs as its input	Pixel which takes the RGB values transformed with a filter
<b>Neighborhood</b>	All inputs from the previous layer contribute to the output calculation	Only a localised neighborhood of inputs is considered for each output pixel.
	The entire convoluted output image corresponds to a whole layer of neurons. With <b>multiple filters</b> , <b>multiple convoluted outputs</b> each <b>correspond to separate layers of neurons</b>	

### The convolution operation and neural networks

#### Part 2

How did we arrive here?

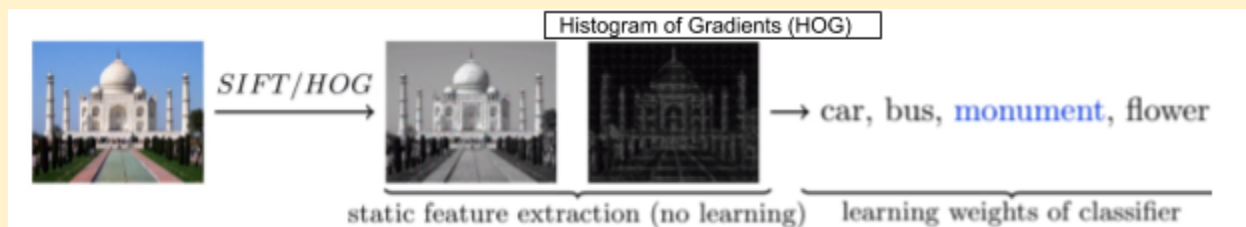
1. Let's look at the image classification task as it would've been performed with Machine Learning.



- a. Here, we flatten a 30x30x3 image into a 2700 raw pixels and feed them as input to a classifier such as a Support Vector Machine or Naive Bayes etc.
  - b. There isn't much intelligence applied on the input side, we just pass the raw pixel data.
2. Now, let's look at the image classification done with some input preprocessing



- a. Here, we realise that there are certain aspects of the image (outlines/edges) that are much more critical to the classification task than other aspects
  - b. So we perform feature engineering, whereby we apply some transformation to the input pixels before passing them into the classifier.
3. Let's look at the use of feature engineering with a 0 Hidden Layer NN to classify the images



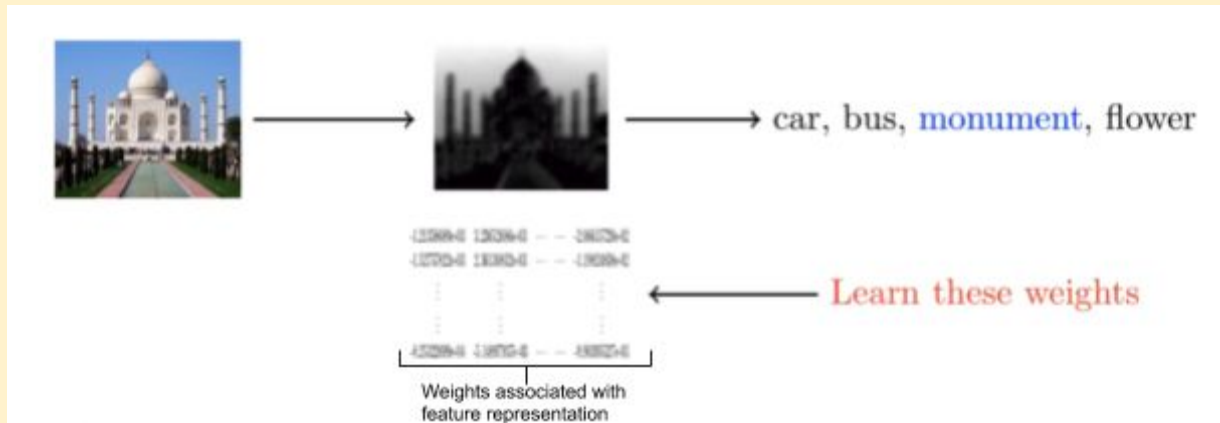
- a. Here, using a deterministic algorithm like HOG or SIFT, we get a better representation of the input by cancelling out useless information.
  - b. We now use these new inputs as features for our Neural Network and learn the weights for the classifier.
  - c. However, in step (a), the transformation performed on the input image was static, without any learning per se, making it a hand-crafted set of features. The only learning that happens is in the classifier.
4. However, in a deep Neural Network, the input features are not directly fed to the classification/output layer, instead they are passed through hidden/representation layers, where they are distilled down to more relevant features, before being passed into the classification layer. This is why Deep Learning is also called Deep Representation Learning.
  5. In the above case, we allow the DNN to learn the representation weights and apply it to the features in steps, before finally passing it onto the output layer.

### The convolution operation and neural networks

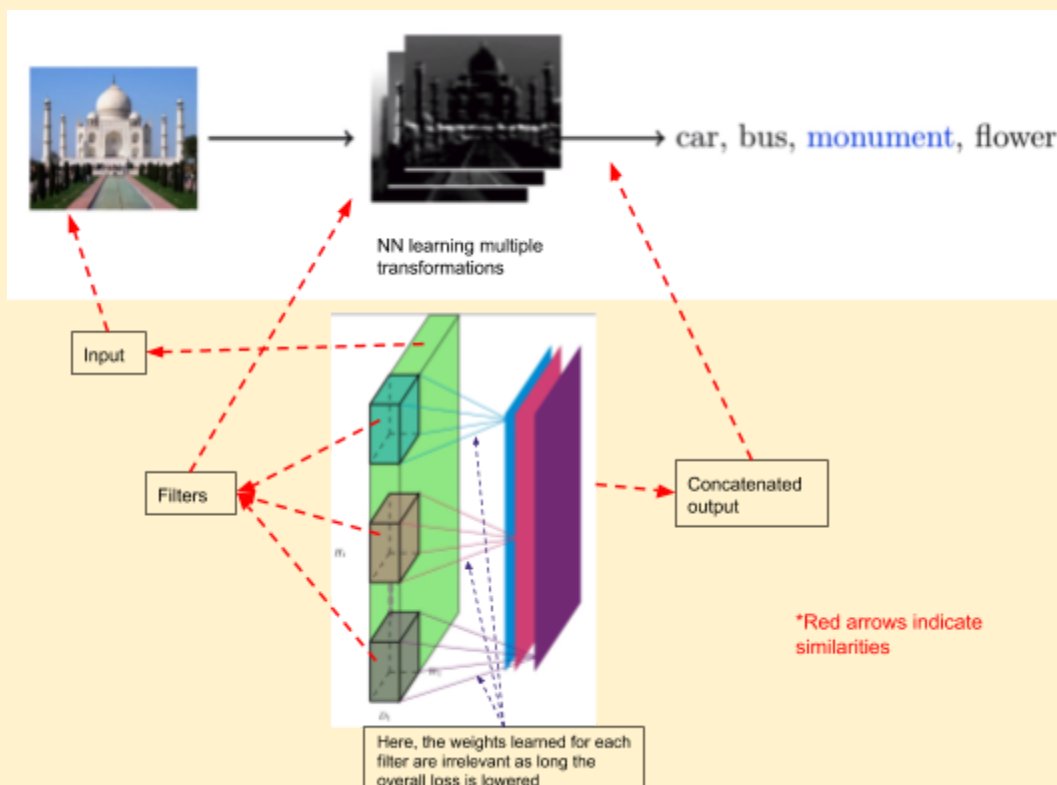
#### Part 3

Why not let the network learn the feature representation also?

1. So far, we have intuitively determined what kind of transformation to apply to the input image before passing it through the classifier. For eg: we saw that performing edge detection yielded higher accuracy.
2. Now, another thing to think about is, why must we leave the choice of transformation to our human intuition? Why can't we let the DNN learn for itself the best transformation to apply?



3. **Why not let the network learn the multiple feature representation?**
4. Another point to consider is that we might not necessarily benefit from only one transformation, so letting the DNN learn the number of transformations to perform is also very useful.



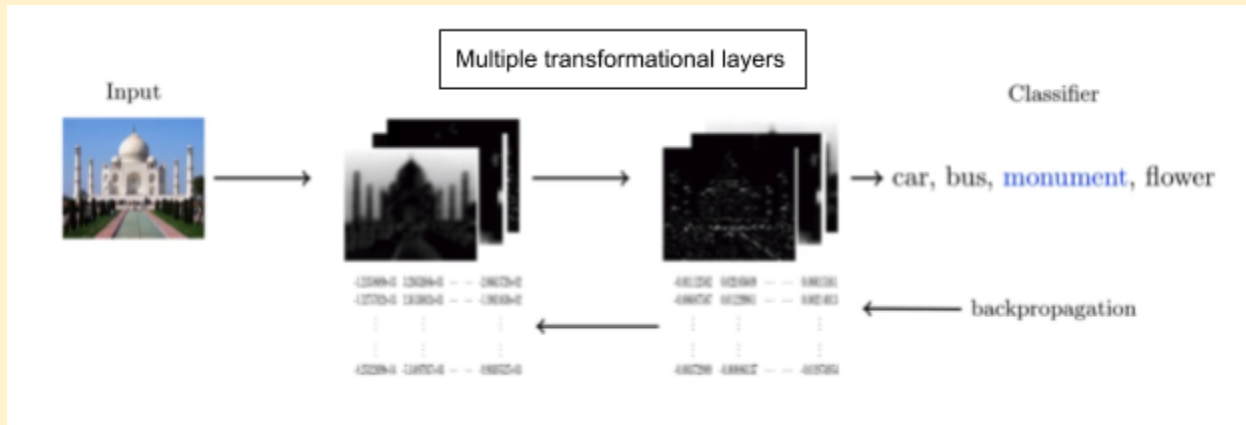
- a. Here, we are not concerned with the type of transformation that occurs, so long as the overall loss/error is reduced. In the above image, we cannot say that the transformations are anything

# PadhAI: From Convolution Operation to Neural Network

## One Fourth Labs

definitive like edge-detection or blurring but still choose to accept them because they lower the loss.

5. **Why not let the network learn the multiple layers of feature representation?**
6. It stands to reason that we can consider several transformational layers like the one seen in the previous example.



- a. The above process is quite similar to what we have learned in the DNN. There are 3 sets of weights, one for each of the two representational layers and one for the classification layer.
  - b. These weights are calculated using backpropagation
  - c. The only difference between a DNN and what we're looking at now (CNN) is that for a CNN we only consider a small localised neighborhood of inputs when calculating the output, instead of the entire input layer as in the case of DNNs.
7. In a nutshell, for CNNs, instead of learning the final classifier weights directly, we should also learn to transform the input into a suitable representation through multiple layers of representations and learn the kernel weights for all of those representations instead of using hand-crafted kernels.

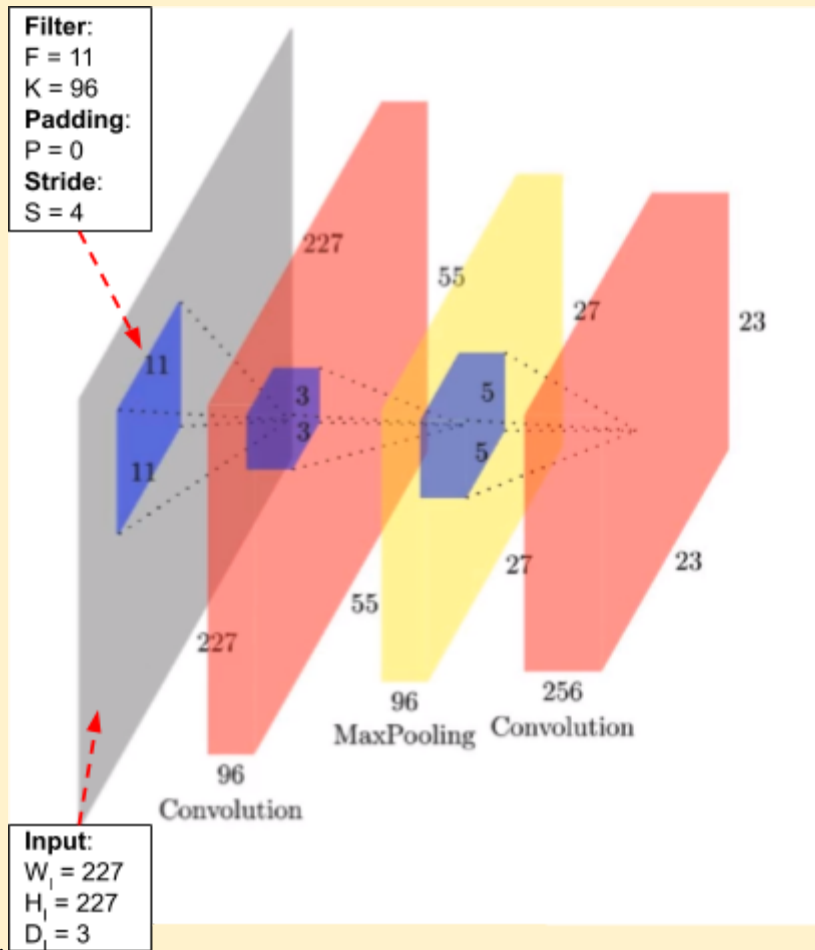


## One Fourth Labs

### Understanding the input/output dimensions

Let's look at the input and output dimensions for a Convolutional Operation

1. As we have seen before, a CNN can be compared to a normal Neural Network, the difference being that CNNs take the RGB pixel values as inputs and output calculation is done with a localised neighborhood of inputs.
2. Consider the following diagram of a CNN. Let us dissect the first convolutional operation in

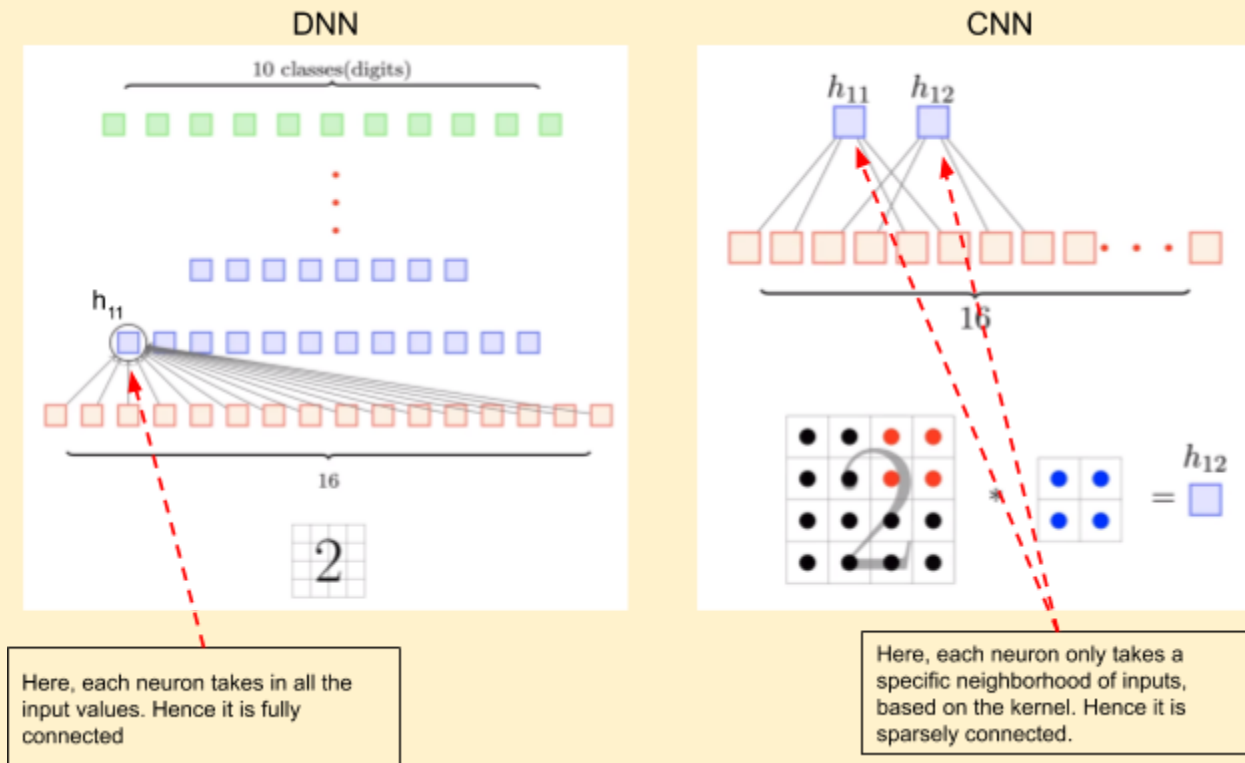


- a. From the above diagram, we are analysing the convolutional operation on the grey input layer.
- b. The input dimensions are as follows
  - i. W<sub>1</sub> = 227
  - ii. H<sub>1</sub> = 227
  - iii. D<sub>1</sub> = 3
- c. The filter is of scale F = 11, i.e 11x11x3, where 3 is the same depth as D<sub>1</sub>
- d. We apply 96 Filter operations, so therefore K = 96
- e. We do not take any padding (P=0) and we choose a stride length of S = 4
- f. Thus, going by the above information, the output volume can be calculated as follows
  - i.  $W_O = \frac{W_I - F + 2P}{S} + 1 = 55$
  - ii.  $H_O = \frac{H_I - F + 2P}{S} + 1 = 55$
  - iii.  $D_O = K = 96$
- g. Thus, the output of the convolutional layer has the dimensions 55x55x96

### Sparse connectivity and Weight Sharing

Let's look at the differences between Fully-connected DNNs and CNNs

1. We already have an intuition of how CNNs are different from DNNs, the key point being that in CNNs we take a weighted aggregation of a neighborhood of inputs instead of all the inputs like in DNNs. Also, it is better to visualise CNNs in terms of matrices than by flat vectors as in the case of FFNs
2. More formally, two of the main aspects of CNNs are **Sparse Connectivity & Weight Sharing**.
3. Let us look at the **connectivity difference** between DNNs and CNNs

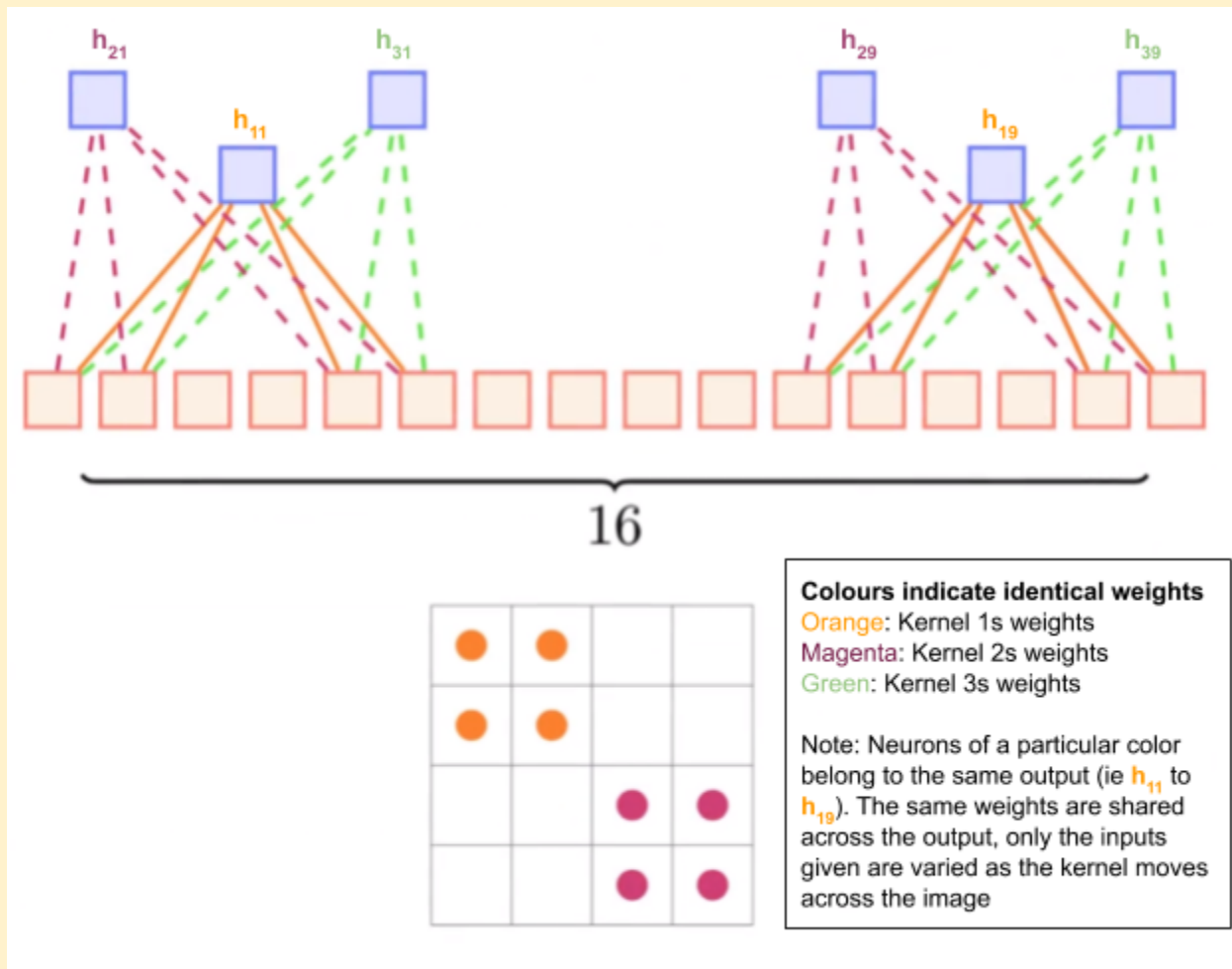


- a. The above diagram is meant to illustrate the difference between DNNs and CNNs with regards to the input-output relationship
- b. In CNNs, as the kernel moves over the input image in strides specified by the stride length  $S$ , it passes over the highlighted regions as shown in the figure. The highlighted regions vary in size with the Kernel size  $F$ , and they correspond with the input regions that are used to calculate the output for that particular neuron.
- c. For the CNN, neuron  $h_{11}$  corresponds to inputs 1, 2, 5 & 6, while neuron  $h_{12}$  corresponds to inputs 3, 4, 7 & 8.
- d. In contrast, in the DNN, neuron  $h_{11}$  and every other neuron corresponds to all inputs 1 to 16.
- e. This is why DNNs are called fully connected and CNNs are said to be sparsely connected

# PadhAI: From Convolution Operation to Neural Network

## One Fourth Labs

4. Now, let us look at the concept of **weight sharing**, as illustrated in the figure.



- Here, we can see how weights are shared across an entire layer.
- In CNNs, **weights are nothing but the kernel**, and the **kernel remains constant as we pass over the entire image**, creating different output values based on the neighborhood of inputs
- One complete pass of the kernel over the image constitutes one Convolutional output.
- As we have seen earlier, it is possible to have multiple convolutional operations by using multiple kernels over the input.
- For each of these convoluted outputs, the kernel is constant, thereby **the weights get shared for all the neurons in that output area**. Only the inputs vary.
- By combining all of these convolutional outputs, we get one convolutional layer
- Instead of treating this Convolutional Layer as a flat vector, we treat it as a volume, whose depth is given by the number of kernels used to process the input.
- By this practice, we are effectively reducing the number of parameters yet still retaining model complexity, thereby overcoming one of the shortcomings of DNNs.

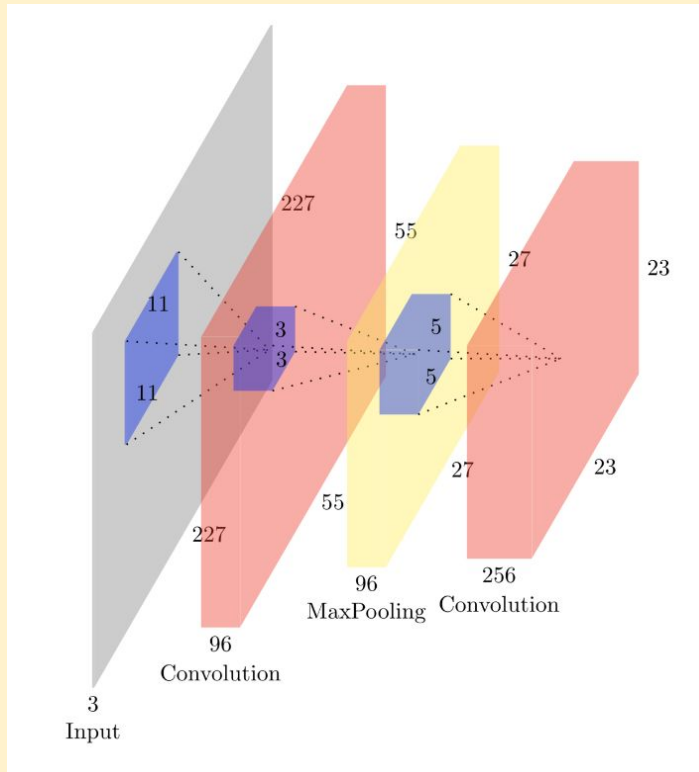
# PadhAI: From Convolution Operation to Neural Network

## One Fourth Labs

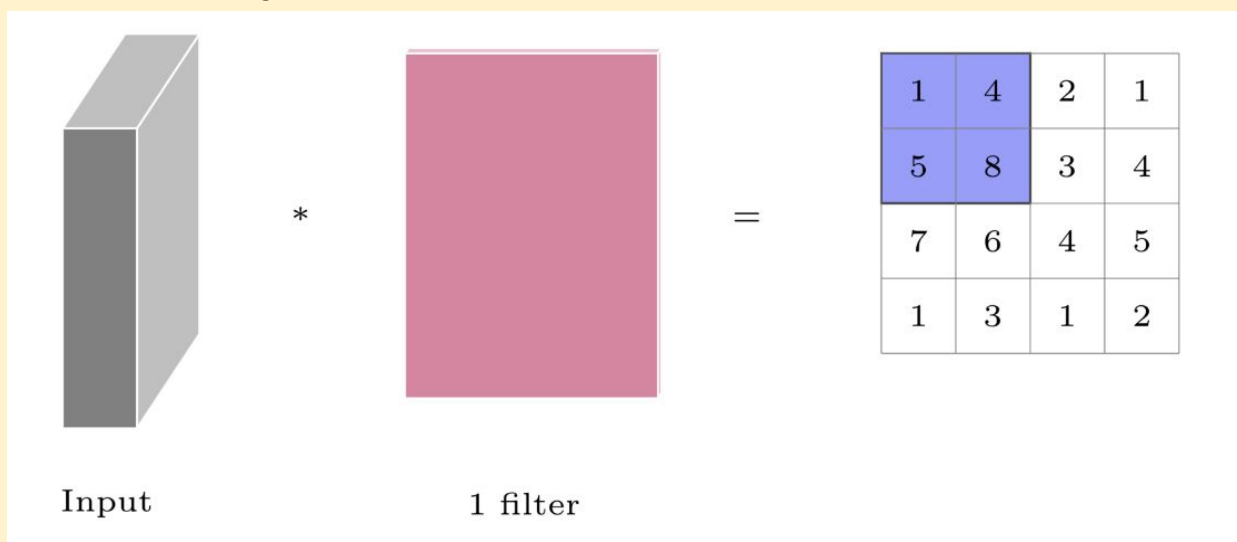
### Max Pooling and Non-Linearities

What is the max pooling operation?

1. Let us look at a diagram of a CNN to better understand what max pooling does.



2. Here, as we have discussed earlier, the first operation performed is a convolutional transformation using a filter.

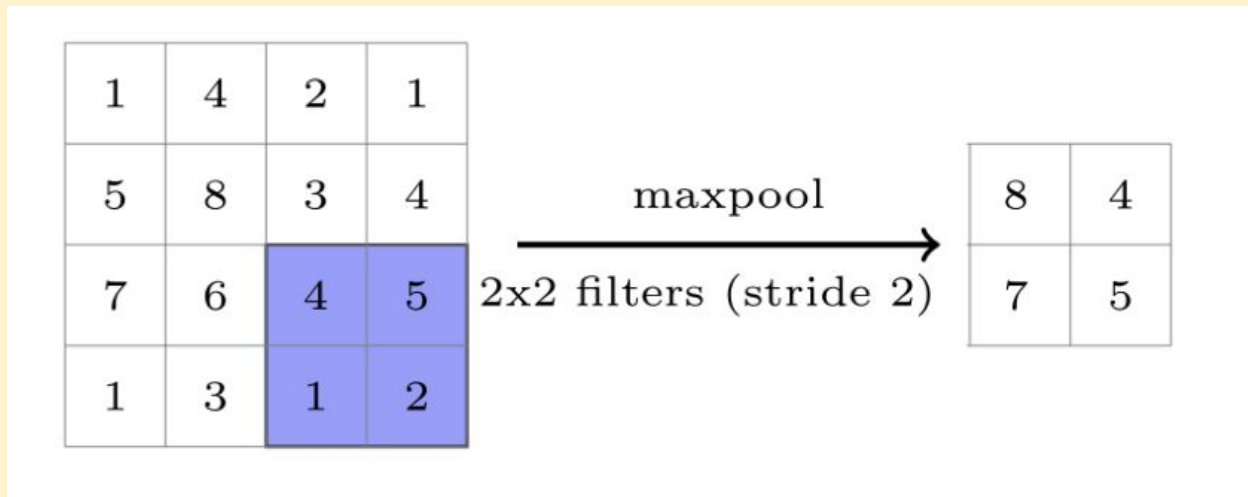


- a. Here, by passing the filter over an image (with or without padding), we get a transformed matrix of values

# PadhAI: From Convolution Operation to Neural Network

## One Fourth Labs

3. Now, we perform max-pooling over the convoluted input to select the max-value from each position of the kernel, as specified by stride length.

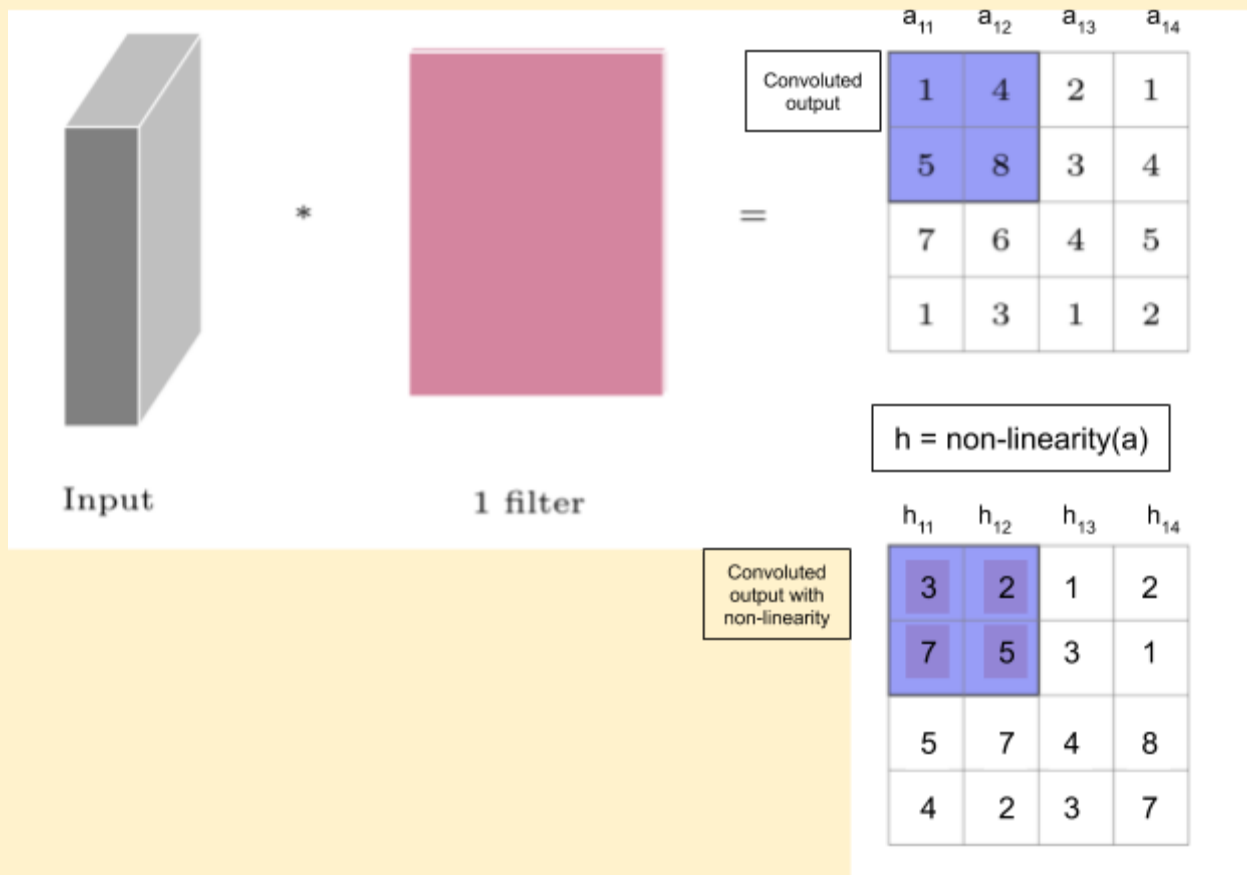


- Here, we select a stride length of 2 and a 2x2 filter, meaning the 4x4 convoluted output is split into 4 quadrants.
- The max value of each of these quadrants is taken and a 2x2 matrix is generated.
- Max pooling is done to select the most prominent or salient point within a neighborhood. It is also known as subsampling, as we are sampling just a single value from a region.
- Similar to Max pooling, average pooling is also done sometimes and it's carried out by taking the average value in a sampled neighborhood.
- The idea behind Max Pooling is to condense the convolutional input into a smaller size, thereby making it easier to manage.

# PadhAI: From Convolution Operation to Neural Network

## One Fourth Labs

4. Another point to consider is the application of nonlinearities to the convoluted output.



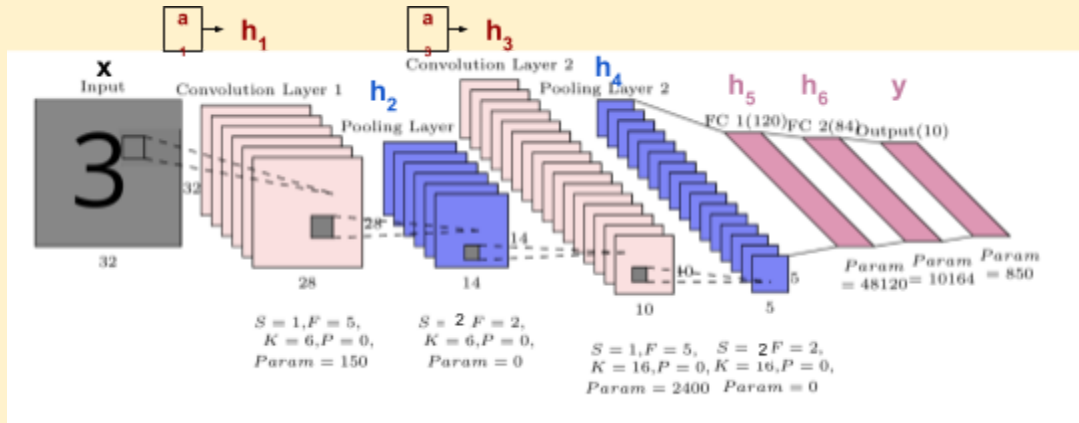
- Here, we can consider the convoluted output to be similar to the pre-activation layer of a neuron.
- By applying a non-linear transformation like sigmoid, tanh, ReLU etc, we are effectively transforming the convoluted output
- The resultant transformed matrix is then passed through the subsequent stages in the CNN, such as Max Pooling etc.
- Thus, we are creating a non-linear relationship between the input and the output, thereby allowing us to approximate more complex functions.
- Yet at the same time, we avoid over-complexity by reducing the number of parameters by weight sharing and condensing the convoluted output using max pooling.

## One Fourth Labs

### Our First Convolutional Neural Network (CNN)

How to use a convolutional neural network for image classification?

1. The following diagram illustrates the configuration and working of a Convolutional Neural Network. It follows the **LeNet** architecture, created by Yann LeCun



2. Let us sequentially break down the various layers in this CNN
3. **Input:**
  - a. The image takes 32x32 pixel inputs.
  - b. There is no depth component because the images are in black & white.
4. **Convolution Layer 1:**
  - a. Here, the filter size  $F = 5$ , and the central cell is the pixel of interest
  - b. Stride length  $S = 1$
  - c. We use a total of 6 filters, i.e.  $K = 6$
  - d. No padding is used, i.e.  $P = 0$
  - e. Each of the filters generate 28x28 output (calculated using  $W_o$ ,  $H_o$  formula).
  - f. Our hidden representation at this layer is  $\mathbf{a}_1 = 28 \times 28 \times 6$  ( $D_o = K$ ).
  - g. Non-linearity like tanh or ReLU(preferred for CNN) is applied to  $\mathbf{a}_1$  making it  $\mathbf{h}_1$
  - h. If we were to proceed as a Fully Connected Network, we would have an extremely large number of parameters ( $32 \times 32 \times 28 \times 28 \times 6 = 4,816,896$  parameters).
  - i. However in this sparsely connected network, each of the 6 filters is of size 5x5x1. So the number of parameters would be much more manageable ( $6 \times 5 \times 5 \times 1 = 150$  parameters).
  - j. This is significantly smaller than in a fully connected network, thereby reducing the chance of overfitting.
  - k. Here, the values  $F$ ,  $S$ ,  $K$ ,  $P$  etc are all counted as hyperparameters.
5. **Max Pooling Layer 1:**
  - a. The hyperparameters are as follows
  - b. Filter size  $F = 2$
  - c. Stride length  $S = 2$
  - d. No. of filters  $K = 6$
  - e. Padding  $P = 0$
  - f. Here, from a 2x2 filter, we select only 1 value. Therefore for a stride of 2, the output dimensions are half of the input( $\mathbf{h}_1$ ) dimensions, i.e 14x14
  - g. We apply the max pooling independently to all 6 of the  $\mathbf{h}_1$  layers, giving us  $\mathbf{h}_2 = 14 \times 14 \times 6$
  - h. No parameters for this layer as we are simply choosing the largest value in the filter and not applying any weights to it.



## One Fourth Labs

---

### 6. Convolutional Layer 2:

- The hyperparameters are as follows
- Filter size  $F = 5$
- Stride length  $S = 1$
- No. of filters  $K = 16$
- Padding  $P = 0$
- Thus, the filter dimensions are  $5 \times 5 \times 6$
- Here, 16 filters are applied to the input  $h_2$ , thereby giving us an output depth of  $D_o = 16$
- Calculating  $W_o$  and  $H_o$  using the formula, we get  $10 \times 10$
- Our hidden representation at this layer is  $\mathbf{a}_3 = 10 \times 10 \times 16$
- Non-linearity like tanh or ReLU (preferred for CNN) is applied to  $\mathbf{a}_3$  making it  $\mathbf{h}_3$
- The number of parameters for the filters ( $16 \times 5 \times 5 \times 6$ ) is 2400 parameters
- This is much smaller than what we would have had in a fully connected network

### 7. Max Pooling Layer 2:

- The hyperparameters are as follows
- Filter size  $F = 2$
- Stride length  $S = 1$
- No. of filters  $K = 16$
- Padding  $P = 0$
- Here, from a  $2 \times 2$  filter, we select only 1 value. Therefore for a stride of 2, the output dimensions are half of the input ( $h_3$ ) dimensions, i.e.  $14 \times 14$
- We apply the max pooling independently to all 16 of the  $h_1$  layers, giving us  $\mathbf{h}_4 = 5 \times 5 \times 16$
- No params for this layer as we are simply choosing the largest value in the filter.

### 8. Fully connected layer 1:

- Number of neurons: 120
- Input is  $\mathbf{h}_4$  flattened, i.e.  $5 \times 5 \times 16 = 400$
- No. of parameters in  $\mathbf{h}_5 = 120 \times 400 + 120$ -bias = 48120 parameters

### 9. Fully connected layer 2:

- Number of neurons: 84
- Input is number of neurons in  $\mathbf{h}_5 = 120$
- No. of parameters in  $\mathbf{h}_6 = 84 \times 120 + 84$ -bias = 10164 parameters

### 10. Output layer:

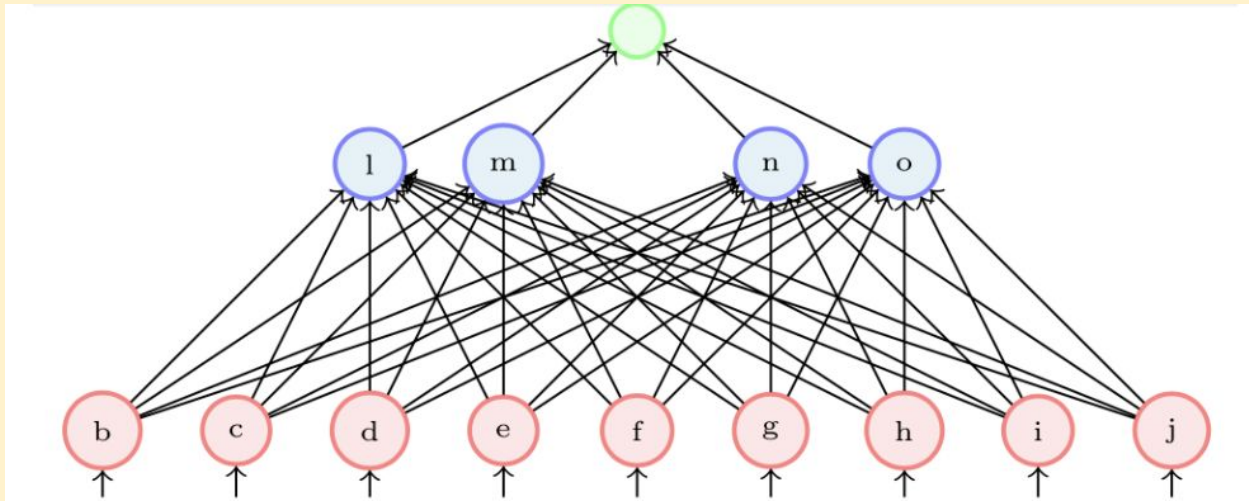
- Number of neurons: 10
- Input is number of neurons in  $\mathbf{h}_6 = 84$
- No. of parameters in  $\mathbf{y} = 10 \times 84 + 10$ -bias = 850 parameters

11. Overall, this combination of Convolutional and fully-connected layers is much more efficient than an entirely fully connected network. It has a significantly lower number of parameters but still is able to estimate functions of very high complexity.

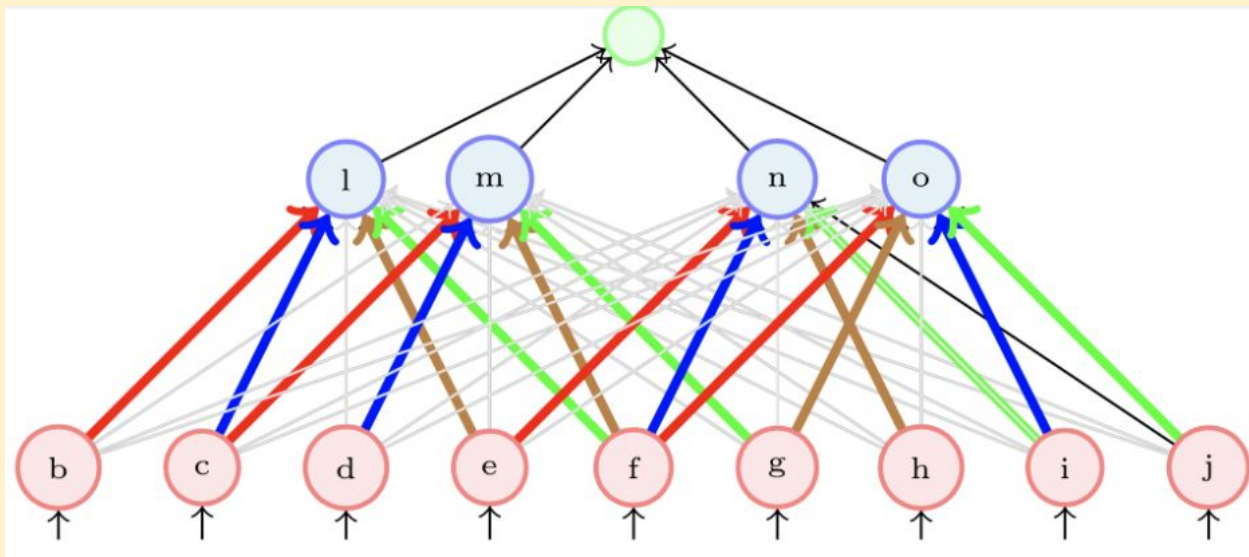
### Training CNNs

How do we train a Convolutional Neural Network?

1. Let us consider a regular Feedforward Network



- a. Here, we obtain  $\hat{y}$  as a prediction, and use it to calculate the loss.
  - b. Using the loss value, we backpropagate to calculate the gradients w.r.t each of the parameters
  - c. Using the gradient descent update rule, we update the weights such that they minimize the loss.
2. Now, we can carry over all of these processes to a CNN, with a few small modifications

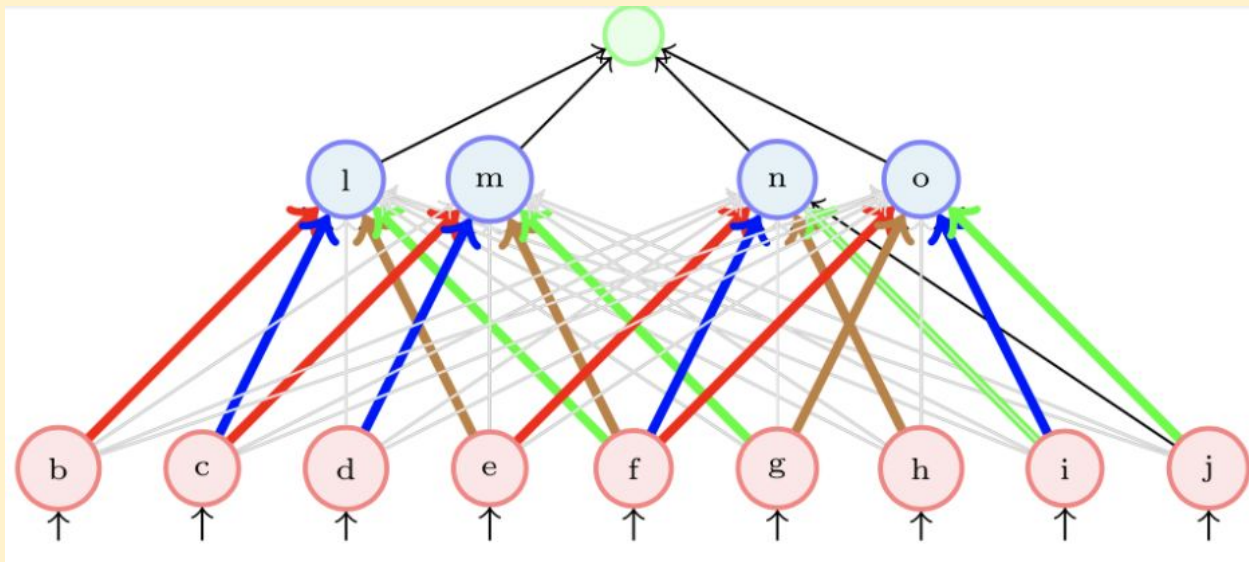


- a. A CNN can be implemented as a feedforward network wherein only a few weights (coloured) are active
  - b. The rest of the weights (grey) remain zero
  - c. Thus, we can train a CNN using backpropagation by thinking of it as a FFN with sparse connections
3. However, in practice we don't do this, as most of the weights in the matrix end up being zero. Frameworks like PyTorch and Tensorflow don't end up creating such large matrices and only focus on dealing with the weights that are to be updated.

### Summary and what next

Making sense of everything we have seen so far

1. By the Universal Approximation Theorem, we have learned that DNNs are powerful function approximators.
2. They can be trained using backpropagation
3. However, due to the large number of parameters, the function can become extremely complex, resulting in a high chance of overfitting the training data.
4. We looked into CNNs to see if we could have a Neural Network which are complex (many non-linearities) but with fewer parameters and hence be less prone to overfitting.



5. CNNs solve both of those shortcomings, with key points such as weight sharing and sparse connectivity. Non-Linearity functions such as ReLU are applied after each convolutional layer.
6. Training CNNs is also very similar to training FFNs, the only difference being we take a 0 value for all weights that we are not interested in.
7. We will then be able to apply learning to the filters(parameters), thereby reducing the overall loss of the CNN.
8. They can be easily implemented with frameworks such as PyTorch or Tensorflow.