These parameters are not learned from the data but are set before the training process begins. They affect various aspects of the learning process and can significantly impact the performance of the model. Some common hyperparameters include:

Learning rate: This is perhaps the most crucial hyperparameter. It determines the size of the step taken during optimization (e.g., gradient descent) to update the model's weights. A higher learning rate can lead to faster convergence but may risk overshooting the optimal solution, while a lower learning rate can lead to slower convergence but more stable training.

Batch size: Batch size refers to the number of samples processed in each iteration of training. A larger batch size can lead to faster training but may require more memory, while a smaller batch size can provide more stable updates but may take longer to converge.

Number of epochs: An epoch is one complete pass through the entire training dataset. The number of epochs determines how many times the model will see the entire dataset during training. Too few epochs may result in underfitting, while too many epochs may lead to overfitting.

Network architecture: This includes the choice of layers (e.g., convolutional, pooling, fully connected), the number of layers, the number of neurons in each layer, and the activation functions used. The architecture of the neural network greatly influences its capacity to learn and its ability to generalize to new data.

Regularization parameters: Regularization techniques such as L1 or L2 regularization, dropout, and batch normalization help prevent overfitting by penalizing large weights or introducing noise during training. The regularization parameters control the strength of regularization applied to the model.

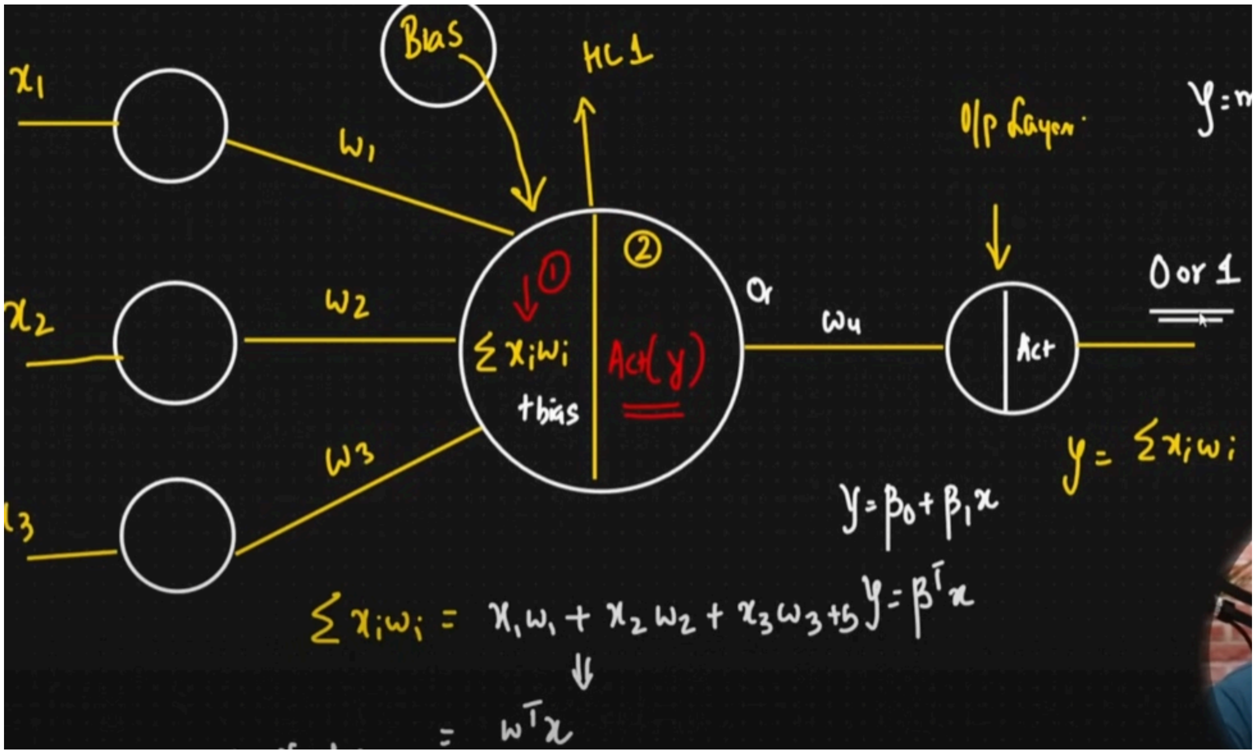
Optimizer parameters: Optimizers such as stochastic gradient descent (SGD), Adam, RMSprop, etc., have their own hyperparameters such as momentum, decay rates, etc., which can affect the optimization process and convergence speed.

Padding and stride: These hyperparameters are specific to convolutional neural networks (CNNs) and affect the spatial dimensions of the feature maps produced by convolutional layers.

Activation functions: The choice of activation functions (e.g., ReLU, sigmoid, tanh) can significantly impact the model's ability to learn complex patterns and gradients during training.

Perceptron -> Single neural network -> Binary classification
The Simplest Neural Network: It's a single layer network that can only do binary classification (think yes/no or spam/not spam). It takes in data points, assigns weights to them, and uses an activation function to make a decision.

Weights - How much neuron should be activated or deactivated?
weights in a neural network are parameters that determine the strength of connections between neurons.



Forward propagation is the process by which input data is processed through a neural network to produce an output or prediction. It involves the following steps:

- Input Layer: The input data is fed into the neural network. Each input feature corresponds to a neuron in the input layer.
- Weighted Sum: For each neuron in the next layer (hidden layers and output layer), the inputs are multiplied by the corresponding weights, and these weighted inputs are summed together.

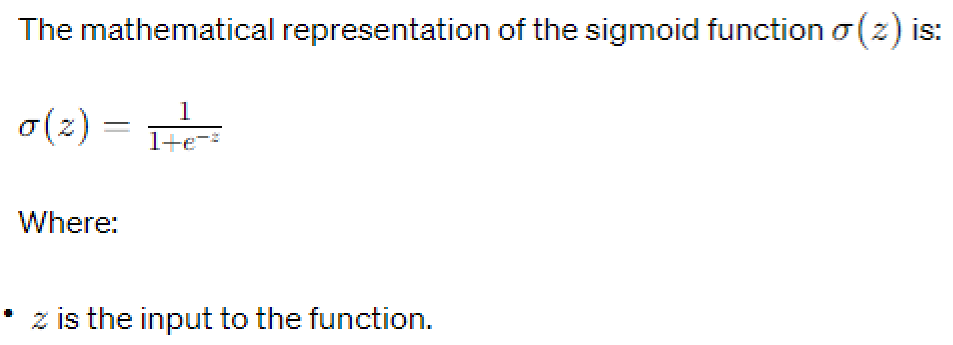
- Activation Function: The weighted sum is then passed through an activation function, which introduces non-linearity into the network. This transformed value becomes the output of the neuron.

- Propagation: The output of each neuron in one layer becomes the input to neurons in the next layer, and this process continues until the output layer is reached.

- Output: Finally, the output layer produces the network's prediction based on the processed input data.

During forward propagation, no learning or weight adjustment occurs. It's simply the process of computing the output of the neural network given a set of input data and the current weights. After forward propagation, the output is compared to the actual target values to compute the loss, which is then used in the subsequent backpropagation step to update the weights of the network.

Sigmoid function- the sigmoid function is a widely used activation function, especially in the context of binary classification problems. It's a smooth, S-shaped function that maps any real-valued number to a value between 0 and 1.



The mathematical representation of the sigmoid function $\sigma(z)$ is:

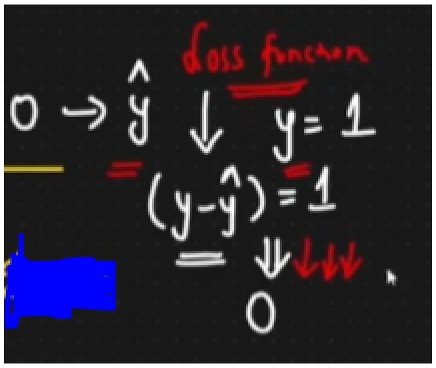
$$\sigma(z) = \frac{1}{1+e^{-z}}$$

Where:

- z is the input to the function.

The sigmoid function is valuable in neural networks because it squashes the input to a range between 0 and 1, making it useful for models where you want to output probabilities. For binary classification problems, it's commonly used as the activation function in the output layer, where values closer to 1 indicate one class and values closer to 0 indicate the other class.

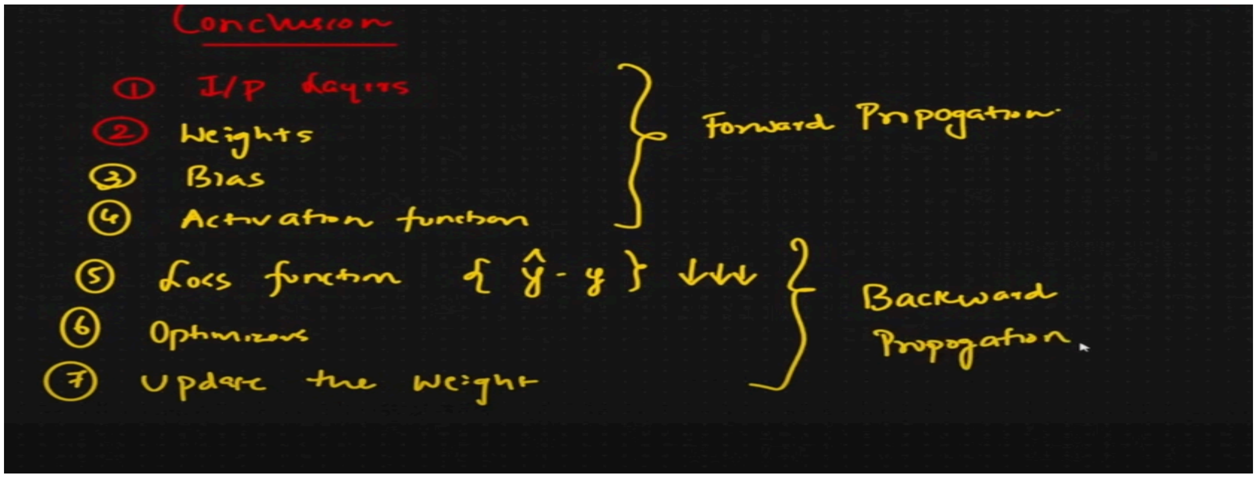
Loss function - Difference between our predicted value and our real value.
Aim - to minimize the difference


Loss function
$0 \rightarrow \hat{y} \downarrow \quad y = 1$
$(y - \hat{y}) = 1$
$\Downarrow \downarrow\downarrow\downarrow$
0

If our difference is too high we have to do back propagation.

Back Propagation - Aim to update the weights because then only predicted value and real value will match.
The key idea behind backpropagation is the chain rule from calculus, which allows the algorithm to efficiently compute the gradients of the loss function with respect to each parameter in the network by recursively applying the chain rule from the output layer to the input layer. Backpropagation enables neural networks to learn from data by adjusting their parameters to minimize the error between predicted and actual outputs.

An **optimizer** is an algorithm that adjusts the parameters (weights and biases) of the model in order to minimize the loss function. The primary goal of an optimizer is to find the optimal set of parameters that result in the best possible performance of the model on the training data.



Conclusion

① I/P Layers
② Weights
③ Bias
④ Activation function

} Forward Propogation

⑤ Loss function $\{ \hat{y} - y \} \downarrow\downarrow\downarrow$
⑥ Optimizers
⑦ Update the weight

} Backward Propogation.

Weight Update formula in backpropagation - The weight update formula in backpropagation relies on the concept of gradient descent. Here's the breakdown:

Formula:

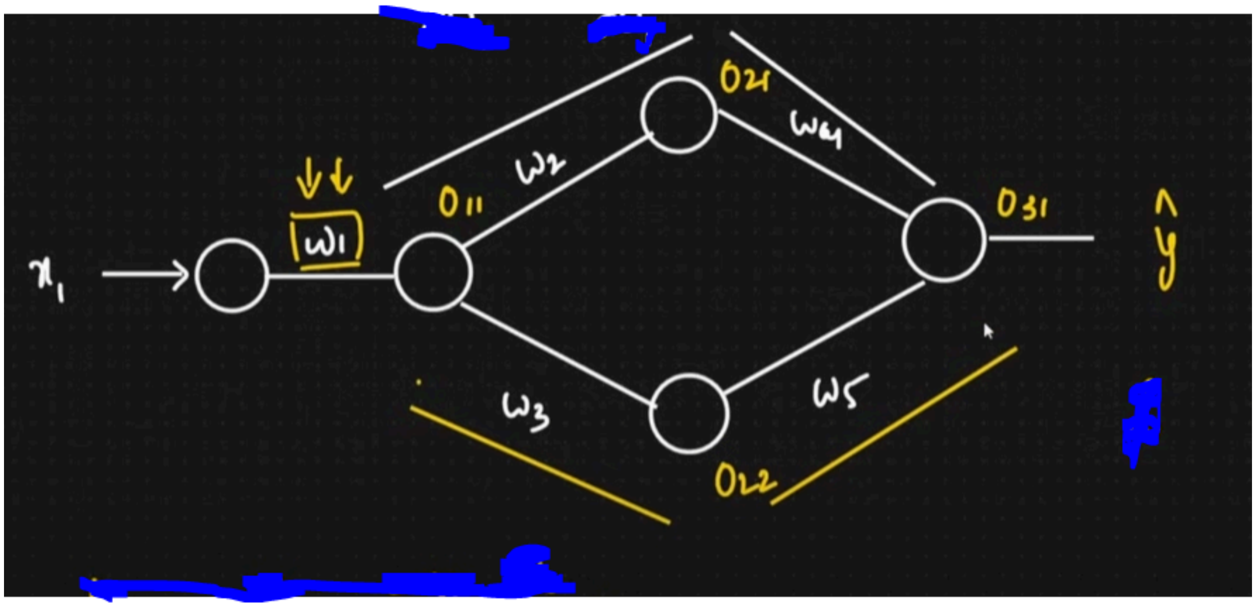
$\Delta w = - \eta * \partial E/\partial w$

Where:

Δw (delta w) represents the update amount for the weight.

η (eta) is the learning rate, a hyperparameter that controls the step size taken during optimization.

$\partial E/\partial w$ represents the partial derivative of the error function (E) with respect to the weight (w). This essentially tells us how much the error changes due to a small change in the weight.



Weight updated formula -

$$w_{1_{new}} = w_{1_{old}} - \eta \frac{\partial L}{\partial w_{1_{old}}}$$

Chain rule of Derivatives -

$$\frac{\partial L}{\partial w_{1old}} = \left[\frac{\partial L}{\partial O_{31}} * \frac{\partial O_{31}}{\partial O_{21}} * \frac{\partial O_{21}}{\partial O_{11}} * \frac{\partial O_{11}}{\partial w_{1old}}\right] + \left[\frac{\partial L}{\partial O_{31}} * \frac{\partial O_{31}}{\partial O_{22}} * \frac{\partial O_{22}}{\partial O_{11}} * \frac{\partial O_{11}}{\partial w_{1old}}\right]$$

∂L -> Derivative of loss

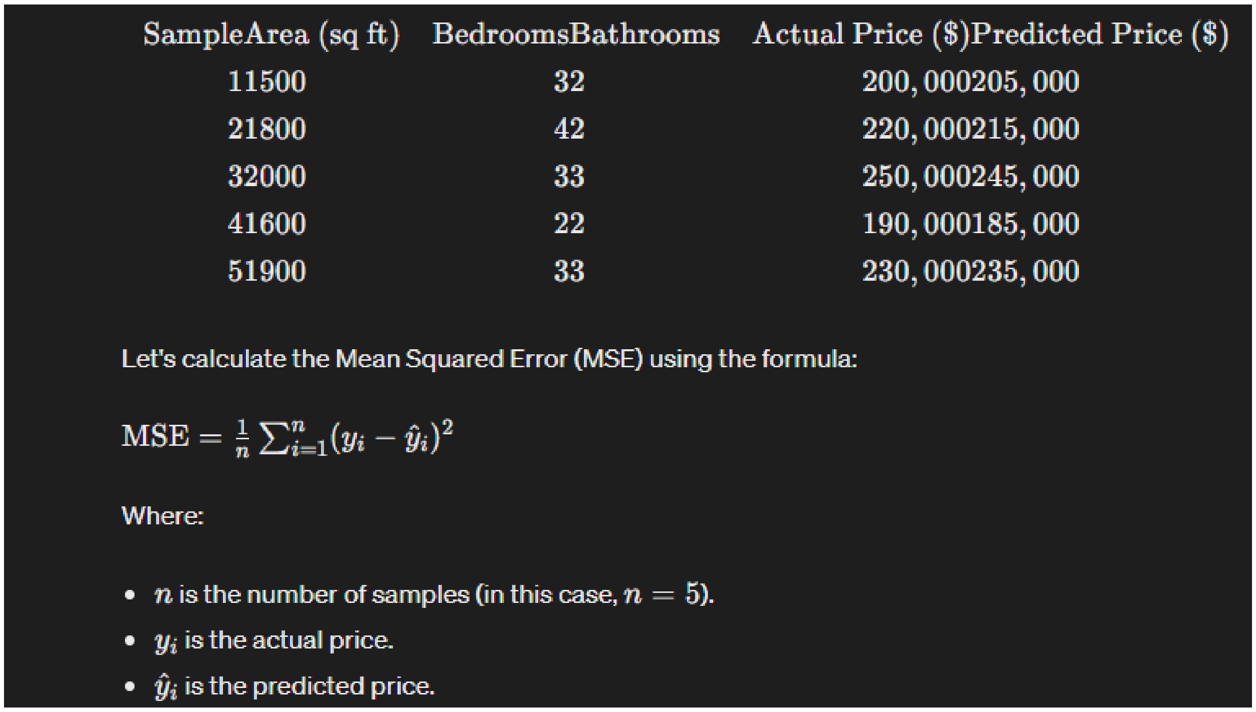
MSE -

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- n is the number of samples in the dataset.
- y_i is the actual target value of the i-th sample.
- $\hat{y}_i$ is the predicted target value of the i-th sample.

MSE is widely used as a loss function in regression problems, where the goal is to minimize the difference between predicted and actual values. In deep learning, it's often used as the loss function for regression tasks, where the model outputs continuous values. By minimizing the MSE during training, the model learns to make predictions that are closer to the actual target values.

Let's say we have a dataset with 5 samples, and we're predicting house prices based on features like area, number of bedrooms, and number of bathrooms. Here's a hypothetical dataset:



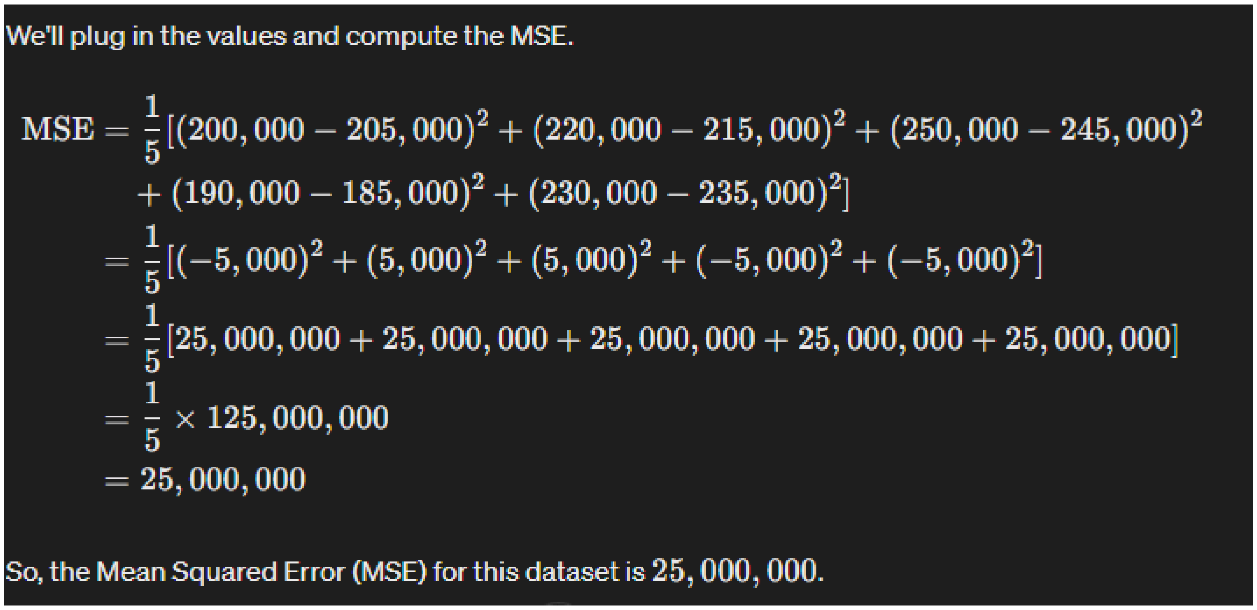
Sample	Area (sq ft)	Bedrooms	Bathrooms	Actual Price ($)	Predicted Price ($)
1	1500	3	2	200,000	205,000
2	1800	4	2	220,000	215,000
3	2000	3	3	250,000	245,000
4	1600	2	2	190,000	185,000
5	1900	3	3	230,000	235,000

Let's calculate the Mean Squared Error (MSE) using the formula:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Where:

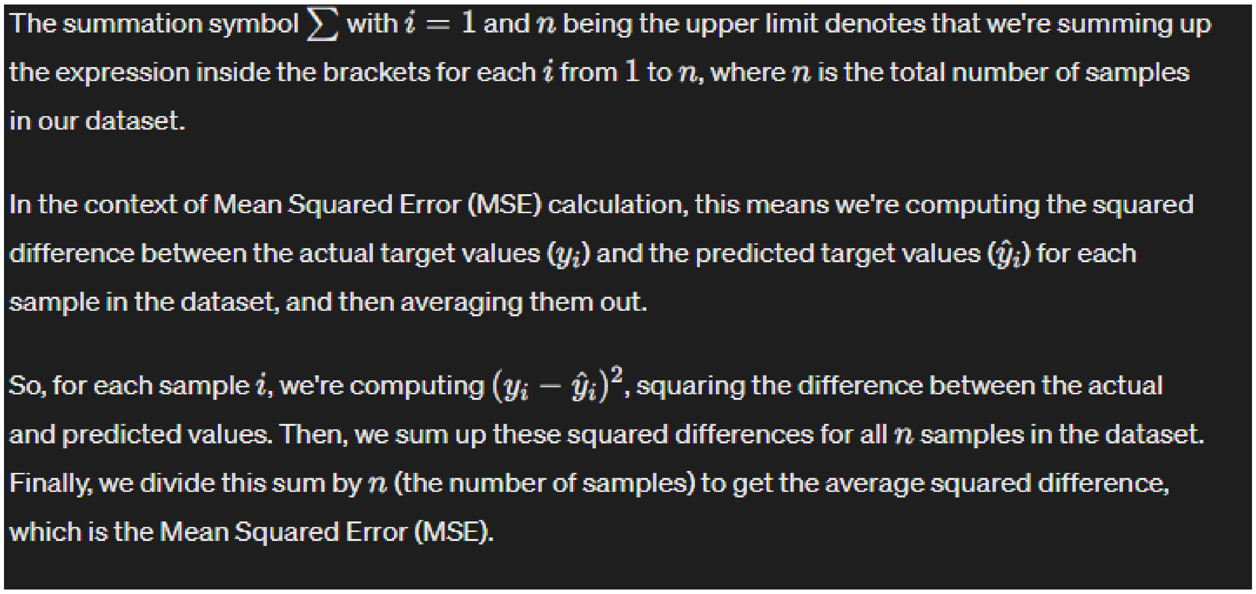
- n is the number of samples (in this case, $n = 5$).
- y_i is the actual price.
- $\hat{y}_i$ is the predicted price.



We'll plug in the values and compute the MSE.

$$\begin{aligned}
\text{MSE} &= \frac{1}{5}[(200,000 - 205,000)^2 + (220,000 - 215,000)^2 + (250,000 - 245,000)^2 \\
&\quad + (190,000 - 185,000)^2 + (230,000 - 235,000)^2] \\
&= \frac{1}{5}[(-5,000)^2 + (5,000)^2 + (5,000)^2 + (-5,000)^2 + (-5,000)^2] \\
&= \frac{1}{5}[25,000,000 + 25,000,000 + 25,000,000 + 25,000,000 + 25,000,000] \\
&= \frac{1}{5} \times 125,000,000 \\
&= 25,000,000
\end{aligned}$$

So, the Mean Squared Error (MSE) for this dataset is $25,000,000$.


The summation symbol $\sum$ with $i = 1$ and n being the upper limit denotes that we're summing up the expression inside the brackets for each i from 1 to n, where n is the total number of samples in our dataset.

In the context of Mean Squared Error (MSE) calculation, this means we're computing the squared difference between the actual target values (y_i) and the predicted target values ($\hat{y}_i$) for each sample in the dataset, and then averaging them out.

So, for each sample i, we're computing $(y_i - \hat{y}_i)^2$, squaring the difference between the actual and predicted values. Then, we sum up these squared differences for all n samples in the dataset. Finally, we divide this sum by n (the number of samples) to get the average squared difference, which is the Mean Squared Error (MSE).

Vanishing Gradient problem - when the gradients of the loss function with respect to the weights of early layers become extremely small as training progresses. Consequently, these small gradients result in negligible updates to the weights of early layers, slowing down or even halting learning in those layers.

In networks with many layers, especially those that use activation functions like the sigmoid or hyperbolic tangent (tanh), the gradients of these activation functions can become extremely small as the inputs move away from the origin. As a result, when these small gradients are multiplied together during backpropagation, they can vanish, becoming so close to zero that they effectively have no impact on the updates to the weights in the earlier layers.

CNN -
Kernel - In Convolutional Neural Networks (CNNs), a kernel, also known as a filter, is a small matrix applied to an input image. The purpose of the kernel is to extract specific features from the input data. During the convolution operation, the kernel slides (convolves) over the input image, performing element-wise multiplication with the overlapping pixels and then summing up the results to produce a single value in the output feature map.

Each kernel is responsible for detecting a particular pattern or feature in the input image. For example, one kernel might be designed to detect edges, while another might be designed to detect textures or corners. CNNs typically consist of multiple layers of kernels, where each layer learns increasingly complex features by combining the information extracted from the previous layers.

Sobel horizontal kernel - (detect horizontal edges) The Sobel horizontal kernel is a 3x3 matrix that is convolved with an image to compute the gradient approximation in the horizontal direction.(The Sobel horizontal kernel is like a little grid, usually 3 squares by 3 squares, that's used to look at an image in a special way. When we put this grid over an image and do some math with it, we can figure out how the brightness changes from one side of the image to the other, but only in a horizontal direction.Imagine you're looking at a picture, and you want to know where things get brighter or darker as you move from left to right. The Sobel horizontal kernel helps us figure that out by giving us a way to measure how quickly things change in brightness along the horizontal lines in the picture. This can be really useful in things like edge detection in images, where we want to find where objects start or end.)

It is designed to highlight vertical edges in the image. The Sobel horizontal kernel typically looks like this:

$$\begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix}$$

During the convolution operation, this kernel slides over the input image, and at each position, it calculates the weighted sum of pixel values. Specifically, it multiplies each pixel in the neighborhood of the current position by the corresponding value in the kernel matrix, and then sums up the results. This process effectively detects changes in intensity along the horizontal direction.

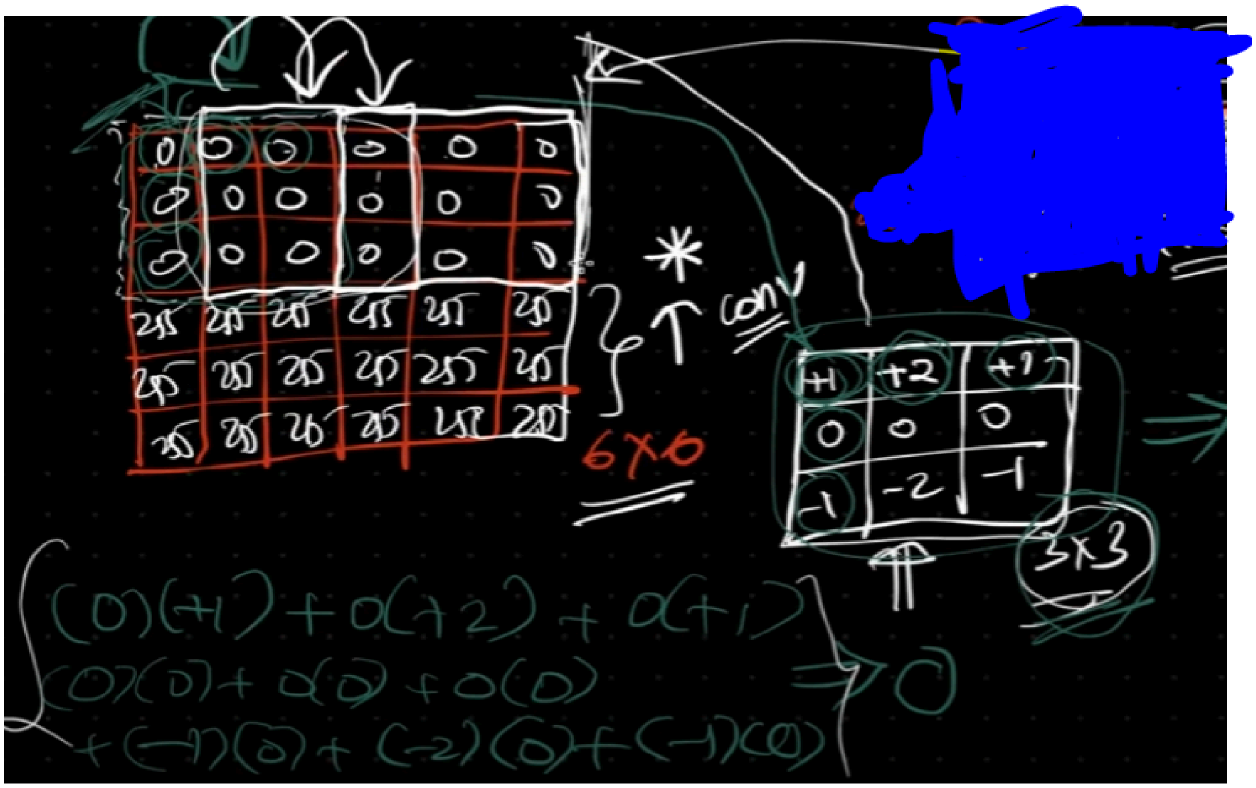
After applying the Sobel horizontal kernel to an image, the resulting output highlights areas where there are significant changes in intensity along the horizontal axis, which often corresponds to edges that run vertically in the image. This information can be further processed for tasks such as edge detection, image segmentation, or feature extraction.

Vertical edge detector - In convolutional neural networks (CNNs), a vertical edge detector is a type of filter or kernel used in the convolutional layers to detect vertical edges in images. These filters are designed to highlight areas of significant contrast along the vertical direction in the input image.

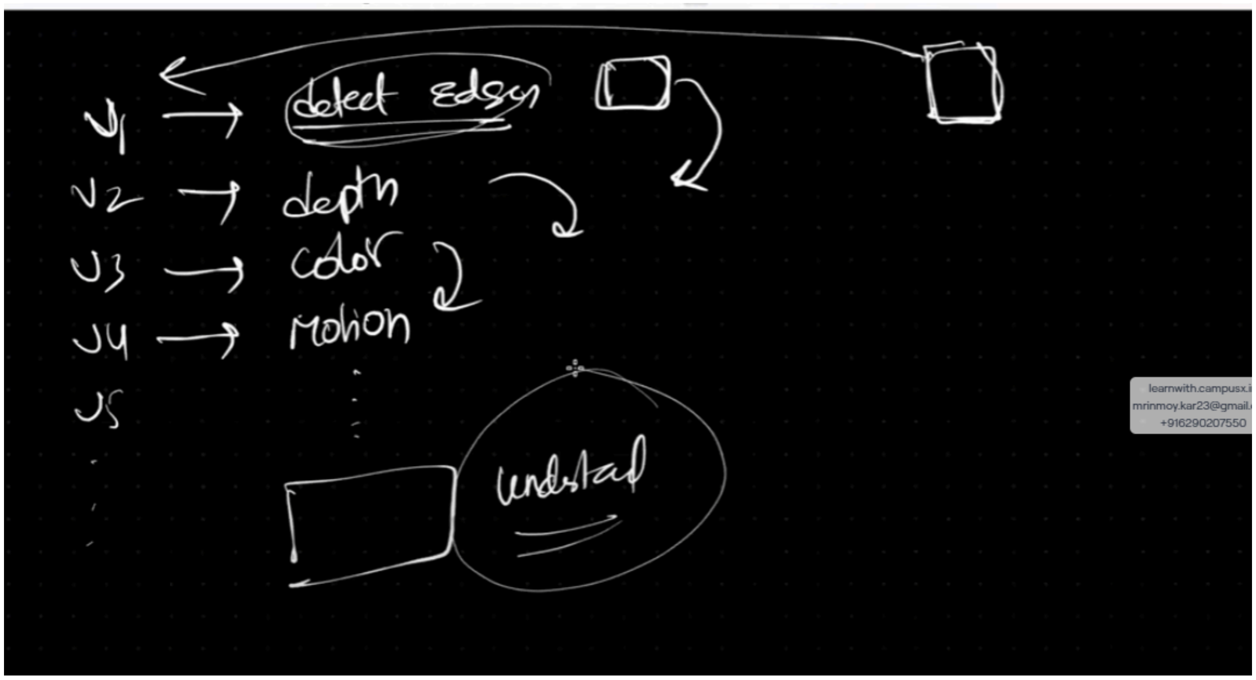
The vertical edge detector filter typically consists of positive values on one side and negative values on the other, with a central region of zeros. When this filter is convolved with an image, it computes the gradient along the vertical direction, emphasizing the regions where there is a rapid change in intensity, indicating the presence of vertical edges.
[-1 0 1]
[-1 0 1]
[-1 0 1]
When this filter is convolved with an image, it produces high activations (positive or negative) at locations where there are vertical edges.

Convolutional - In a Convolutional Neural Network (CNN), the term "convolutional" refers to the operation of applying a filter (also called a kernel or convolutional kernel) to the input image. The filter typically has smaller dimensions than the input image.

Let's break down the example:

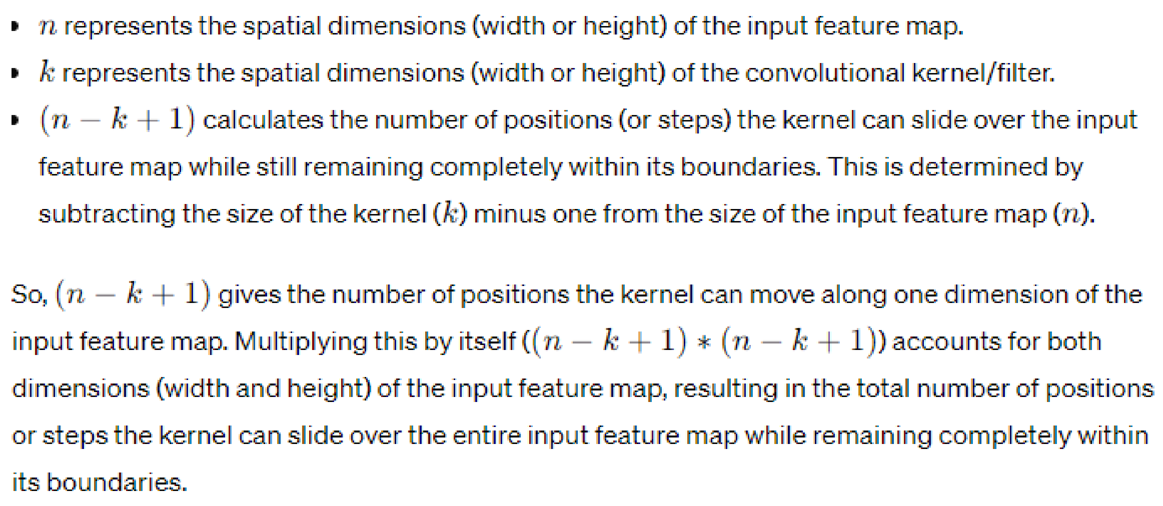

[redacted]

If you have a 6x6 input image (matrix) and you apply a 3x3 filter to it, you're performing convolution.
At each step, you place the 3x3 filter over a 3x3 region of the input image and perform an element-wise multiplication.
Then, you sum up the results of these multiplications to get a single value.
This process is repeated by sliding the filter over the entire input image, one position at a time.
The result of this process is a new matrix, which is typically called a feature map or a convolutional feature map.
So, in summary, convolution in CNNs involves the multiplication and summing up of elements of a filter with corresponding elements of the input image, followed by sliding the filter over the entire image to generate a new feature map. This process is what enables CNN to learn and extract features from the input data.
Final goal to understand the image -



$$(n-K+1) \times (n-K+1)$$

although it assumes no padding by default.



- n represents the spatial dimensions (width or height) of the input feature map.
- k represents the spatial dimensions (width or height) of the convolutional kernel/filter.
- $(n - k + 1)$ calculates the number of positions (or steps) the kernel can slide over the input feature map while still remaining completely within its boundaries. This is determined by subtracting the size of the kernel (k) minus one from the size of the input feature map (n).

So, $(n - k + 1)$ gives the number of positions the kernel can move along one dimension of the input feature map. Multiplying this by itself ($(n - k + 1) * (n - k + 1)$) accounts for both dimensions (width and height) of the input feature map, resulting in the total number of positions or steps the kernel can slide over the entire input feature map while remaining completely within its boundaries.

spatial dimensions typically refer to the width, height, and depth (or channels) of the image.

Valid Padding (No Padding): In valid padding, no extra pixels are added around the edges of the image. As a result, the convolutional operation is only applied to positions where the entire kernel can fit within the boundaries of the input image.

```
[1   2  3  4  5 ]
[6   7  8  9 10 ]
[11 12 13 14 15 ]
[16 17 18 19 20 ]
[21 22 23 24 25 ]
```

Convolutional Kernel (3x3):

```
[ a  b  c ]
[ d  e  f ]
[ g  h  i ]
```

Output Feature Map (3x3) after applying convolution (Valid Padding):

```
[ Result1 Result2 Result3 ]
[ Result4 Result5 Result6 ]
[ Result7 Result8 Result9 ]
```

In this case, the output feature map has dimensions 3x3 because the kernel can fit into the input image without extending beyond its boundaries.

Same Padding: In the same padding, extra pixels are added around the edges of the input image to ensure that the spatial dimensions of the output feature map remain the same as the input.

Input Image with Same Padding (7x7):

```
[ 0  0   0   0   0  0    0 ]
[ 0  1   2   3   4   5   0 ]
[ 0  6   7   8   9  10  0 ]
[0 11 12 13 14 15  0 ]
[ 0 16 17 18 19 20 0 ]
[ 0 21 22 23 24 25 0 ]
[ 0  0    0   0   0   0  0 ]
```

Convolutional Kernel (3x3):

```
[ a  b  c ]
[ d  e  f ]
[ g  h  i ]
```

Output Feature Map (5x5) after applying convolution (Same Padding):

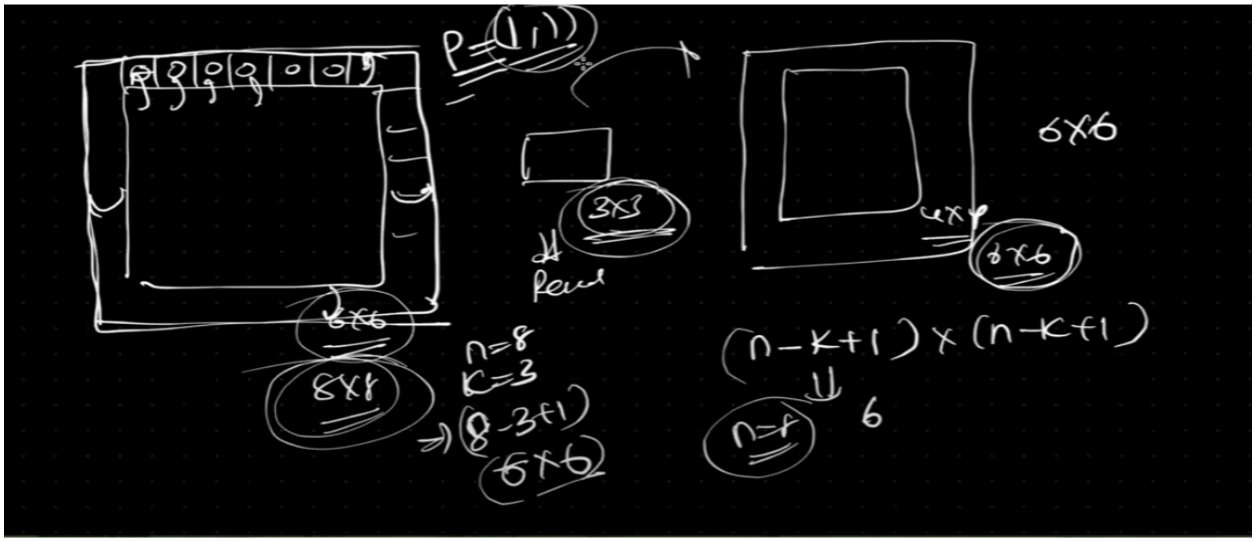
```
[ Result1 Result2 Result3 Result4 Result5 ]
[ Result6 Result7 Result8 Result9 Result10 ]
[ Result11 Result12 Result13 Result14 Result15 ]
[ Result16 Result17 Result18 Result19 Result20 ]
[ Result21 Result22 Result23 Result24 Result25 ]
```

In this case, extra zeros are added symmetrically around the edges of the input image, creating a padded image with dimensions 7x7. This ensures that the kernel can slide over the entire input image while preserving the spatial dimensions of the output feature map.

Padding diagram -> P = (1 , 1) [in padding, you can use values other than zero. This technique is known as "constant padding" or "value padding," where you pad the input image or feature map with a constant value instead of zeros.

The constant value used for padding can be any arbitrary number, typically chosen based on the specific requirements of the task or the characteristics of the input data. For example, you might choose to pad with the nearest pixel value or with a specific intensity value that makes sense for the context of the problem you're working on.]

In this image we use zero.



$$(n-k+\boxed{2P}+1)\times(n-k+2P+1)$$

- n is the spatial dimension (width or height) of the input feature map.
- k is the spatial dimension (width or height) of the convolutional kernel/filter.
- P represents the amount of padding applied to the input feature map.
- The expression $(n-k+2P+1)$ calculates the number of positions or steps the kernel can slide over the padded input feature map while remaining completely within its boundaries.

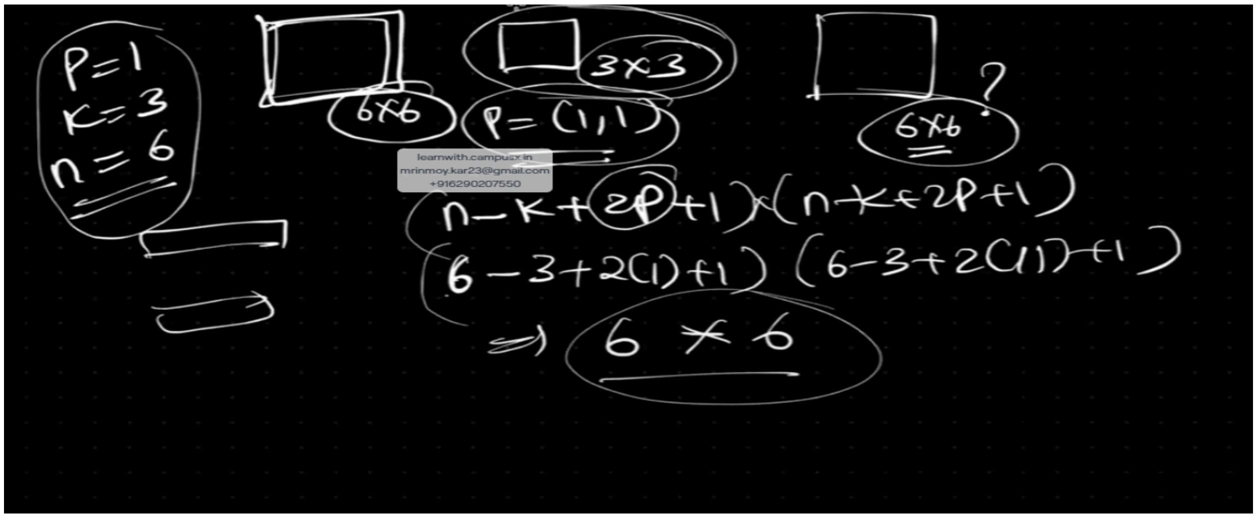
Here's how the terms are used:

- $(n-k+2P+1)$ calculates the spatial dimension of the output feature map along one dimension (either width or height).
- Multiplying $(n-k+2P+1)$ by itself ($\times(n-k+2P+1)$) accounts for both dimensions (width and height) of the output feature map, resulting in the total number of positions or steps the kernel can slide over the entire padded input feature map while remaining completely within its boundaries.

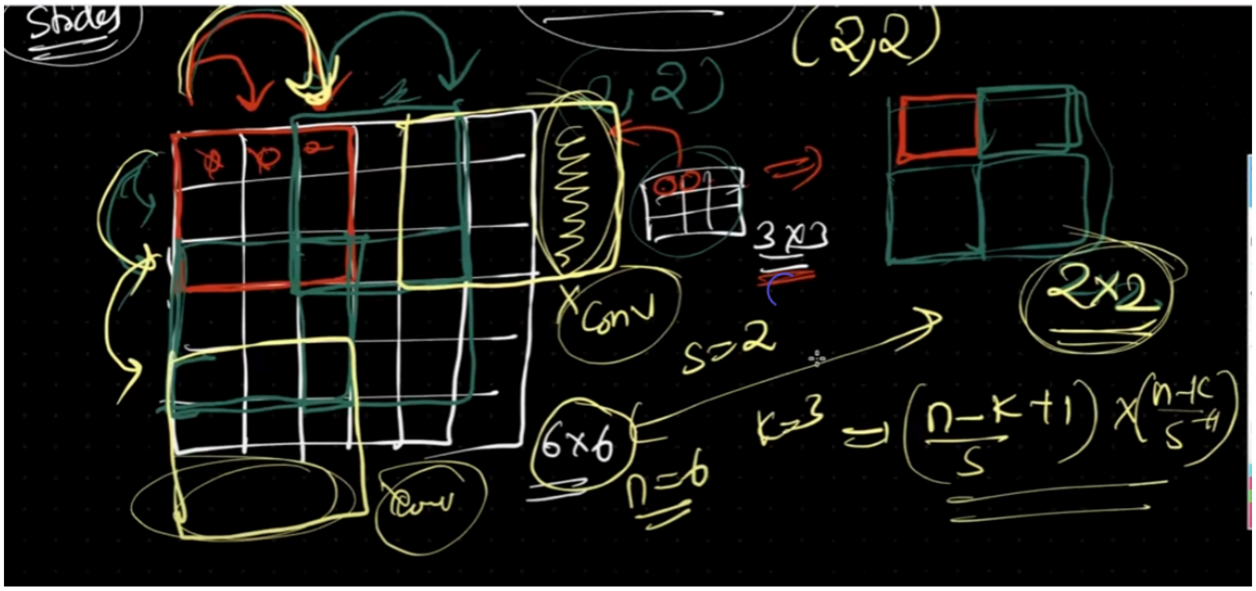
In summary, this expression helps calculate the spatial dimensions of the output feature map after applying convolutional operations with padding. It takes into account the size of the input feature map (n), the size of the convolutional kernel (k), and the amount of padding applied (P).

Why 2p?

In the formula $(n-k+2P+1)\times(n-k+2P+1)$, the term $2P$ accounts for the padding added on both sides (top and bottom, left and right) of the input feature map.



Strides - means jumping horizontally and vertically.



Without padding, the formula to compute the output size simplifies:

$$\text{Output Size} = \frac{\text{Input Size} - \text{Kernel Size}}{\text{Stride}} + 1$$

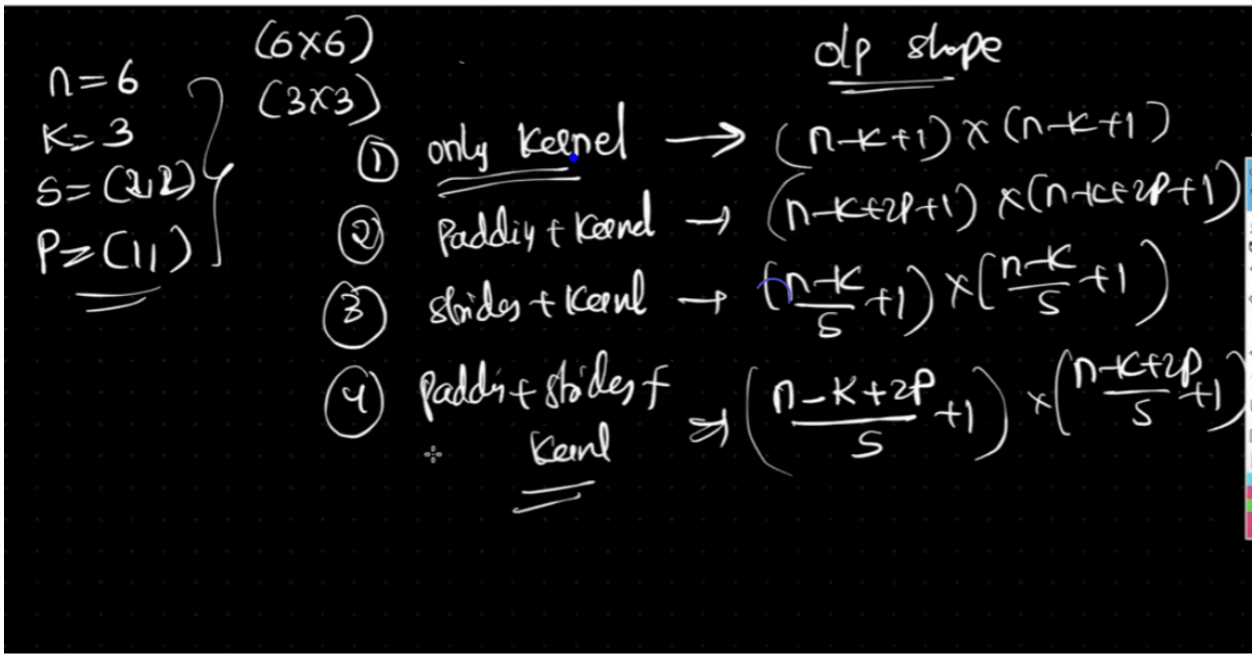
With padding:

$$\text{Output Size} = \frac{\text{Input Size} - \text{Kernel Size} + 2 \times \text{Padding}}{\text{Stride}} + 1$$

For example, if you have an input image of size 32x32, a 3x3 kernel and a stride of 1, the output size would be:

$$\text{Output Size} = \frac{32-3+2\times0}{1} + 1 = 30$$

So, the output feature map would be 30x30.



$n=6$
$K=3$
$S=(2,2)$
$P=(1,1)$

(6×6)
(3×3)

o/p shape

① only kernel $\rightarrow (n-k+1)\times(n-k+1)$

② Padding + kernel $\rightarrow (n-k+2p+1)\times(n-k+2p+1)$

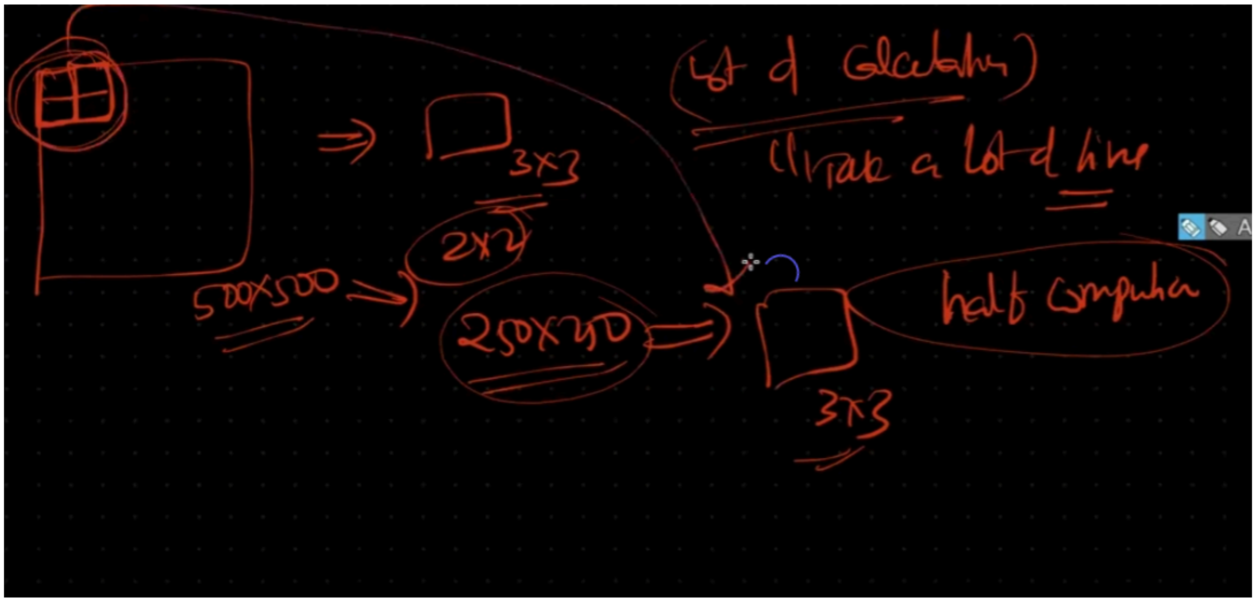
③ strides + kernel $\rightarrow \left(\frac{n-k}{s}+1\right)\times\left(\frac{n-k}{s}+1\right)$

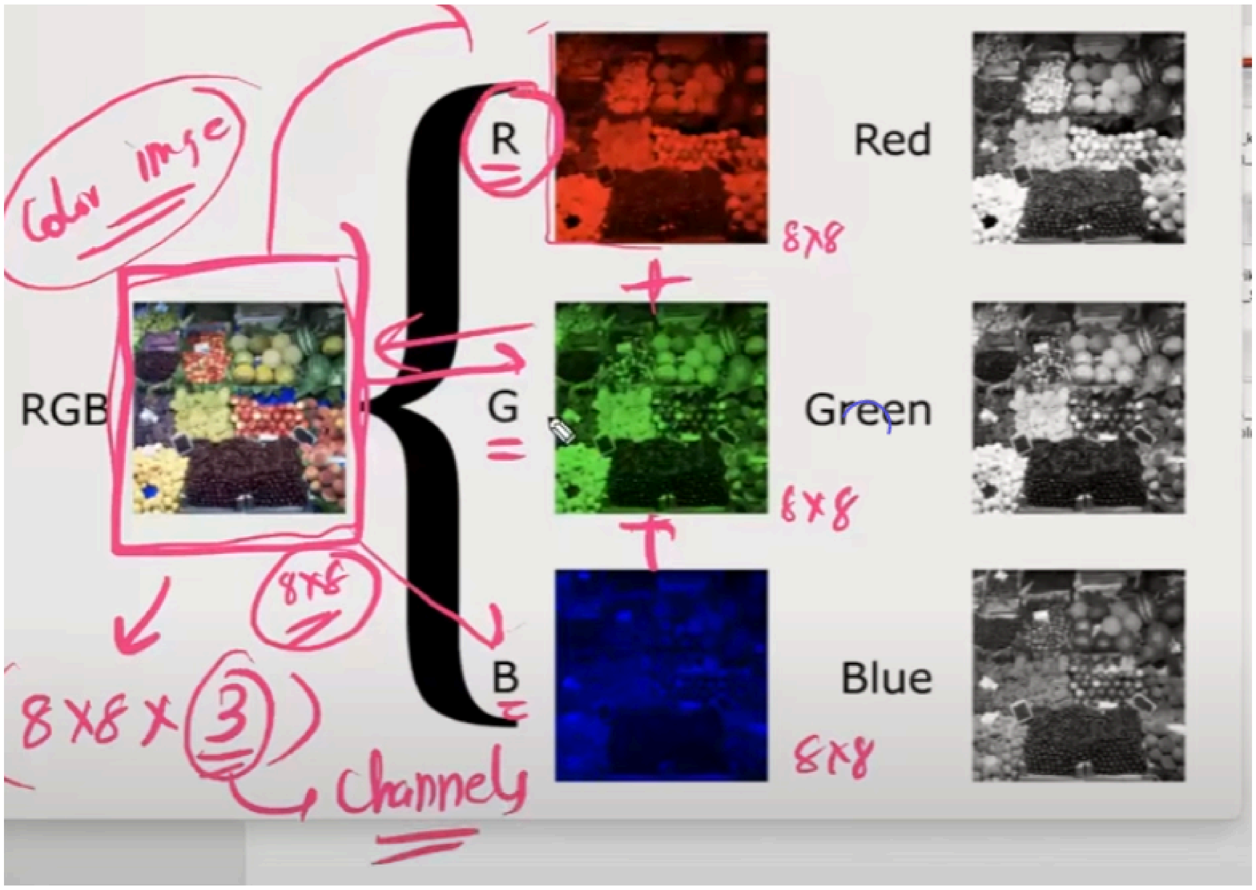
④ padding + strides + kernel $\rightarrow \left(\frac{n-k+2p}{s}+1\right)\times\left(\frac{n-k+2p}{s}+1\right)$

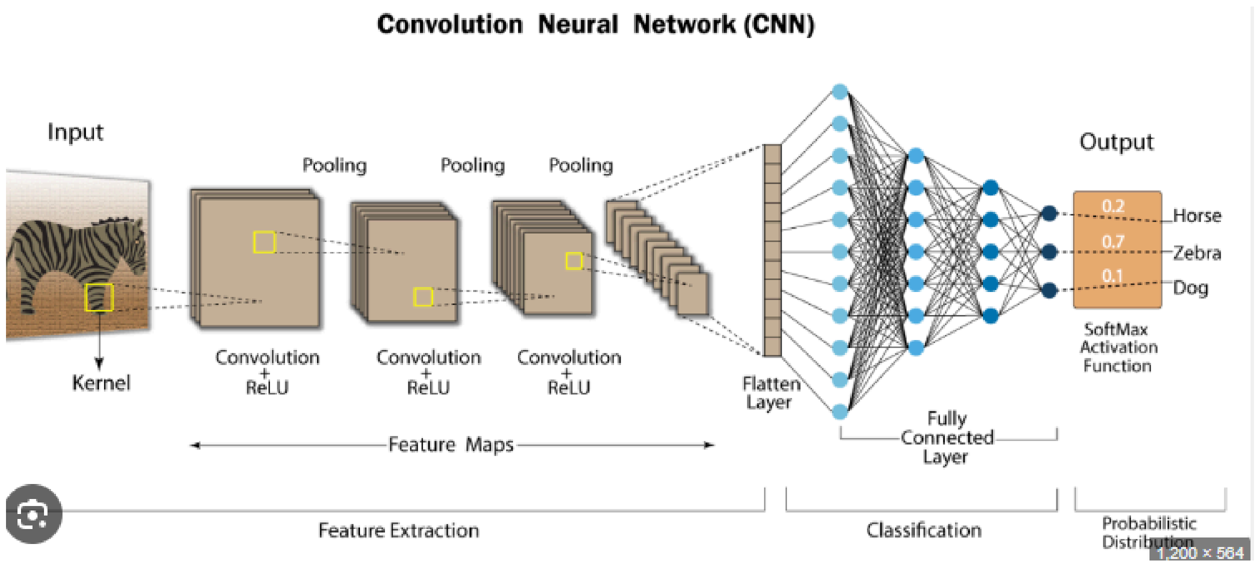
Pooling - Getting the max information
Pooling, in the context of deep learning and specifically convolutional neural networks (CNNs), is a downsampling operation applied to feature maps produced by convolutional layers. Pooling helps reduce the spatial dimensions (width and height) of the input volume, while retaining important information.

The most common type of pooling is max pooling, where a window (typically of size 2x2) is applied to the input feature map, and the maximum value within the window is selected as the output. This process is repeated across the entire feature map, with the window moving by a certain amount called the stride. Max pooling helps capture the most salient features in a given region of the input feature map while discarding less relevant information.

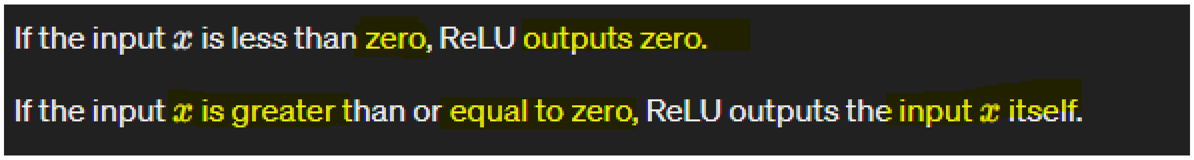
Another type of pooling is average pooling, where instead of taking the maximum value, the average value within the window is computed and used as the output





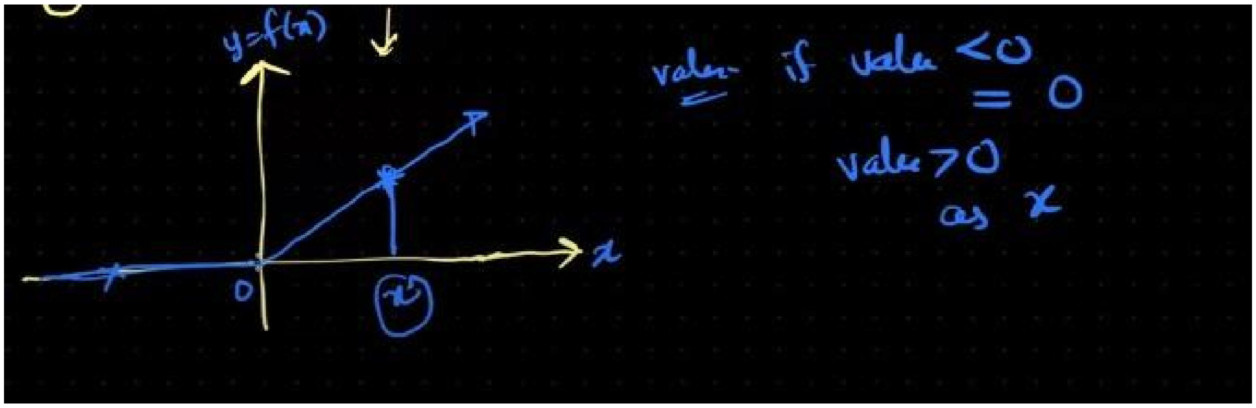


Relu -



If the input x is less than zero, ReLU outputs zero.

If the input x is greater than or equal to zero, ReLU outputs the input x itself.



Bias - is a trainable parameter

+
bias
3x3
Trainable
parameters

In a neural network, each neuron typically has its own bias parameter. The bias is an additional input to the neuron, just like the weighted sum of inputs. It allows the neuron to adjust its output independently of the inputs.

Mathematically, the bias b is added to the weighted sum of inputs $w \cdot x + b$
where w represents the weights,
x represents the inputs, and
b represents the bias.
The bias is trainable because it is adjusted during the training process along with the weights to minimize the loss function and improve the performance of the network.

Including biases in neural networks allows them to learn more complex functions and better fit the training data. They provide the network with additional degrees of freedom to model relationships between inputs and outputs.