Computer Vision

There are **two parts** to building a computer vision application:

1. Developing a Model.
2. Deploying it via an application.

CV Studio is a computer vision learning tool that can help you build your computer vision application in an interactive web application.

Image Processing Using (OpenCV & Pillow):

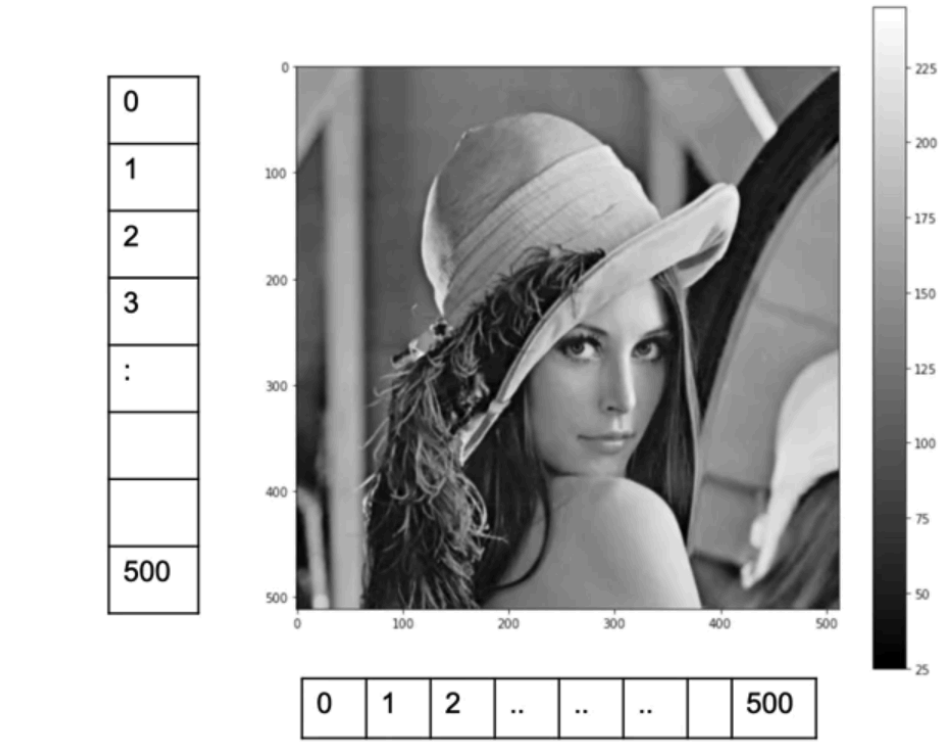
- Image Processing with OpenCV & Pillow:

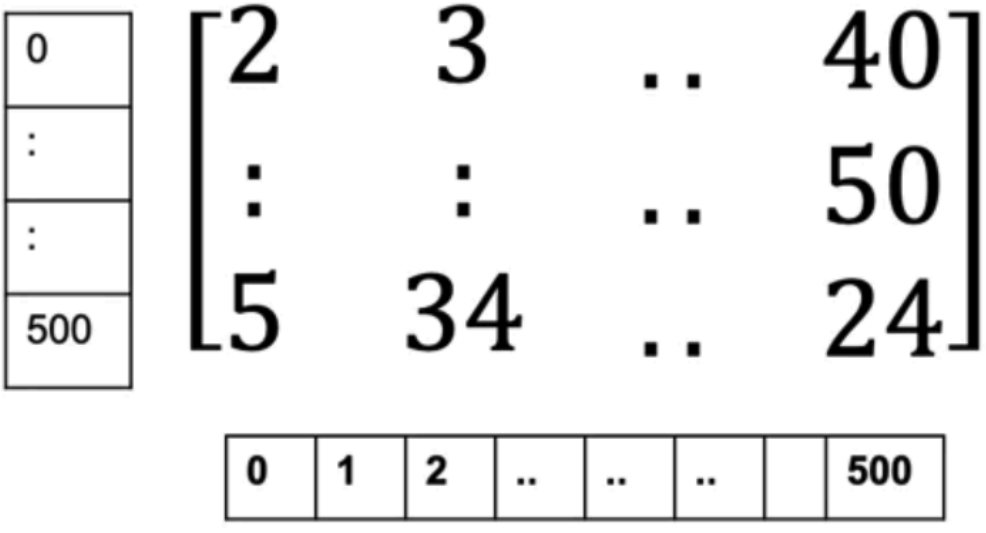
Image processing enhances images or extracts useful information from the image.

- What is a Digital Image?

- It's a **rectangular array of numbers**.
- The image is comprised of a rectangular grid of blocks called (**Pixels**).
- Pixels are represented by numbers called **intensity values.**
- Digital Images can be represented by intensity values between **0 to 255** - A total of **256**.
- **Darker shades** of gray have Lower values, with **zero as black**, and **Lighter shades** have Higher values, with **255 as white**.
- The array's **Raw** is the Digital Image **Height**, and The array's **Columan** is the Digital Image **Width**.

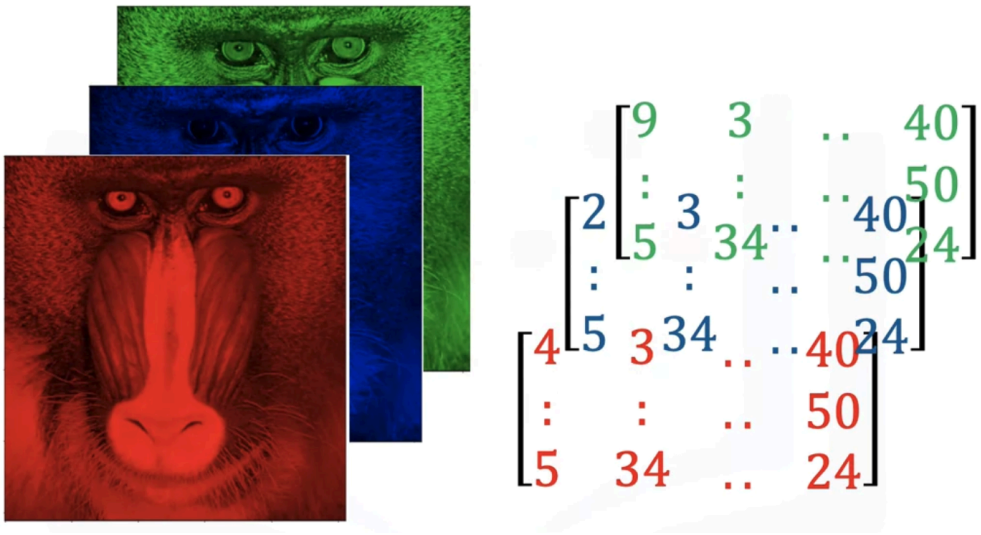
- The Digital Image array starts from the top left corner, where rows start at the top of the image and move down. For columns, we start from the left of the image and move down.



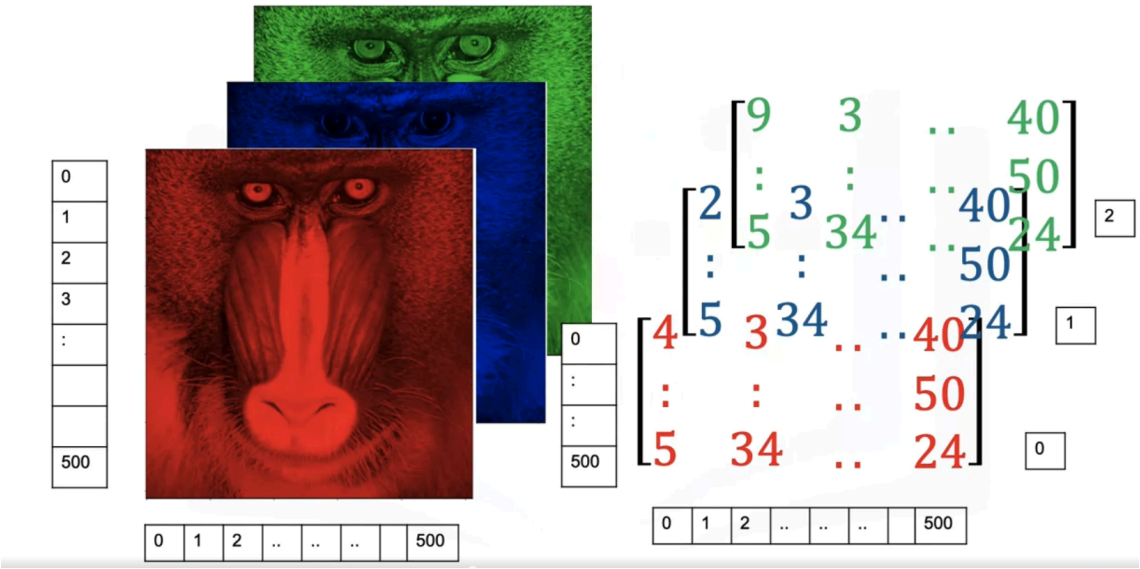


- A **<u>colored digital image is a competition of Red, Green, and Blue images</u>**. Also, it's referred to as an **RGB** image.

- A colored digital image is one of many color representations. These color values are represented as different channels, where in our case the colored digital image is Three channels (RGB). Each channel is represented by an array (Pixel index).



- In addition to accessing each image intensity with a row and column index, we also have an index for each channel, where (Zero) for Red, (1) for Green, and (2) for Blue.



Two popular image forms:

- JPEG - Joint Photographic Expert Group.
- PNG - Portable Network Graphics.

Python Code to Load an Image:

```
my_image= ”Image/Directory/ImageName”
```

Python has two main Image **Libraries** to work with images:

1. Pillow (PIL)
2. OpenCV

- OpenCV is a library used for computer vision, it has more functionality than PIL, but it is more difficult to use.

- **OpenCV** image channel order is (**BGR**), where **PIL** image order channels is (**RGB**)

==========

- Flip, Copy, and Crop images:

Flipping images changes the image orientation, and we can flip an image by changing the index values of a pixel or intensity. For example the following array:

I[0,0]	I[0,1]	I[0,2]	I[0,3]	I[0,4]	I[0,5]
I[1,0]	I[1,1]	I[1,2]	I[1,3]	I[1,4]	I[1,5]
I[2,0]	I[2,1]	I[2,2]	I[2,3]	I[2,4]	I[2,5]
I[3,0]	I[3,1]	I[3,2]	I[3,3]	I[3,4]	I[3,5]
I[4,0]	I[4,1]	I[4,2]	I[4,3]	I[4,4]	I[4,5]
I[5,0]	I[5,1]	I[5,2]	I[5,3]	I[5,4]	I[5,5]

I[0,0]	I[1,0]	I[2,0]	I[3,0]	I[4,0]	I[5,0]
I[0,1]	I[1,1]	I[2,1]	I[3,1]	I[4,1]	I[5,1]
I[0,2]	I[1,2]	I[2,2]	I[3,2]	I[4,2]	I[5,2]
I[0,3]	I[1,3]	I[2,3]	I[3,3]	I[4,3]	I[5,3]
I[0,4]	I[1,4]	I[2,4]	I[3,4]	I[4,4]	I[5,4]
I[0,5]	I[1,5]	I[2,5]	I[3,5]	I[4,5]	I[5,5]

Converting the column indexes to row indexes, the image will have a different orientation (Flipped). For the colored image, we need to flip all the channels (Red, Green & Blue).

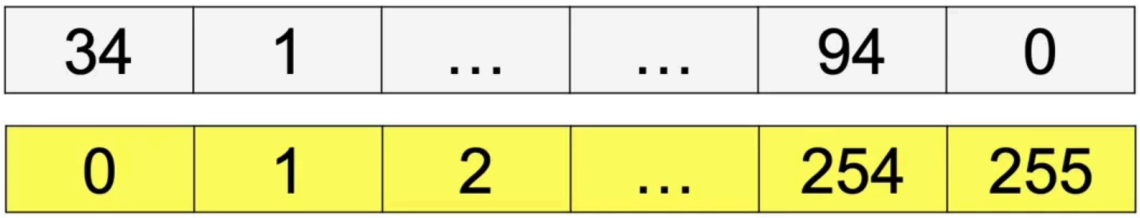
==========

- Pixel Transformation:

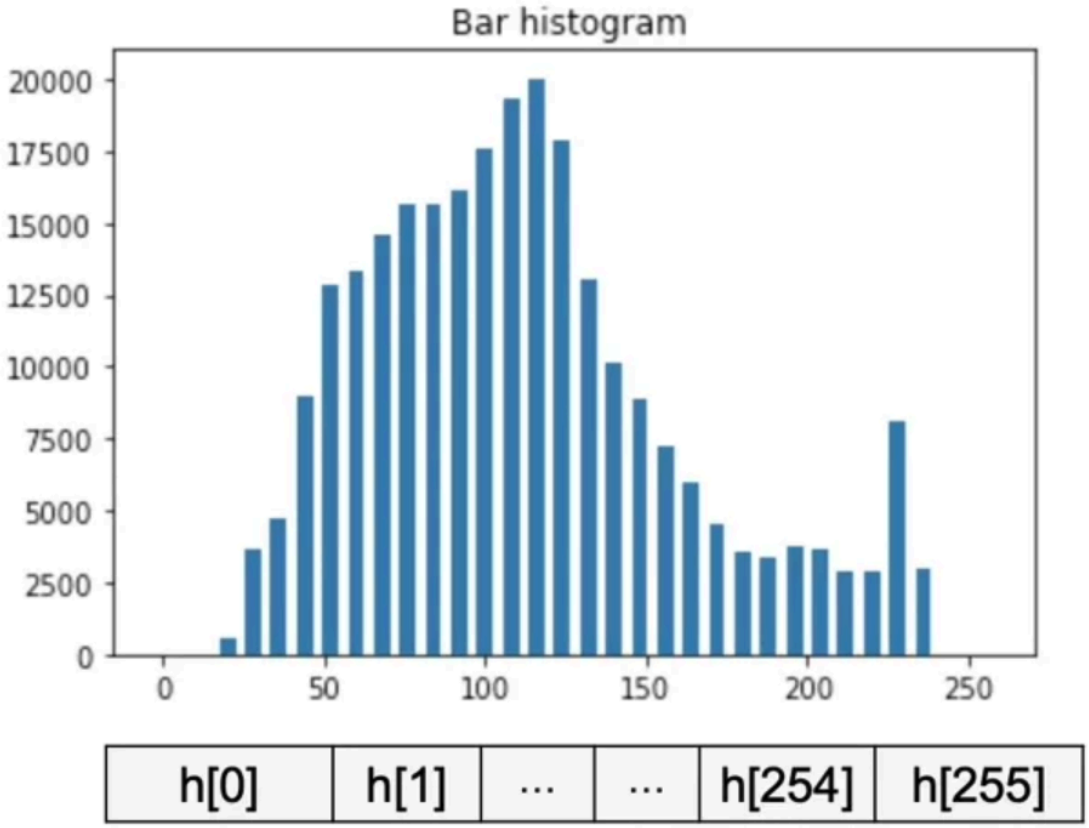
Pixel Transforms are **operations you perform one pixel**.

- Histograms:

A **histogram** counts the number of occurrences of the intensity values of pixels, and it's a useful tool for understanding and manipulating images.



The above figure shows the range of intensity values from (0) to (255), and shows how many times they appear in an image, and this is [***Histograms***]. We can plot the histograms as a graph shown below:



- **Intensity Transformation:**

It's helpful to think of an image as a function f(x,y) instead of an array at this point, where `x` is the row index and `y` is the column index. You can apply a transformation (T) to the image and get a new image:

$$g(x,y)=T(f(x,y))$$

An Intensity Transformation depends on **only one single point (x,y)**.

For example, you can apply a linear transform g(x,y) = 2f(x,y) + 1; this will multiply each image pixel by two and add one.

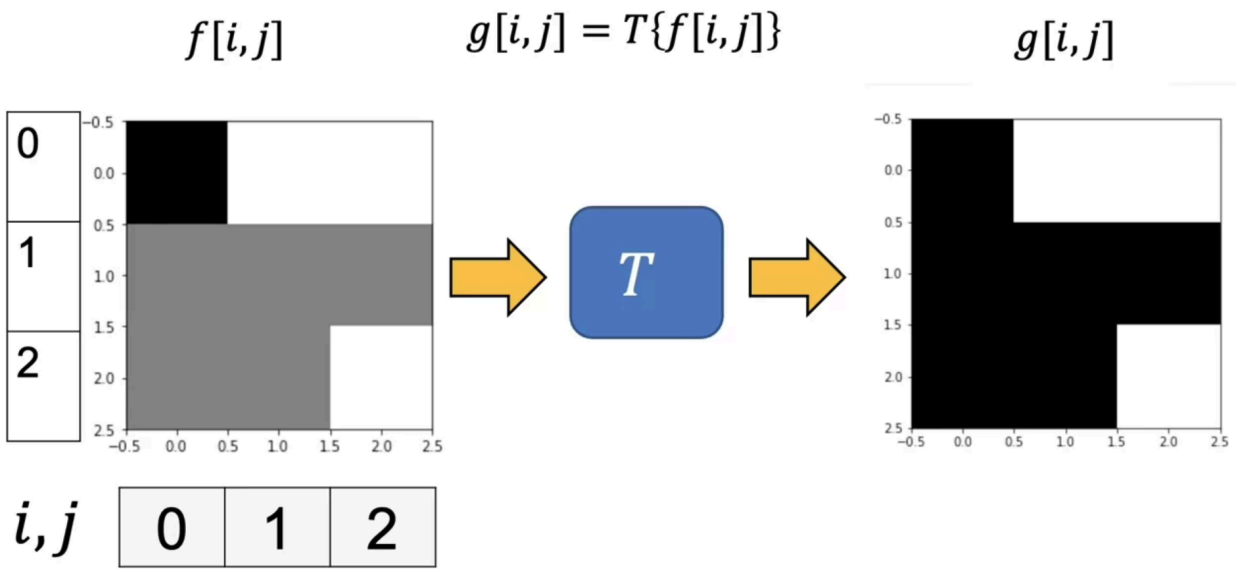
As the **Intensity transforms only depend on one valu**e; as a result, it is sometimes referred to as a gray-level mapping.

The variable if (r) is the gray level intensity, similar to the histogram values. The new output s is given by:

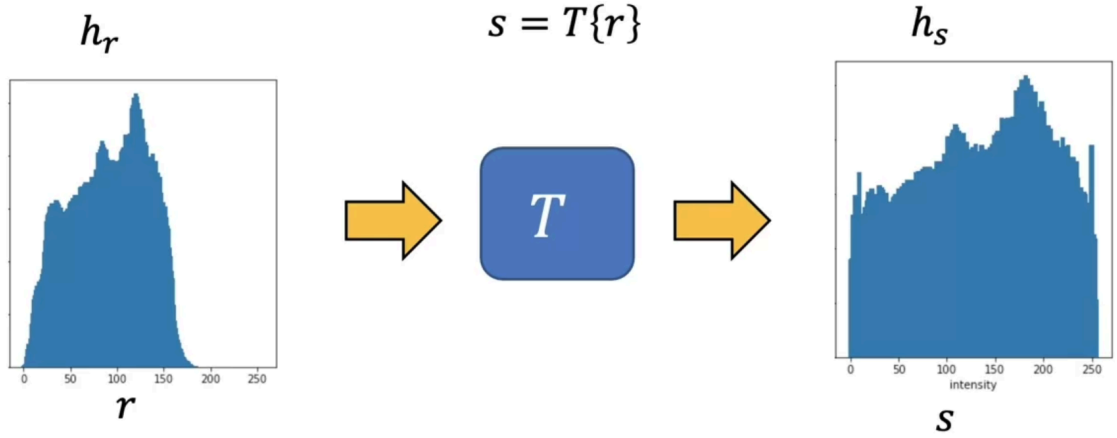
s=T(r)

Changing an image's pixel at a time. An ***intensity transformation (T)*** depends on only one single point **(Pixel) (i,j)** in the image array (f).

The **image array (f)** is converted to array **(g)**.



For the ***gray image array*** **(r)** is converted to **arry (s).**



Using Intensity Transformation (T), allows us to apply linear transformation to the image array (f).

$f[i,j]$ $\quad g[i,j] = 2f[i,j] + 1 \quad$ $g[i,j]$

0	2	2
1	1	1
1	1	2

1	5	5
3	3	3
3	3	5

The above transformation changes the intensity value of each pixel, where we apply the following function.

The colored image's pixel intensity values are a function of (f) or histogram (f), The transform maps each intensity value at (f) to (g).

$$g[i,j] = 2f[i,j] + 1$$

The gray image's pixel intensity values are a function of (r) or histogram (r), The transform maps each intensity value at (r) to (s).

$$s[i,j] = 2r[i,j] + 1$$

r	h_r
0	1
1	5
2	3
3	0
4	0
5	0

$s = 2r + 1$

s	h_s
0	0
1	1
2	0
3	5
4	0
5	3

- ***Image Negative:***

Image negative is where we can **reverse the intensity levels** of an image. **From High Intensity values to Low Intensity Values, or vice versa.** We can use the function below:

$$s[i,j] = -[i,j] + 255$$

- **Brightness and Contrast Adjusting:**

A linear transformation can be applied to adjust the Brightness and Contrast, we can use the following linear model:

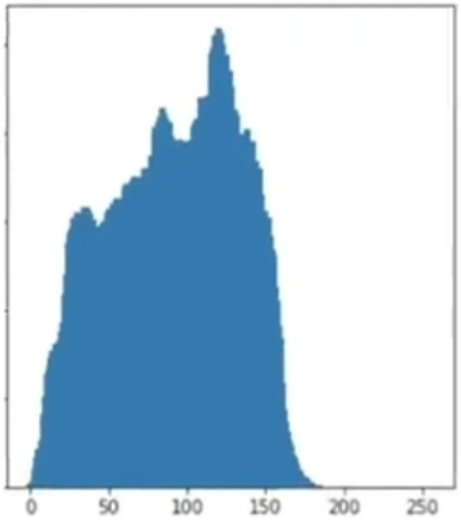
$$g[i,j] = \alpha f[i,j] + \beta$$

Where:

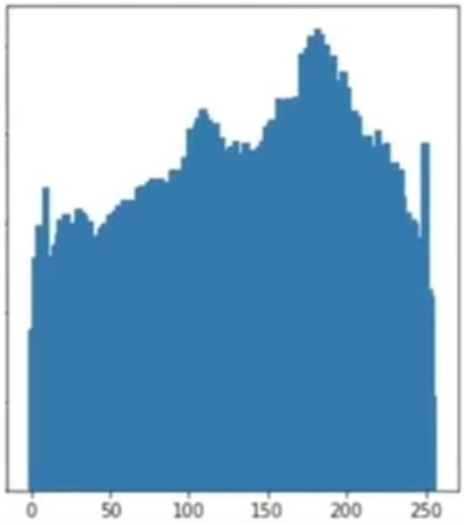
- **Alpha** is Contrast Control.
- **Beta** is Brightness Control.

- ## Histogram Equalization:

Histogram Equalization is an algorithm that uses the image's Histogram to adjust the contrast, where **using the histogram to determine a transform that flattens the histogram**.



To



- ## Thresholding and Simple Segmentation:

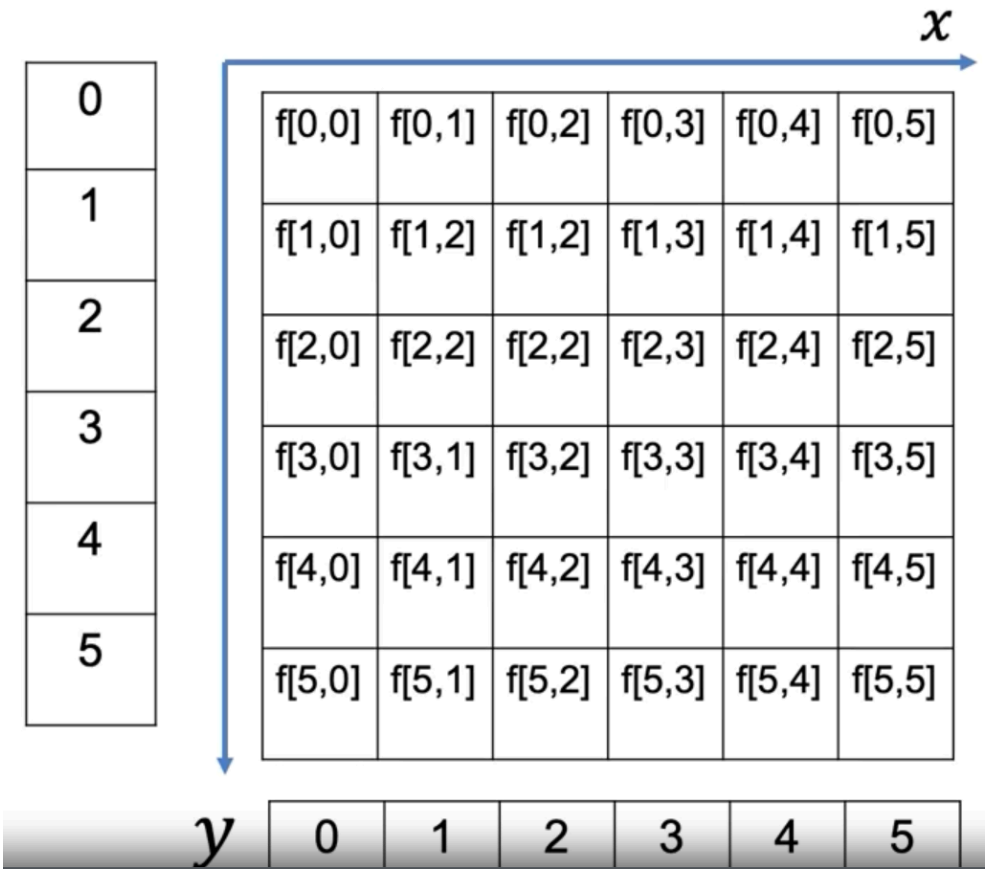
It can be used in extracting objects from an image (**Segmentation**).

Thresholding is used in image segmentation; this means extracting objects from an image. Image segmentation is used in many applications including extracting text, medical imaging, and industrial imaging. Thresholding an image takes a threshold; If a particular pixel (i,j) is greater than that threshold it will set that pixel to some value usually 1 or 255, otherwise, it will set it to another value, usually zero.

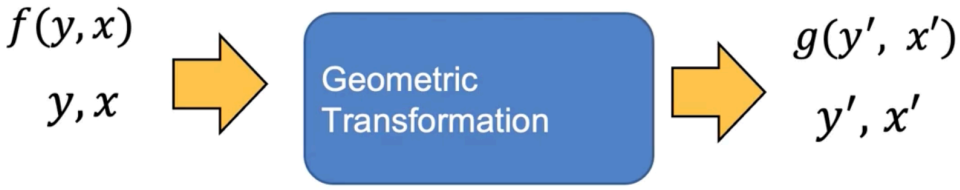
Geometric Operations:

Geometric Operations are: Scaling, translation, and Rotation of images.

The figure below shows one channel representation, we treat the image as a function of (y) and (x), where (y) is the vertical direction, and (x) is the horizontal direction.



The transformation is for a function of **f(y,x)** transform to a function of **g(y',x')** - [x Prime & y prime]

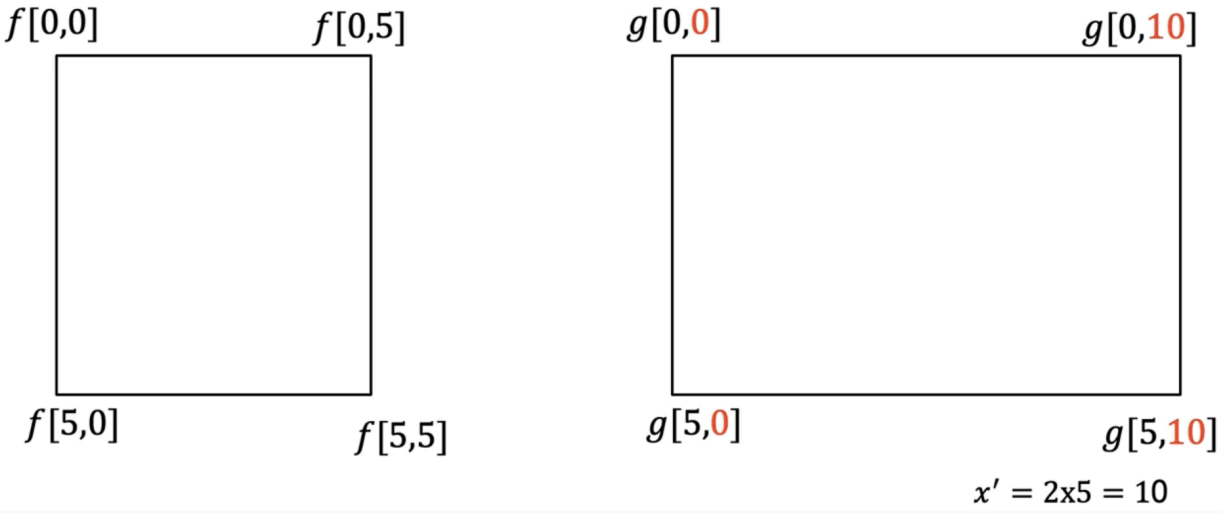


- **Scale**:

Scaling is reshaping the image, we can shrink or extend the image in a horizontal and/or vertical direction.

$$g(y,2x) = f(y,x)$$

$x' = ax$ $x' = 2x$



The relationship between the pixels of **f(y,x)** & **g(y',x')**:

f[0,0]	f[0,1]	f[0,2]	f[0,3]	f[0,4]	f[0,5]
f[1,0]	f[1,1]	f[1,2]	f[1,3]	f[1,4]	f[1,5]
f[2,0]	f[2,1]	f[2,2]	f[2,3]	f[2,4]	f[2,5]
f[3,0]	f[3,1]	f[3,2]	f[3,3]	f[3,4]	f[3,5]
f[4,0]	f[4,1]	f[4,2]	f[4,3]	f[4,4]	f[4,5]
f[5,0]	f[5,1]	f[5,2]	f[5,3]	f[5,4]	f[5,5]

g[0,0]	g[0,1]	g[0,2]	g[0,3]	g[0,4]	g[0,5]	g[0,6]	g[0,7]	g[0,8]	g[0,9]	g[0,10]
g[1,0]	g[1,1]	g[1,2]	g[1,3]	g[1,4]	f[1,5]	g[1,6]	g[1,7]	g[1,8]	g[1,9]	g[1,10]
g[2,0]	g[2,1]	g[2,2]	g[2,3]	g[2,4]	g[2,5]	g[2,6]	g[2,7]	g[2,8]	g[2,9]	g[2,10]
g[3,0]	g[3,1]	g[3,2]	g[3,3]	g[3,4]	g[3,5]	g[3,6]	g[3,7]	g[3,8]	g[3,9]	g[3,10]
g[4,0]	g[4,1]	g[4,2]	g[4,3]	g[4,4]	g[4,5]	g[4,6]	g[4,7]	g[4,8]	g[4,9]	g[4,10]
g[5,0]	g[5,1]	g[5,2]	g[5,3]	g[5,4]	g[5,5]	g[5,6]	g[5,7]	g[5,8]	g[5,9]	g[5,10]

In the figure above, we see **several columns** of g(y',x') *do not have corresponding* values in f(y,x). To determine the unknown column's pixel values, we use **Interpolation** to fill the gaps, where we use neighboring pixels to determine the value of an unknown pixel - In this case, we use the nearest neighbouring pixel.

f[0,0]	f[0,1]	f[0,2]	f[0,3]	f[0,4]	f[0,5]
f[1,0]	f[1,1]	f[1,2]	f[1,3]	f[1,4]	f[1,5]
f[2,0]	f[2,1]	f[2,2]	f[2,3]	f[2,4]	f[2,5]
f[3,0]	f[3,1]	f[3,2]	f[3,3]	f[3,4]	f[3,5]
f[4,0]	f[4,1]	f[4,2]	f[4,3]	f[4,4]	f[4,5]
f[5,0]	f[5,1]	f[5,2]	f[5,3]	f[5,4]	f[5,5]

g[0,0]	g[0,0]	g[0,2]	g[0,2]	g[0,4]	g[0,4]	g[0,6]	g[0,6]	g[0,8]	g[0,8]	g[0,10]
g[1,0]	g[1,0]	g[1,2]	g[1,2]	g[1,4]	g[1,4]	g[1,6]	g[1,6]	g[1,8]	g[1,8]	g[1,10]
g[2,0]	g[2,0]	g[2,2]	g[2,2]	g[2,4]	g[2,4]	g[2,6]	g[2,6]	g[2,8]	g[2,8]	g[2,10]
g[3,0]	g[3,0]	g[3,2]	g[3,2]	g[3,4]	g[3,4]	g[3,6]	g[3,6]	g[3,8]	g[3,8]	g[3,10]
g[4,0]	g[4,0]	g[4,2]	g[4,2]	g[4,4]	g[4,4]	g[4,6]	g[4,6]	g[4,8]	g[4,8]	g[4,10]
g[5,0]	g[5,0]	g[5,2]	g[5,2]	g[5,4]	g[5,4]	g[5,6]	g[5,6]	g[5,8]	g[5,8]	g[5,10]

We can use the same concept to scale the image Vertically, where we ***multiply*** (x) in function (f).

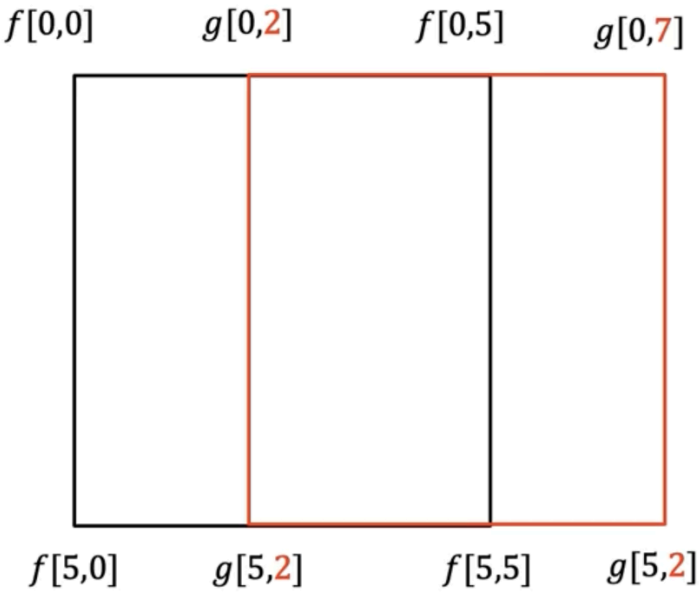
- ## Translation:

The translation is shifting the image. We can shift an image horizontally by adding the number of pixels (x) in function (f).

$$g(y', x') = f(y, x + t)$$

Where:

$$x' = x + t$$



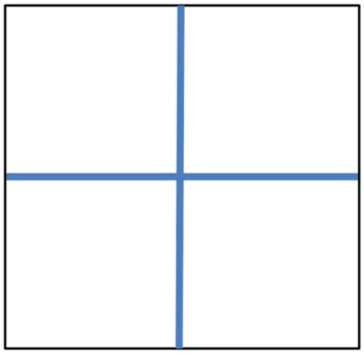
f[0,0]	f[0,1]	f[0,2]	f[0,3]	f[0,4]	f[0,5]
f[1,0]	f[1,2]	f[1,2]	f[1,3]	f[1,4]	f[1,5]
f[2,0]	f[2,2]	f[2,2]	f[2,3]	f[2,4]	f[2,5]
f[3,0]	f[3,1]	f[3,2]	f[3,3]	f[3,4]	f[3,5]
f[4,0]	f[4,1]	f[4,2]	f[4,3]	f[4,4]	f[4,5]
f[5,0]	f[5,1]	f[5,2]	f[5,3]	f[5,4]	f[5,5]

	g[0,1]	g[0,2]	g[0,3]	g[0,4]	g[0,5]
	g[1,1]	[1,2]	g[1,3]	g[1,4]	f[1,5]
	g[2,1]	f[2,2]	g[2,3]	g[2,4]	g[2,5]
	g[3,1]	f[3,2]	g[3,3]	g[3,4]	g[3,5]
	g[4,1]	f[4,2]	g[4,3]	g[4,4]	g[4,5]
	g[5,1]	f[5,2]	g[5,3]	g[5,4]	g[5,5]

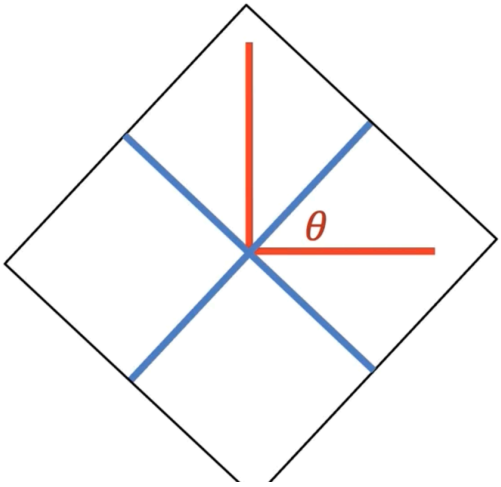
We can apply the same concept to shifting the image vertically by adding pixels (t) to (y) in function (f).

- Rotation:

We can rotate an image by angle (theta).



θ
Theta



The rotation of the image is from the center of the image.

- Geometric Transformation (GT) Representation:

We can represent (GT) as a set of equations:

$$x' = ax + t_x$$
$$y' = dy + t_y$$

a - for Horizontal Scale
d - for Vertical Scale
tx - for Horizontal Translation
ty - Vertical Translation

Then, we can put the **equations** ***in a matrix form***:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & 0 \\ 0 & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

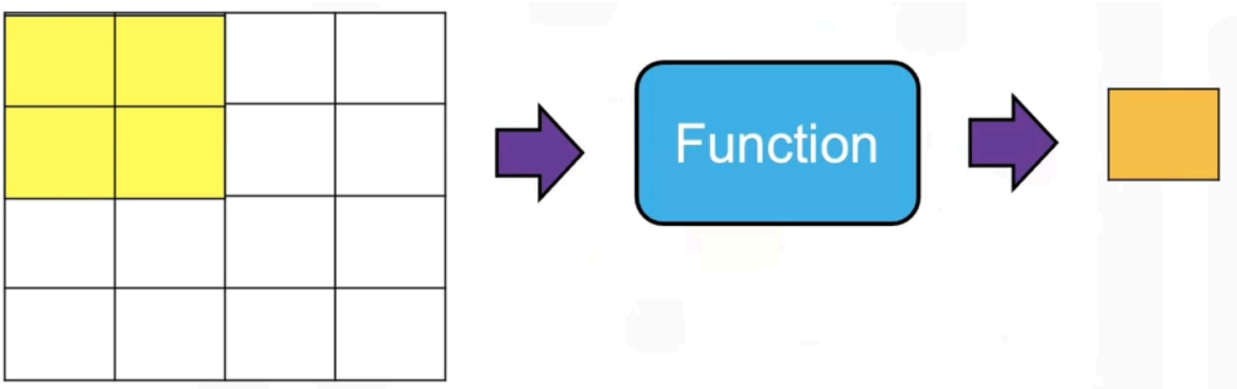
After that, we can ***get the Affine Transformation Matrix***:

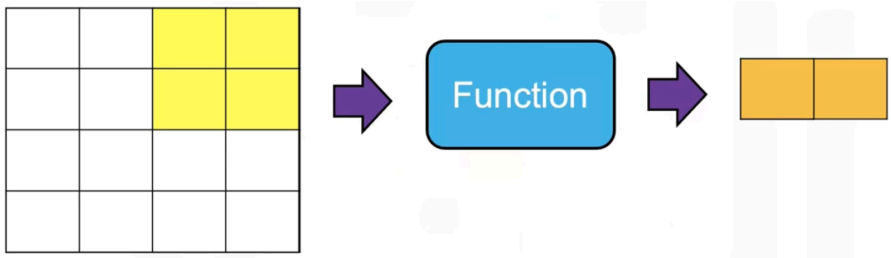
$$\begin{bmatrix} a & 0 & t_x \\ 0 & b & t_y \end{bmatrix}$$

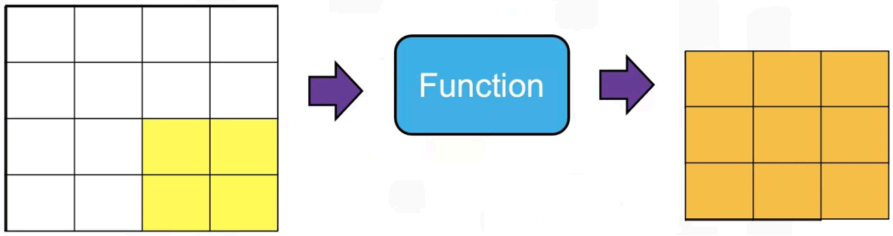
Affine Matrix is used in the OpenCV library for ***Translation & Rotation***.

Spatial Operations in Image Processing:

Spatial Operations consist of neighborhood pixels applying a function on them and **output the results**. Then, shift the neighborhood and repeat the process for each pixel in the image. The **result is a new image with enhanced characteristics**.







These spatial operations take advantage of **spatial patterns** in the image.

- Convolution: Linear Filtering:

Convolution is a standard way to filter an image, the **filter is called the Kernel**. The different Kernel performs different tasks.

Filtering involves enhancing an image, for example, removing the Noise from an image. Noise is caused by a bad camera or bad image compression. The same factors that cause noise may lead to blurry images, we can apply filters to sharpen these images. Convolution is a standard way to Filter an image the filter is called the kernel and different kernels perform different tasks. In addition, Convolution is used for many of the most advanced artificial intelligence algorithms. We simply take the dot product of the kernel as an equally sized portion of the image. We then shift the kernel and repeat.

Convolution is a linear equation:

$$z = wx$$

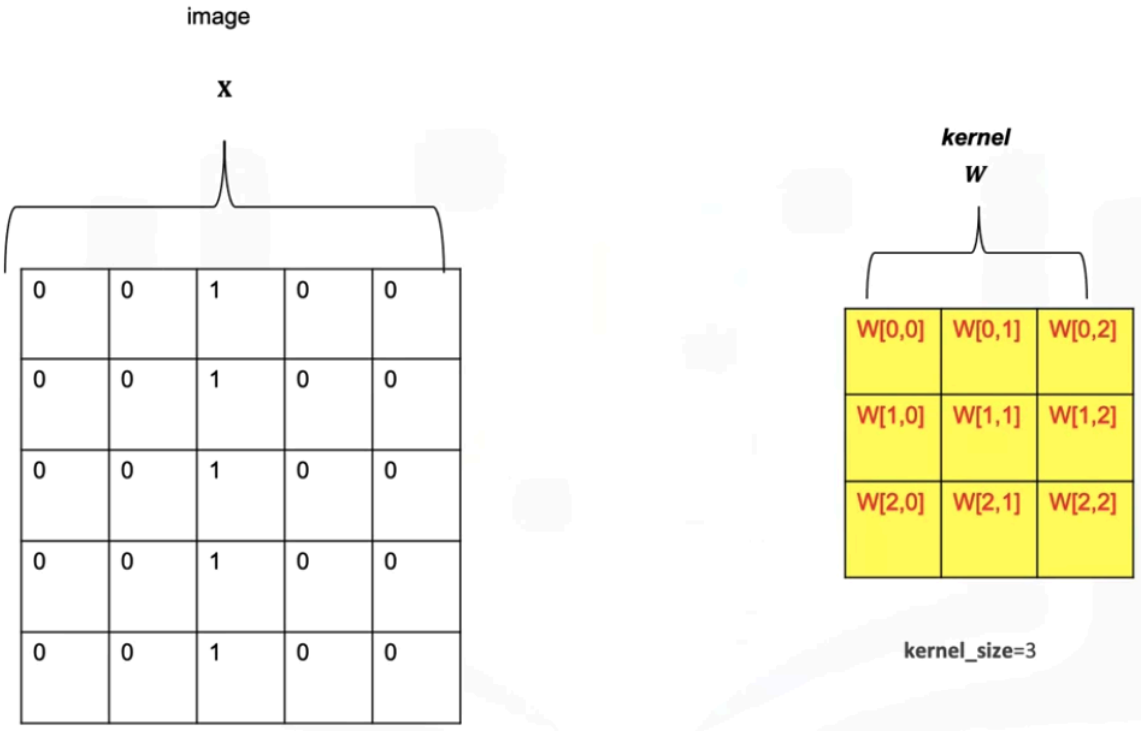
Where:

x - The image input.
z - The different image output.
w - The Kernel parameter (Filter).

The **star is the Convolution Operation**:

$$Z = W * X$$

Example of Linear Filtering:



The Kernel parameter is an **array** that is made up of **values that have been predetermined for some operations**.

The operation starts in the top right corner, and overlays the kernel values with that region (Red Values) as shown below:

X

0	1	0	0	1	-1	0		0			
0	2	0	0	1	-2	0		0			
0	1	0	0	1	-1	0		0			
0		0		1		0		0			
0		0		1		0		0			

$$Z = W * X$$

X

0 1	0 0	1 -1	0	0
0 2	0 0	1 -2	0	0
0 1	0 0	1 -1	0	0
0	0	1	0	0
0	0	1	0	0

0x1+0x0+1x-1	-1
+	+
0x2+0x0+1x-2	-2
+	+
0x1+0x0+1x-1	-1
	4

Z

-4

Then, we shift the kernel to the left and do the same process. As shown below

$$Z = W * X$$

X

0	0 1	1 0	0 -1	0
0	0 2	1 0	0 -2	0
0	0 1	1 0	0 -1	0
0	0	1	0	0
0	0	1	0	0

0x1+1x0+0x-1	0
+	+
0x2+1x0+0x-2	0
+	+
0x1+0x0+0x-1	0
	0

Z

-4	0

$$Z = W * \mathbf{X}$$

X

0	0	1 1	0 0	0 -1
0	0	1 2	0 0	0 -2
0	0	1 1	0 0	0 -1
0	0	1	0	0
0	0	1	0	0

Z

-4	0	4

$$Z = W * \mathbf{X}$$

X

0	0	1	0	0
0 1	0 0	1 -1	0	0
0 2	0 0	1 -2	0	0
0 1	0 0	1 -1	0	0
0	0	1	0	0

Z

-4	0	4
-4		

$$Z = W * X$$

X

0	0	1	0	0
0	0	1	0	0
0	0	1 1	0 0	0 -1
0	0	1 2	0 0	0 -2
0	0	1 1	0 0	0 -1

Z

-4	0	4
-4	0	4
-4	0	4

We keep shifting the Kernel until we finish all the pixels. Then, we will **get a new image**.

- The problem is that the **new image** is **different in size than the original image** as shown above (X) & (Z).
    - To fix this, we can change the image (X) size.

We can **change the image (X) size** in two ways:

- **Padding**:
    - Adding two zeros of columns and rows in image (X) - [Zero Padding], as shown below:

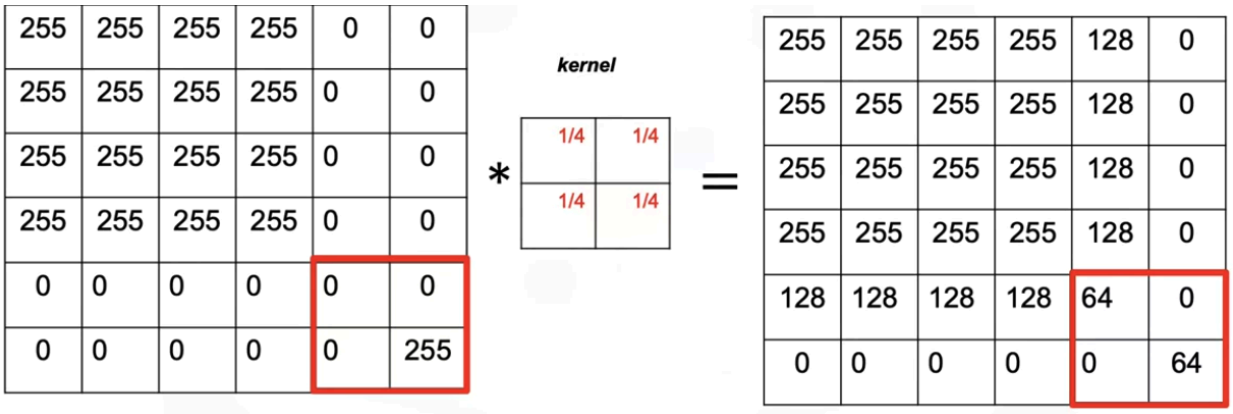
0	0	0	0	0	0
0					0
0					0
0					0
0					0
0	0	0	0	0	0

- **<u>Replicate the values of the Border:</u>**

12	12	8	4	2	2
12	12	8	4	2	2
2	2	2	4	4	4
3	3	4	2	3	3
4	4	20	3	4	4
4	4	20	3	4	4

- Low Pass Filter:

Low pass filters are used to **smooth the image** and get **rid of noise**. Something Filters **average out Pixels** within a neighborhood.



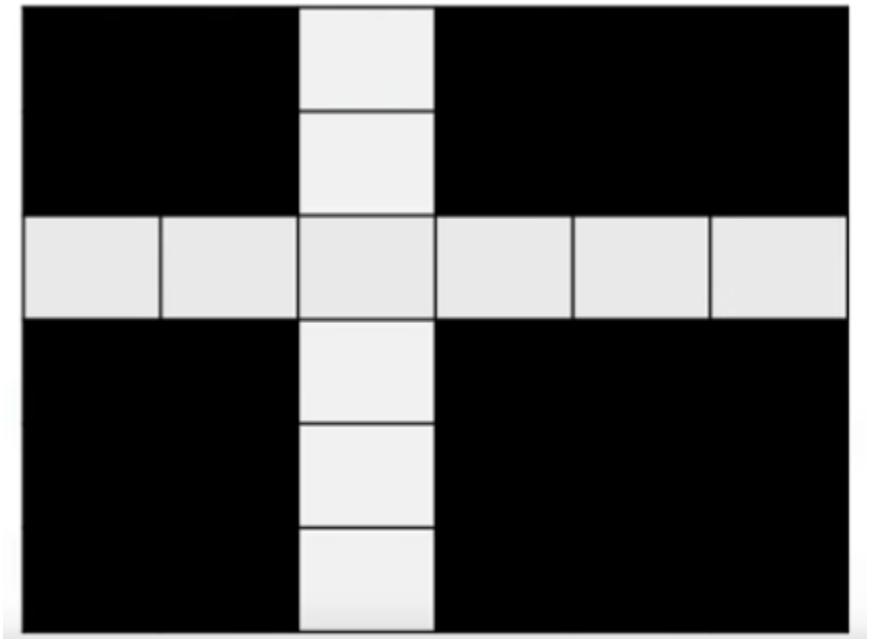
255	255	255	255	0	0
255	255	255	255	0	0
255	255	255	255	0	0
255	255	255	255	0	0
0	0	0	0	0	0
0	0	0	0	0	255

255	255	255	255	128	0
255	255	255	255	128	0
255	255	255	255	128	0
255	255	255	255	128	0
128	128	128	128	64	0
0	0	0	0	0	64

There is a trade-off between Sharpness and Smoothness.

- **Edge Detection:**

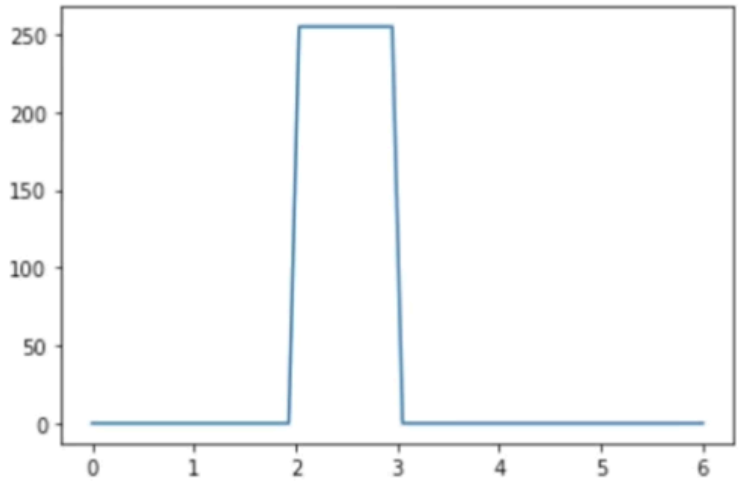
Edges in a digital image are the **image brightness changes sharply**, as shown below:



Edge detection is a **vision algorithm that detects edges** in a digital image.

- Edge detection uses methods to **approximate derivatives and gradients** to identify edges.

The figure below shows a plot of the first rows (Horizontal-axis)(Index) - (Vertical-axis) (Intensity Value):



0	0	255	0	0	0
0	0	255	0	0	0
255	255	255	255	255	255
0	0	255	0	0	0
0	0	255	0	0	0
0	0	255	0	0	0

From Pixel (1) to (2), the intensity value increases, and **we can represent this in the image as a vector**. The same thing if we move from Pixel (2) to (3) the **intensity value decreases**.

[Horizontal Changes (Derivative)]

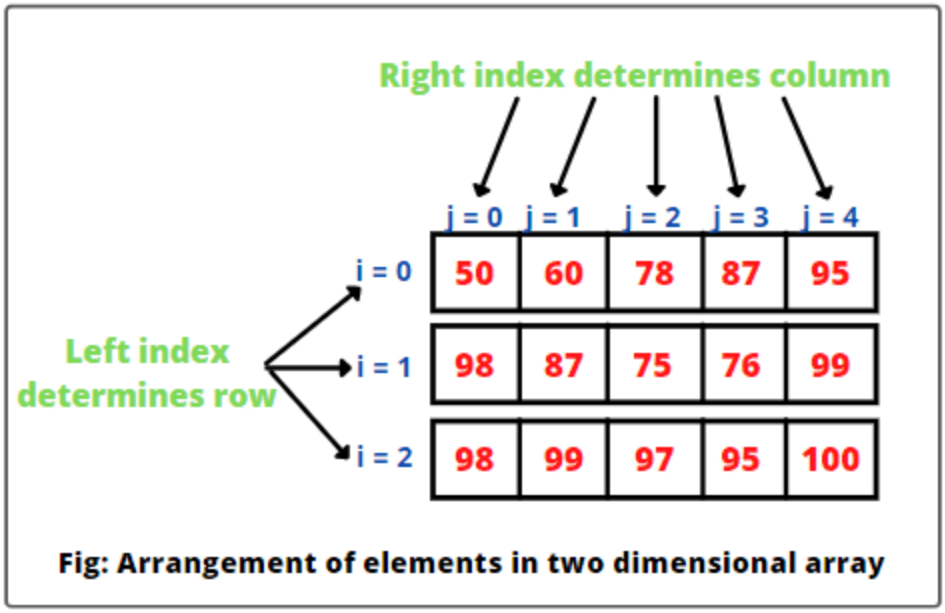
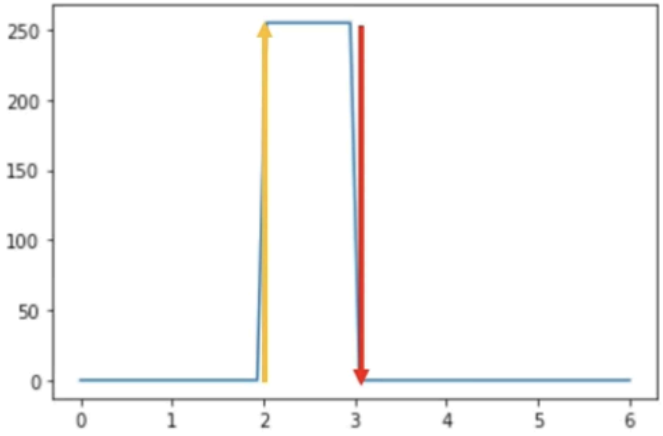


Fig: Arrangement of elements in two dimensional array



0	0	255	0	0	0
0	0	255	0	0	0
255	255	255	255	255	255
0	0	255	0	0	0
0	0	255	0	0	0
0	0	255	0	0	0

The Vector direction represents **if the adjacent pixel is increasing**. This means the vector is pointing from a small value to a large one.

We can **represent these vectors (Changes)** by applying the **following difference equation**:

0	1	2	3	4	5
0	0	255	0	0	0
0	0	255	0	0	0
255	255	255	255	255	255
0	0	255	0	0	0
0	0	255	0	0	0
0	0	255	0	0	0

$$I[i, j+1] - I[i, j]$$

It works by subtracting the **intensity value** of the a**djacent column (j) and (j+1) in raw (i)**. This computes the gradiance (Direction and Magnitude) in the (x-direction).

- **Gradiance**: ∇ f whose value at a point (p) gives the direction and the rate of fastest increase.

  Operation for **column (1) and (2)**:

0	1	2	3	4	5
0	0	255	0	0	0
0	0	255	0	0	0
255	255	255	255	255	255
0	0	255	0	0	0
0	0	255	0	0	0
0	0	255	0	0	0

$$I[i, j+1] - I[i, j]$$

$$\frac{\partial I}{\partial x}$$

i	$I[i,2]$	$I[i,1]$	$I[i,j+1]-I[i,j]$
0	255	0	255
1	255	0	255
2	255	255	0
3	255	0	255
4	255	0	255
6	255	0	255

We can overlay the results over the image as vectors:

0	1	2	3	4	5
0	0	255	0	0	0
0	0	255	0	0	0
255	255	255	255	255	255
0	0	255	0	0	0
0	0	255	0	0	0
0	0	255	0	0	0

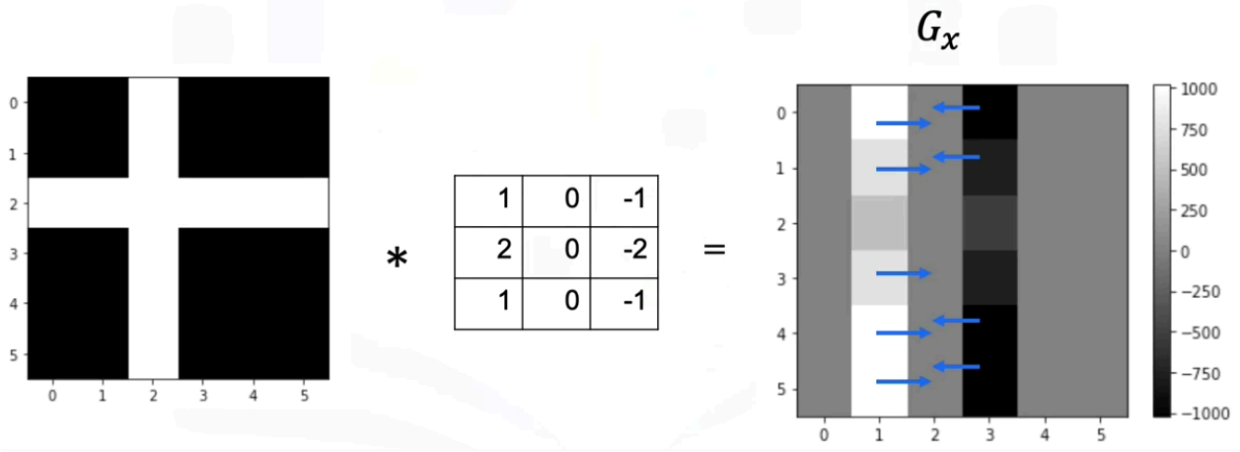
Operation for **columns (2) and (3)**:

0	1	2	3	4	5
0	0	255	0	0	0
0	0	255	0	0	0
255	255	255	255	255	255
0	0	255	0	0	0
0	0	255	0	0	0
0	0	255	0	0	0

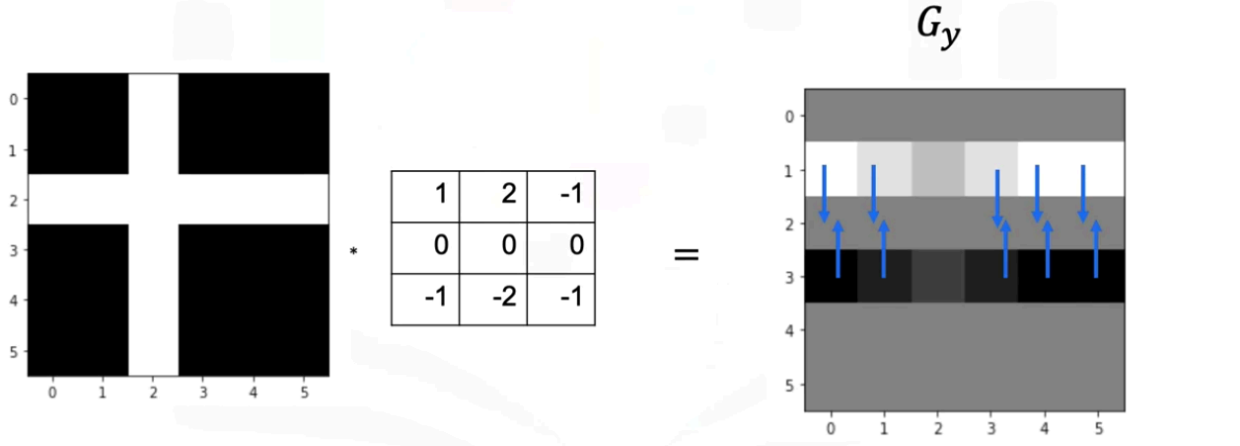
i	$I[i,3]$	$I[i,2]$	$I[i,j+1]-I[i,j]$
0	0	255	-255
1	0	255	-255
2	255	255	0
3	0	255	-255
4	0	255	-255
6	0	255	-255

The results are negative, so the **arrow is in the other direction**.

- We can perform (**Horizontal** Derivative (changes)) using (**Convolution**), using **Sobel Kernel.**

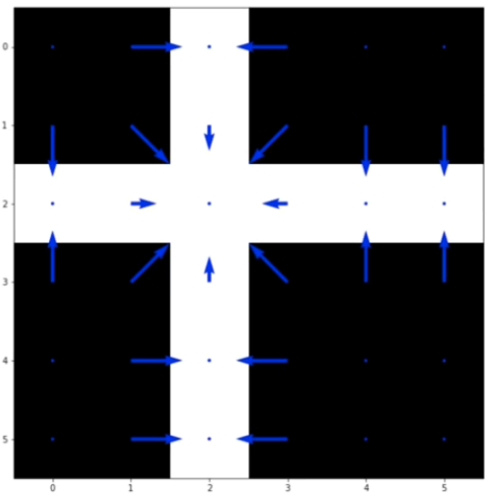


- We can perform (**Vertical** Derivative (changes)) using (**Convolution**).



We can combine the output of each filter into a vector.

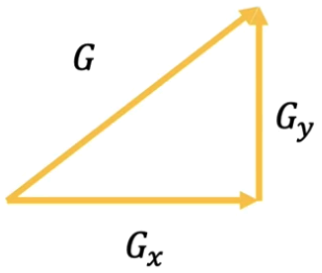
$$G = [G_x, G_y]$$

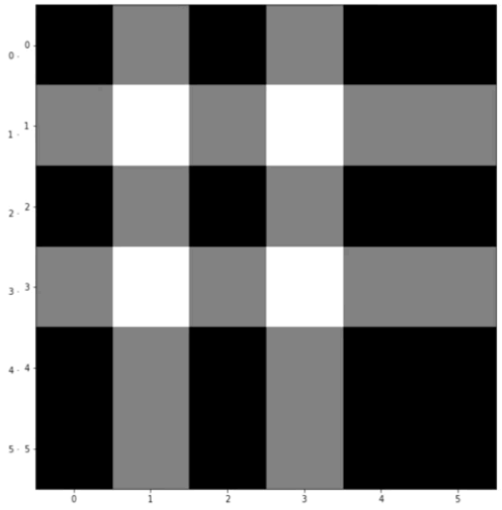


$$\|G\|$$

$$G = [G_x, G_y]$$

$$\|G\| = \sqrt{{G_x}^2 + {G_y}^2}$$



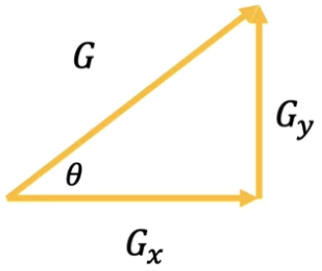


$\|G\|$

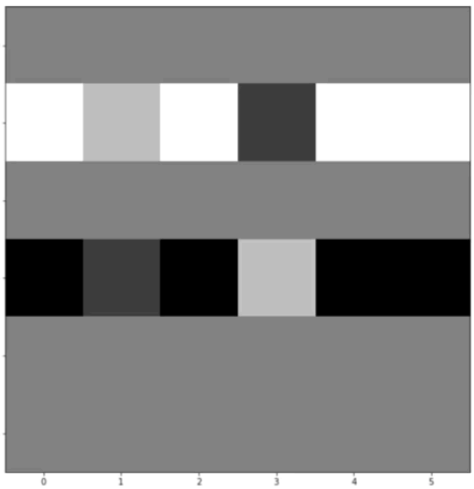
The above figure shows the image that represents the edges. Also, we can calculate the **angle** that represents the direction.

$$G = [G_x, G_y]$$

$$\|G\| = \sqrt{{G_x}^2 + {G_y}^2}$$



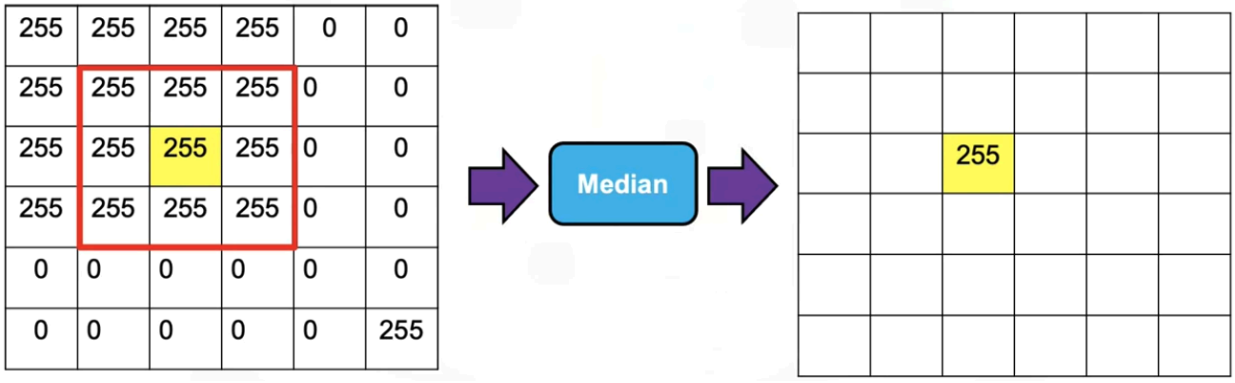
$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$



$\|G\|$

- **Median Filter:**

Median Filters are filters to **remove some type of noise** but may **distort** the image. The Median Filter **outputs the median values** of the **neighborhood pixels**.



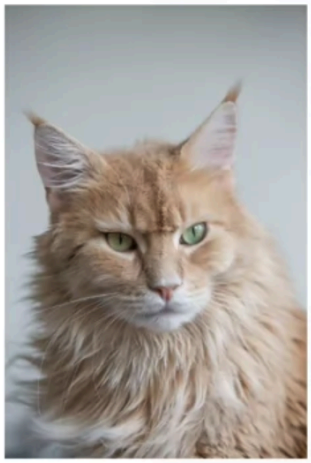
Machine Learning Image Classifications:

- **Introduction to image classifications:**

  It's the process of taking an image & getting a computer to classify it or provide the probability of the class of the image.

  - **Class** is a label for objects.

- Process of taking an image or picture



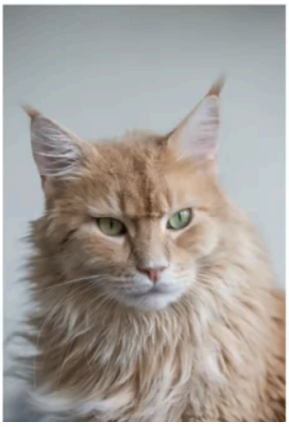
Cat

98%

We use (**y**) to represent the class of the image (Label) as shown below:

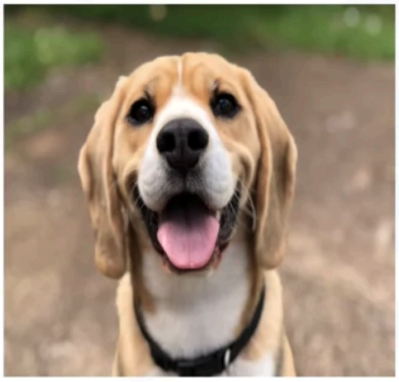
$y = 0$

$y = \text{“}cat\text{”}$



$y = 1$

$y = \text{“}dog\text{”}$



- **Computers** can understand the image by **using the intensity values** of the digital image to **classify** the image.

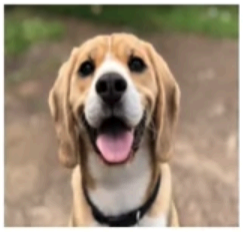
We represent (**X**) the image that we want to classify as shown below for (RGB):

$$X = \begin{bmatrix} \begin{bmatrix} 2 & 3 & .. & 40 \\ 3 & 10 & .. & 50 \\ 2 & 3 & .. & 40 \\ 5 & 34 & .. & 24 \end{bmatrix} \\ \begin{bmatrix} 3 & 10 & .. & 50 \\ 2 & 3 & .. & 40 \\ 5 & 34 & .. & 24 \\ 3 & 10 & .. & 50 \end{bmatrix} \\ \begin{bmatrix} 5 & 34 & .. & 24 \end{bmatrix} \end{bmatrix}$$

- Three Dimensions array (**Tensor**).

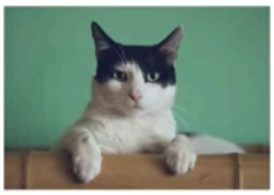
The **dataset** is a set of images (x) and labels (y):

$\mathbf{x}_1, y_1 = 1$



$\mathbf{x}_3, y_3 = 0$

$\mathbf{x}_5, y_5 = 0$

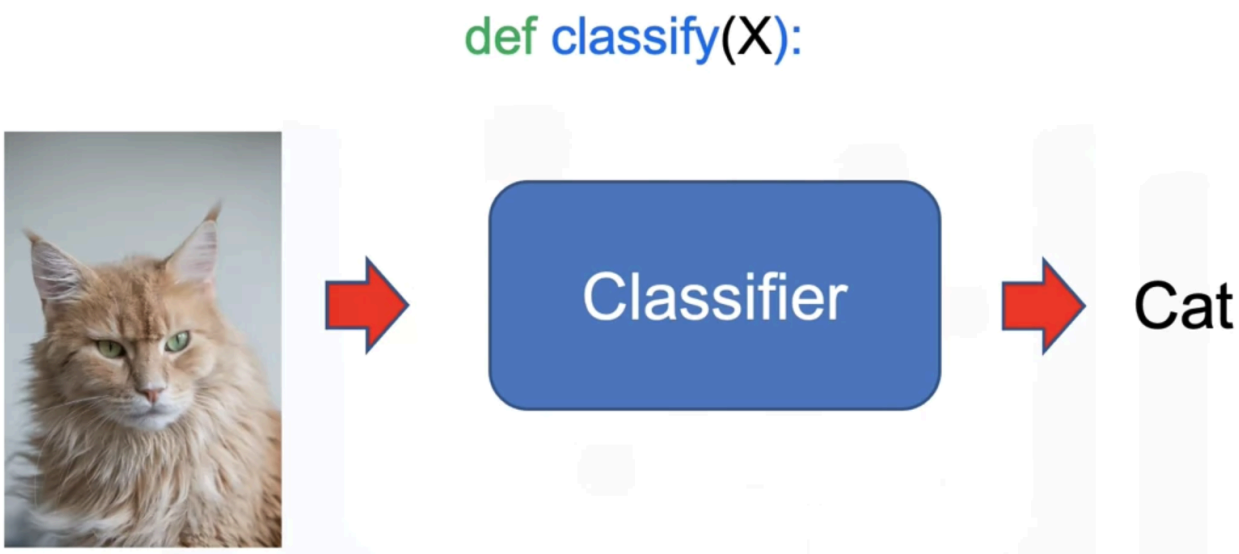


$\mathbf{x}_2, y_2 = 1$

$\mathbf{x}_4, y_4 = 1$

$\mathbf{x}_6, y_6 = 0$

The final goal is to come up with a Python function that takes the input image and output as a class



- **Image Classification Challenges:**

1. Changing in the viewpoint.
2. Change in the illumination.
3. Background Clutter.
4. Deformation.
5. Occlusion.

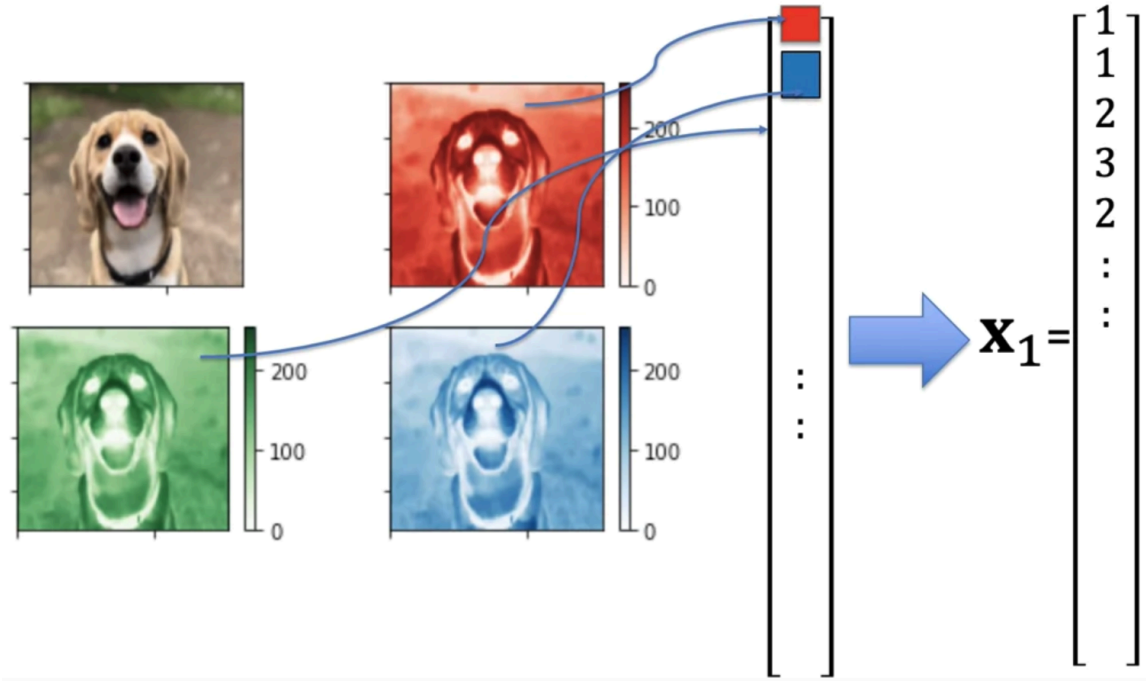
- **Methods for image classification** (**Supervised Learning**):

1. K-Nearest Neighbors.
2. Features Extractions.
3. Linear Classifier.

● Image Classification with KNN:

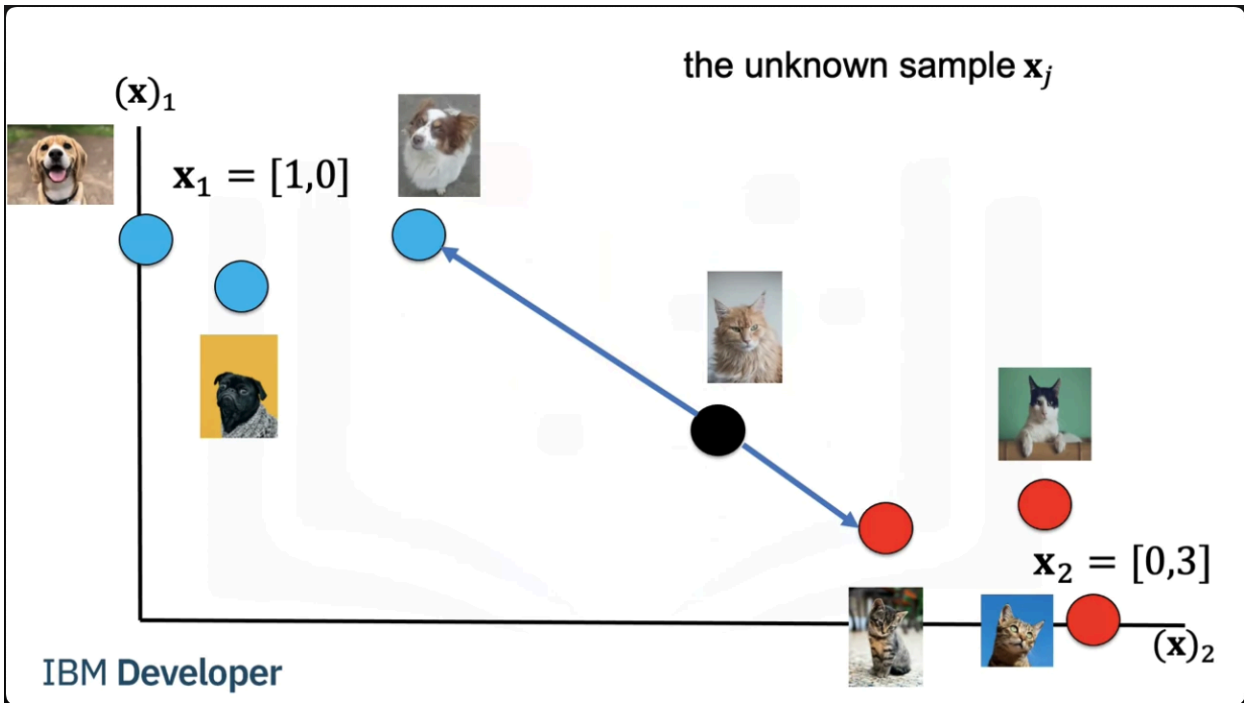
KNN is the **simplest classification algorithm**. KNN classified the unknown data points by **finding the most common** classes in the **k-nearest neighbor example**. (**KNN**) is a **supervised learning algorithm.** This means it **relies** on **labeled training data** to make predictions.

For **RGB**, we will concatenate the three-channel images to a vector, or for a **grayscale** image to (**X**) input as shown below:



For example (Two-Dimensions):

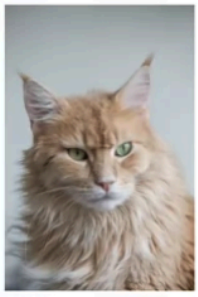
Red Dots: Label 1 **(Training Samples).**
Blue Dots: Label 2 **(Training Samples).**
Black Dot: Unknown Sample.



Then, we **calculate the distance** between the **unknown image and each image in the dataset**. We calculate the distance between two images using **Euclidean Distance**.

$$d'(\mathbf{x}_1,\mathbf{x}_2) = \|\mathbf{x}_1 - \mathbf{x}_2\|$$

n	y	$d(\mathbf{x}_n, \mathbf{x}_j)$
1 		1.0
2 		1.5
3 		1.7
4 		2.0
5 		2.5
6 		3.0


the unknown sample

SKILLS NETW

(**y**) the **labels**, (**n**) the **training images**, and (**d**) the **measured distance** between the unknown sample and each image.

The closet (nearest) value, is the label for the unknown image.

- The new sample is assigned to the label as ($\hat{y}$), (y-hat) where this is a prediction (guess)

n	y	$d(\mathbf{x}_n, \mathbf{x}_j)$
1 		1.0
2 		1.5

$\hat{y} =$ 

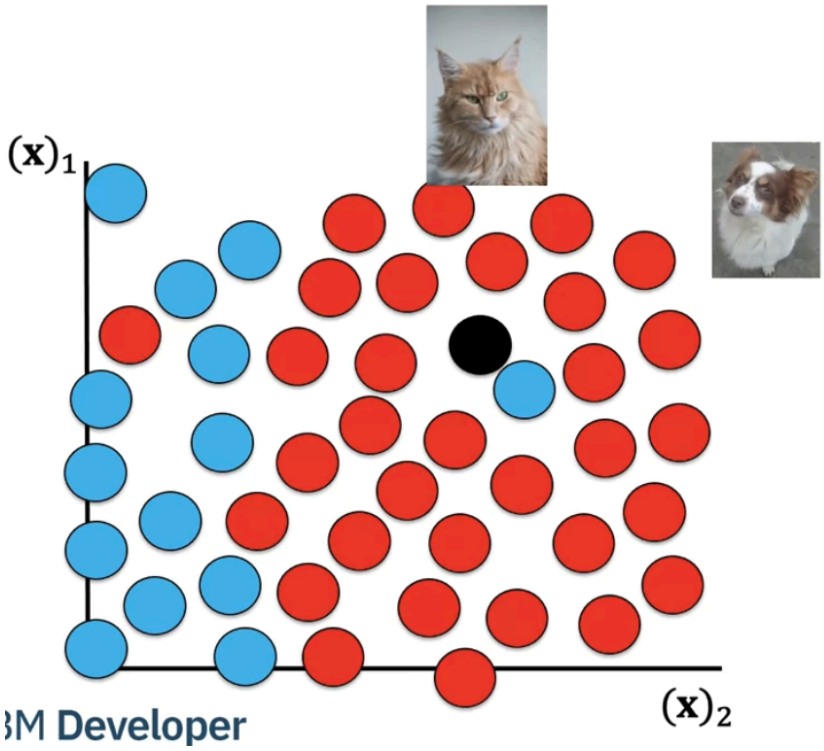
To know that our Model is working, we need to **split** our dataset as training and testing data, where a large portion is used for training. The test data will **evaluate the model** performance.

The **model accuracy** is the **number** of samples that have been **predicted correctly** divided by the total **number of samples**. (**1**) is correct prediction and (**0**) is incorrect prediction. (**Compare the testing data prediction with the actual label**)

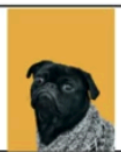
Samples: n	1	2	3	4
y	1	0	1	0
$\hat{y}$	1	0	0	1
$Correct$	1	1	0	0

$$= \frac{1}{4}(1+1+0+0) = 0.5$$

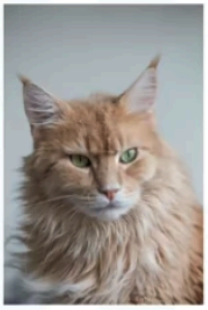
Finding the nearest samples does not always work best, as below example:



The unknown sample is the Black-dot (Cat) is next to a Blue-dot (Dogs), **but all the other samples next to are Red-dots (Cat**). In this case, we **Find (K) nearest samples (The number of samples to take that are near to the unknown sample)**, then we **perform majority vote**, where we assign the **unknown sample with the most number** of samples.

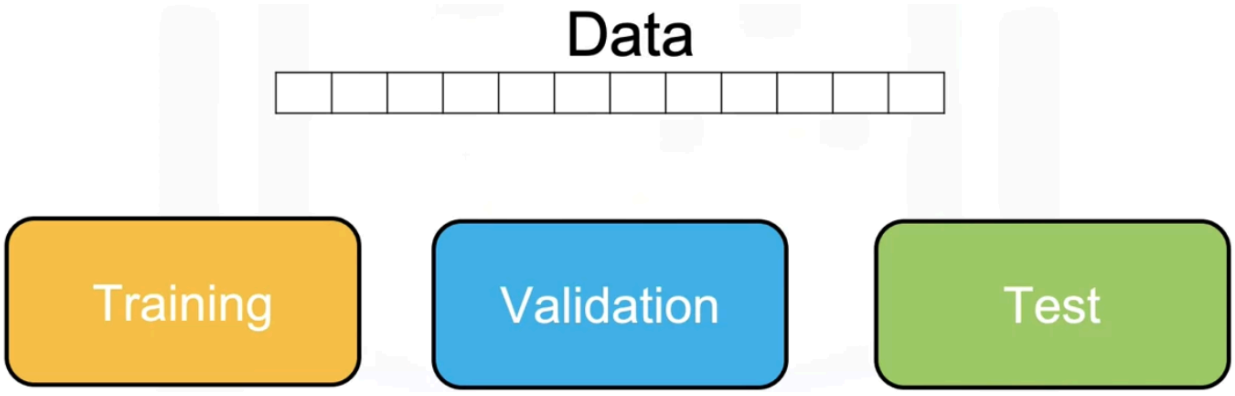
n	y	$d(\mathbf{x}_n, \mathbf{x}_j)$
1 		1.0
2 		1.5
3 		1.7
4 		2.0
5 		2.5

$\hat{y} =$ 

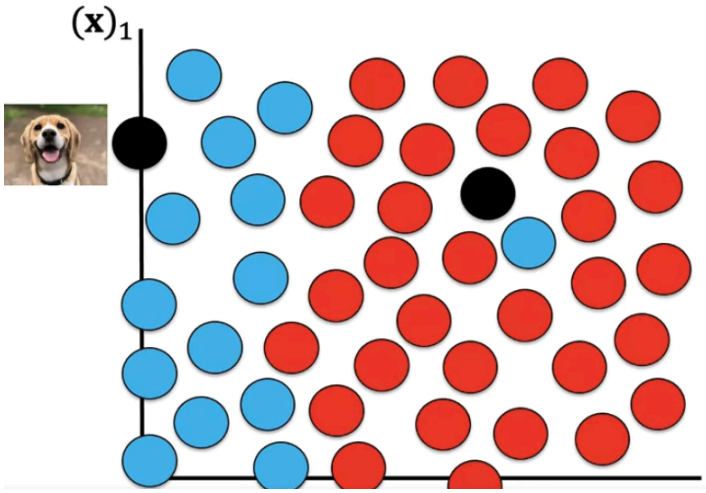


The above example is taking (K=3), where we t**ake the first (3) nearest samples distance**, then the **assign the most number (Red)** to the unlnown sample.

To **determine** the best (K) value, we **use a validation** data in the model dataset, and this called [**Hyper Parameter**].



We select Hyper Parameter (K) in the validation dataset to maximizes the accuracy. The figure below shows the process:



Validation Data

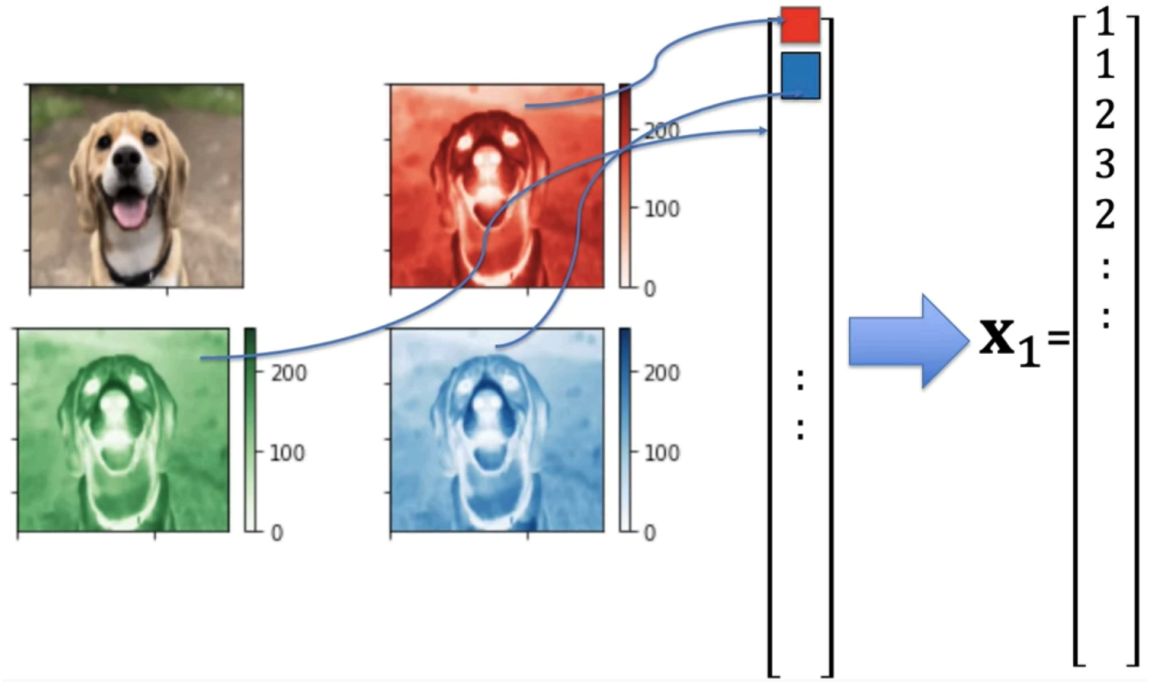
$\hat{y}$ \ y			Accuracy
K=1			50%
K=3			100%

KNN is not used parctice, **KNN is extremely slow**

- ## Linear Classifier:

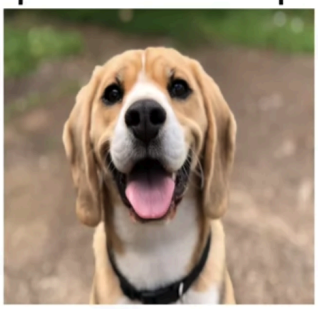
Linear classifiers are widely used in classification, and they are the building blocks for more advanced classification methods.

The first thing for an **RGB image**, we will concatenate the three-channel images to a vector, or for a **grayscale** image to ($\mathbf{X}$) input as shown below:



Then, a **simple function** that can take an **image as an input** and **output the image class**.

$$\hat{y} = f(\overbrace{\quad\quad}^{\mathbf{x}}) = 1$$



The equation of a **line in one dimension** is given by the following:

$$z = wx + b$$

Where:

w - Weight Term.
b - Bias Term.

These are **Learnable Parameters**, and we will use them to classify the image.

The above equation can be **generalized** to a hyperplane (Arbitrary Dimensions):

$$z = w_1 x_1 + .. + w_d x_d + b$$

Hyperplane:

In an (n)-dimensional space, a hyperplane is an ((n-1))-dimensional subspace. For example: • In a 2-dimensional space (a plane), a hyperplane is a line. • In a 3-dimensional space, a hyperplane is an ordinary plane. • In general, in an (n)-dimensional space, a hyperplane is defined by a linear equation involving (n) variables.

We can represent the above equation as a **dot product** of **row vector [w]** and **image [x]**:

$$z = \mathbf{w}\mathbf{x} + b$$

This **(Z)** equation is called the **Decision Plane**.

- **Dot Product:**

$$C = A \cdot B = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

- **Calculate (c_{11}):**

$$c_{11} = (1 \times 5) + (2 \times 7) = 5 + 14 = 19$$

- **Calculate (c_{12}):**

$$c_{12} = (1 \times 6) + (2 \times 8) = 6 + 16 = 22$$

- **Calculate (c_{21}):**

$$c_{21} = (3 \times 5) + (4 \times 7) = 15 + 28 = 43$$

- **Calculate (c_{22}):**

$$c_{22} = (3 \times 6) + (4 \times 8) = 18 + 32 = 50$$

Resulting Matrix

So the resulting matrix C is:

$$C = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

Summary

The product of the matrices (A) and (B) is:

$$\begin{pmatrix} 1 & 2 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ \end{pmatrix}$$

- **Ensure the number of columns in the matrix (A) is equal to the number of rows in the matrix (B).**

Two dimensions line equation:

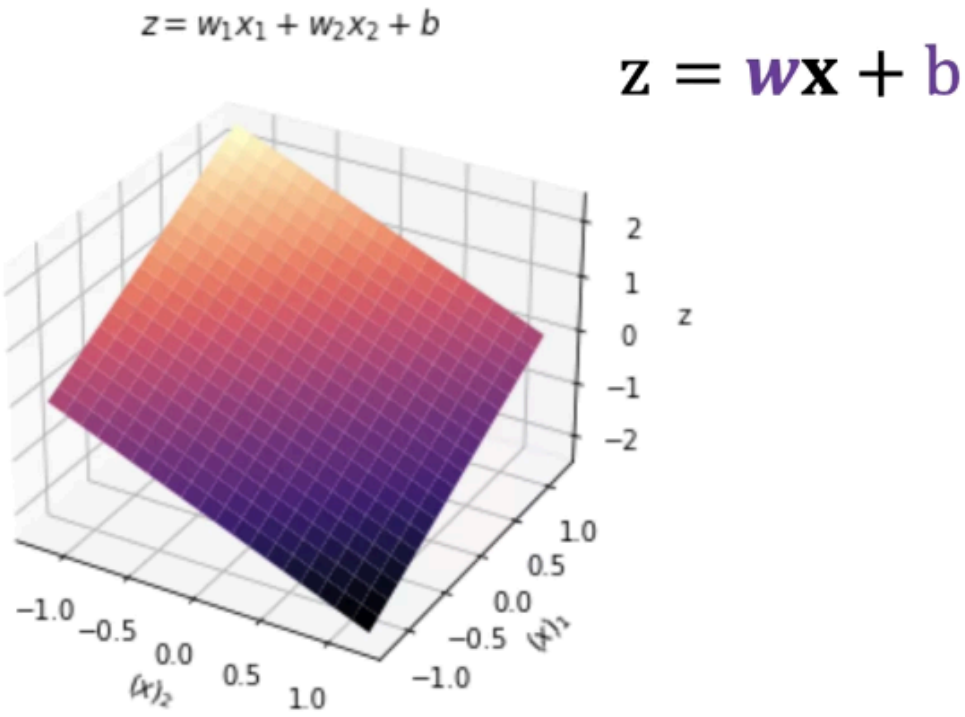
$$z = \boldsymbol{w}\boldsymbol{x} + \mathrm{b}$$

$$z = w_1 x_1 + w_2 x_2 + \mathrm{b}$$

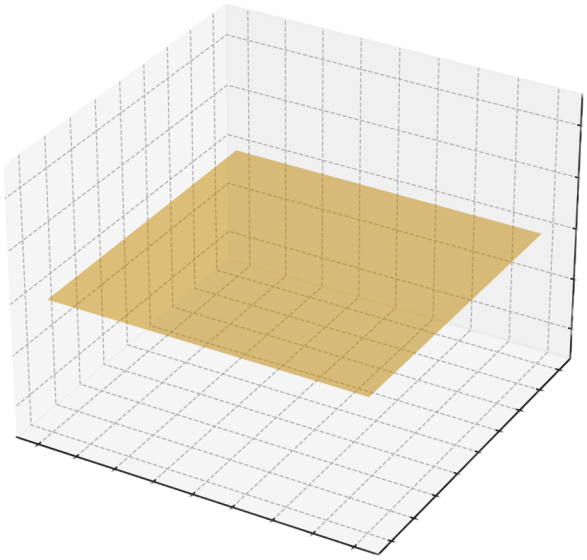
If we use **w1=1, w2=-1, and b=1 (Decision Plane):**

$$z = 1x_1 - 1x_2 + 1$$

The **plot as a plane (Decision Plane)**:



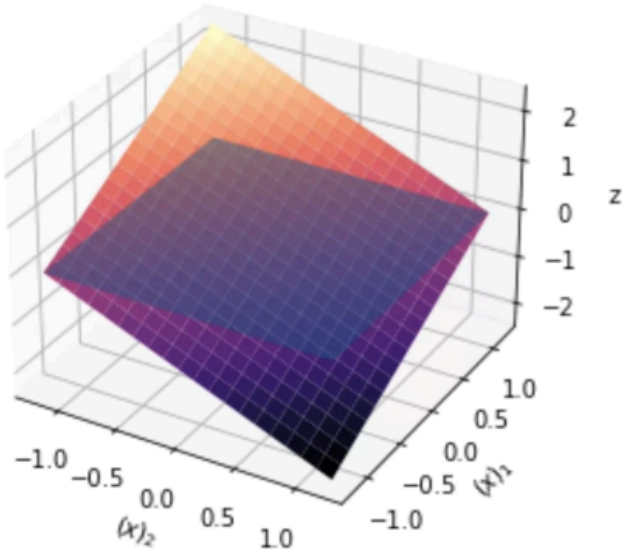
We can plot the plan where **z=o** as shown below:



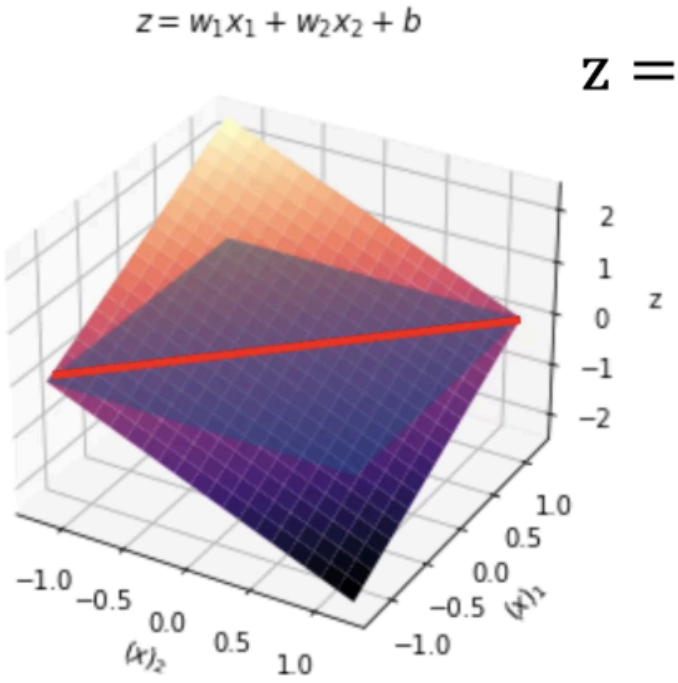
Merge these two planes:

$$z = w_1 x_1 + w_2 x_2 + b$$

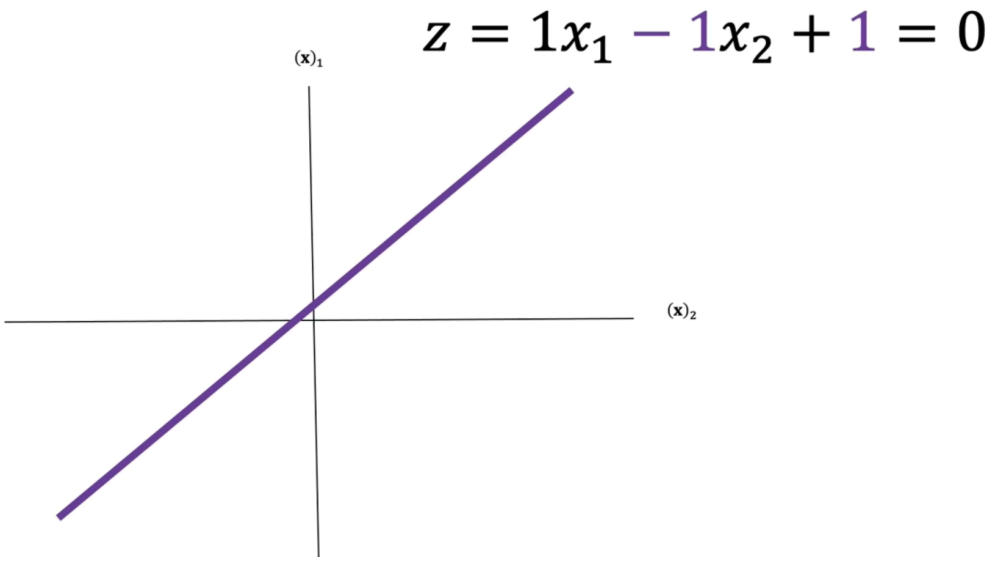
$$\mathbf{z} = \boldsymbol{w}\mathbf{x} + \mathbf{b}$$



We can see the **line where the two planes are intersect**:

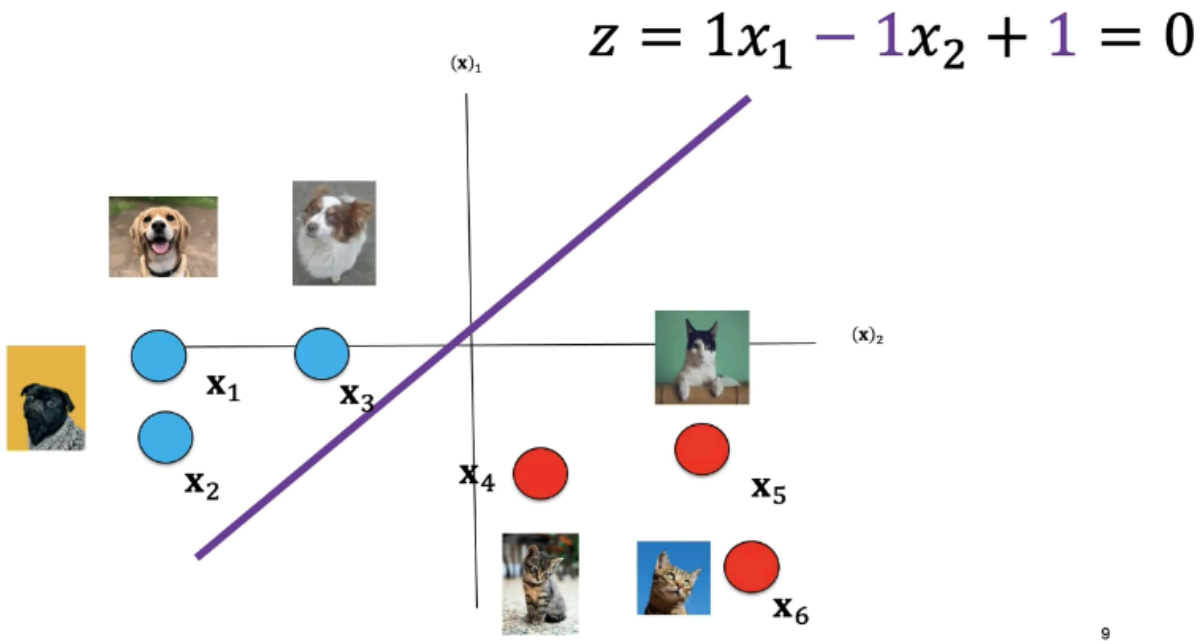


If we look at the plot above from above where the plane [z=0] **with the intersected line**:



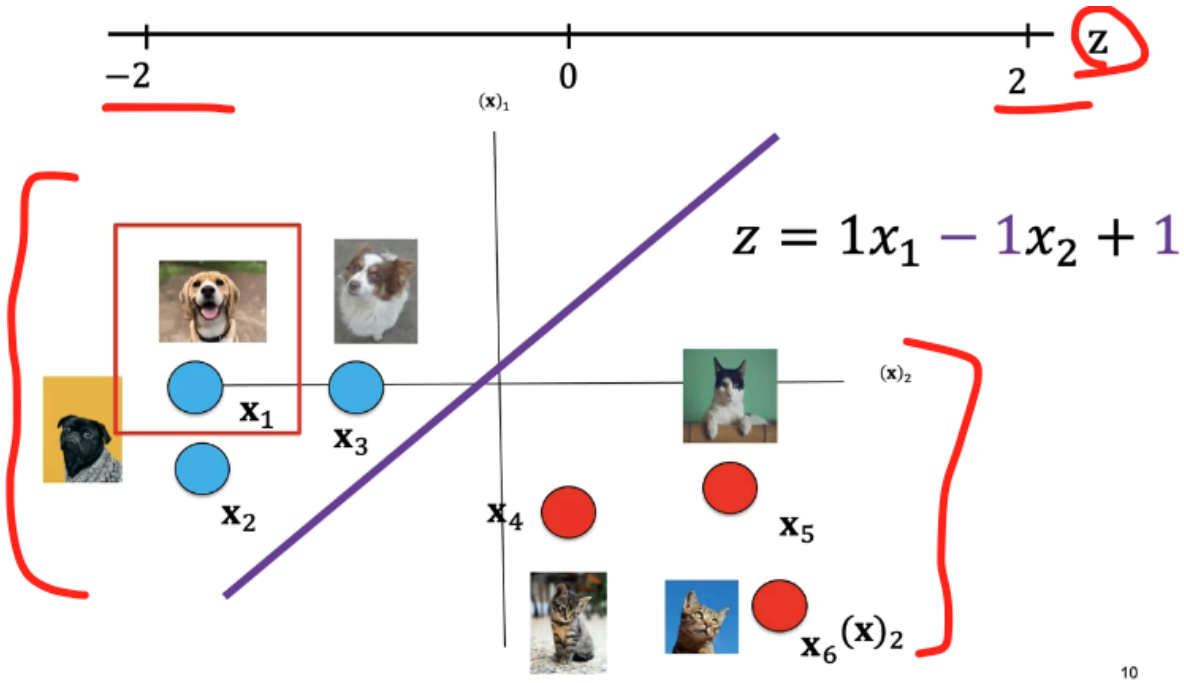
This line is called (**Dicision Boundry**).

We can overlay our sample images

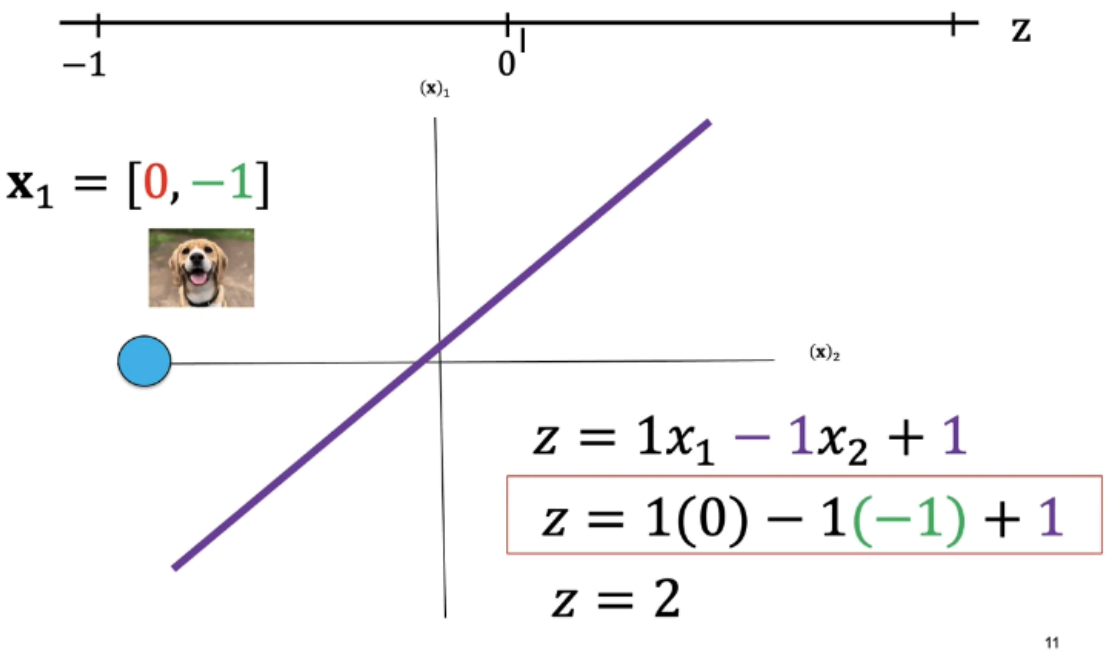


(Doges **Samples on the left**, and Cats **samples on the right**)

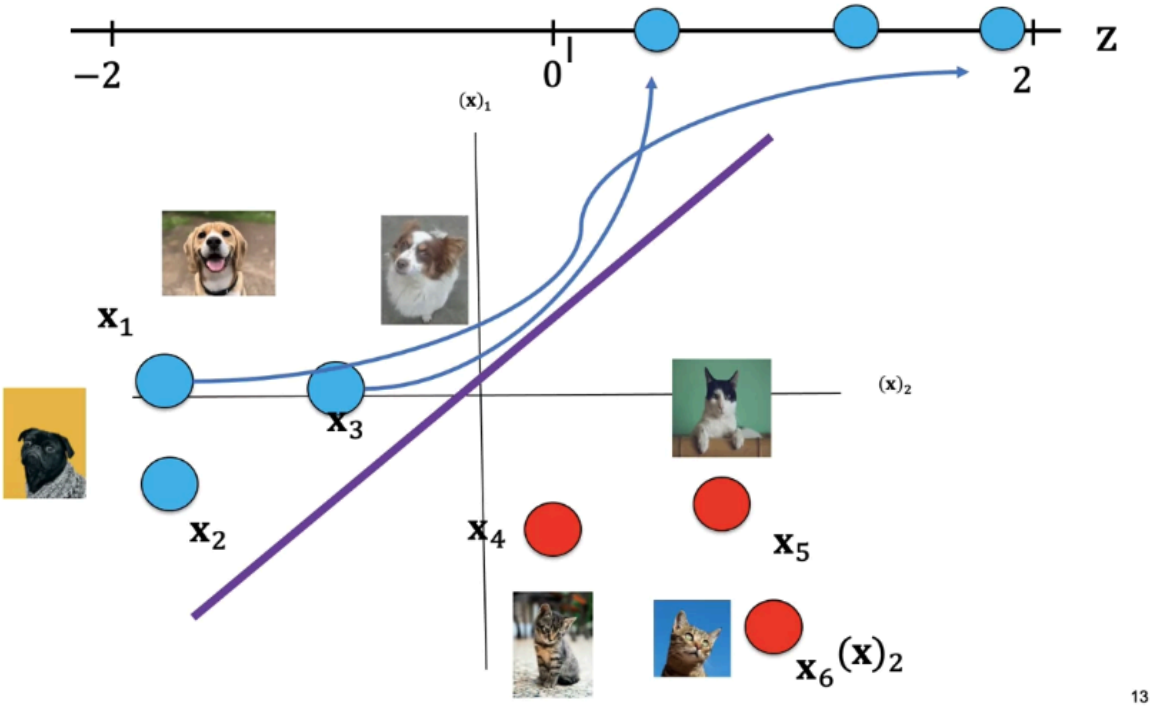
We can use the value of (z) to **determine the classification**,



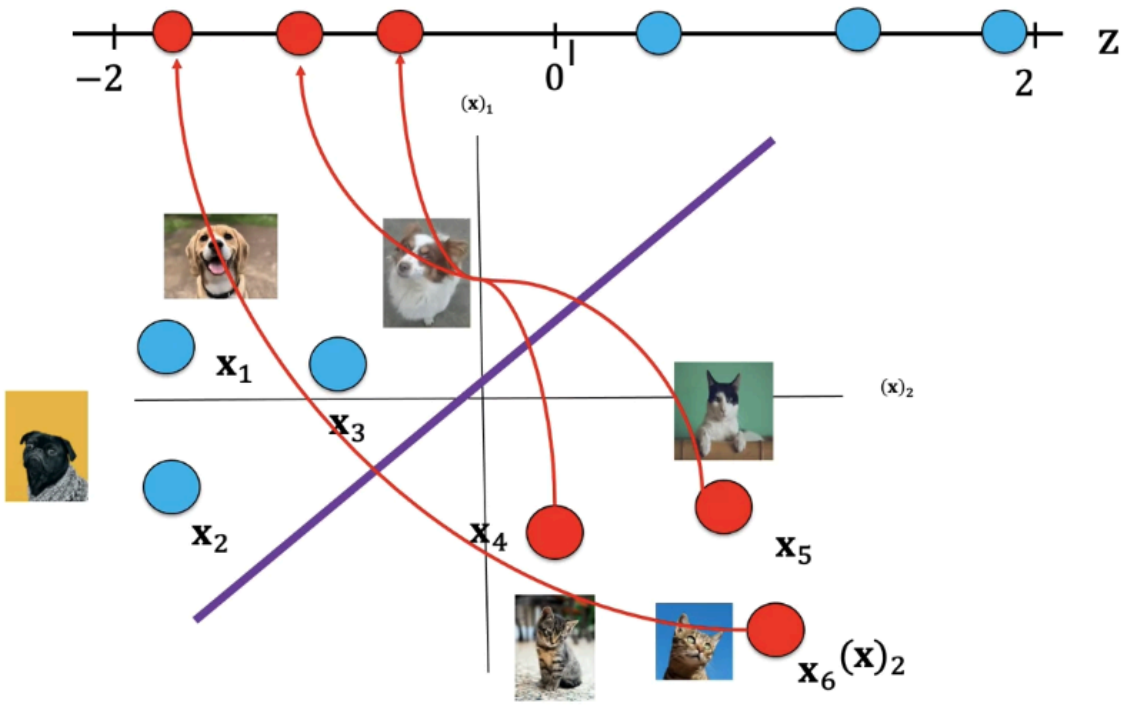
If we take (x1) as a sample:



And, every (**Dogs**) on the right has a (**Positive**) Values:



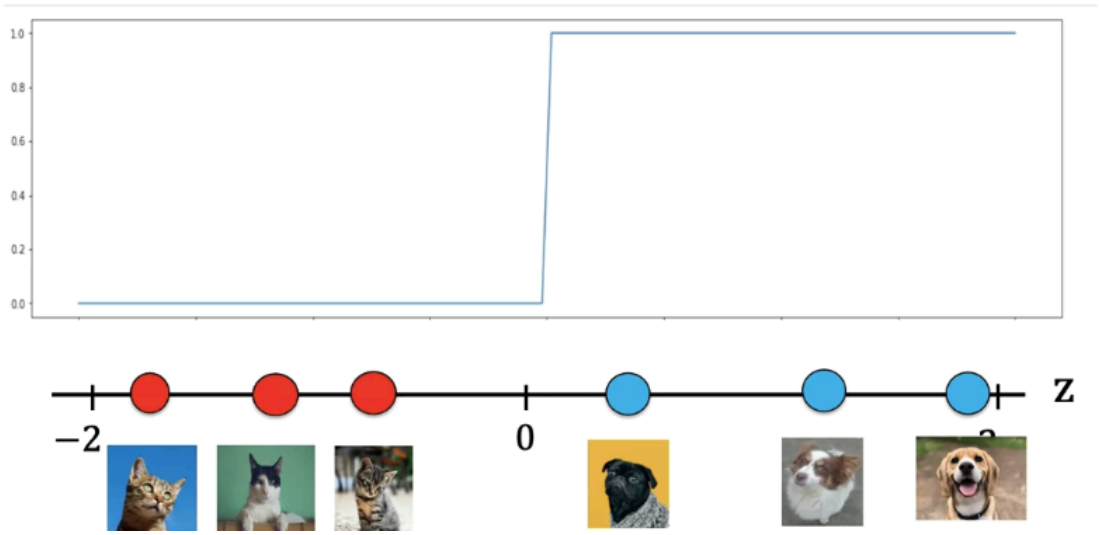
And, every (**Cat**) on the right has a (**Negative**) Value:



15

If we use (z) to **calculate the class** of points, it will return real numbers **(Positive) & (Negative).**

$$\hat{y} = f(z)$$

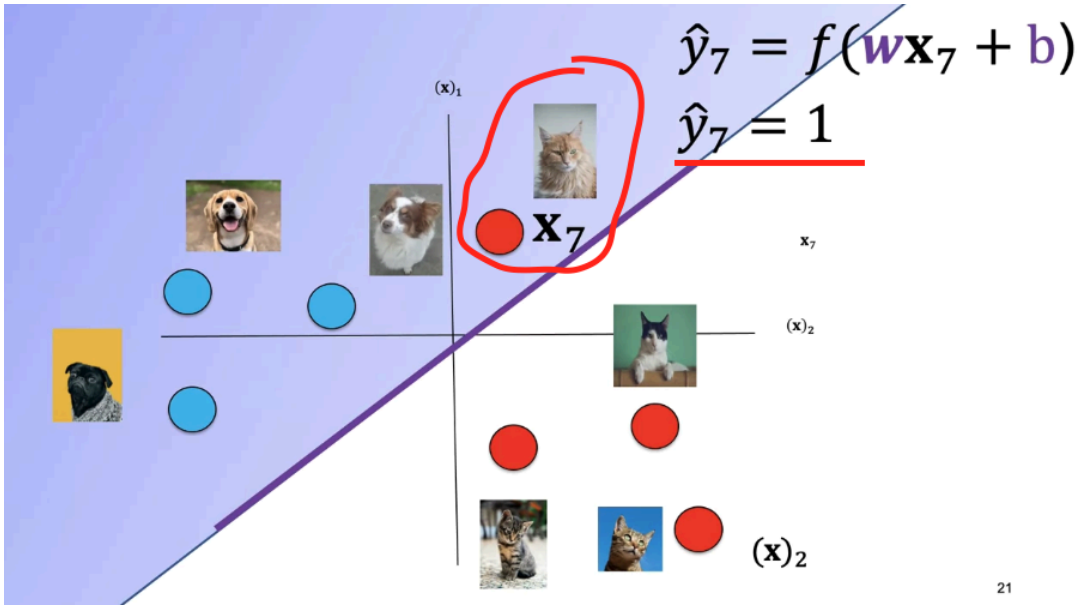


16

The (Z) **values** are (Negative & Positive), **but we need a class between (0) & (1)**.

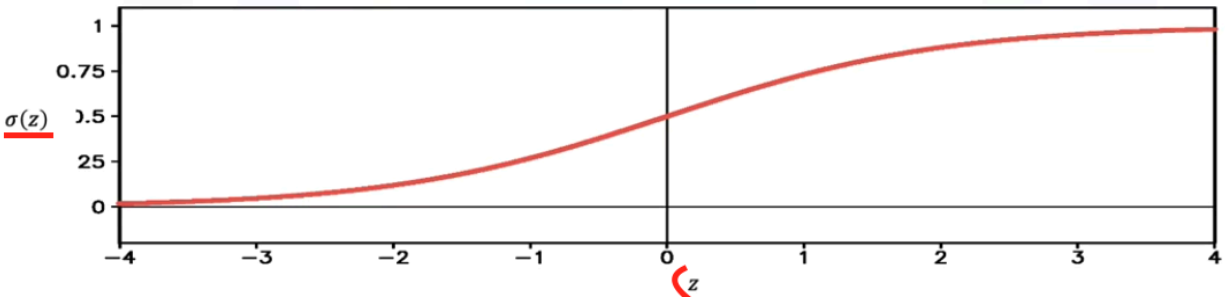
So, we **need to convert these numbers** by using [**Threshold Function**] as shown in the figure above. When **(Z) is greater than (0)**, it will **return (1)**, and when **(Z) is less than (0)**, it will **return (0)**.

A **plane can't always separate the data**, the sample is misclassified, and this **data is not linearly separatable**:



The **[Logistics Regression Function]** resembles the **[Threshold Function]:**

The **logistic function** is also known as the **sigmoid function.** The **logistic function** is a fundamental concept in logistic regression, a type of **statistical model used for binary classification**. **Logistic regression** uses the logistic function to model the **probability of a binary outcome** (0 or 1).

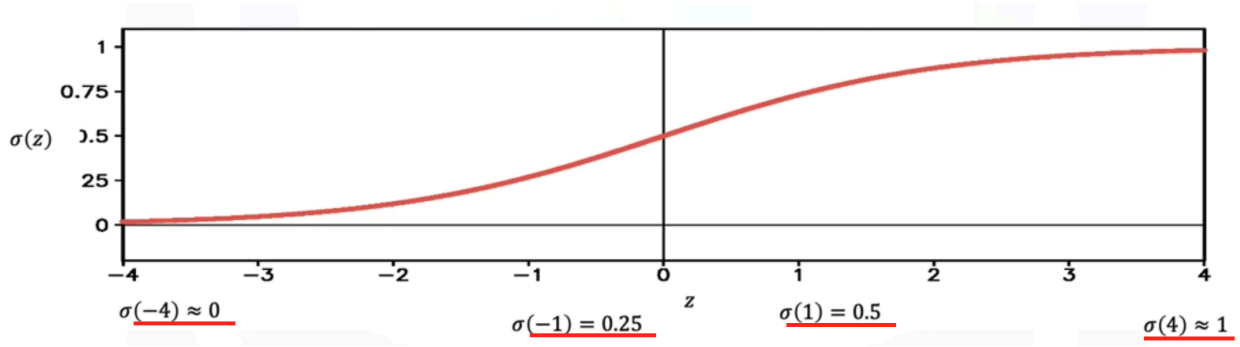


The above function is a Logistics Regression

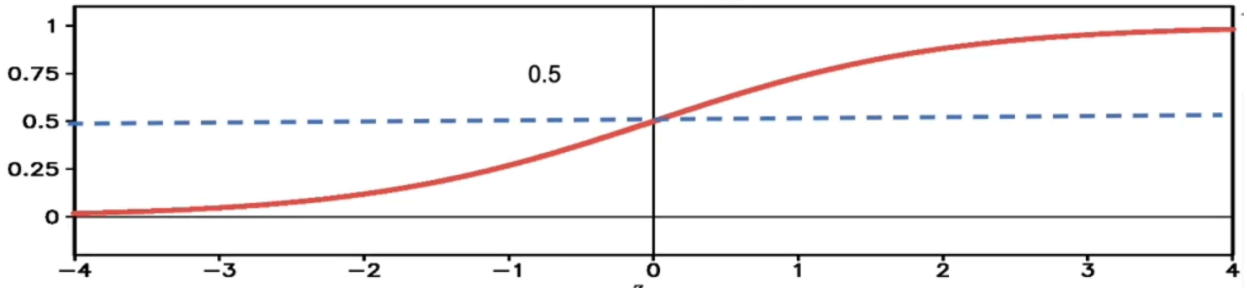
The **[Sigmoid Function]** gives us the **probability** of how likely our estimate is.

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

It has **better performance** than **Threshold Function. (e = 2.72)**

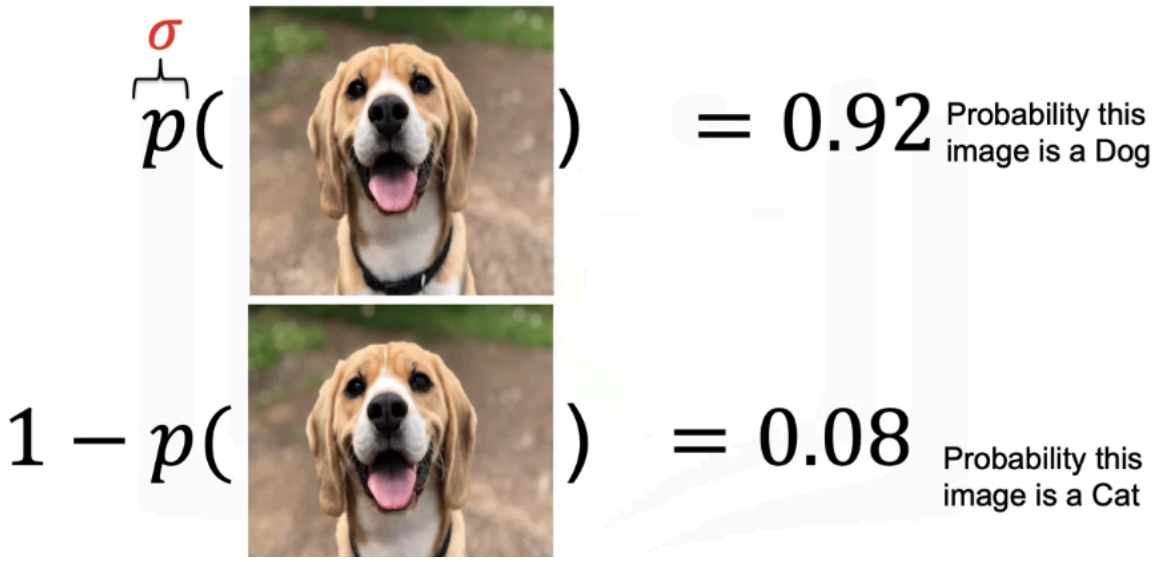


To determine ($\hat{y}$) as a class, we use a threshold as shown below by the line:



If the output of the Logistic Function value is larger than (0.5), we set the prediction value to **(1)**, and if the **value is less than (0.5)**, we set the prediction **value to (0).**

We can represent the **Logistic Function** as a **probability**:



- ## Logistic Regression Training: Gradient Descent:

In this section, we will discuss how to determine the **plane**. Training is where to find the best (**Learnable Parameters**) (**w & b**) of the **decision boundary**. The decision boundary is to **separate the dataset samples**.

- To determine how our learnable parameters are good, we can use (**Cost & Lost Functions**).

The Lost function tells us how **good the classifier prediction** is, if the prediction is correct for a sample, the **loss is (0)**, and if the prediction is incorrect, the **loss is (1)**.

$loss(y, \hat{y})$	y	$\hat{y}$
0	1	1
0	0	0
1	1	0
1	0	1

The **Cost Function** (**Classification Error**) tells us **how good our learnable parameters** on the dataset, and it can be calculated by the **sum of the loss**:

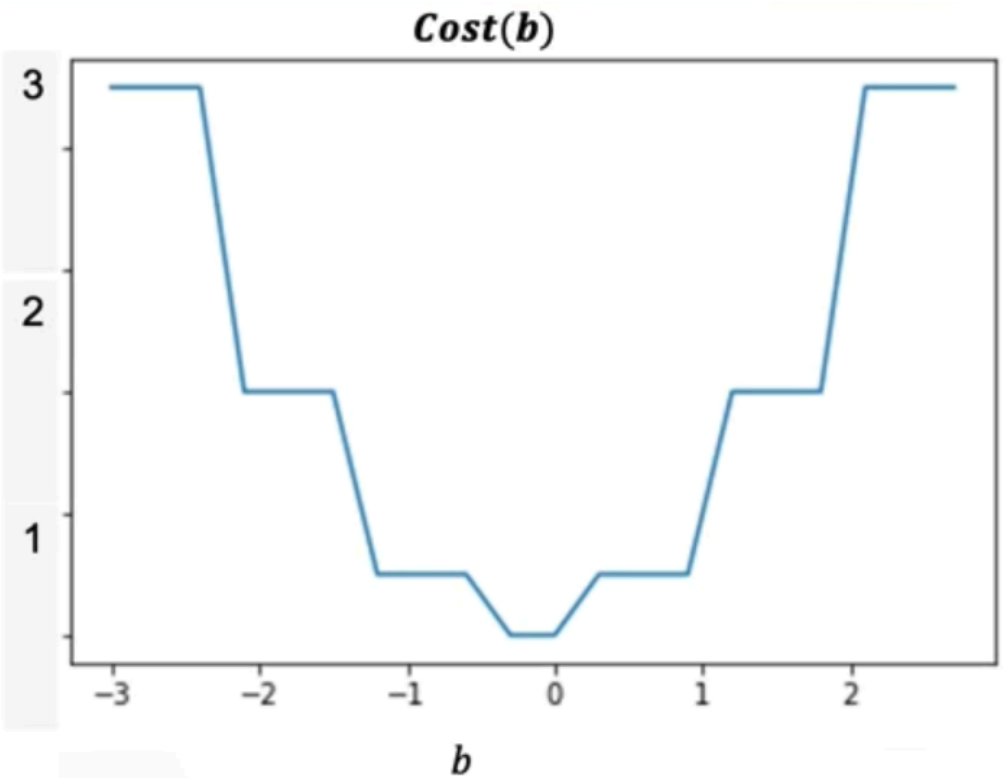
$$Cost = loss(y_0, \hat{y}_0) + loss(y_1, \hat{y}_1) \ ..$$

The Cost Function is a **function of the learnable parameters**, where we set the learnable parameters, and check the decision boundary classification (Cost).

iteration	0	1	2
Cost	3	2	0

$$(w^0, b^0)\ (w^1, b^1)\ (w^2, b^2)$$

To simplify, we will take the cost function as a **function of the parameter (b)** - [Bias]:

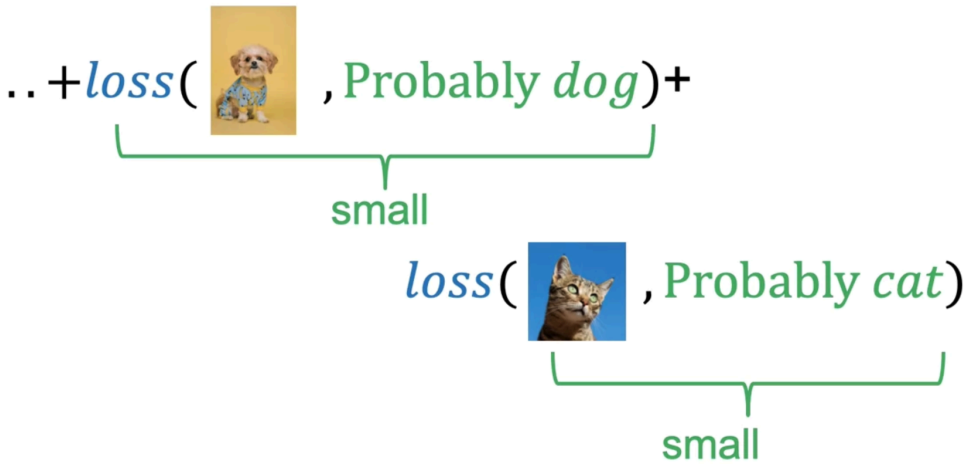


In reality, the **Cost Function** is a function of **multiple parameters (w & b). [Plot of threshold]**

Classification Error is difficult to work with. So, we use **Cross Entropy Loss** that uses the output of the **Logistic Function (σ) [Probability]** insisted of (**ŷ**).

$$Cost(w,b) = loss\big(y_1, \underbrace{\sigma(w\mathbf{x}_1 + \mathrm{b})}_{\text{probability}}\big) + loss\big(y_2, \underbrace{\sigma(w\mathbf{x}_2 + \mathrm{b})}_{\text{probability}}\big) + ..$$

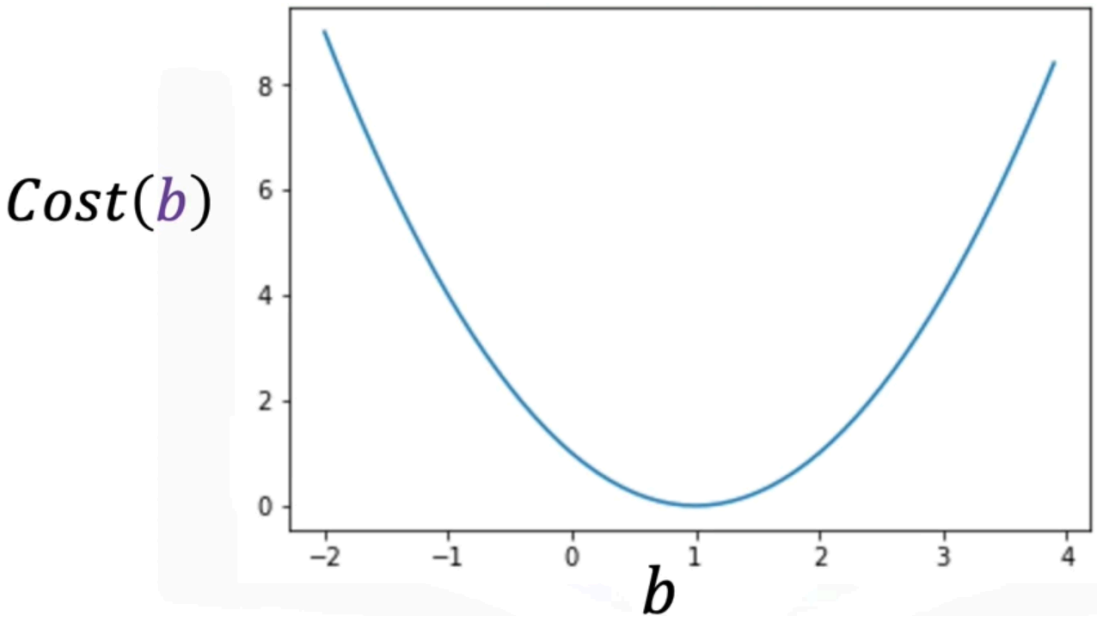
The cost is still the sum of the loss, the **cross entropy** deals with **how likely the image belongs to a specific class**. If the is probability incorrect, the **cross entropy is large**, and if the probability is correct, the **cross entropy is small, but not (0)**.



- **Gradient Descent:**

A method to find the **best learnable parameters (Minimum The Cost Function)**, the **gradient** gives us the **slope of a function at any point**.

The plot below shows the **cost using Cost Entropy**:



This plot is smoother than the plot for **the cost using Classification Error.**

If we find the minimum of the cost, we **can find the best parameters**. We can use a gradient to find the slop in each point until reaches the minimum slope (Close to Zero).

Gradient Descent is an optimization **algorithm** used to minimize the cost function in various machine learning algorithms. It works by iteratively moving towards the minimum of the cost function.

Gradient Descent Equation:

$$b^i = b^{i-1} - \eta slope(b^{i-1})$$

η: learning rate

η: 0.01

Usually, it's a small number.

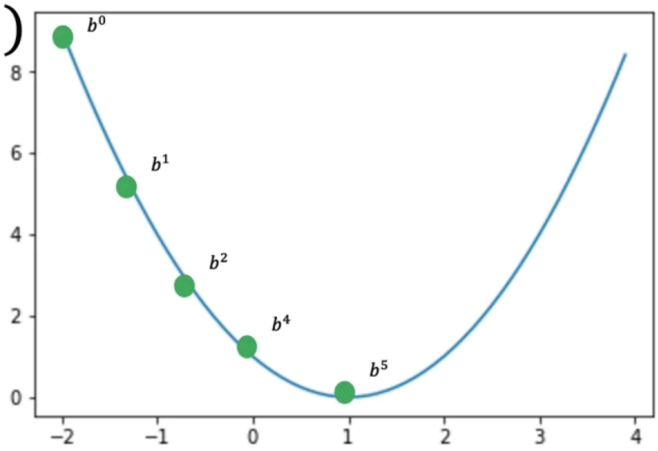
$$b^i = b^{i-1} - \eta slope(b^{i-1})$$

η: learning rate

η: 0.01

$$b^5 = b^5 - \eta 0$$

$$b^5 = b^5$$



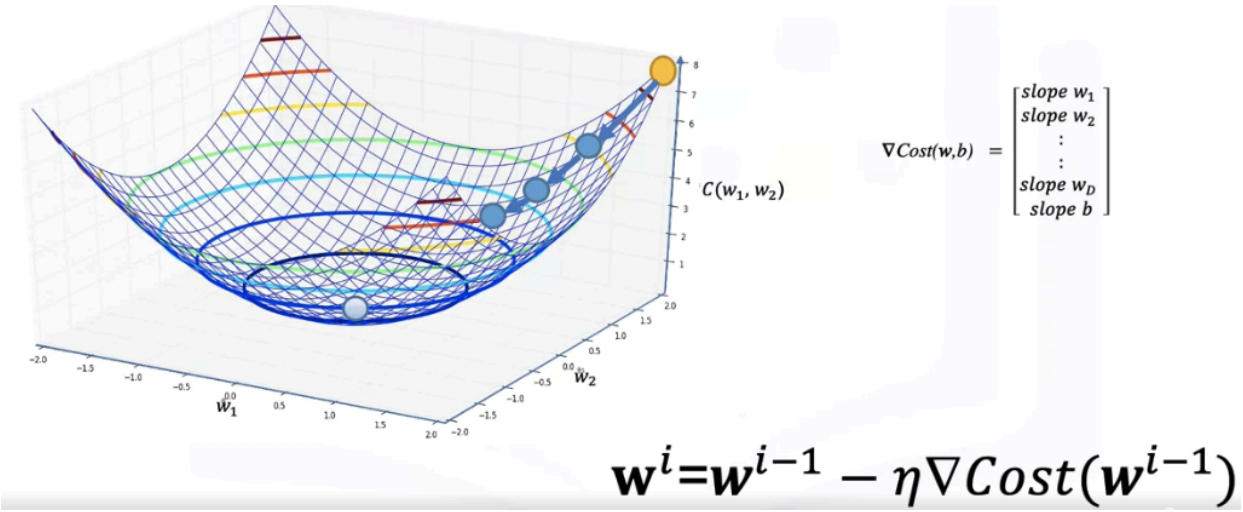
When we reach the minimum cost, the **gradient (Slope) is zero**. If we use too low (Learning Rate), we may **never reach a minimum**, and if we use too large (Learning Rate), we may **never reach a minimum.**

The learning rate is a **hyperparameter** (Hyperparameters are parameters that are set before the learning process begins and control the behavior of the learning algorithm).

We **select** the learning rate by finding a value that has the **best accuracy using validation data**.

- **Using gradient descent to minimize the cost:**

The decision plan has multiple parameters, as a result, the gradient is a vector



- **<u>Mini-Batch Gradient Descent:</u>**

<u>Mini-Batch Gradient Descent</u> allows me to **<u>train models with more data</u>**. **<u>[Batch Gradient Descent]</u>** uses **<u>all the samples</u>** in the dataset to find the minimum cost, whereas **<u>[Mini-Batch Gradient Descent]</u>** uses **<u>a few samples</u>** at a time in the dataset for iteration. All <u>the dataset samples</u> are called [**Epoch**].

Stochastic Gradient Descent is the gradient descent that will be performed for each sample.

$$\textbf{Iterations=} \frac{\text{training size}}{\text{batch size}} = \frac{6}{1} = 6$$

Batch Size
1

x_1, y_1	x_2, y_2	x_3, y_3	x_4, y_4	x_5, y_5	x_6, y_6
1	2	3	4	5	6

Iterations
6

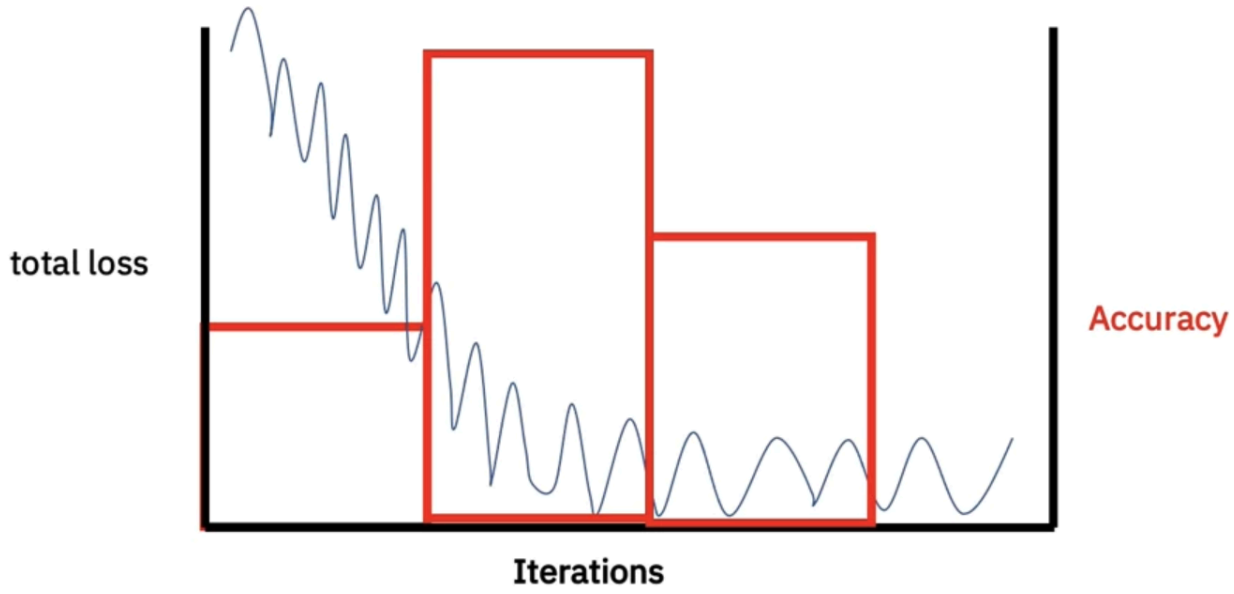
$$\text{Iterations} = \frac{\text{training size}}{\text{batch size}} = \frac{6}{2} = 3$$

Batch Size	x_1, y_1	x_2, y_2	x_3, y_3	x_4, y_4	x_5, y_5	x_6, y_6	Iterations
1	1		2		3		6
2							3

$$\text{Iterations} = \frac{\text{training size}}{\text{batch size}} = \frac{6}{3} = 2$$

Batch Size	x_1, y_1	x_2, y_2	x_3, y_3	x_4, y_4	x_5, y_5	x_6, y_6	Iterations
1	1			2			6
2							3
3							2

We calculate the total loss for **each iteration**, and at the end of each epoch, we **calculate the accuracy**.



- **SoftMax and Multi-Class Classification:**

Argmax function, it's a function that **returns the index** of the **largest value** in a sequence of numbers.

i	0	1	2	3	4	5	6	7	8	9

$$z = [100 \quad 0.2 \quad 0 \quad 2 \quad 3 \quad 2 \quad 0 \quad 3 \quad 0 \quad 2]$$

$$\hat{y} = argmax_i\{z_i\}$$

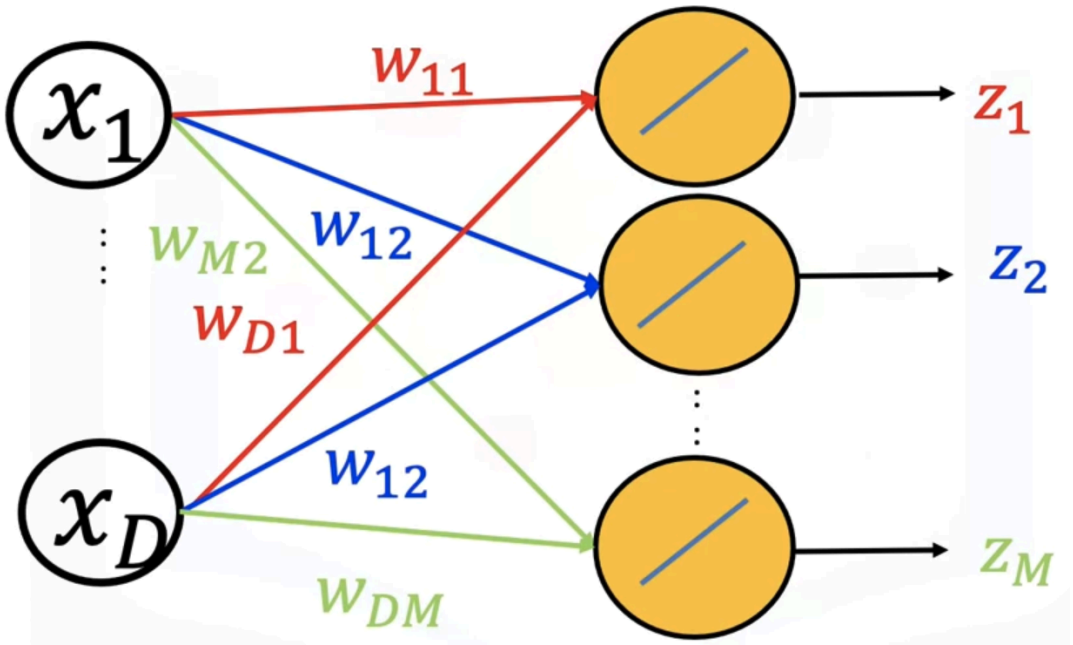
Logistic Regression can be used **for two classes**, if we are using more than (2) classes, we can use **multi-plane to classify the data**. We will use **one plane for each class**.

$$z_0 = w_0 x + b_0$$

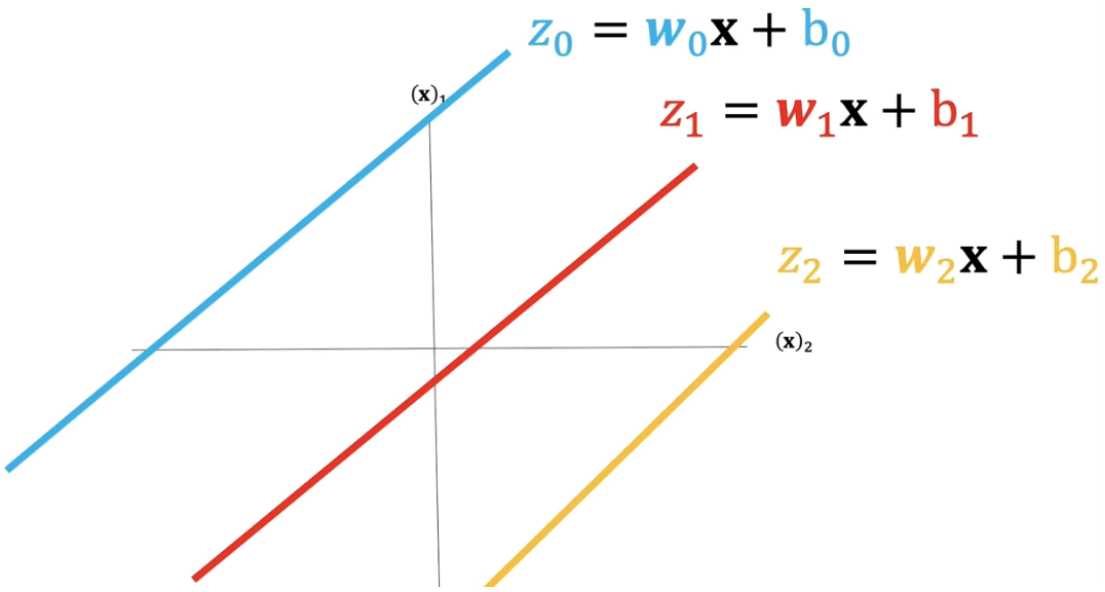
$$z_1 = w_1 x + b_1$$

$$z_2 = w_2 x + b_2$$

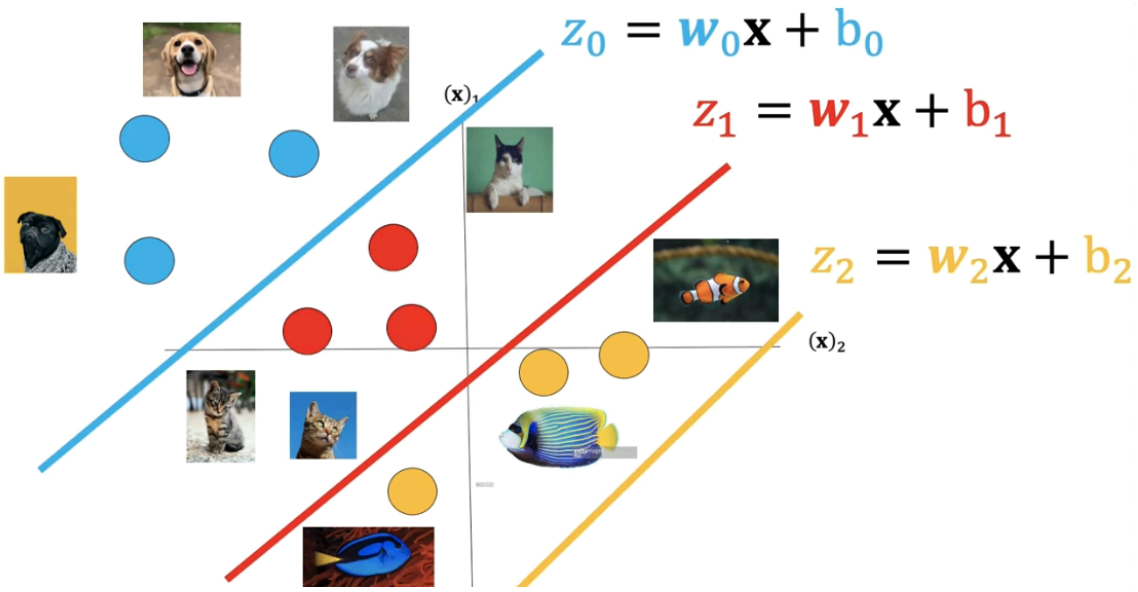
The above equations are for 3 classes. We can use **graph to represent any equations (Nodes)**.



We can see the **decision planes** intersect with z=0:

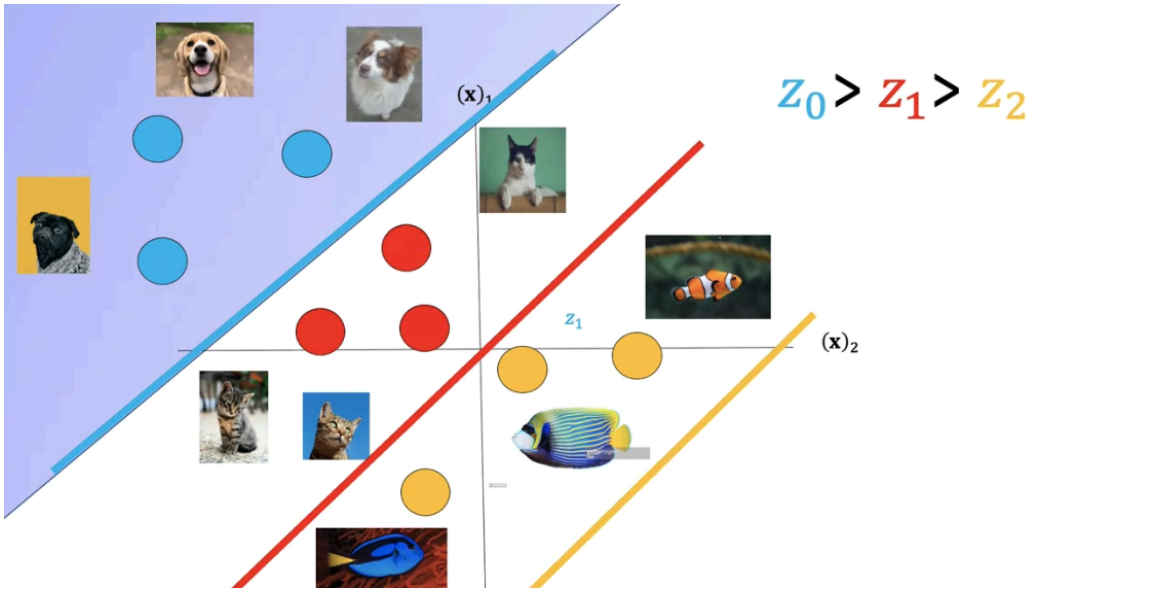


We can overlay the sample images:

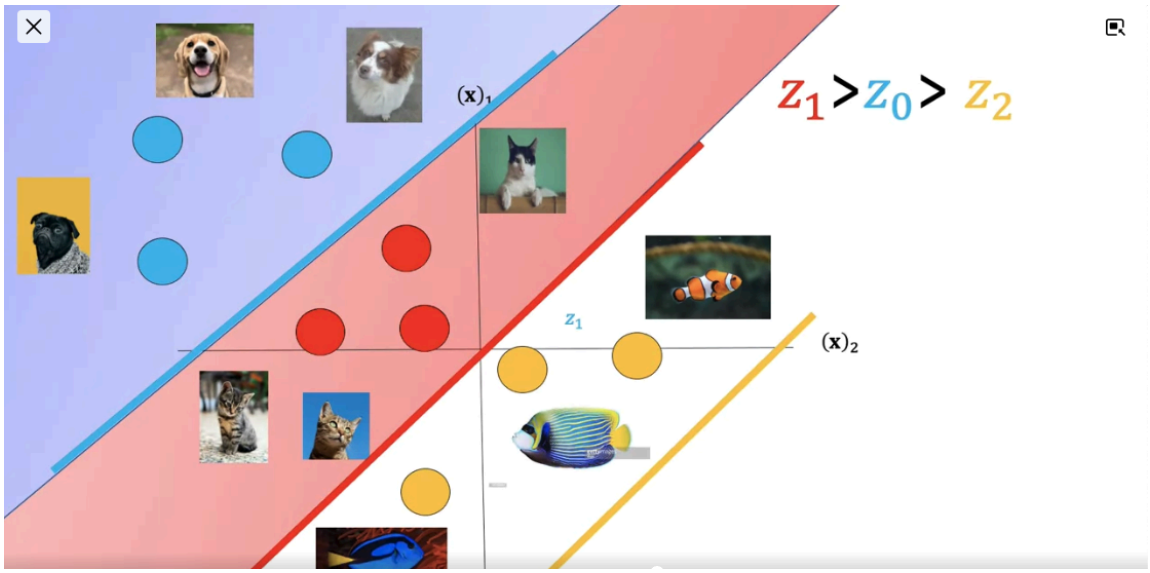


The **decision boundaries split the classes**.

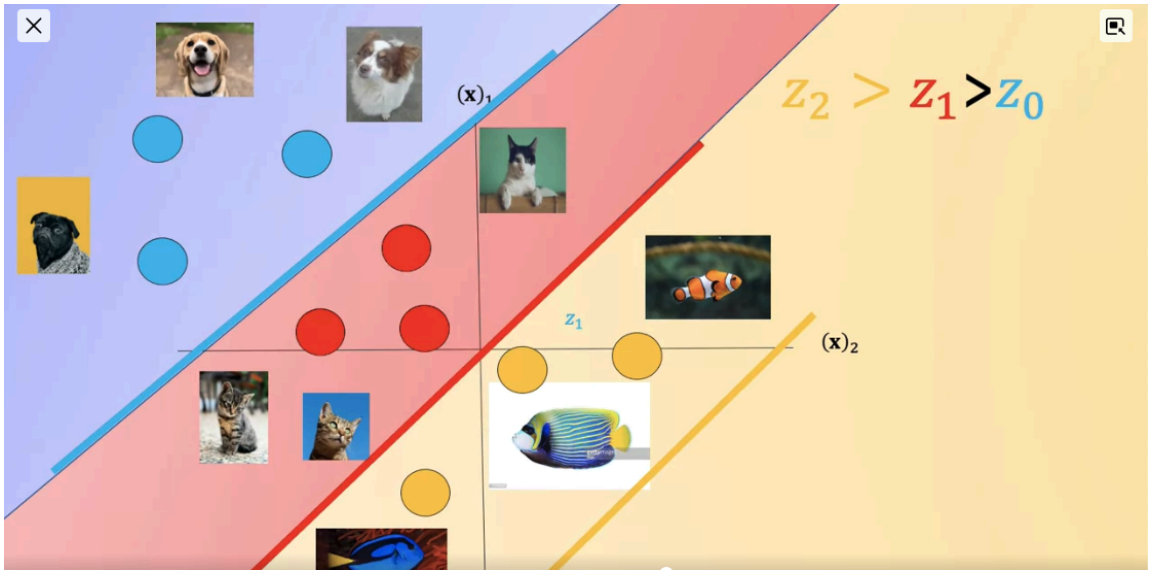
If the image input is in the **blue** region, the (**Z0) is the largest number**, which means the class is in class 0.



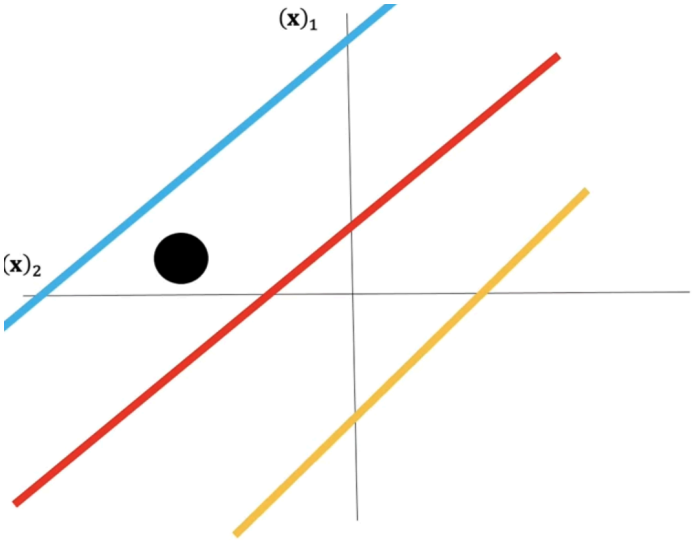
If the image input is in the red region, t**he (Z1) is the largest number**, which means the class is in class 1.



If the image input is in the yellow region, the **(Z2) is the largest number**, which means the class is in class 2.

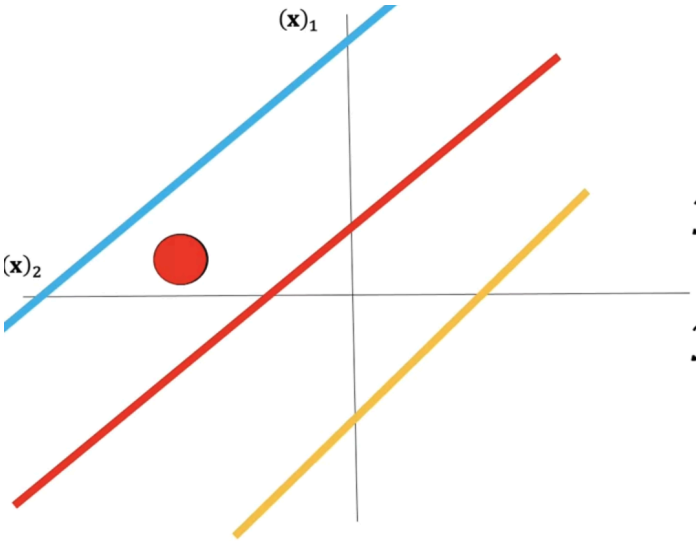


We can use the planes to classify the unknown sample:



$$\hat{y} = argmax_i\{z_0, z_1, z_2\}$$

In the above case, (Z1) is the largest number, so the class is (y1). We use the [**argmax**] function to find the largest number.

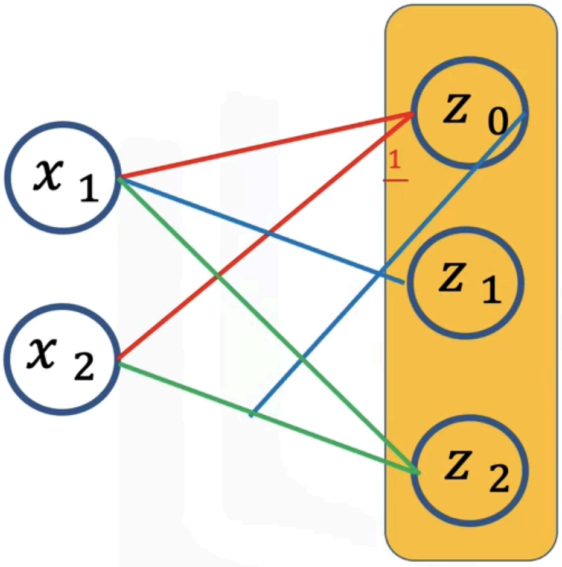


$$\hat{y} = argmax_i\{z_0, z_1, z_2\}$$

$$\hat{y} = argmax_i\{-2, 10, 6\}$$

$$\hat{y} = 1$$

It's called **SoftMax** since the actual distances are (Dot) product for input vector (Xi) with the parameters converted to **probabilities**.

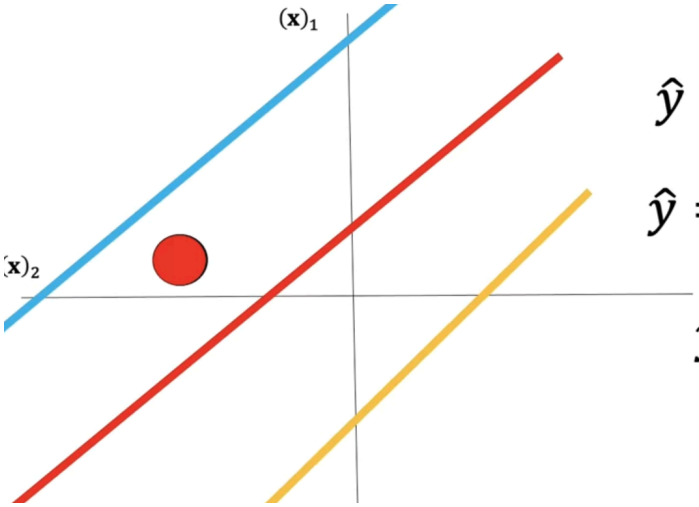


$$P(y=0|x)=\frac{e^{z_0}}{e^{z_0}+e^{z_1}+e^{z_2}}$$

$$P(y=1|x)=\frac{e^{z_1}}{e^{z_0}+e^{z_1}+e^{z_2}}$$

$$P(y=2|x)=\frac{e^{z_1}}{e^{z_0}+e^{z_1}+e^{z_2}}$$

$$\begin{bmatrix} P(y=0|x) \\ P(y=1|x) \\ P(y=2|x) \end{bmatrix}$$



$$\hat{y} = argmax_i\{\mathrm{p}(\mathrm{y} = i|\boldsymbol{x})\}$$

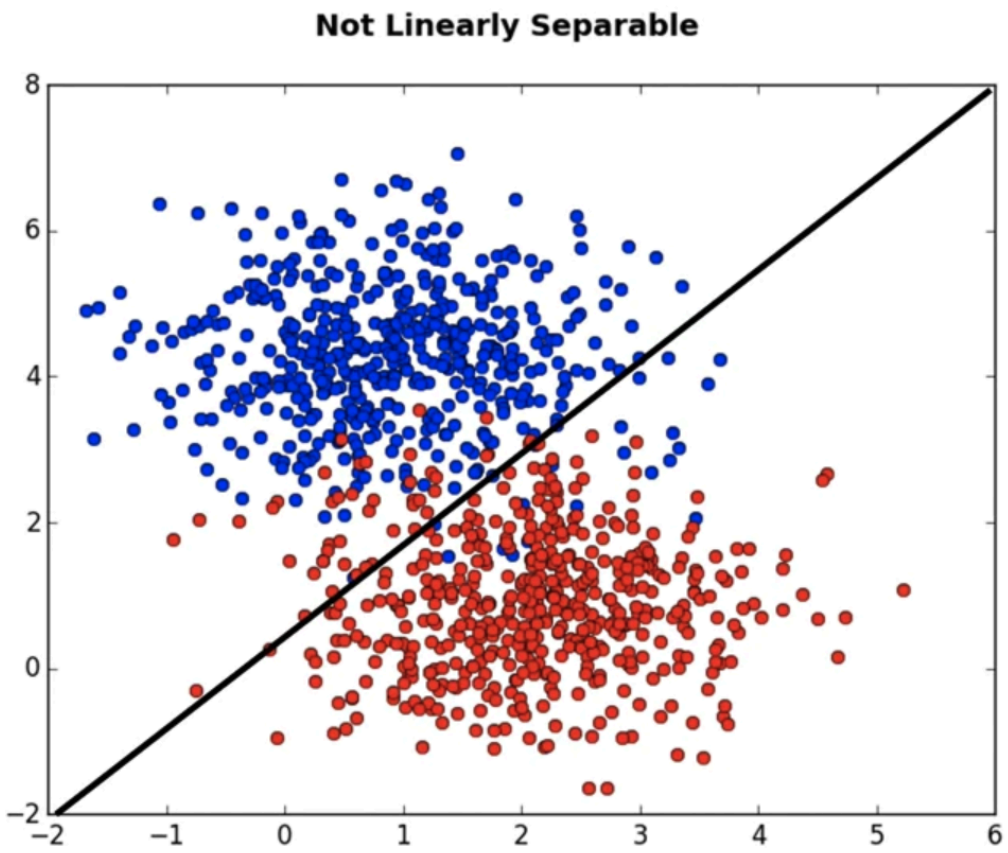
$$\hat{y} = argmax_i\{0.1, 0.7, 0.2\}$$

$$\hat{y} = 1$$

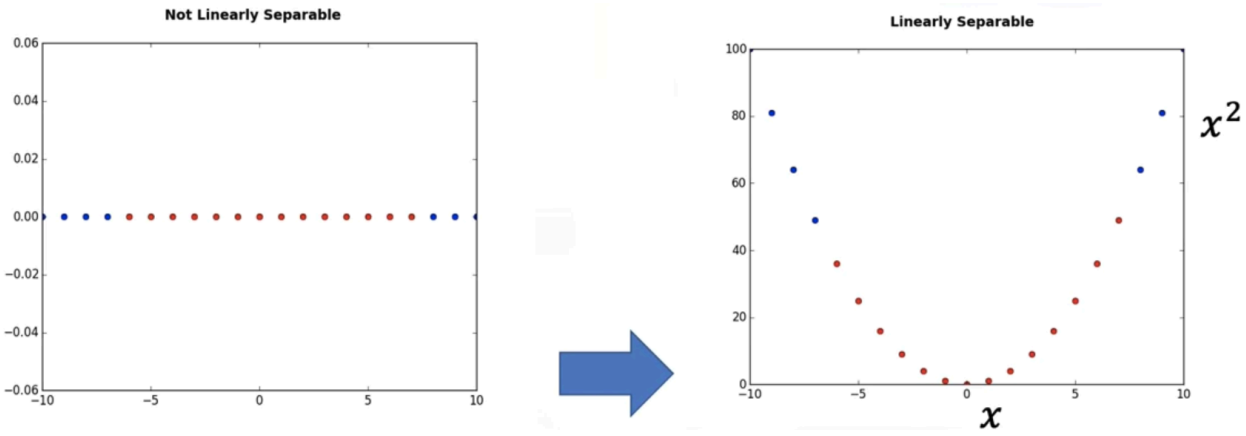
- There are **several methods to convert (2) class classifiers to multiclass**.
    - One-Vs-Rest.
    - One-Vs-One.

- **Support Vector Machines (SVM):**

A dataset is linearly separable if we can use a plane to separate each class, but not all data sets are linearly separable. As shown below:

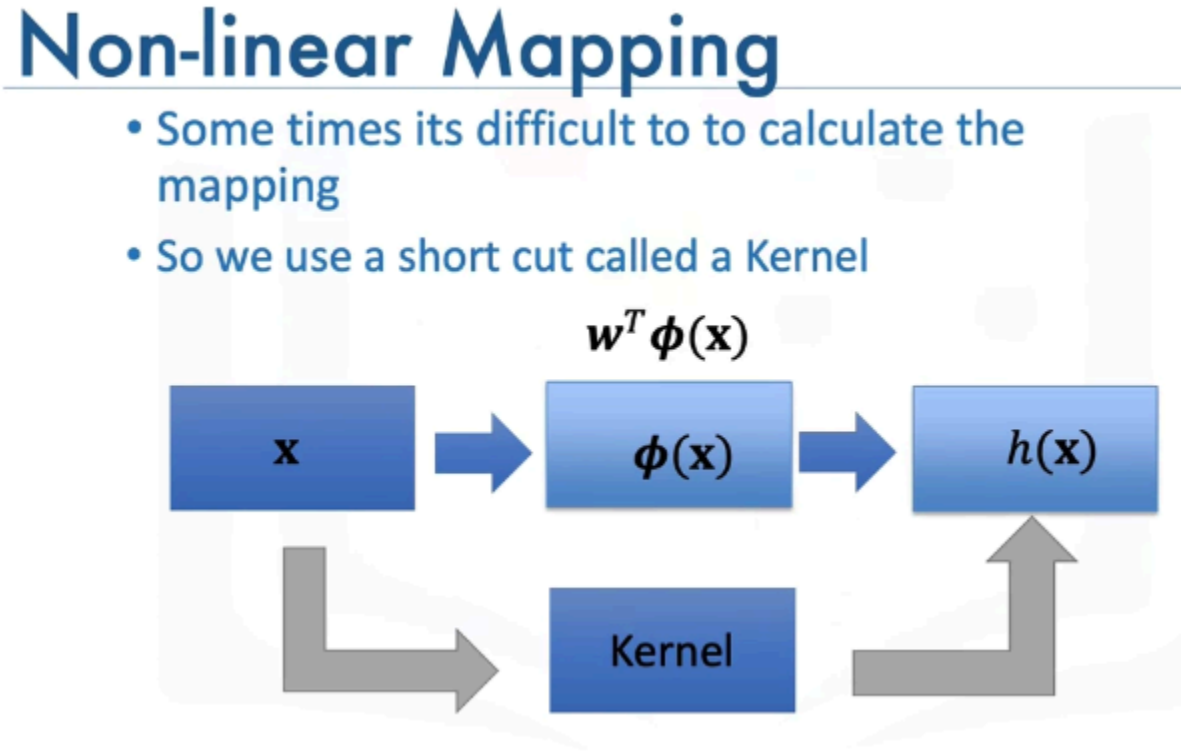


We can **transform the data to make a space** where it's linearly separatable. We **can transfer it into a higher dimensions space**. We can increase the dimensions of data by **mapping** (X) into a **new space using a function** with outputs ($\mathcal{X}$) and ($\mathcal{X}^2$).



$$\phi(x) = [x, x^2]$$

Sometimes it's difficult to calculate the mapping, so we **use a shortcut called [Kernel]**:

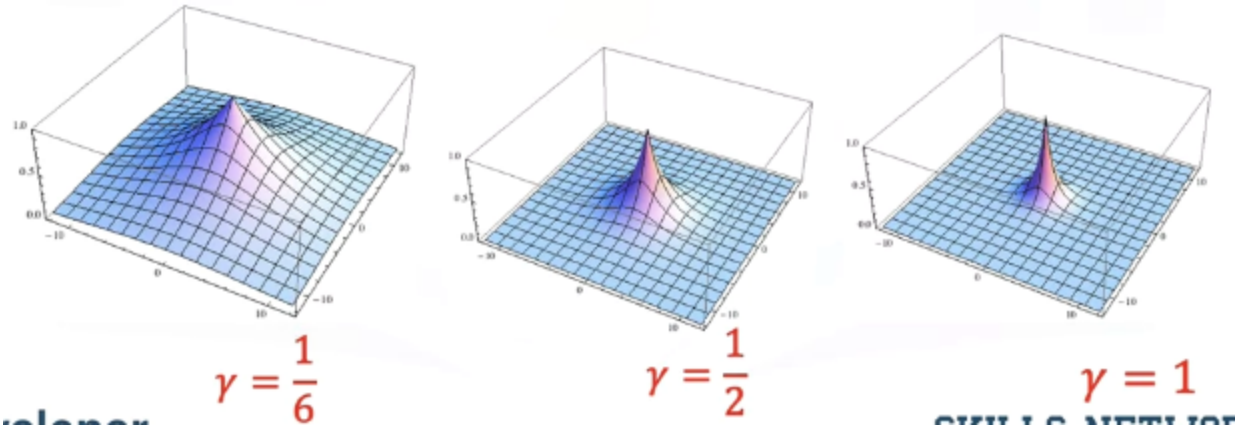


- There are **different** types of **Kernels**:
  - Linear.
  - Polynomial.
  - Radial basis function (RBF). - Mostly used

RBF kernel finds the **differences between two inputs (X) & (X')**[Prime] which is called **Support Vector.** It has the parameter **Gamma.**

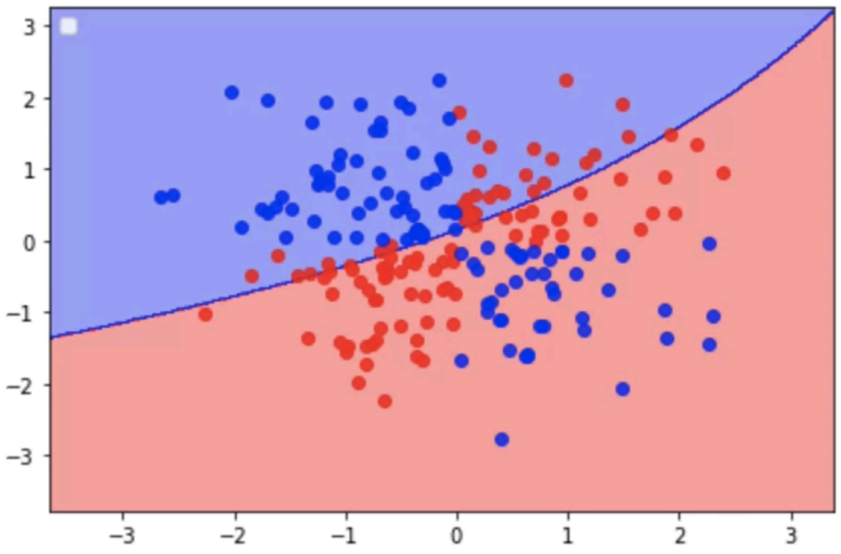
$$K(\mathbf{x}, \mathbf{x}') = \exp(\gamma \|\mathbf{x}' - \mathbf{x}\|^2)$$

- Where **gamma** controls the shape of the kernel

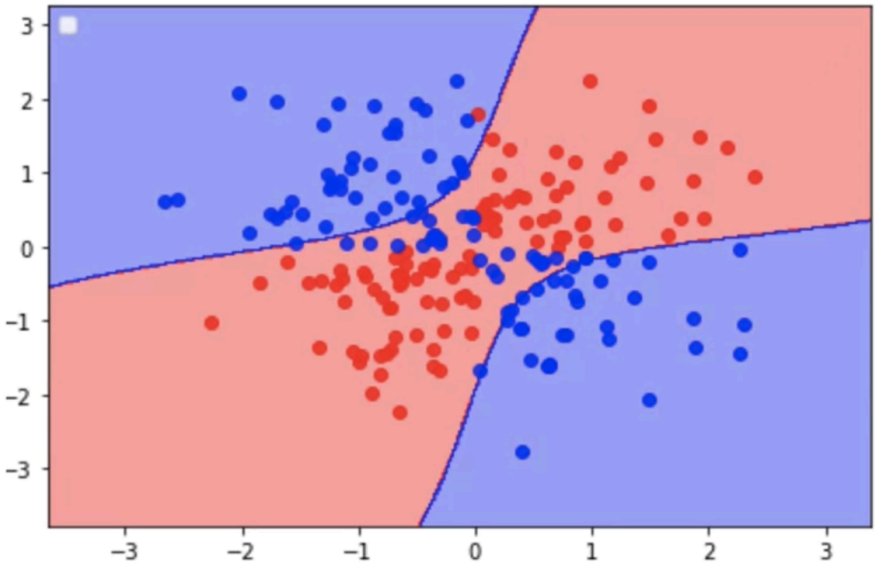


Using (Gamma) **increases the flexibility of the classifier**,

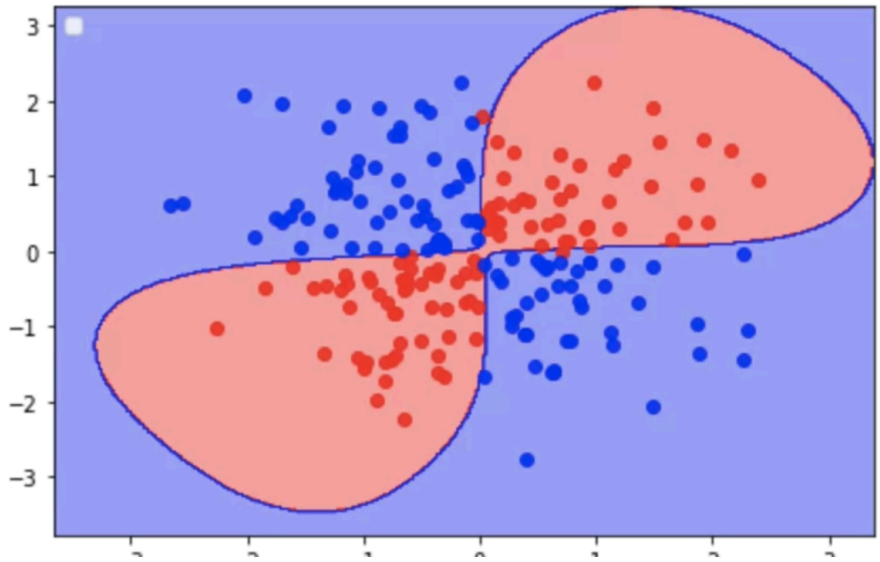
gamma	0.01							



gamma	0.01	0.1						



gamma	0.01	0.1	0.5	...				



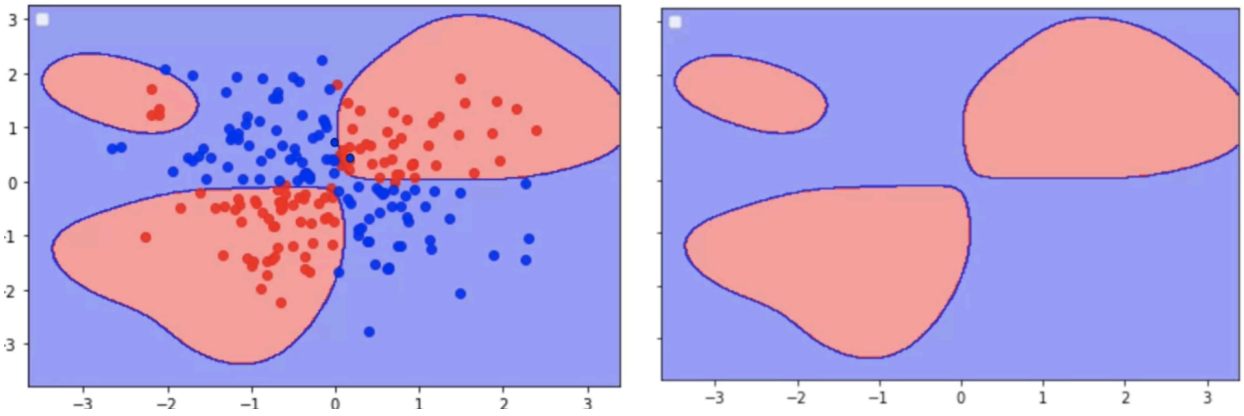
Increasing to higher (**Gamma**) is more likely to **overfit**, where the classifier fits the data points not the actual pattern.

Overfitting in the context of a Support Vector Machine (SVM) occurs when the model learns not only the underlying pattern in the training data but also the noise or random fluctuations. This leads to a model that performs well on the training data but poorly on unseen data, as it has essentially memorized the training data rather than generalized from it.

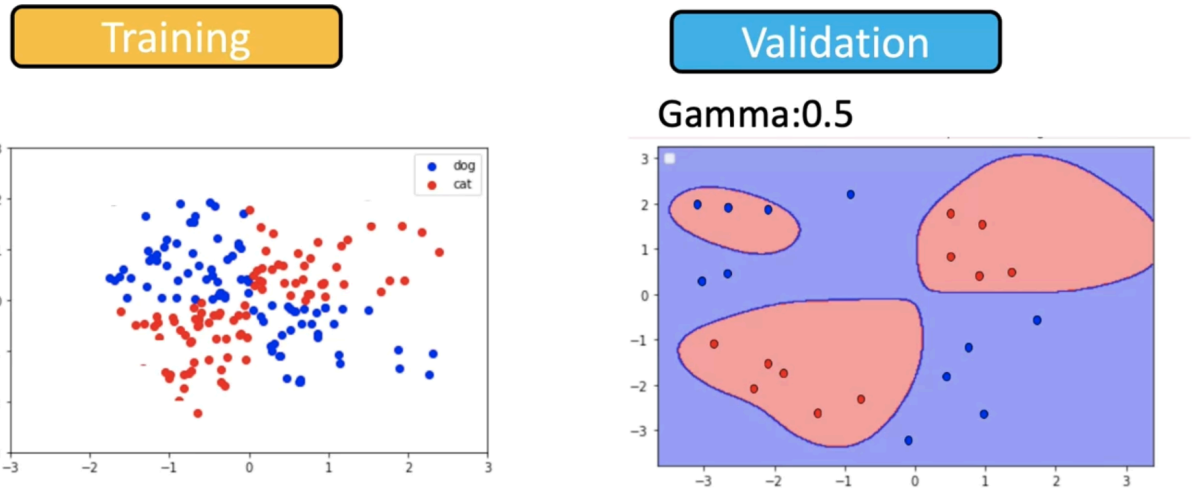
Key Points of Overfitting in SVM:

1. **Complex Decision Boundary**: Overfitting typically results in a very complex decision boundary that tries to perfectly classify all training examples, including outliers and noise.
2. **High Variance**: An overfitted model has high variance and low bias. This means it is very sensitive to the specific details of the training data, resulting in poor generalization to new data.
3. **Performance Metrics**: During overfitting, the model shows a high accuracy on training data but a significantly lower accuracy on validation or test data.

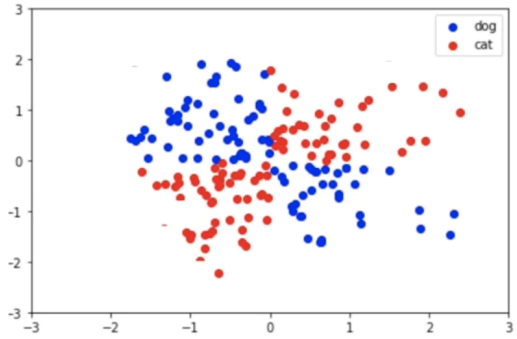
The classifier with the decision region, and does not match our Decision Region.



To avoid this, we find the **best value of Gamma** using **Validation Data**.

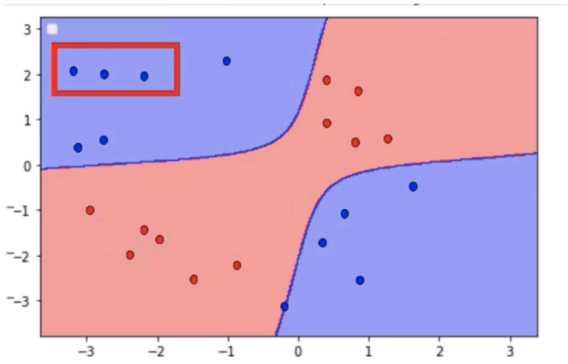


Training

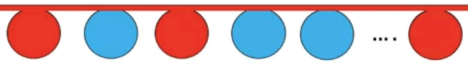


Validation

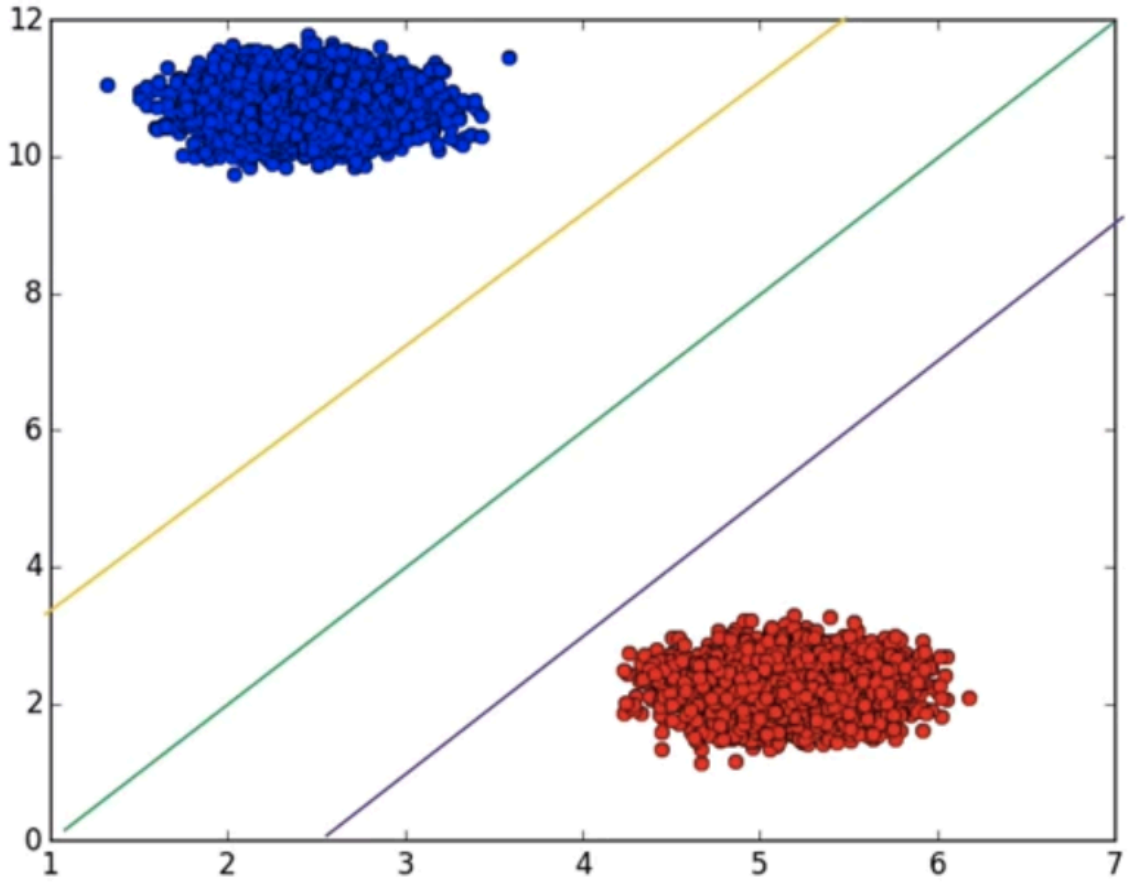
Gamma:0.1

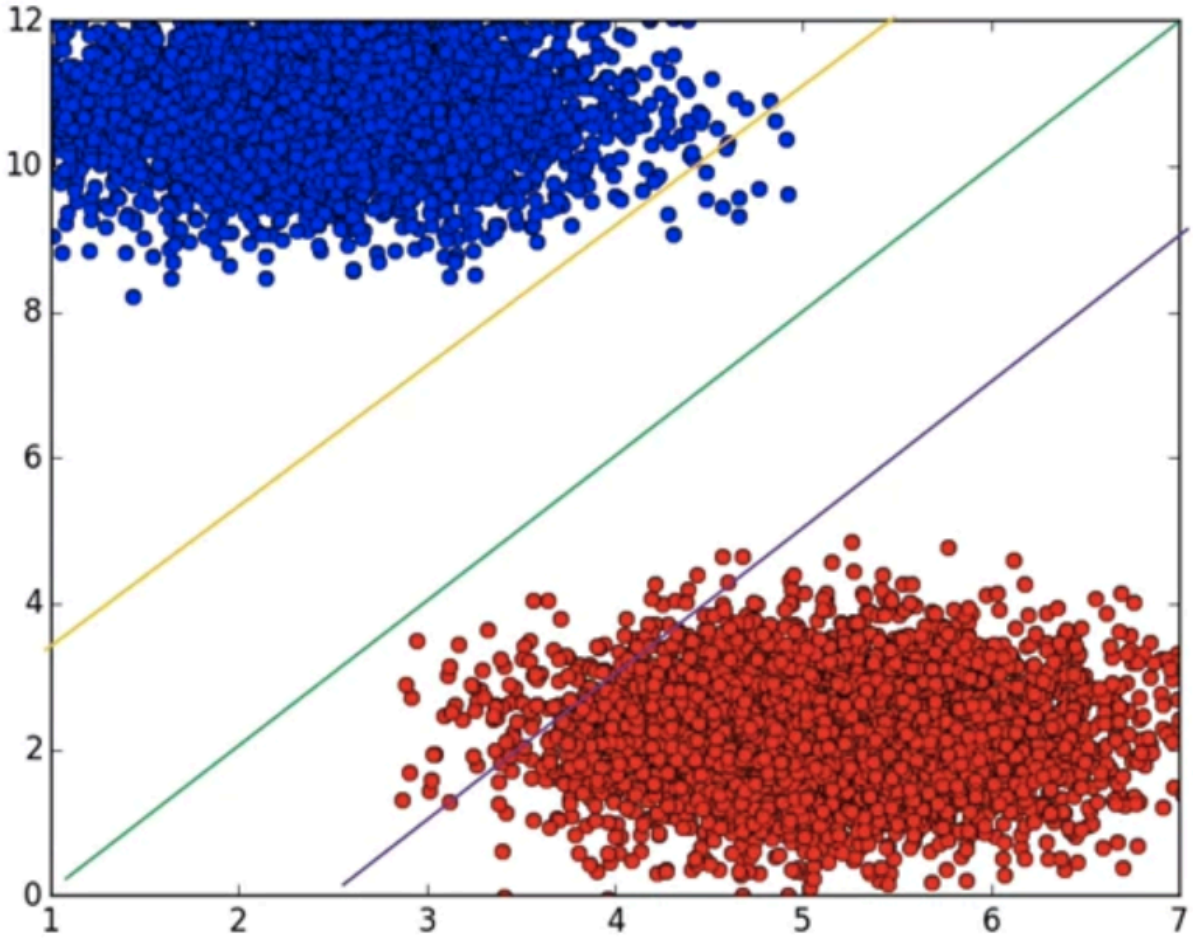


We select Gamma which gives us the **best results in Validation Accuracy**.

Gamma	y	$\hat{y}$	Validation Accuracy
0.001			75%
0.01			85%
0. 1			95%
1			85%

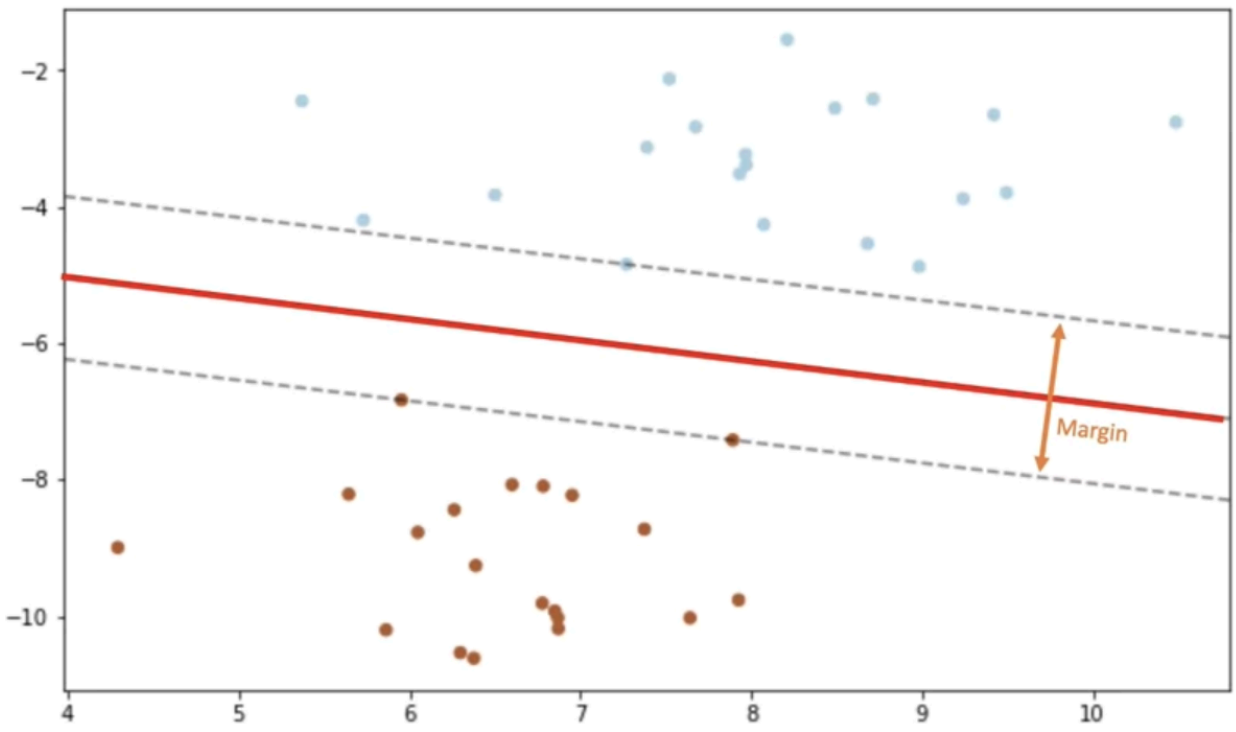
SVM works by finding the **Maximum Margin**, for the example below, the best plane for classifying the data is the green line, whereas the dataset caused by noise would not **affect the classification accuracy**.


12
10
8
6
4
2
0
1
2
3
4
5
6
7

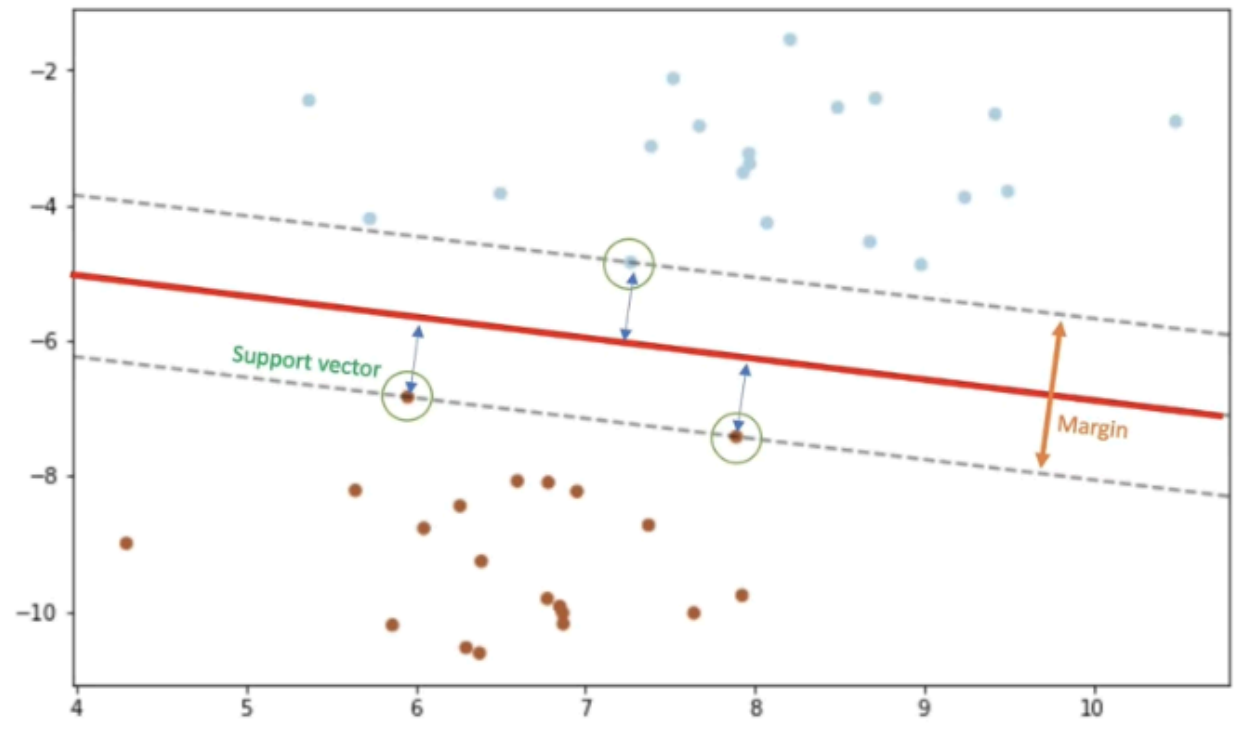


- **SVM** is the **best for selecting the best decision plane**, where SVM is based on finding a plane that best divides a dataset into two classes.

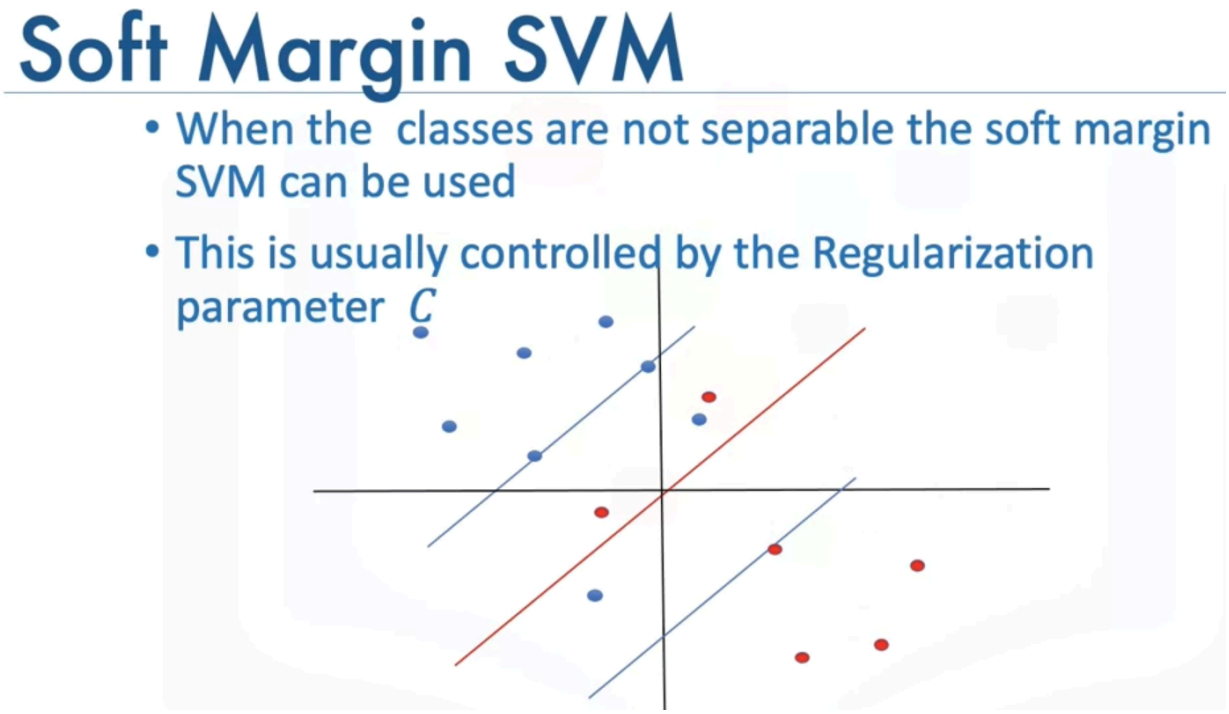
- The best **(Hyperplane)** is the one that represents **the largest separation** (**Margin**) between the two classes.



Samples **closest** to the Hyperplane are **Supported Vectors**.



To get the Maximum Margin, we use **these supported vectors to select the best plane**.

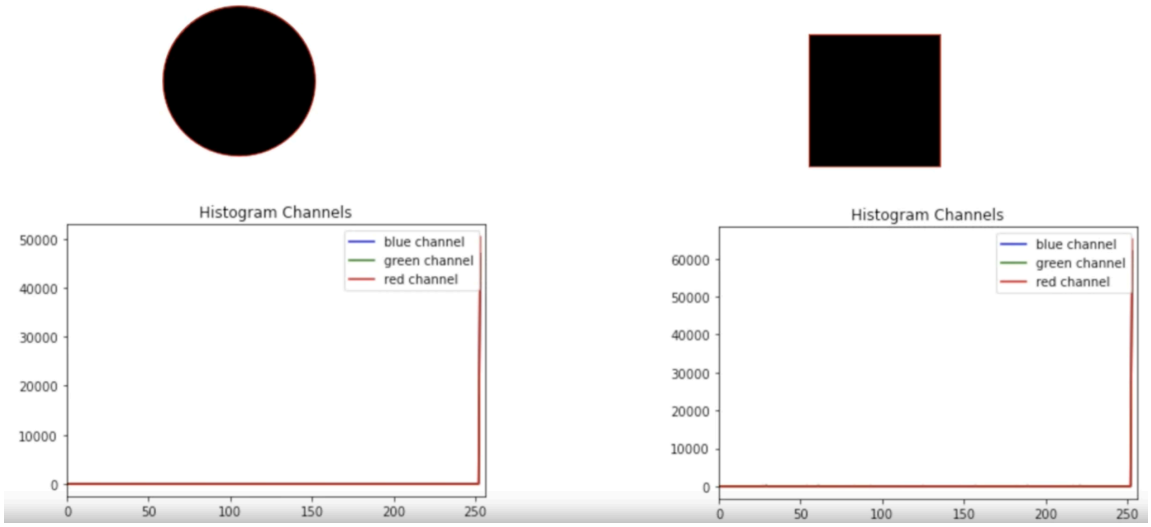


- **C** - **Regulation Parameter**, this allows **some samples to be misclassified**.

- **Image Features:**

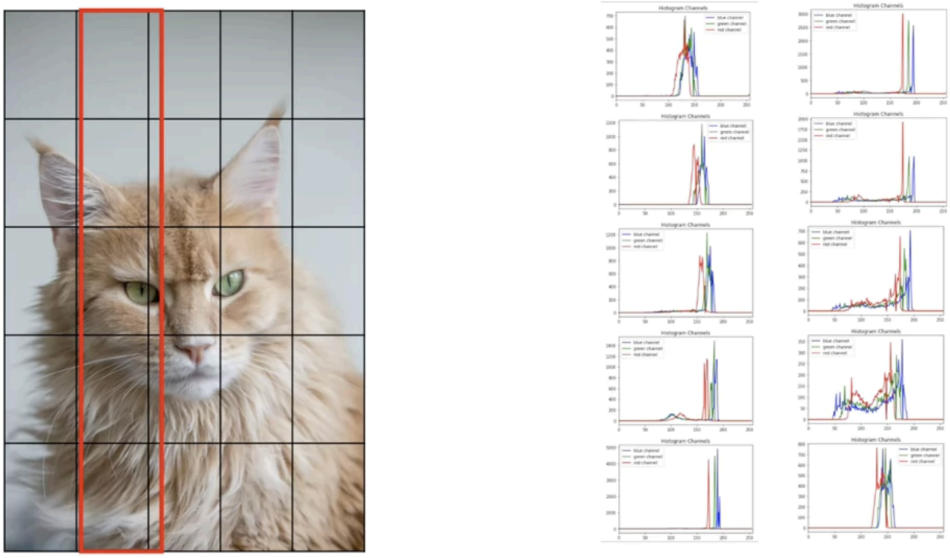
Using Intensity Values for the image **classification** does **not function well**. Classifying an image involves the relationship between pixels. Features are measurements taken from the image that **help with classification**.

Histograms count the intensities and do not consider the relationship between the pixels.



The above figure shows a circle and a square, the histogram counts the pixels' color, where both shapes have the same number of black pixels.

To solve this problem, we can split the image into sub-images and calculate the histogram for each sub-images.



We use features based on "**Image Gradients**"

Image Gradient's directional change in the intensity or color in an image. Essentially, gradients represent the rate of change in pixel values and are used to highlight edges or transitions in images.

We use **HOG (Histogram of Oriented Gradients) - Kernel**, The technique counts occurrences of gradient orientation in localized portions of an image. **H.O.G** generates a Histogram for each of these regions separately. **Histograms** are created using the gradients and orientations of the pixel values, hence the name ‘Histogram of Oriented Gradients’

H.O.G process as follows:

1- Convert the image to

2- Calculate the magnitude and angles of the **gradients** using **Sobel**.

H.O.G.

$$\|G\| = \sqrt{{G_x}^2 + {G_y}^2}$$

$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

3- Split the image into cells and calculate the histogram of Gradient Directions is applied for each sub-images (Cells).

	1	12	1
θ	0	....	180

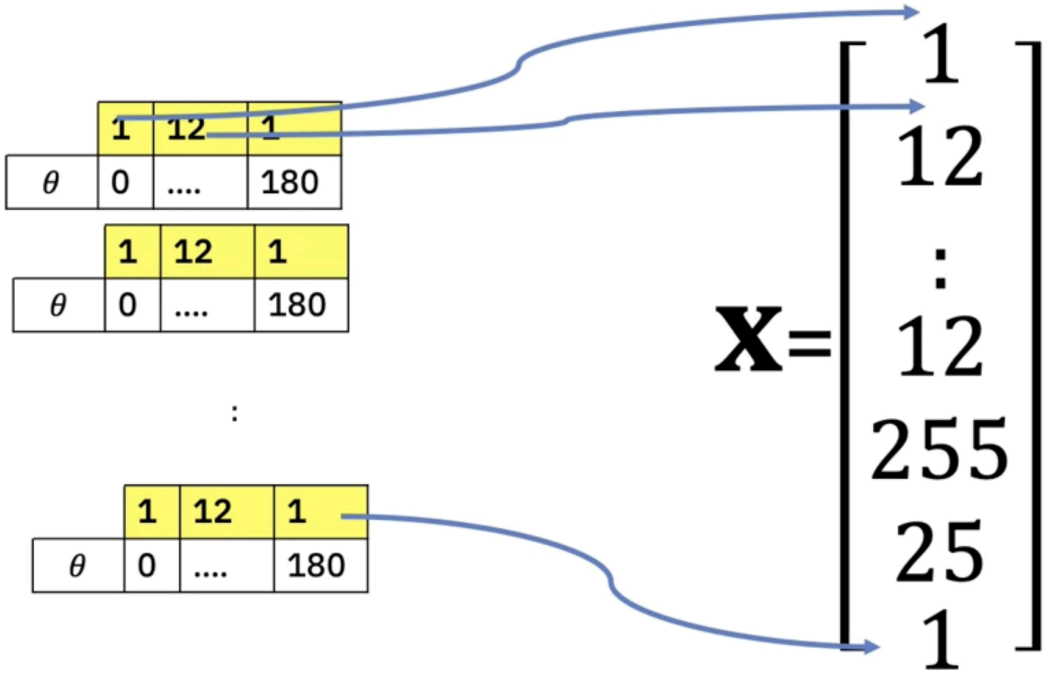
	1	12	1
θ	0	....	180

:

	1	12	1
θ	0	....	180

- To improve imbalance to highlights and shadows in the image, **cells are block normalized**.

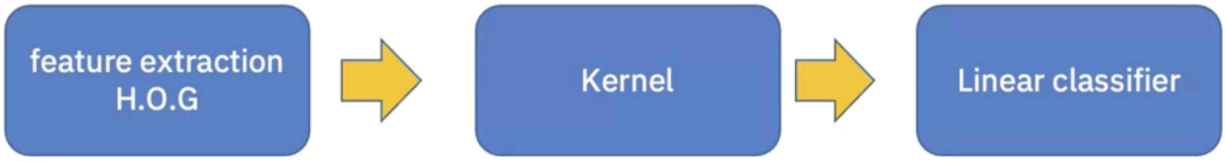
The **HOG feature vector** is a combination of all **pixel-level histograms** and used with **SVM to classify the image.**



we must also consider other free parameters like **number of image cells** or how **many angle bins in the histogram**.

- There are other types of features for images like (**SURF**) & (**SIFT**).

Machine learning Process:



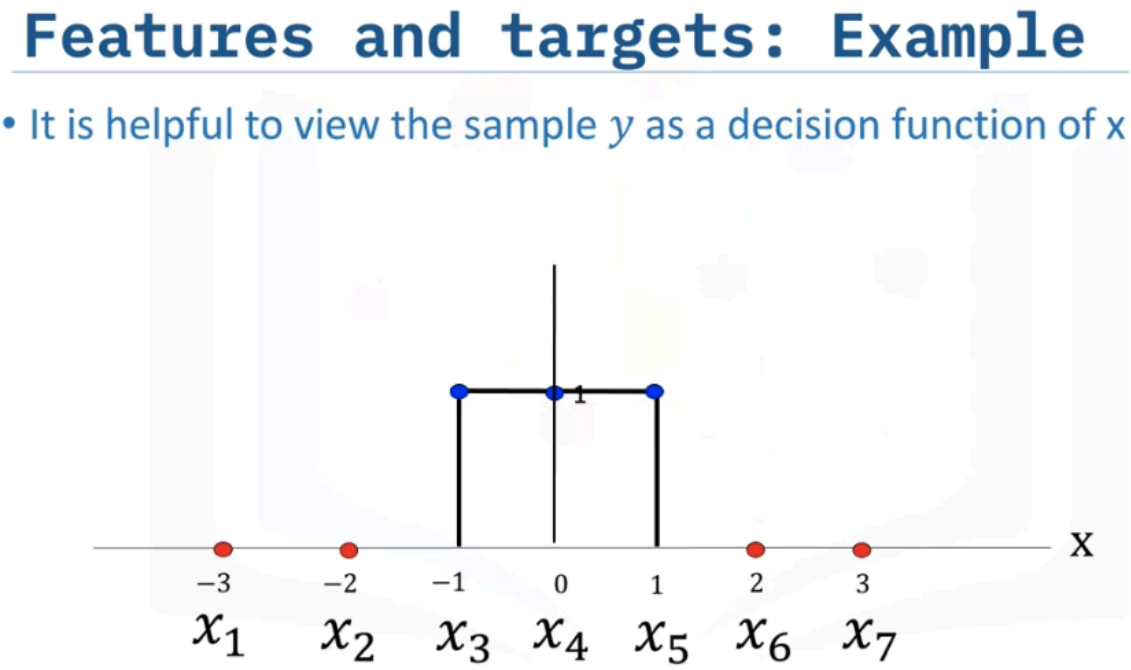
(Kernel for **Non-Linear Mapping**)

linear

https://www.coursera.org/learn/introduction-computer-vision-watson-opencv/lecture/A4BgA/image-features

Neural Networks & Deep Learning:

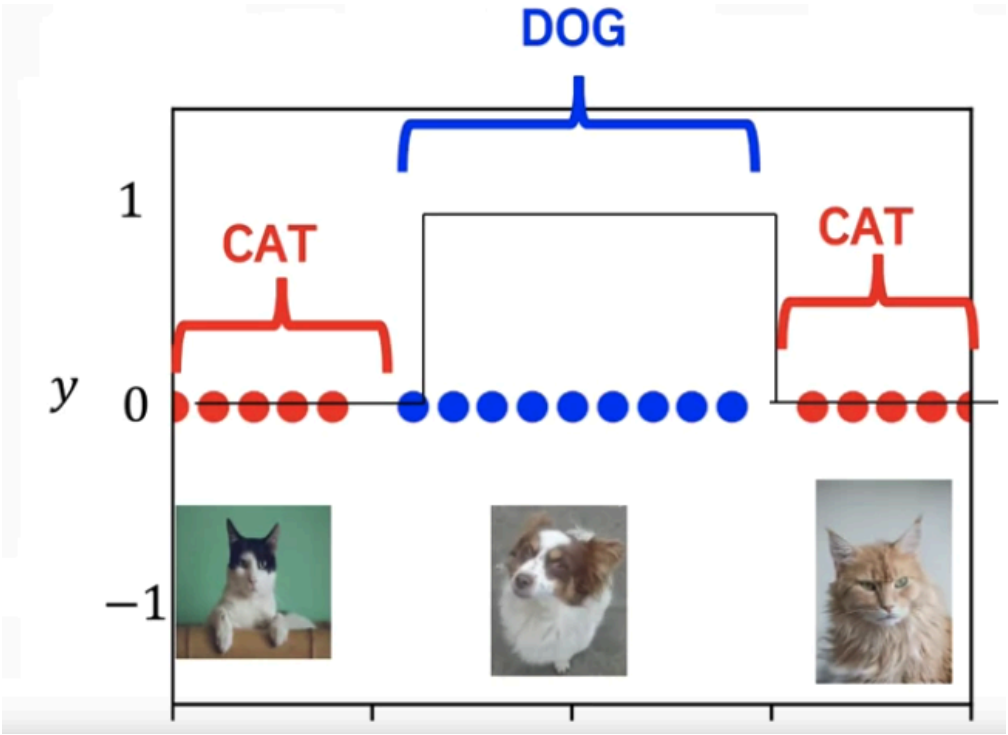
In Neural Networks, we think of the **classification** problem as **a decision function** like a function when (y) equals (1), the value is mapped to one in the vertical axis. We can represent the function as a box (Box Function) example of a [**Decision Fuction**] as shown below:



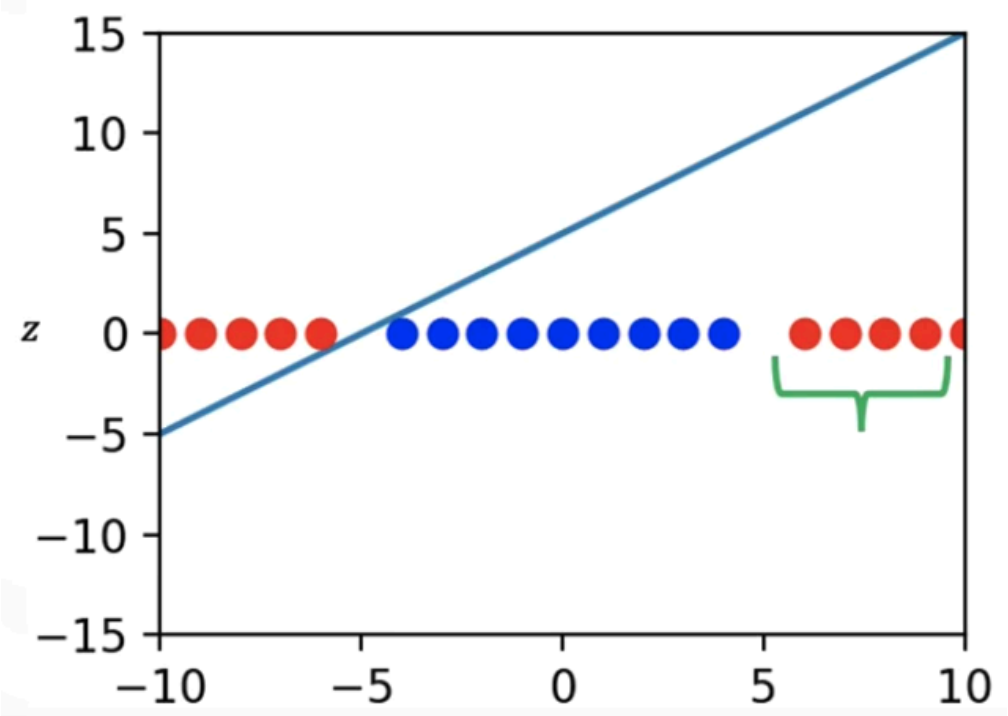
Where every value of (X) in the (Box) **regin is mapped to (0)**, and out of (Box) is mapped to (0).

- A Neural Network will **approximate the function** using **Learnable Parameters**.

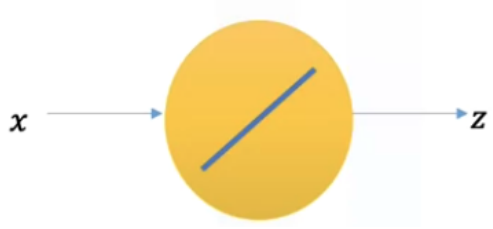
We can view the problem as trying to approximate the box function **using logistic regression**,



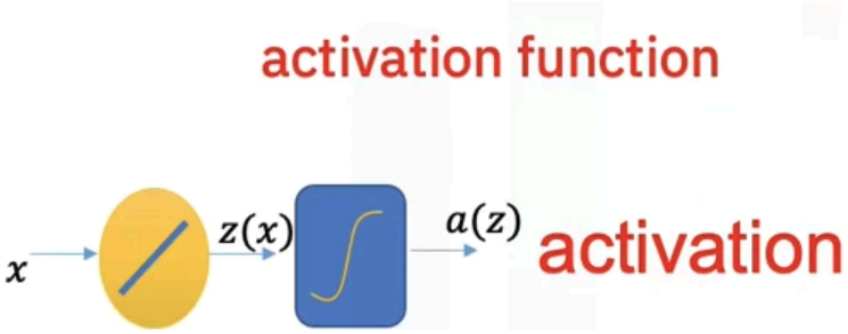
The above example, can't **separate the dataset with a straight line**. The straight line can be used to linerly separate some of the data:



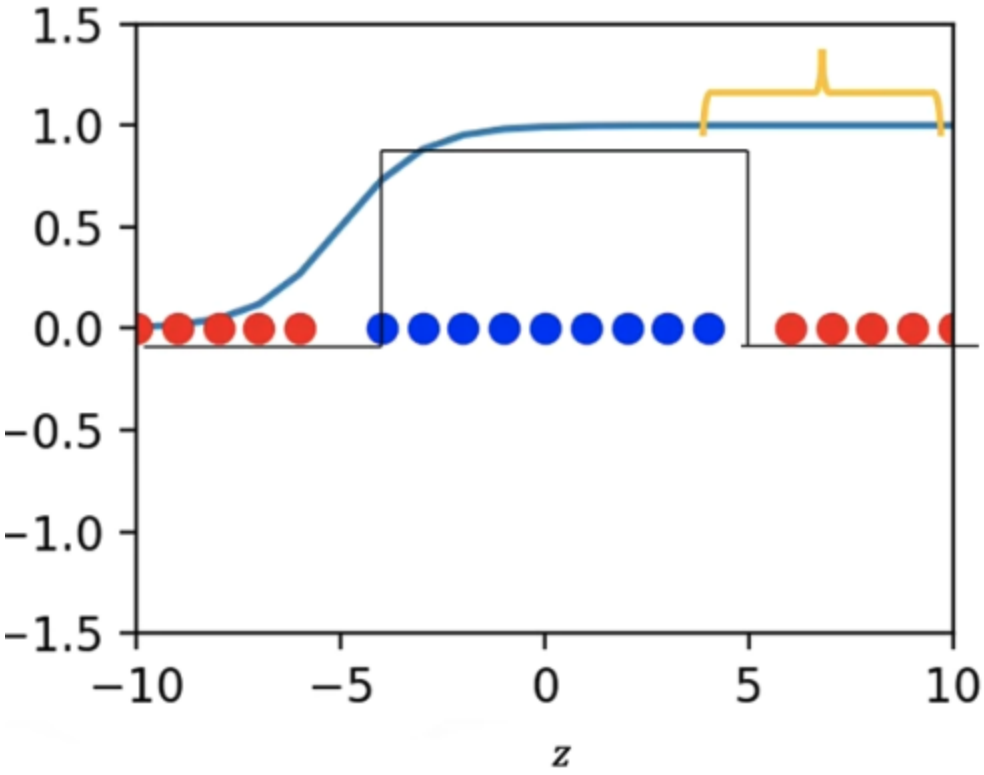
We can **use node** to represent the **line and the edges** to represent the input (x) and the output (z):



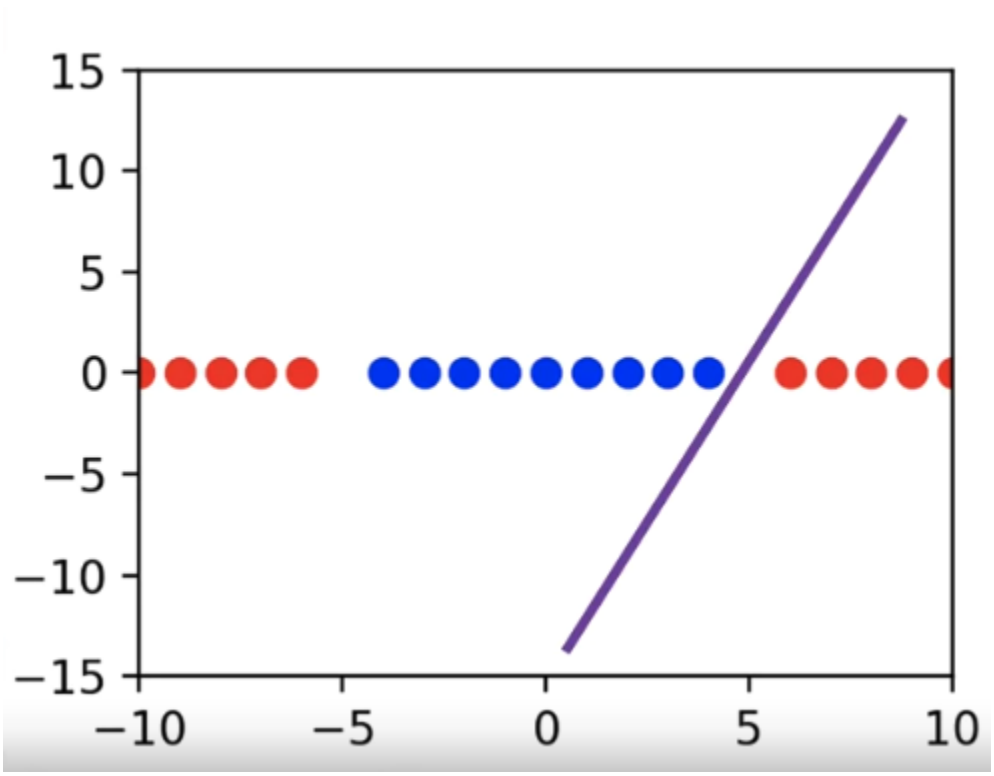
Applying Logistic Function in **Neural Network** is called **Activation Function.**



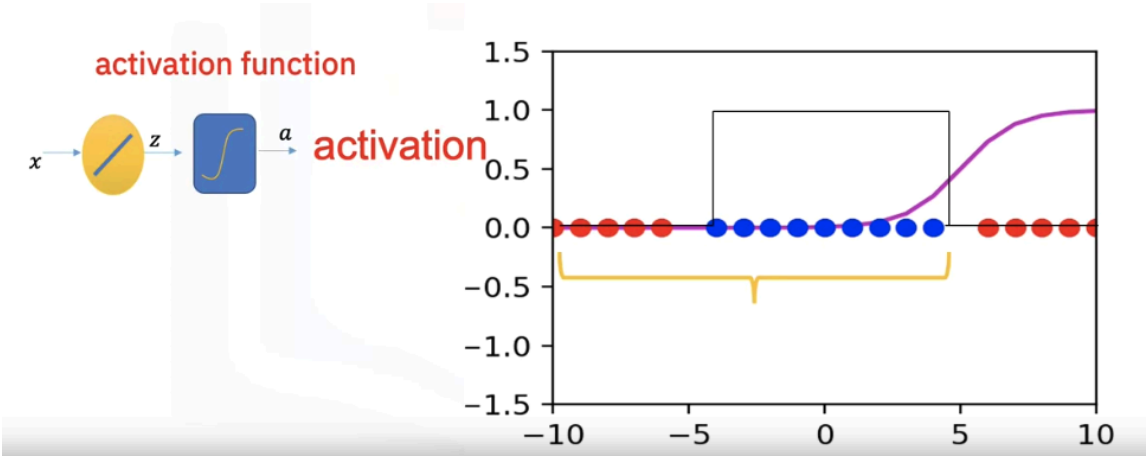
The (**Yellow**) classification **is incorrect.**



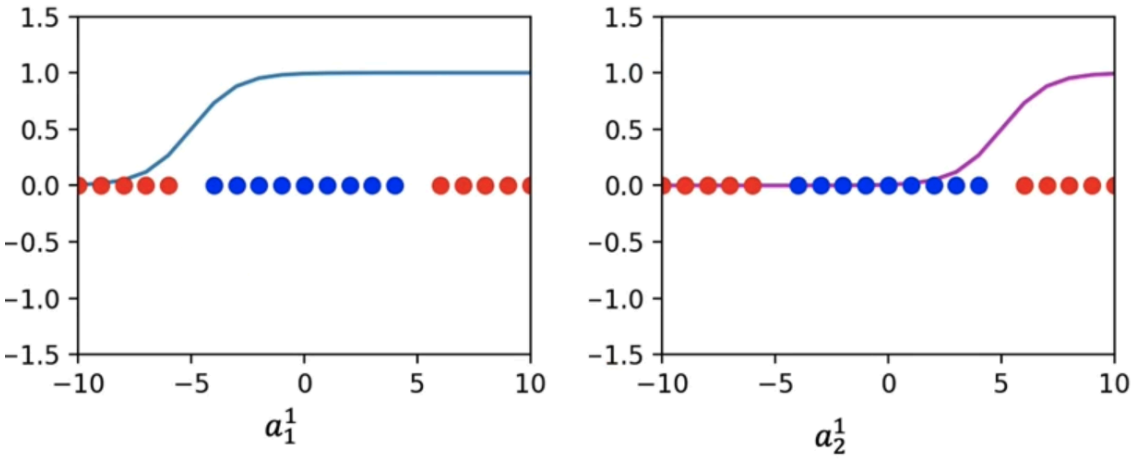
We can use other line to separate the data:



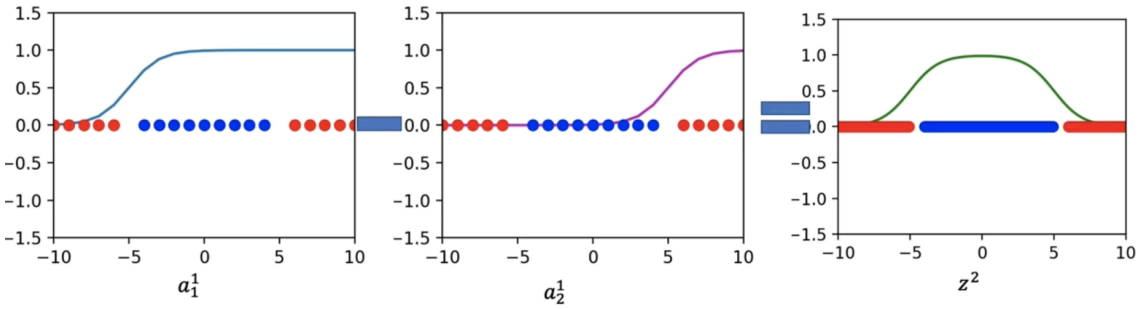
And if we **apply the Sigmoid/Activation function**:



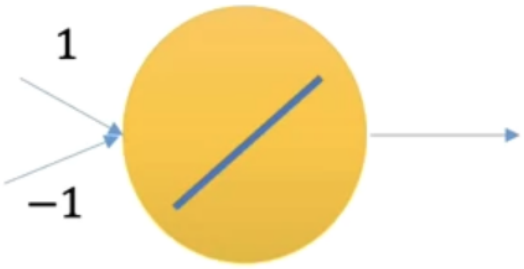
We will get some incorrect classifications in (Yellow) Samples. Now, we will have **Two Sigmoid Functions**:



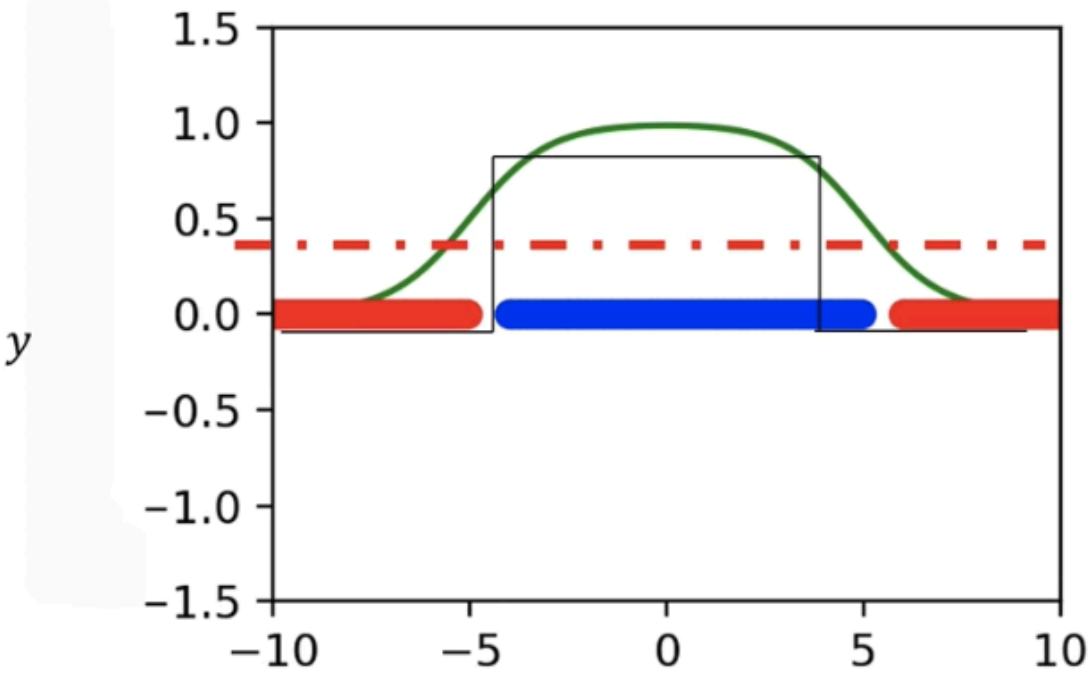
If we **subtract** the **second Sigmoid function** from the **first Sigmoid Function**, we will get the following **Decision Function:**



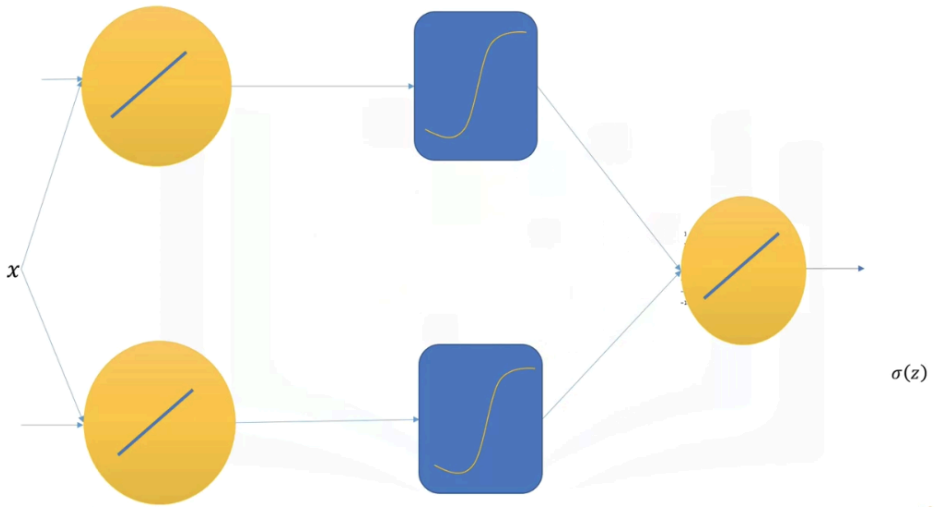
We can apply it in **nodes** by subtract the **Second Activiation Function** from the **First Activiation Function**:



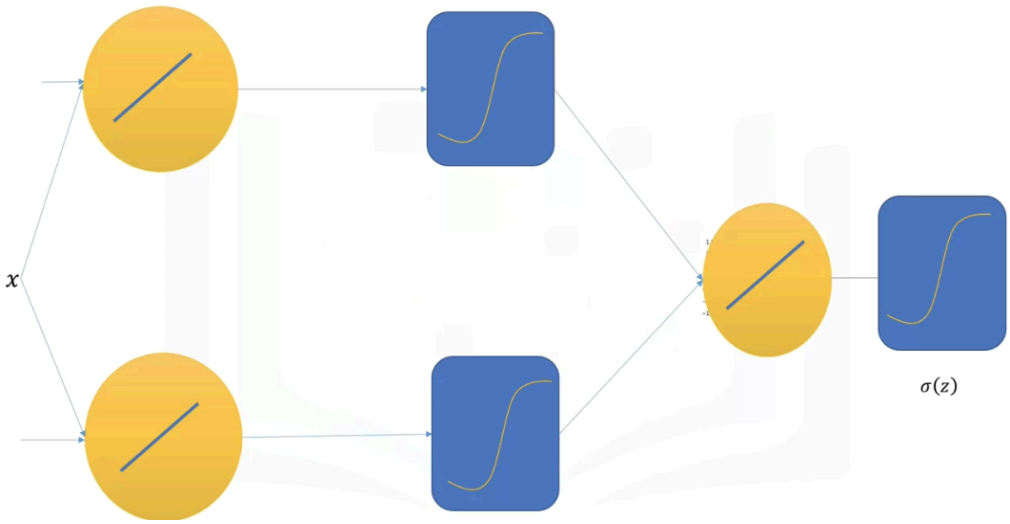
If we apply **threshold** setting every value less than (0.5) to (0) and greater than (0.5) to (1):



Now we can classify the data, and we can obtain the **parameters** via (**Gradient Descent**). We can use the graph to represent the process:

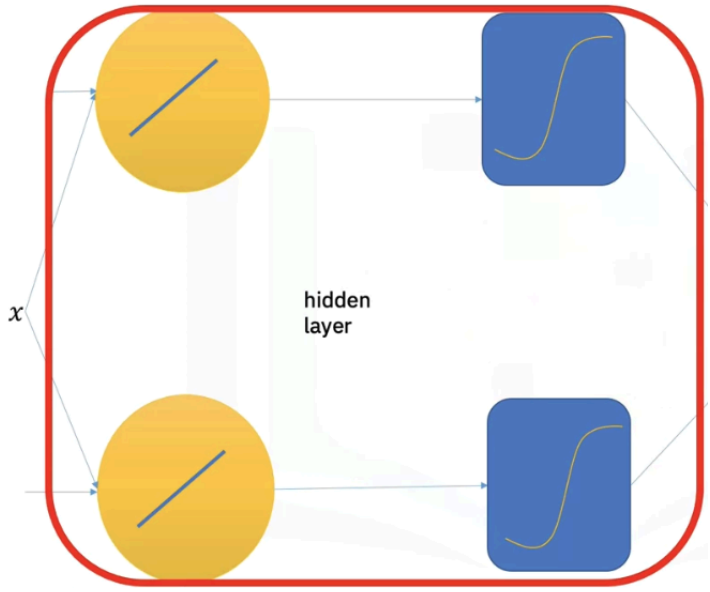


The Yellow Nodes are (**Linear Functions**) and the Blue Nodes are (**Sigomid Functions**). We apply (**Sigomid Function**) to the **output of this (Linear Function)**, then we apply **threshold**.

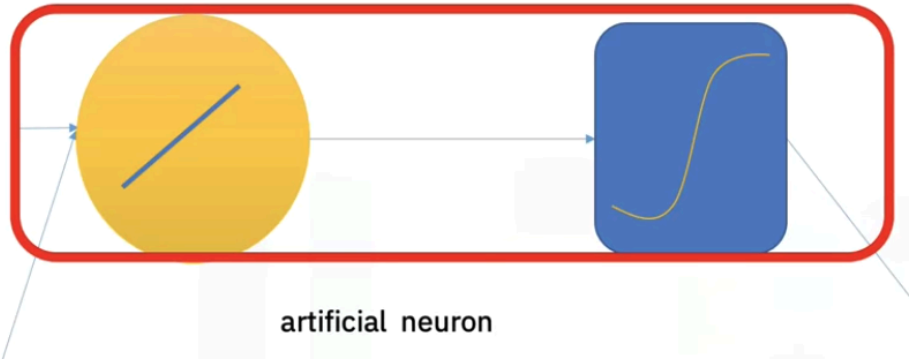


The above diagram represents a (**Two-Layer) Neural Network**.

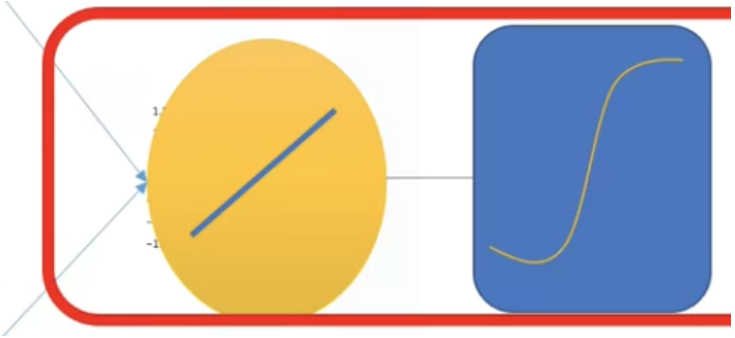
We have the Hidden Layer:



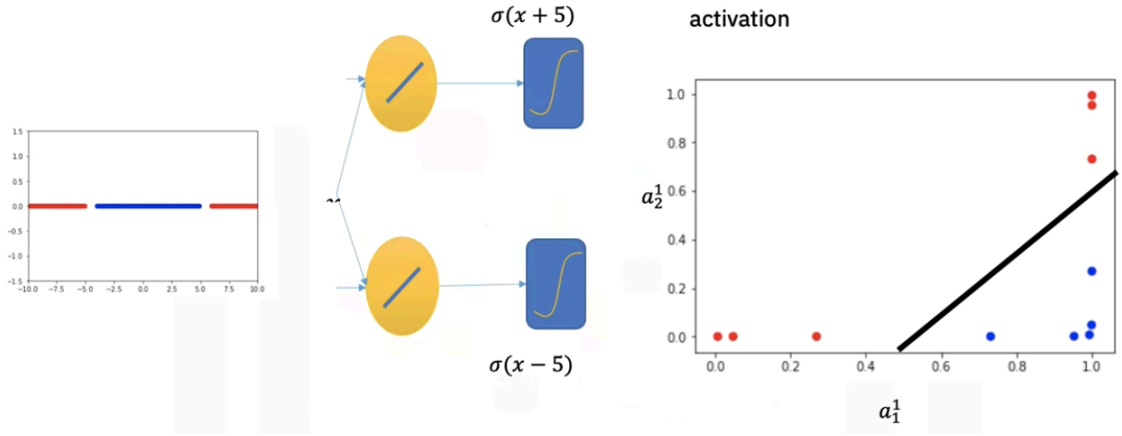
Each **Linear function and Activiation Function** is known as **Artificial Neuron**. Where the above hidden layer has two **Artificial Neurons.**



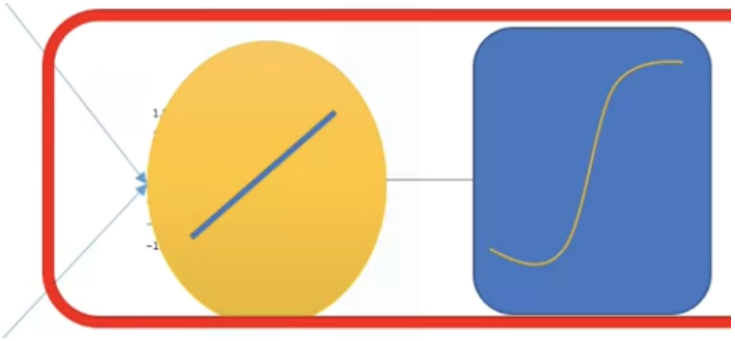
The output has one **Artificial Neuron:**



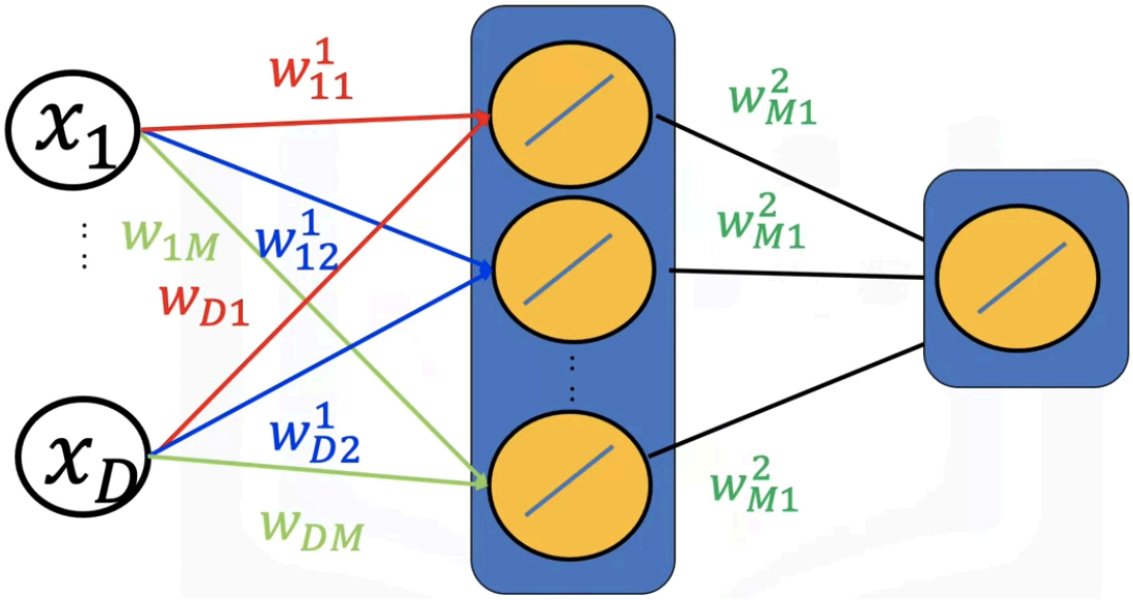
The **output** of the **Activation Function is a 2D plane** (**The Inputs are 1D Plane**), and we can split the points **using a plane**:



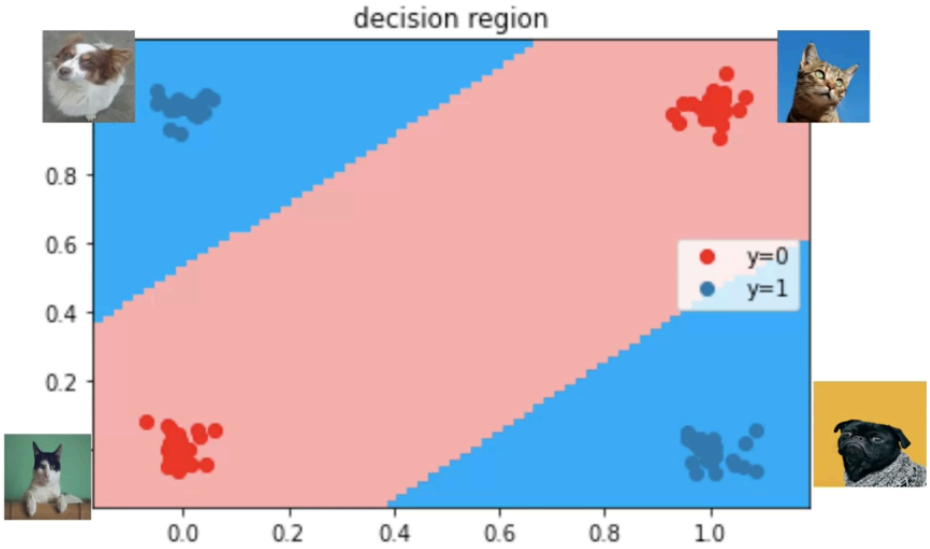
The **linear function** in the second (Out Layer) layer is for the **split plane**.



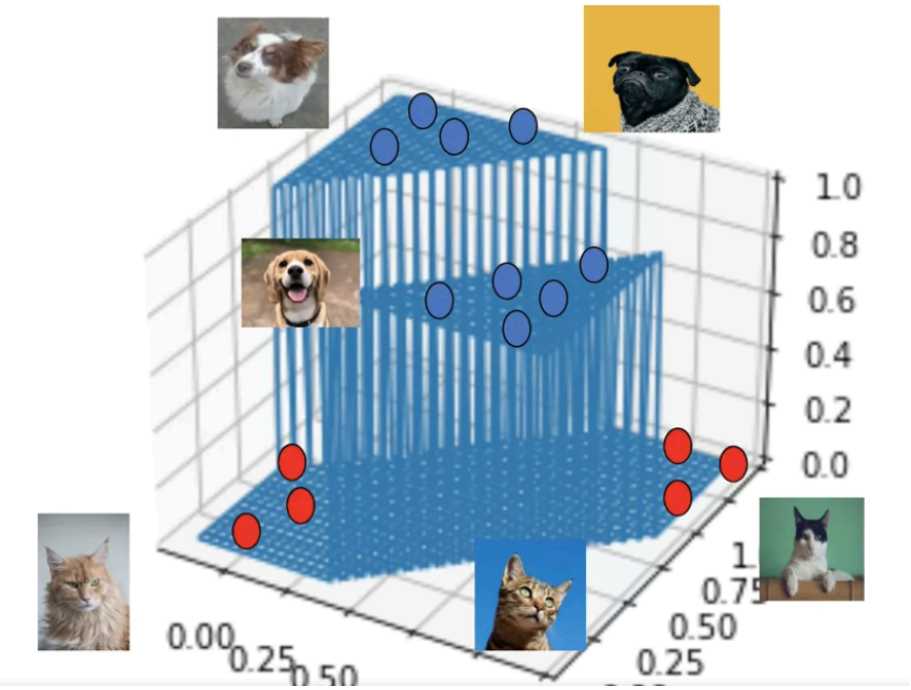
We can apply the same concept to **higher dimensions inputs** as shown below:



This type of Neural Networks is called **Feedforward Neural Networks**. The more dimensions the **more Neurons** we need.

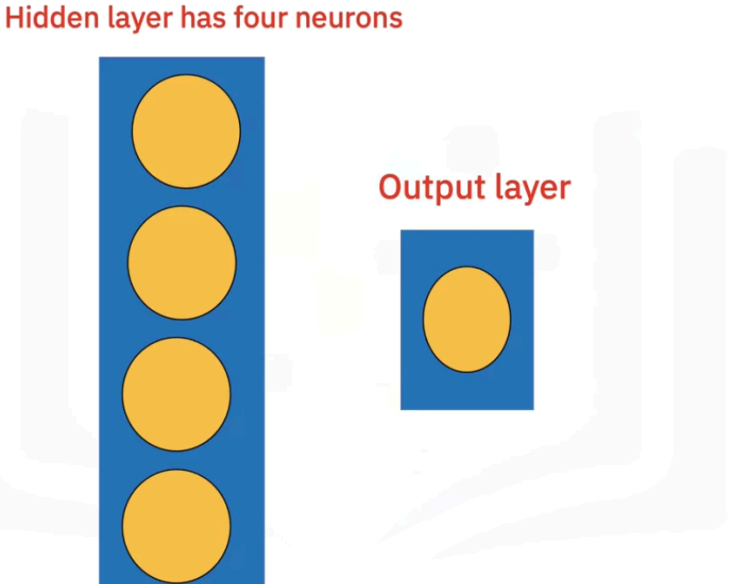


The decision planes

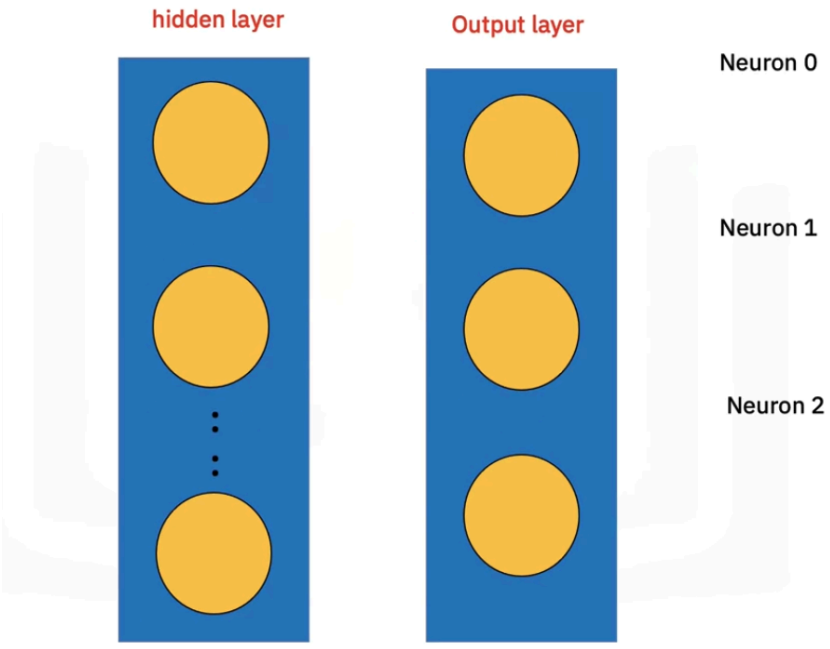


- **<u>Fully Connected Neural Network:</u>**

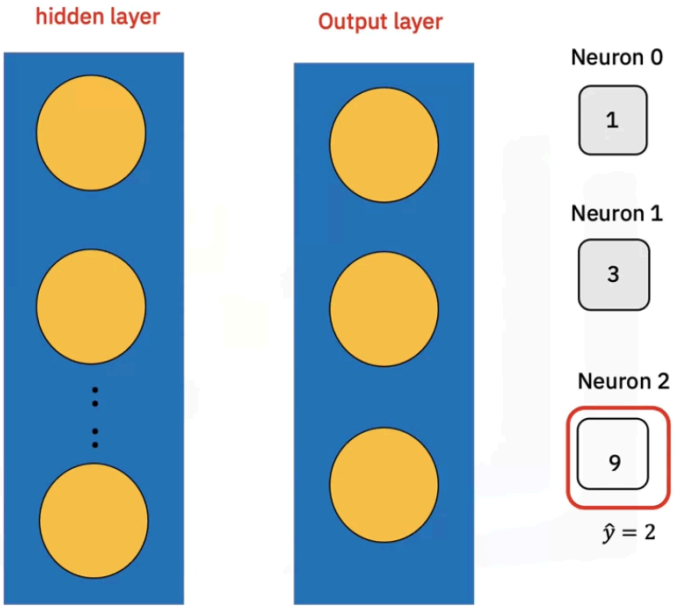
The example below shows a <u>hidden</u> layer with **<u>(4) neuron</u>**s, and the **<u>output has (1) neuron</u>**.



We can make **multiclass prediction** by using Neurol Network, we add **more Neurons to the output** layer.

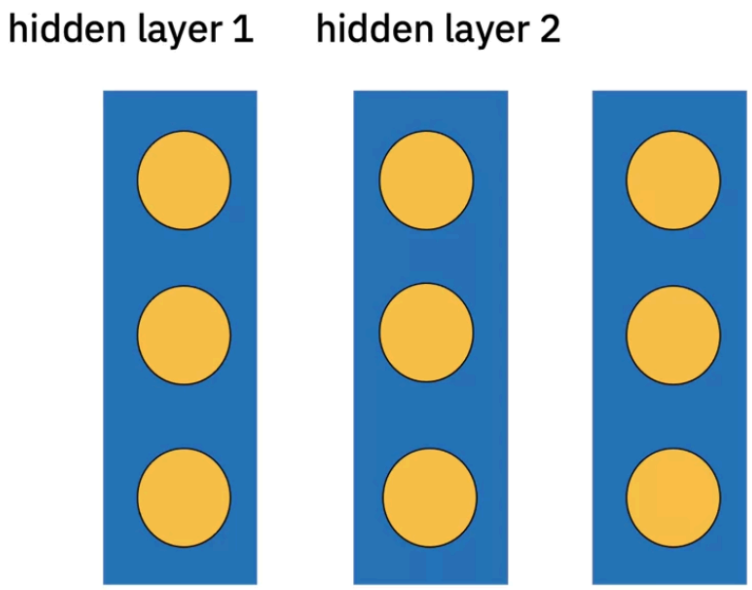


We can replace the **output layer** with **SoftMax Function**. Each **output neuron** has a **class** whereas three output neurons have three classes. For a given input, we **obtain an output for each neuron**.



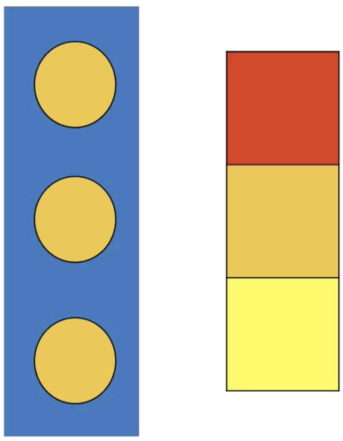
Then, we can use the **SoftMax** Function to choose the class **that is the index** of the neuron that has **the largest value**.

If we have **more than one hidden layer**, the Neural Network is called a (**Deep Neural Network**).



- **More Hidden Layers** or **Neurons** may lead to **overfitting**.

The output or Activation (**a**) of each **layer** is the **same dimension** as the **number of Neurons**.

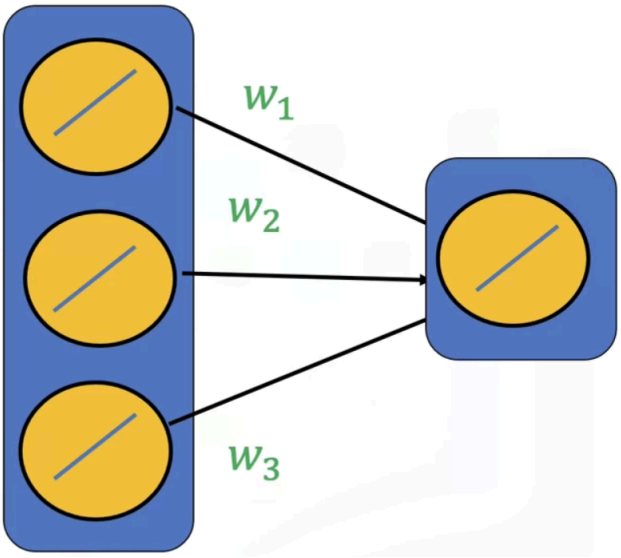


The above Layer has **three Neurons**, therefore the output (**Activation**) has **three dimensions**.

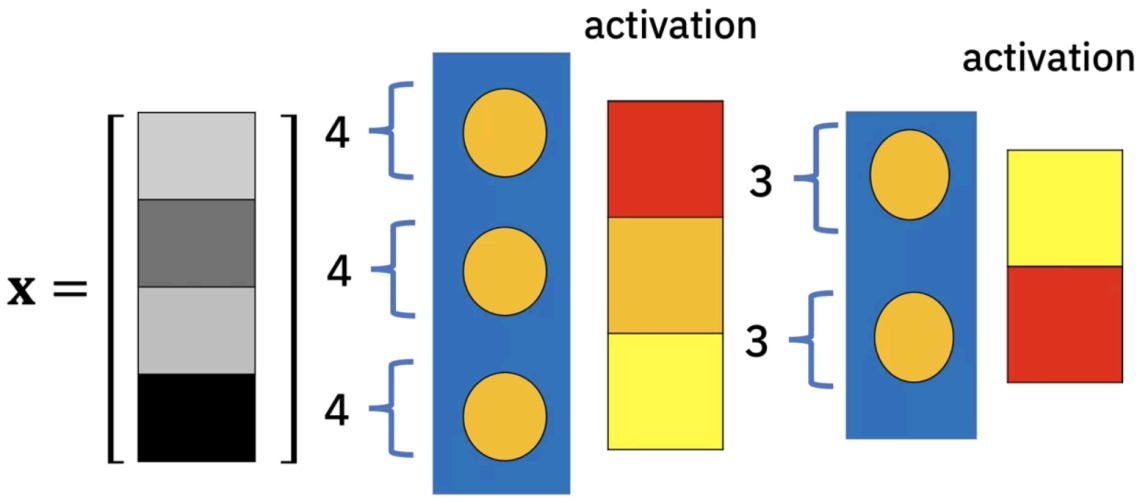
- (As we know the output of the hidden and the output layer are activation (Sigmoid)).

Each neurons is like a linear classifier, therefore **each neurons must have the same number of the input** as the previous layer.

The below example the previous layer has three Neurons, so the next layer has three inputs.

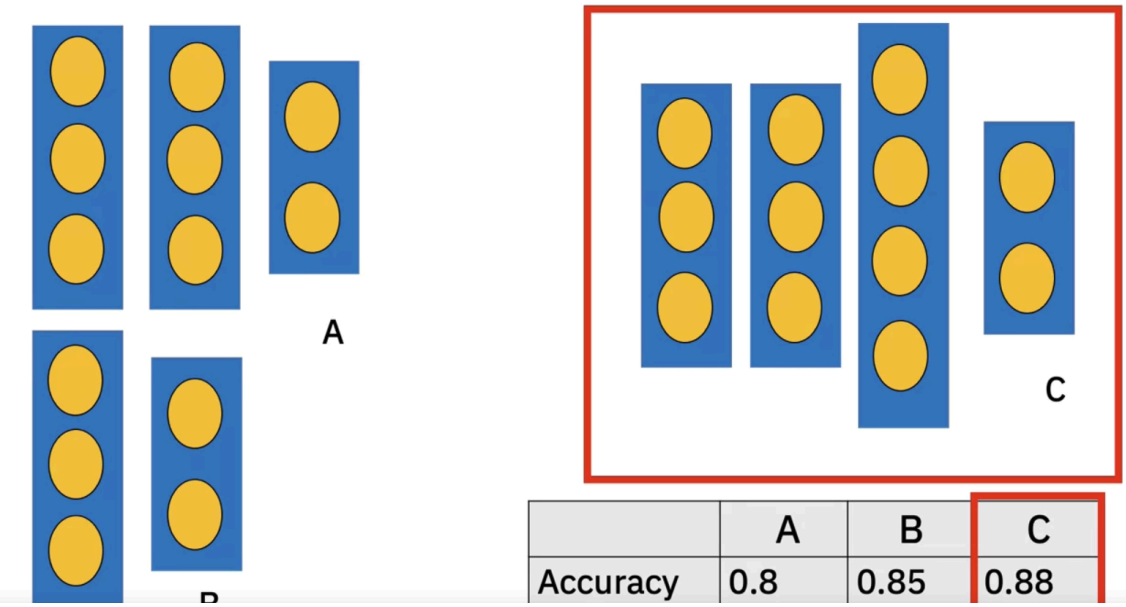


Example



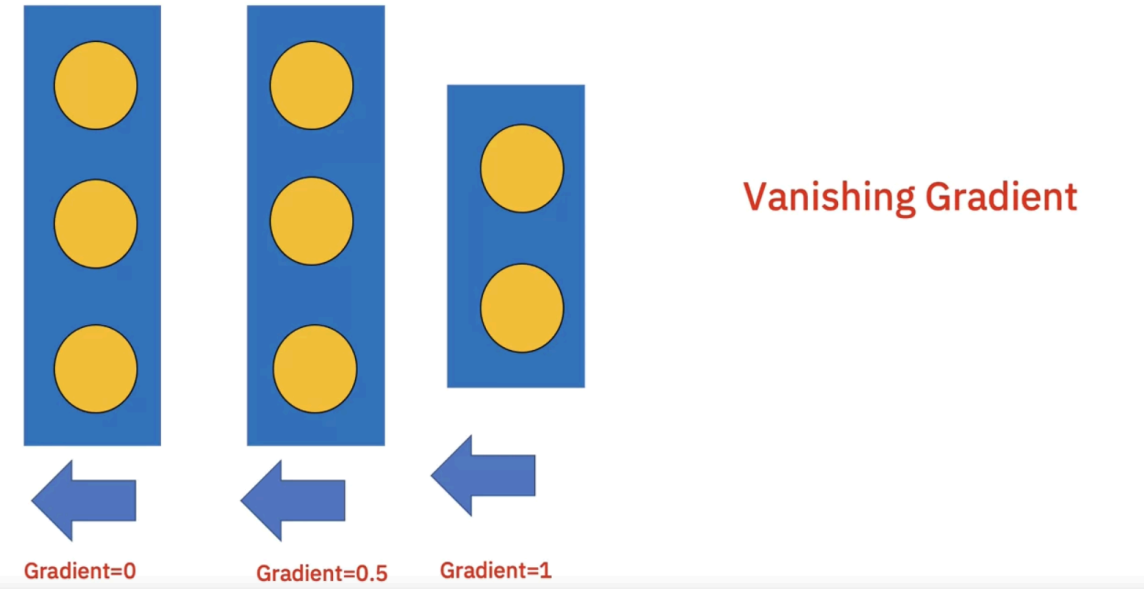
The image vector has (4) dimensions (After flattend), the hidden layer has (3) neurons, so the output has three dimensions. And so on.

One way to select a Fully Connected Neural Network Architure is to use the **validation data**. We select the Architure with the best **performance on the validation data**.



	A	B	C
Accuracy	0.8	0.85	0.88

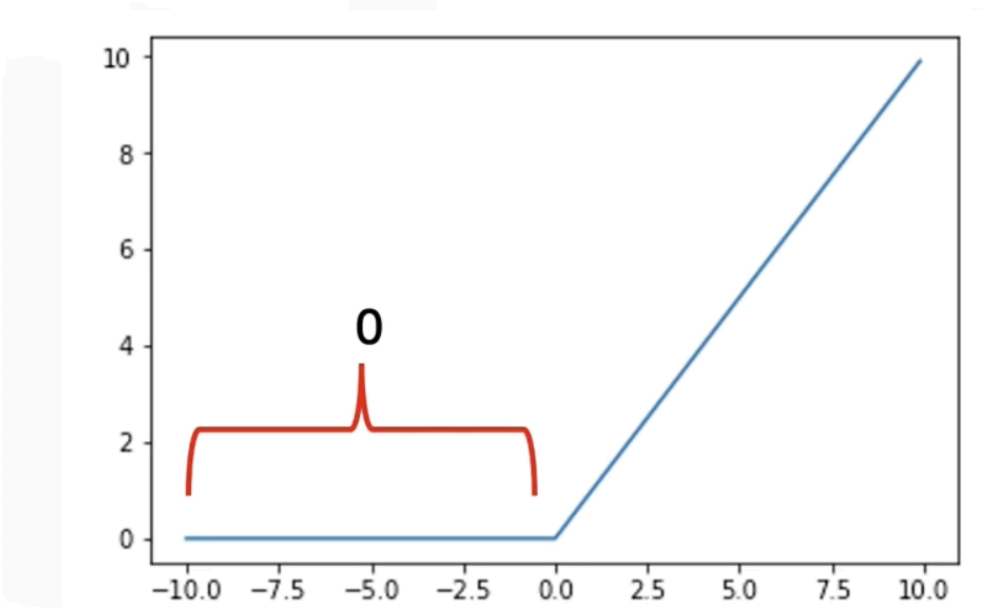
The deep network work **better**, but it **hard to train**, where we perform (Gradient Descent) to **obtain the Learning Parameters**. But the deeper the network, the **smaller the Gradient (Slope)** gets. This is called the (**Vanishing Gradient**). As a result, it's **hard** to train the **deeper layer.**



One of the drawbacks with using **Sigmoid** (Activation) function is the **Vanishing Graident.**

One Activation Function is the **Rectified Linear Unit Function** (ReLU), where the value of the ReLU function is (**0**) when the **input (Z) is less than (0)**. If the input (Z) is **larger then (0)**, the ReLU function **output equals its input (Z)**. **ReLU** is only used in the **Hidden Layers**.

Relu



Networks have **layers** that **help with the training**. Some methods are used in the network:

- **Dropout layers**: Helps with overfitting.
- **Batch Normalization**: Improve the training.
- **Skip Skip Connection**: Helps deeper network training by **connecting deeper layers** during training.

The Hidden Layers of Neural Networks (NN) replace the **Kernels in (SVM)**. Or, we use **Features like (H.O.G).**

The Art of Training neural networks

- Neural networks are trained in a similar manner to logistic regression and Softmax
- The Loss or cost surface is complicated making training difficult

---

Logistic regression:

•Type: Logistic regression is a type of linear model used for binary classification.
•Purpose: It predicts the probability that a given input belongs to a particular class.

Sigmoid Function:

•Type: The sigmoid function is a type of activation function used in neural networks.
•Purpose: It introduces non-linearity to the model and maps input values to an output range between 0 and 1.

ReLU (Rectified Linear Unit):

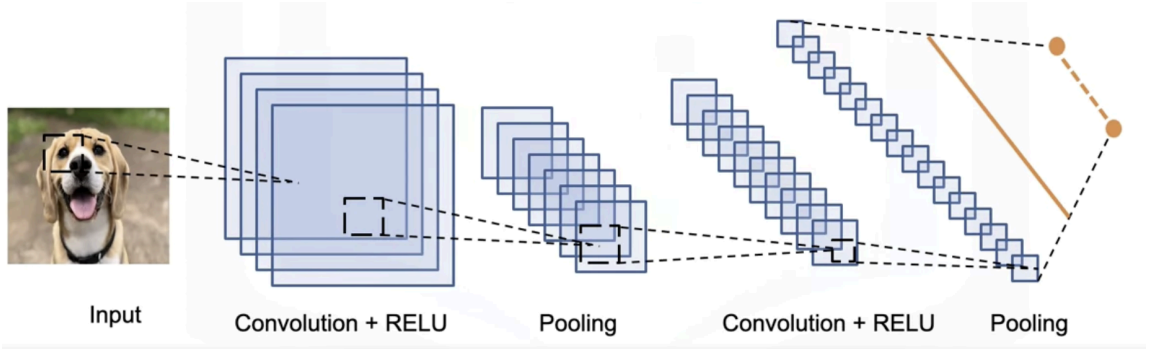
•Type: ReLU is another activation function used in neural networks.
•Purpose: It introduces non-linearity and helps mitigate the **vanishing gradient problem**, making it more suitable for deeper

---

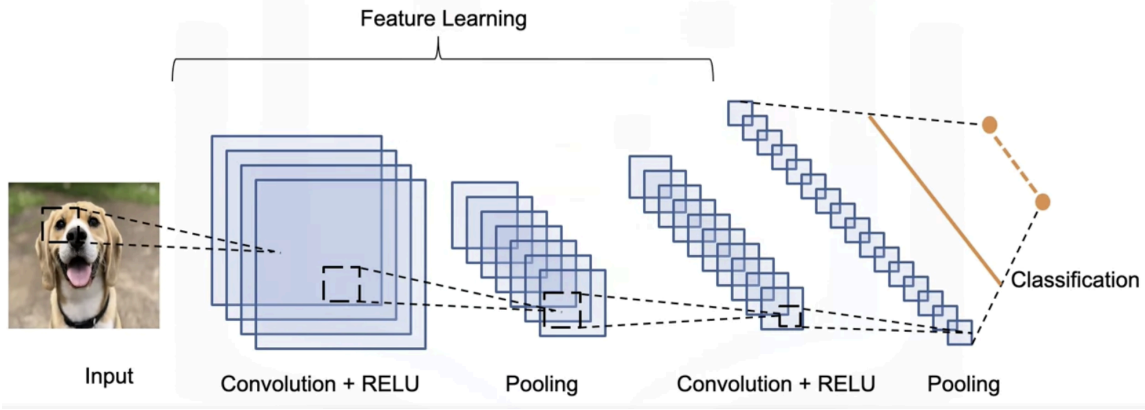
Training a neural network with **momentum** is a common technique used to **accelerate the training process and improve convergence**. Momentum helps in smoothing the updates of the weights by incorporating a fraction of the previous update in the current one. This helps in escaping local minima and dampening oscillations. Here's an overview of how momentum works in neural network training:

- ## Convolutional Neural Network:

A convolutional Neural Network (CNN) is a neural network with **special layers**, where the model classifies an image by **taking a part of the image**. Each input image will pass through a **series of convolution layers with filters**, **pooling layers**, **fully connected layers** and while **applying activation functions to classify an object**.

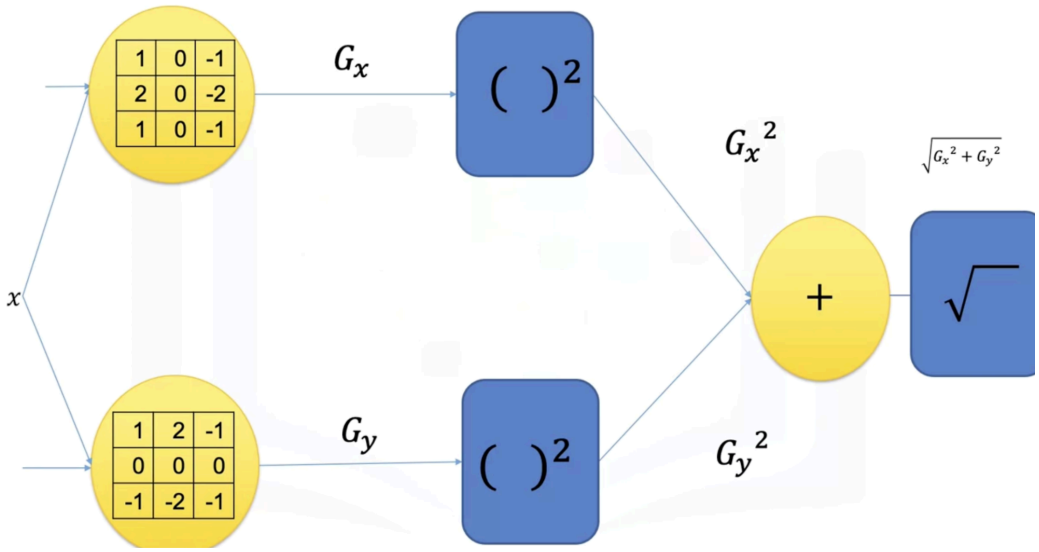


Convolution and pooling layers are the first layers used to extract features from an input - (**Feature Learning Layers**).



How CNN's Build Features:

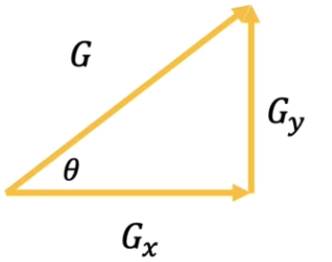
We can represent **(H.O.G)** with a diagram similar to Neural Network, as shown below



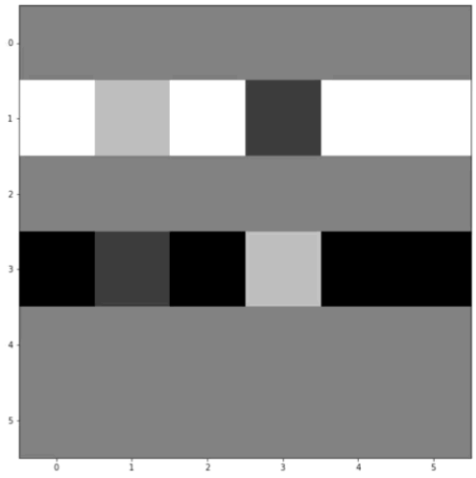
We replace the linear function with the convolution (**Kerenl**), and we have the squaring and the seuqare root operations. We represent the gradient equations below:

$$G = [G_x, G_y]$$

$$\|G\| = \sqrt{G_x{}^2 + G_y{}^2}$$

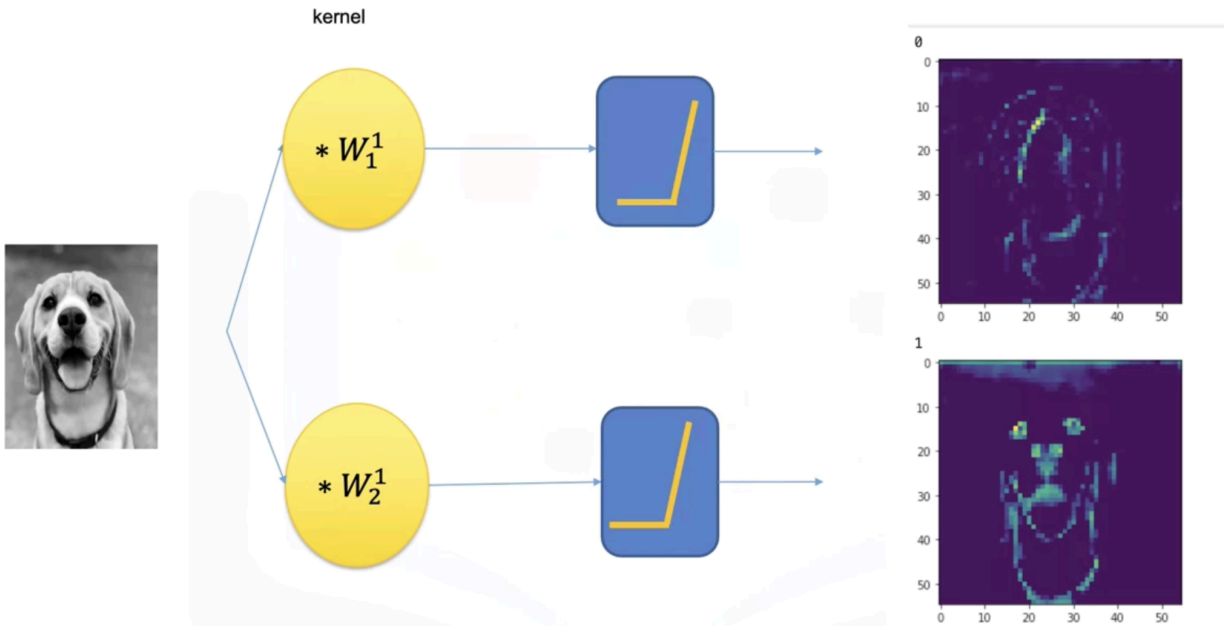


$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

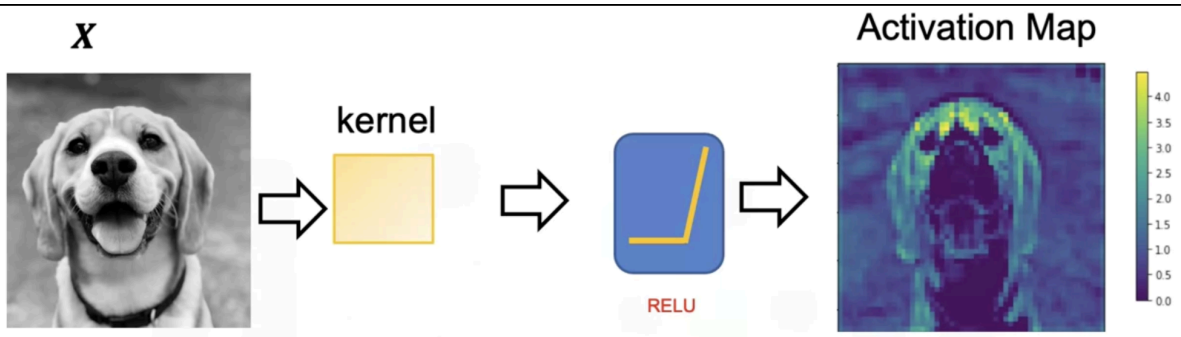


$\|G\|$

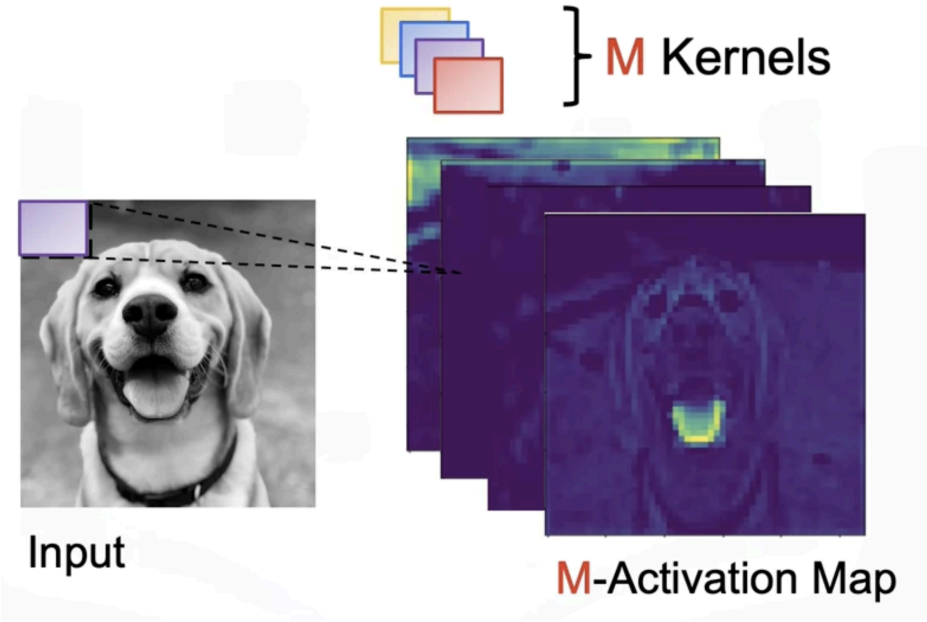
In CNN, we have neurons, but the (Kernels) are learnable parameters. The **activation function is Relu** is applied to each pixel, and instead of the activation, the output is an **activation map or feature map**.



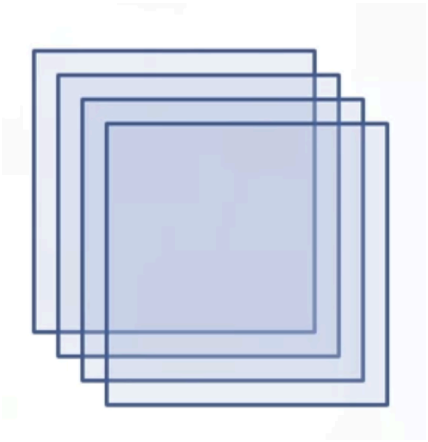
In the CNN, each kernel will detect a different property of the image similar to H.O.G kernel.



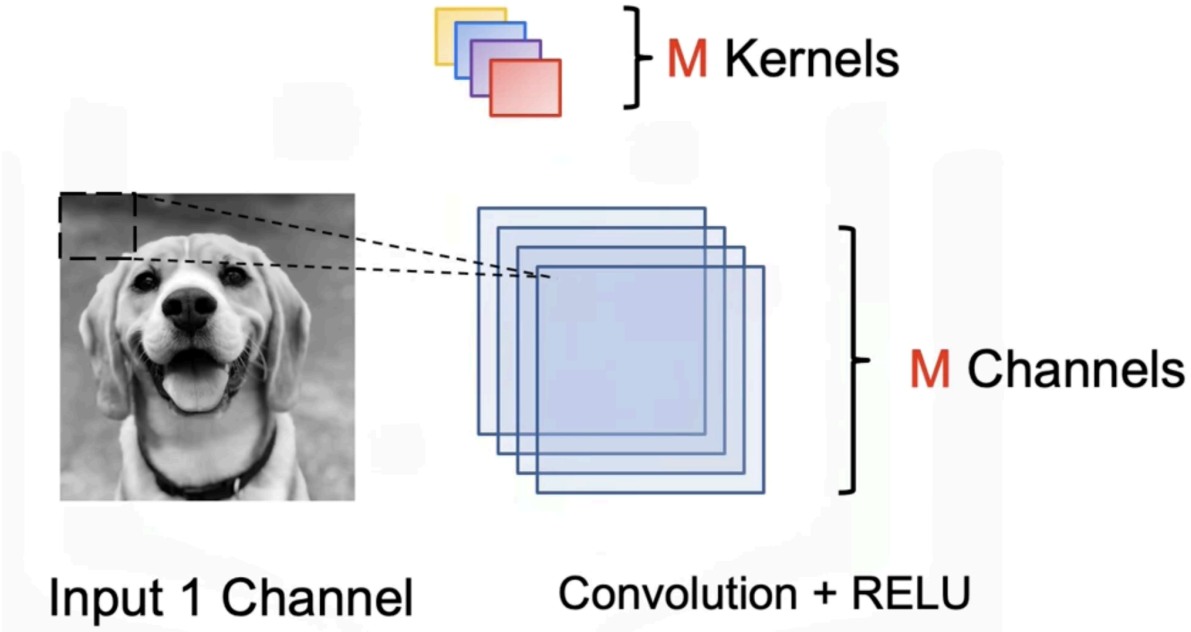
We use **multiple kernels** analogous to **multiple neurons**, if we have **M Kernels** we will have **M feature maps**.



Each activation map reprsent (**Convolution + Relu**), we represnt the activation or feature map with clear sequre

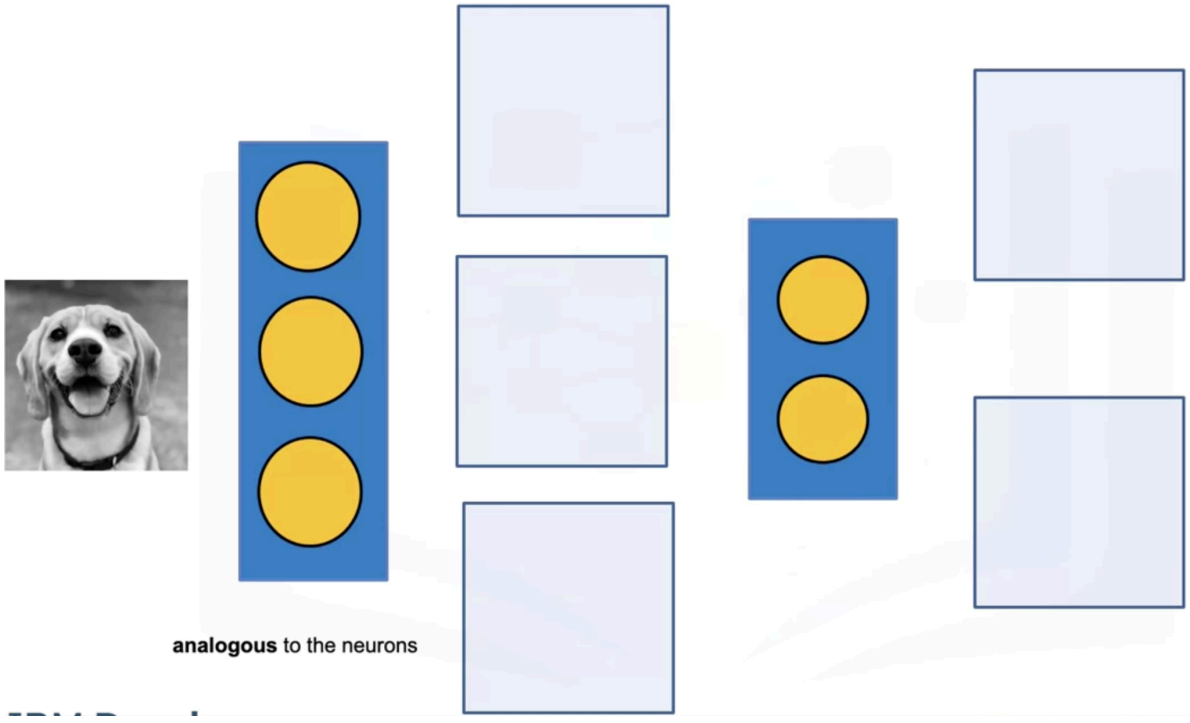


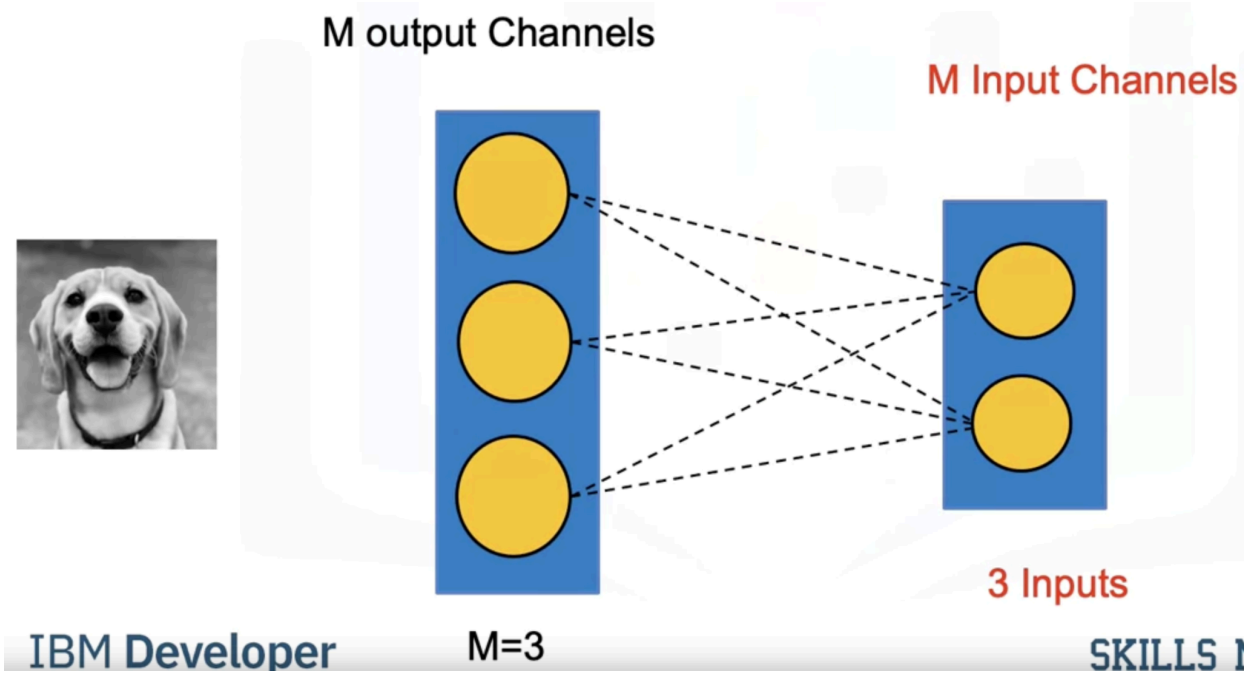
The Gray scale image input is **one channel**, the output is **M-Channels** (**based on the numer of Kernels**):



Adding layers:

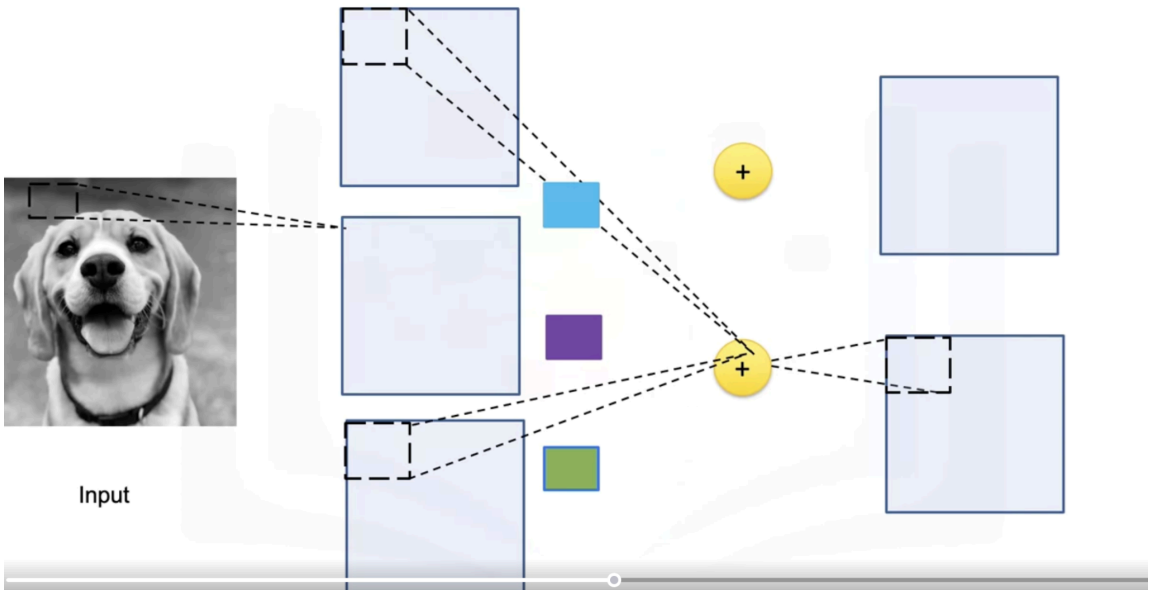
The number of **channels** is based on **how many neurons** in the layer,



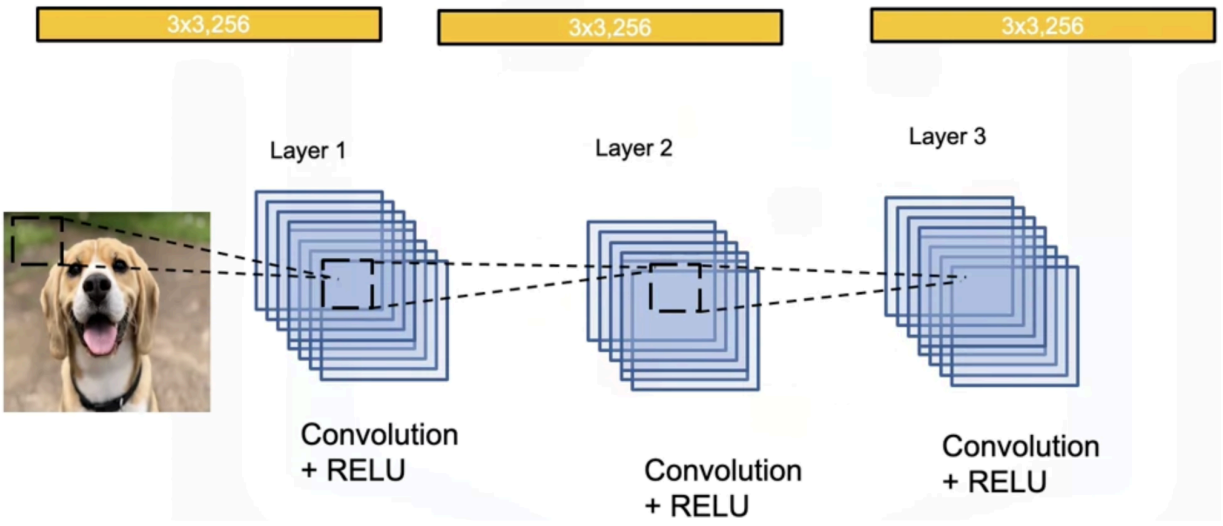


Each neurons will receive the **same output channels from the previous layers as shonw above**. The next layer will **apply convolution** kernel to each input, **and add them** together, then apply **activation function**.

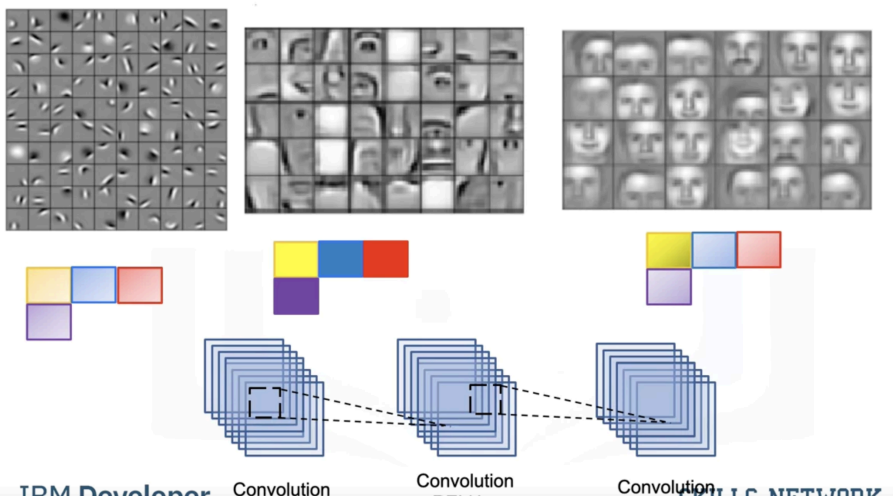
×



For colored image (3 channels), we **stack three CNN layers (3)** with kernels in yellow represent **Kerenl size** and the **number of channels**.

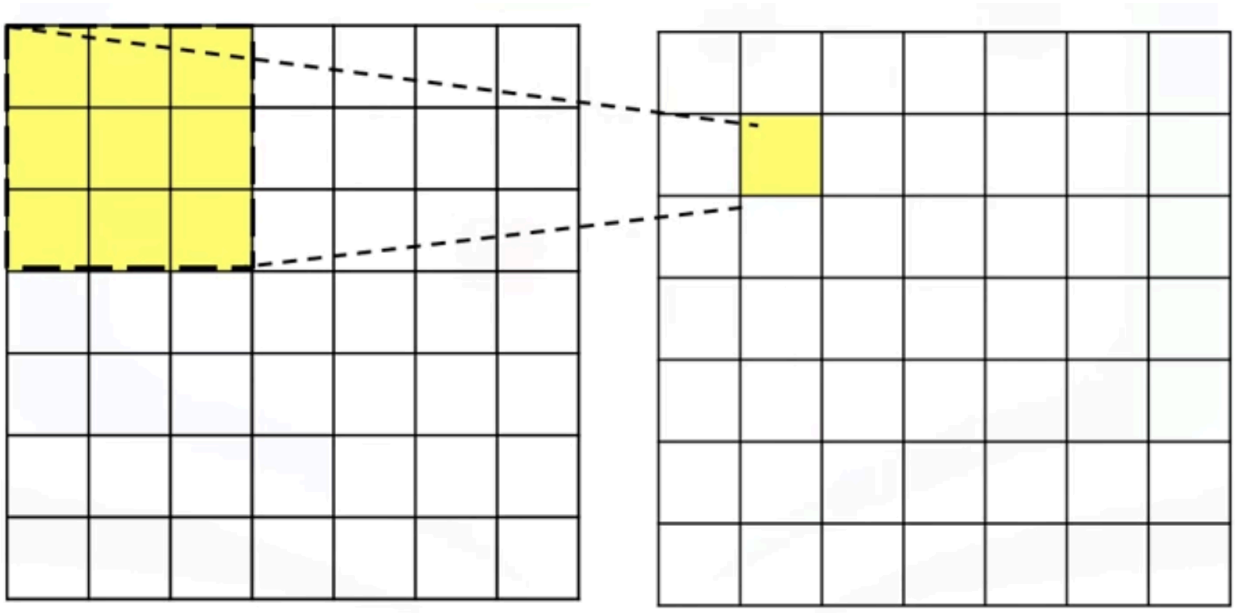


Each **layer detects specific features**, such as face parts, edges and corners, and faces. **Adding more layers**, builds more **complex features**.



Receptive Field:

Receptive Field is is the size of the region in the input that **produces a pixel value in the activation map**.

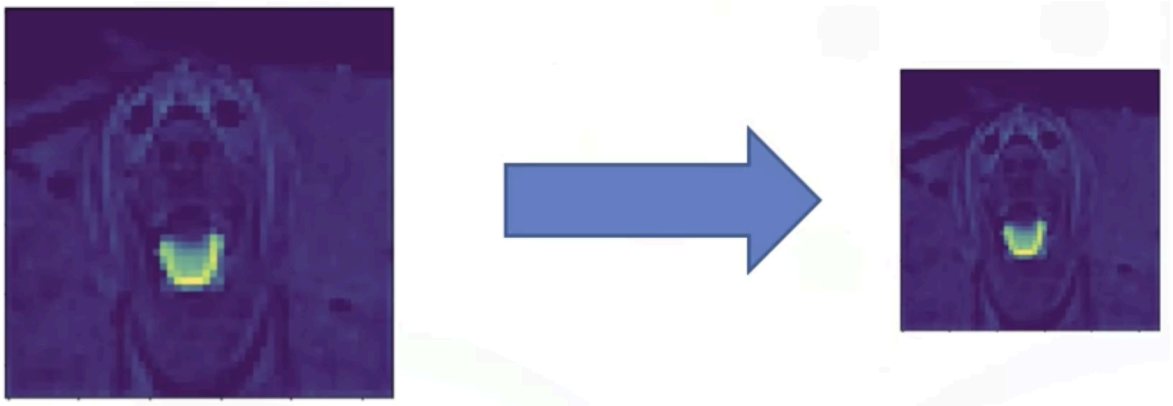


- The **larger the Receptive Field**, the **more information** the **activation** map contains about the **entire image**.

- We can increase the **Receptive Field** by adding **more layers**, this **requires less parameters** than increase the **kernel size**.

Pooling:

Pooling helps to **reduce the number of parameters**, **increases the receptive field** while **preserving the important features**. It's **resizing** the image.



Max pooling is the most popular type of pooling. The example below take **Max pooling with 2x2** dimensions:

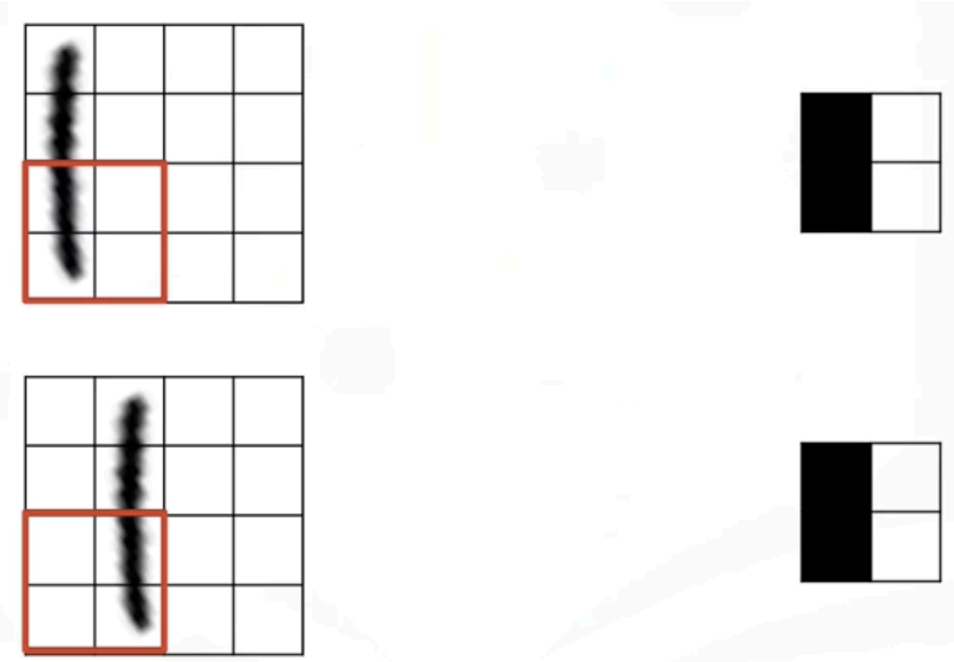
1	0	0	0	0	1
0	1	6	0	1	6
3	0	1	1	0	0
1	0	3	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

Max pooling with 2x2

6	9
3	1

- It gets the **maximum pixel values**, and we get a **smaller feature map**.
- Pooling makes **CNN immutable to to small** changes in the image.

The figure below shows how Pooling **immutable** to small changes in the feature mapping.

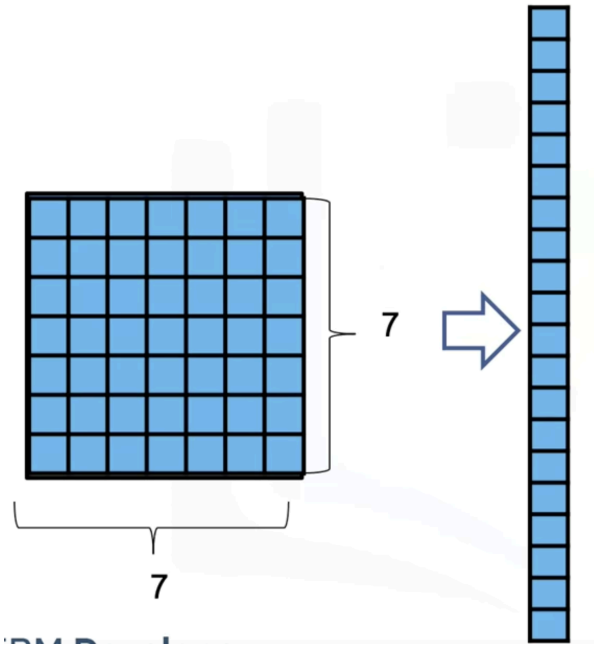


We can see that the small shift in the two images has the same feature mapping.

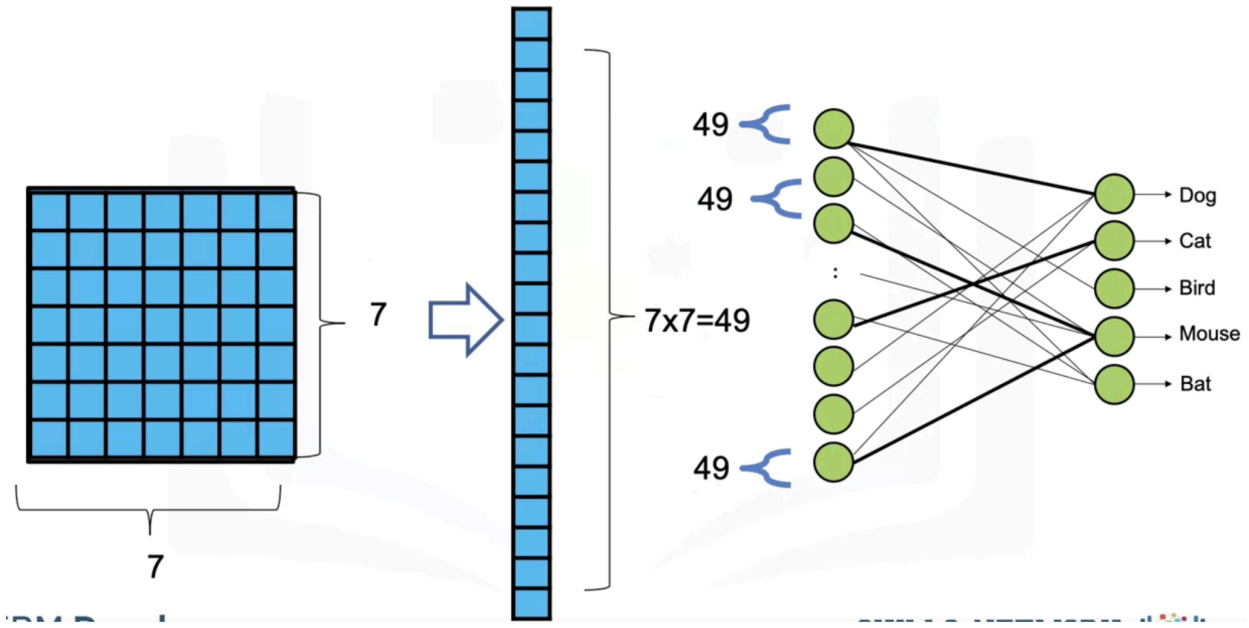
Flattening and the Fully Connected layers:

Flattening and the Fully Connected layers is **flatten** or **reshape** the output of the **Feature Learning layers** and use them as an **input** to the **fully connected layers**.

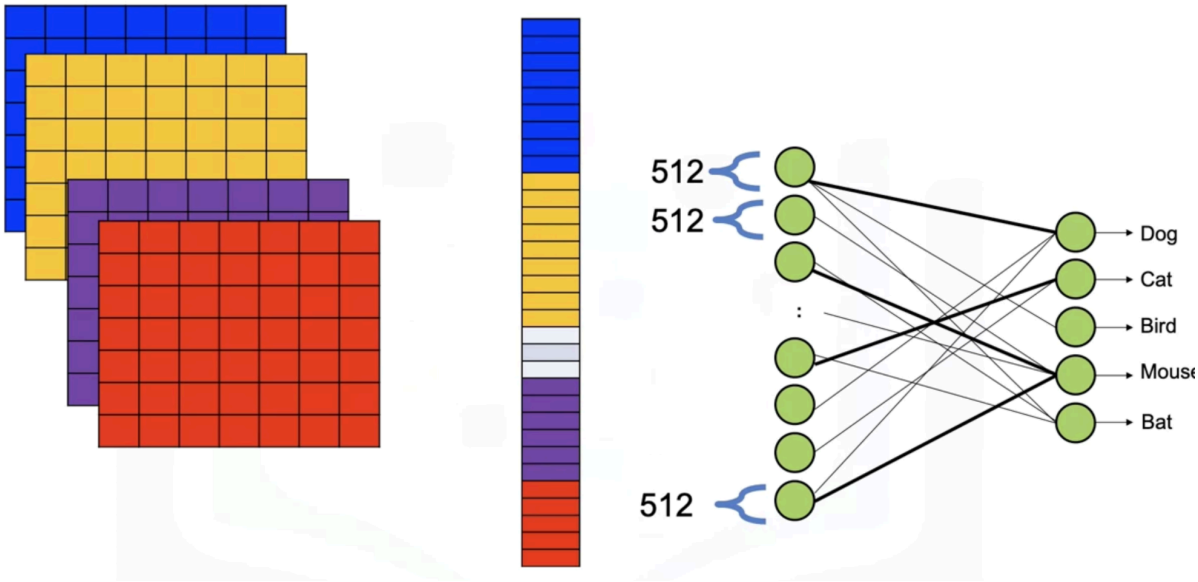
The example below shows the output of a Max Pooling layer size (7x7), we **flatten tham into a feature vector** and this is an **input to the fully connected layer**.



Each Neurons has an input dimensions of the flatten output, which is (7x7=49):

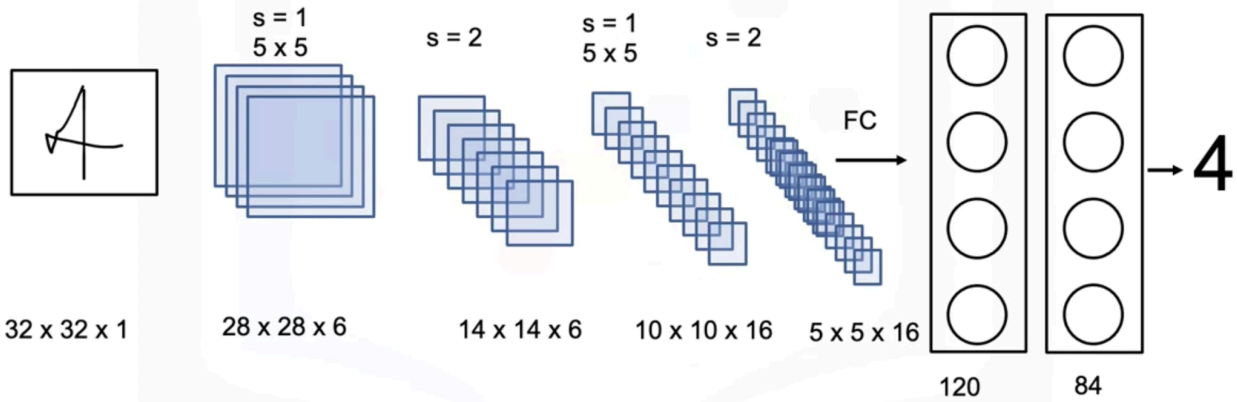


For more channels, we apply the same concept, if each layer (4x4) and we have (32) channels, the feature vector is (4x4x32=512) elements.

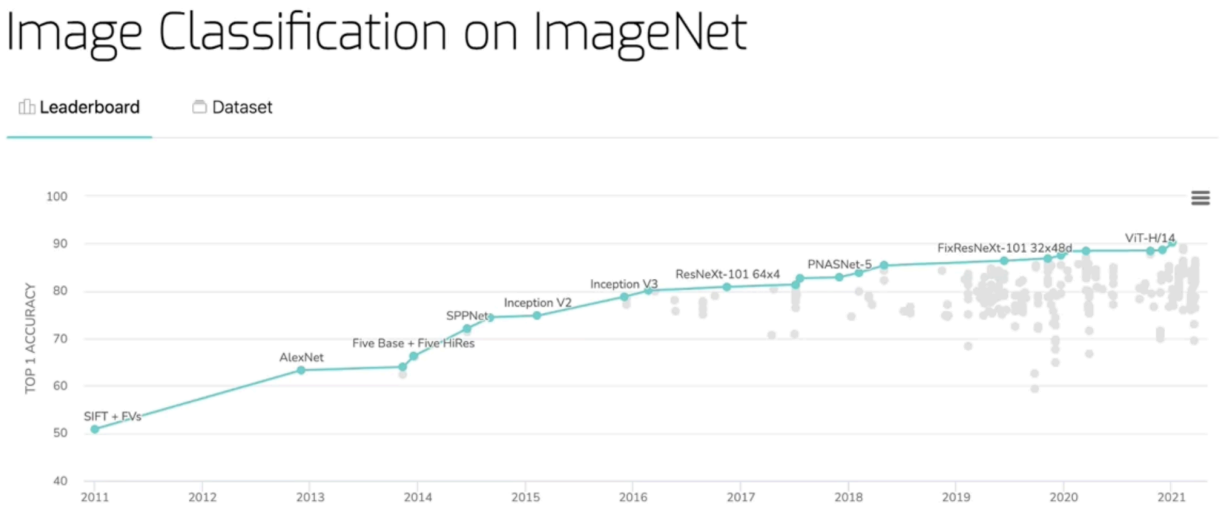


- **<u>CNN Architecture:</u>**

- **LeNet-5:**



- **AlexNet:**



If you want to compare any image classification methods, you compare the classification accuracy on a dataset. **ImageNet** is a **benchmark dataset** i.e **this is the one that everyone uses to see who has the best image classification method**.

- **ResNet:**

As CNN gets deeper, the **vanishing gradient** became a problem, ResNet introduce **Residual Learning**. **Residual Learning or Skip connection** allows the gradient to bypass different layers **improving performance**,

ResNet

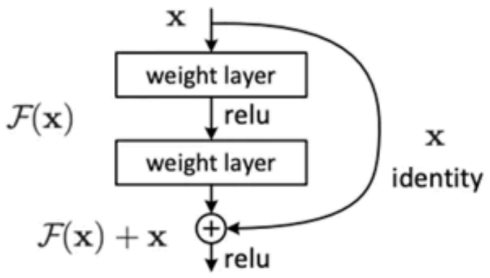
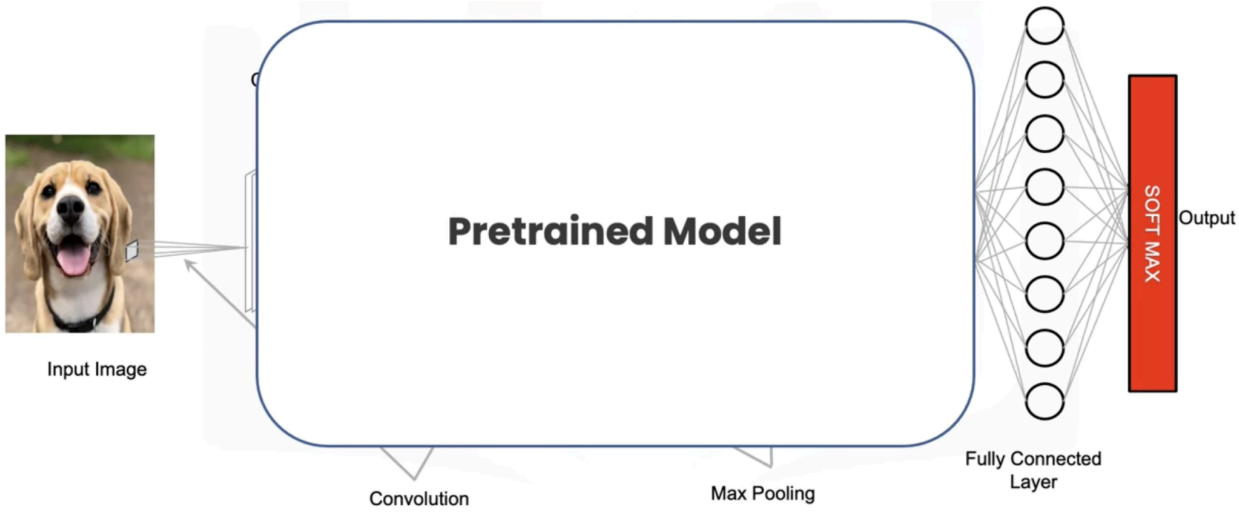


Figure 2. Residual learning: a building block.

Transfer learning:

Transfer Learning is where you **use pre-trained CNN to classify an image**.

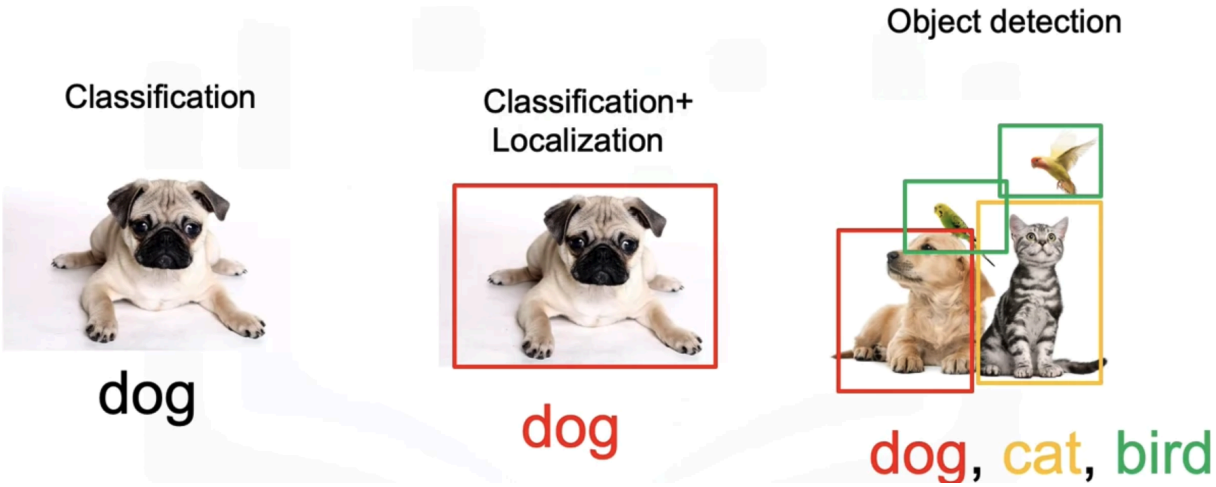


We can replace the **SoftMax with our own SoftMax layer**. The number of neurons is **equal** the **number of classes**. Then, we train the SoftMax layer on the **dataset we would like to classify**.

The pre-trained CNN's have been trained with vast amount of data. Depending on the size of your dataset, **you can use SVM** instead of the **SoftMax layer**.

Object Detection:

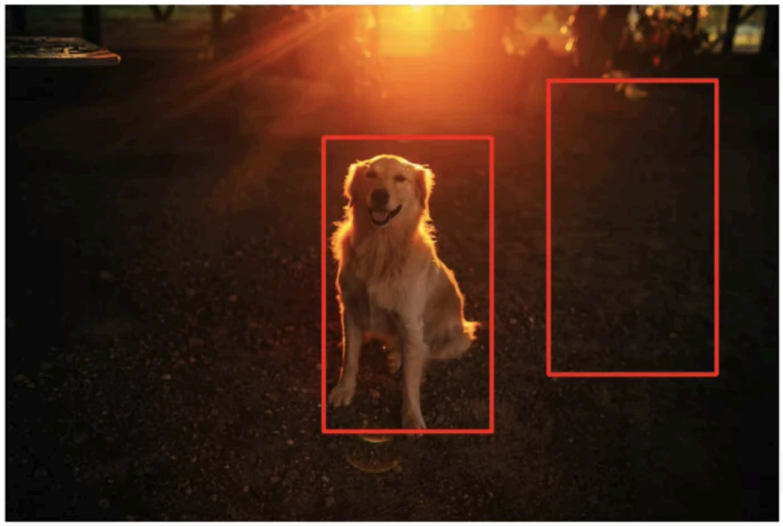
Image classification predicts the **class of an object** in an image. Classification and Object Localization **locate** the presence of an **object** and **indicate the location with a bounding box**. Object Detection locates **multiple objects with a bounding box** and their classes.



- **Sliding Windows:**

In object detection, we find all the possible objects in the image. To achieve this, we use an algorithm known as **Sliding Windows Detection**.

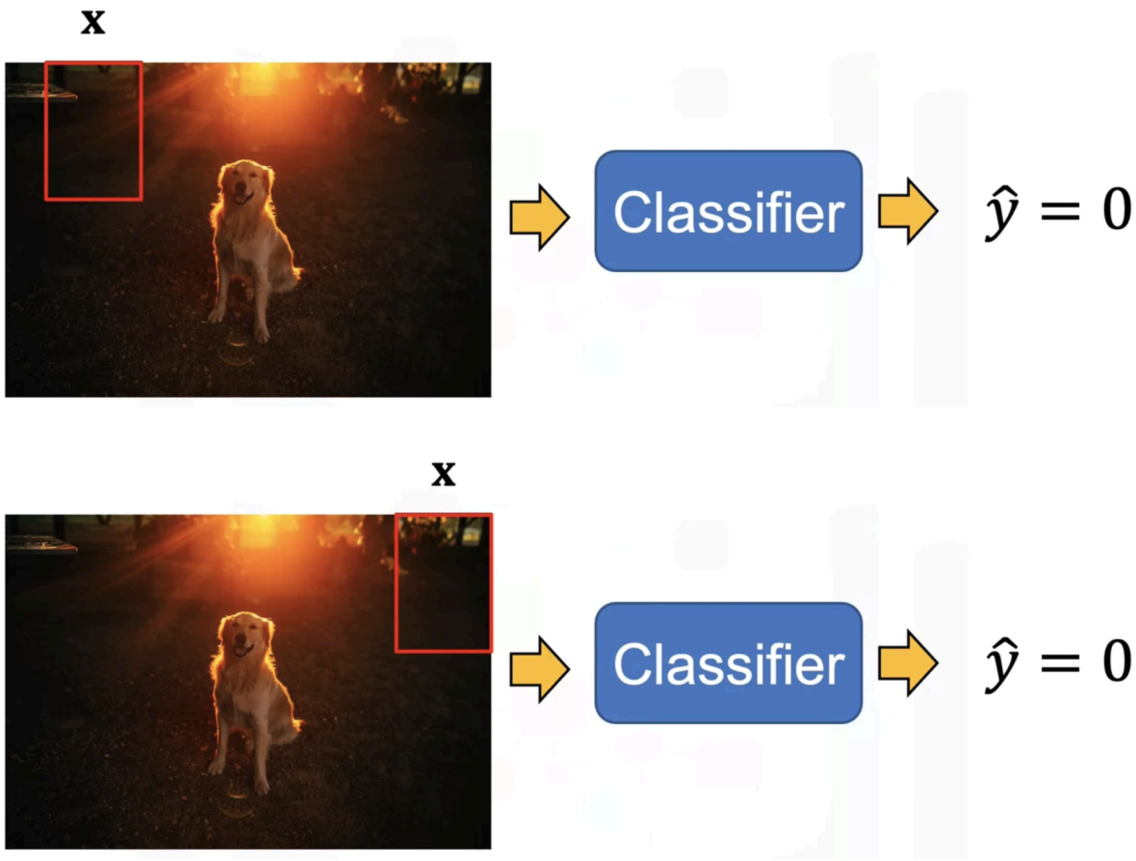
If we want to detect an object, we **consider a fixed window size**, where the detected object is **considered a sub-image** with **object class**, and the other sub-images would be **classified as background**.

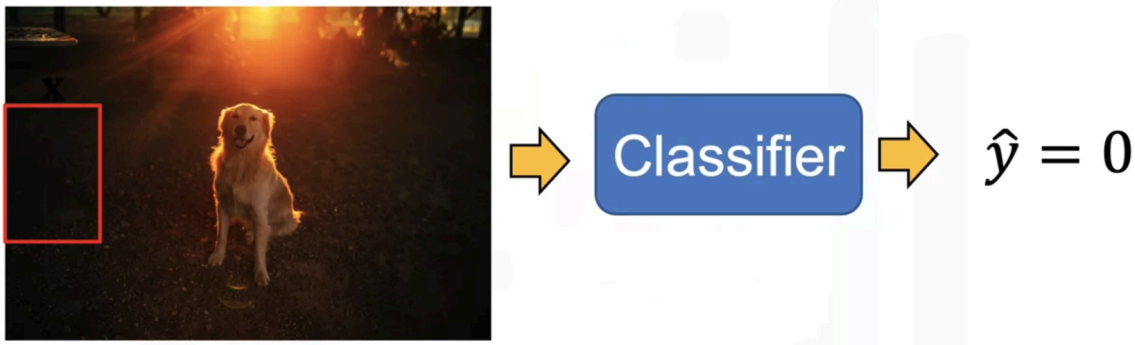


$dog\ y = 1$

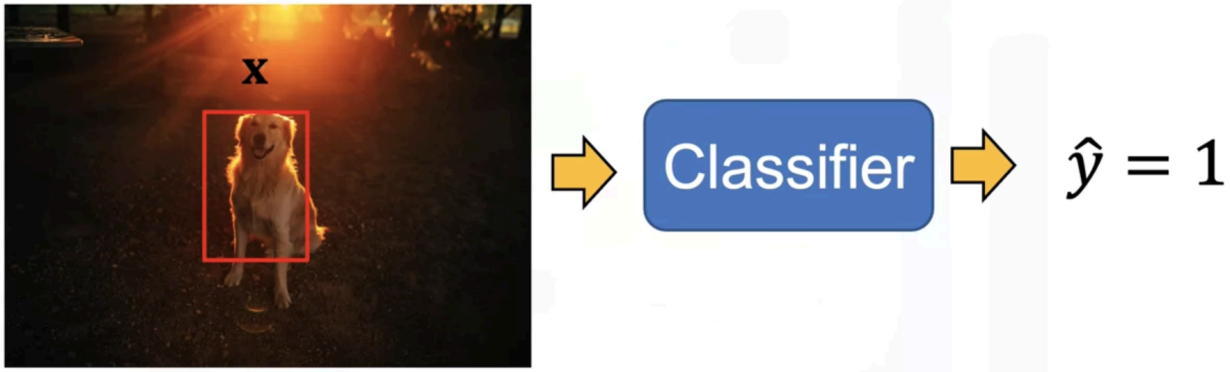
$background\ y = 0$

We start in one region in the image, **classify that sub-image**, and **then we shift the window and classify the next sub-image**. We repeat the process, and when we get to the horizontal border, we **move a few pixels down in the vertical** direction and repeat the process.





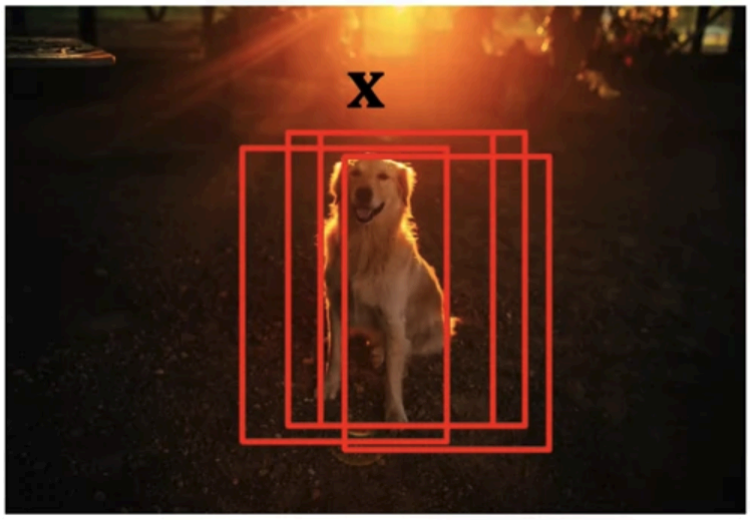
When an **object occupies most of the window**, it will be **classified as the object class**.



- **Problems of Sliding Window**:

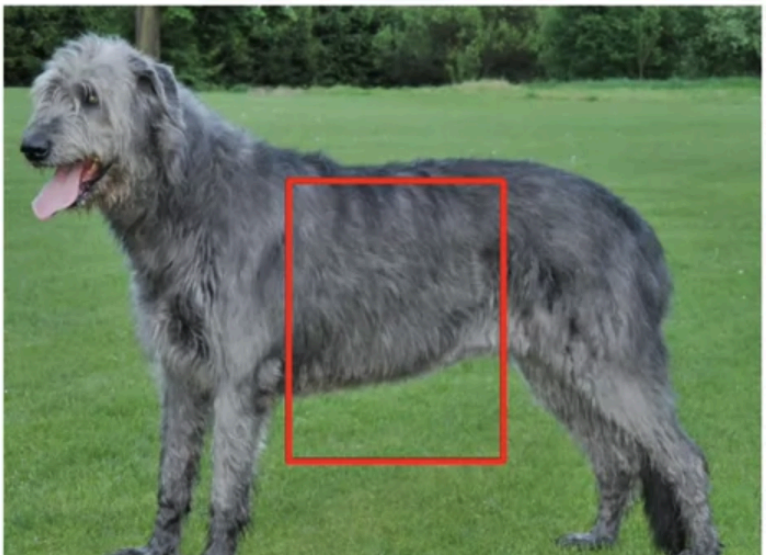
1- Object detection often outputs many **overlapping detections**.

- Overlapping Boxes



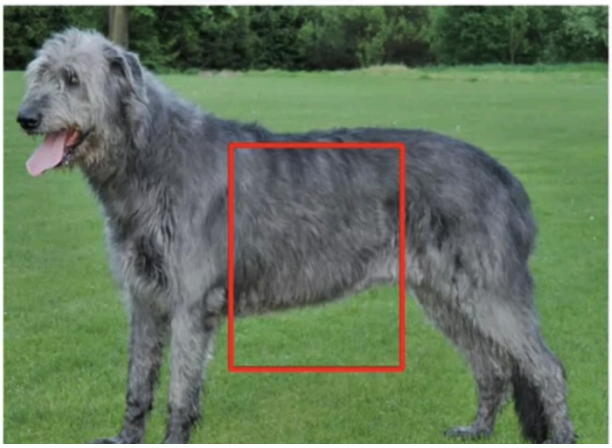
2- The issue of the **object size**, where the same object can come in different sizes.

- Object sizes



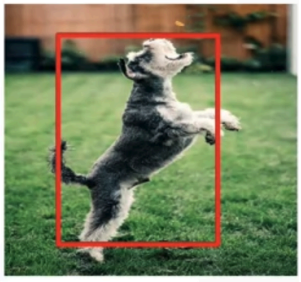
One way to solve this is to **reshape/resize the image**.

- Object sizes



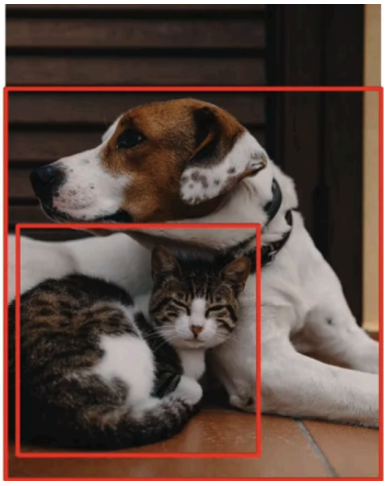
3- Objects can have **different shapes**, r**eshaping/resizing the image is one solution**.

- Object shape



4- **Overlapping Objects** in pictures.

- Overlapping objects

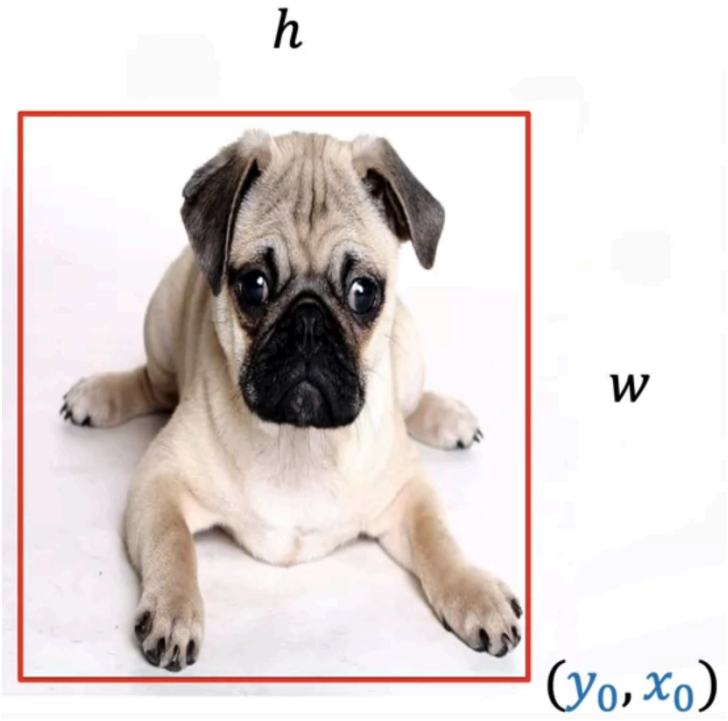


- **Bounding Box:**

A bounding box is another method for object detection, it can be **used independently with sliding windows** or **other advanced methods**.

The bounding box is a rectangular box that can be **determined by**:

1. Determined with the **lower-right corner** of the rectangular with **coordinates (y0,x0)** and the width (w) and height (h).

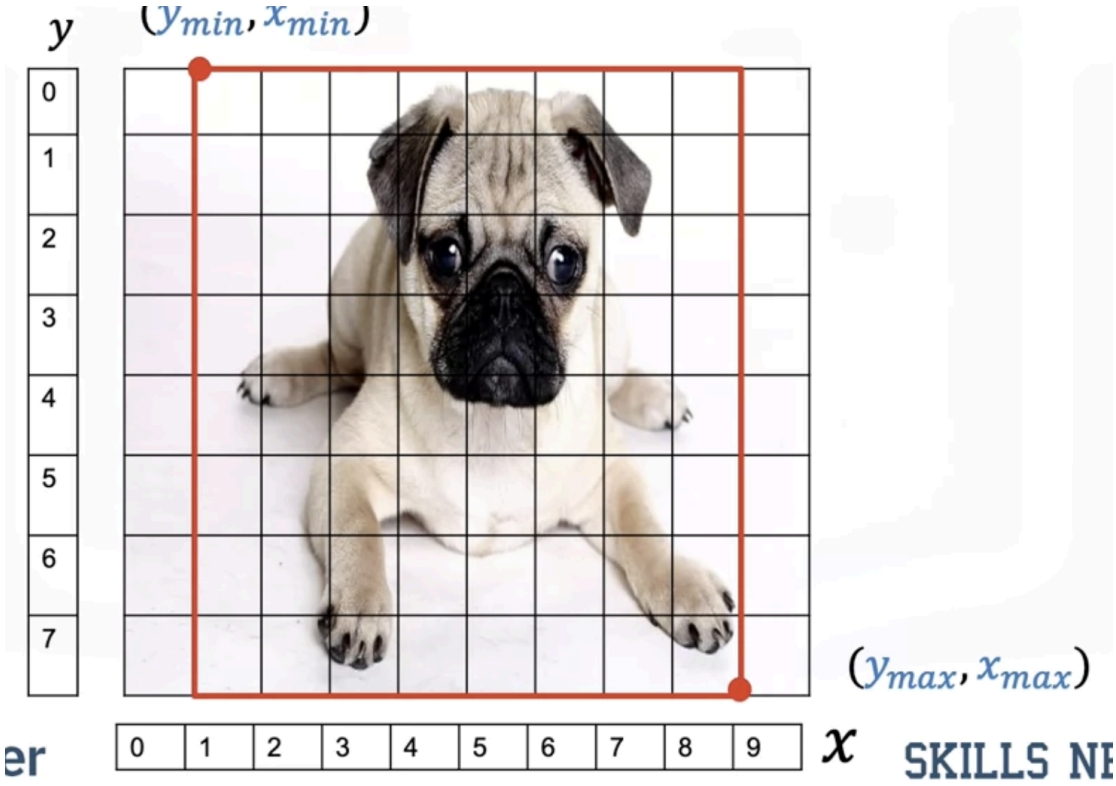


2. Determined by the coordinates in the **upper-left corner (y-Min,x-Min)**, and the **lower-right corner the (y-Max,x-Max)** dd

The **bounding box coordinates**:

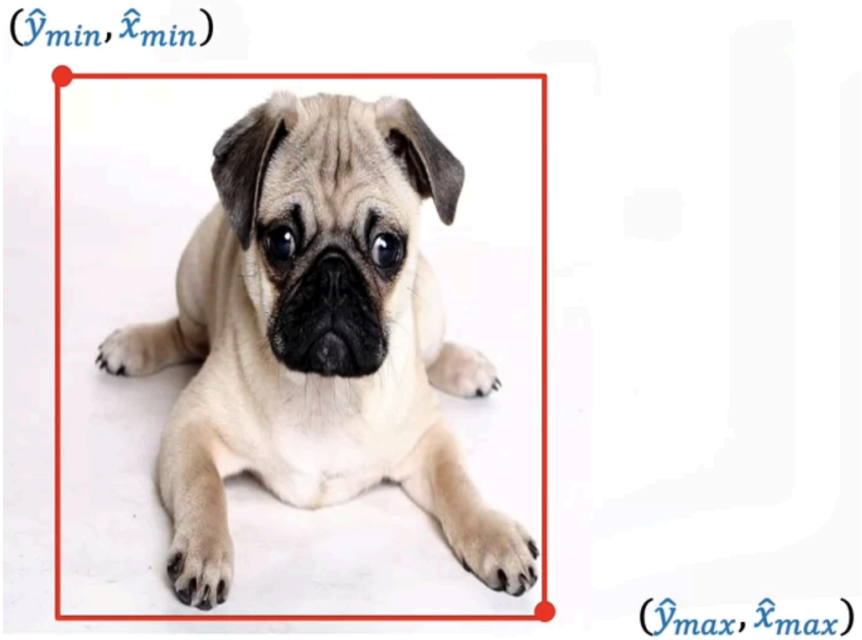
$$box = [(y_{min}, x_{min}), (y_{max}, x_{max})]$$

Example:



In the above example, the **upper-left (1,7)** and the **lower-right (7,8)**. The goal of object detection is to **predict these points**, so we add a **"Hat" to indicate it's a prediction**.

$$\widehat{box} = [(\hat{y}_{min}, \hat{x}_{min}), (\hat{y}_{max}, \hat{x}_{max})]$$



- The Bounding Box Pipeline:

In classification, we have the **class (y)** and **image (x)**.

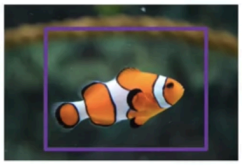
$$y_1, \mathbf{x}_1$$

In object detection (**Bounding Box**), we have a dataset of classes and their Bounding Box.

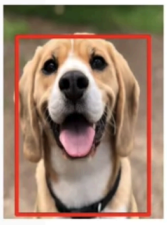
$$y_1, \mathbf{x}_1, box_1$$

$$y_4, \mathbf{x}_4, box_4$$

$$y_2, \mathbf{x}_2, box_2$$

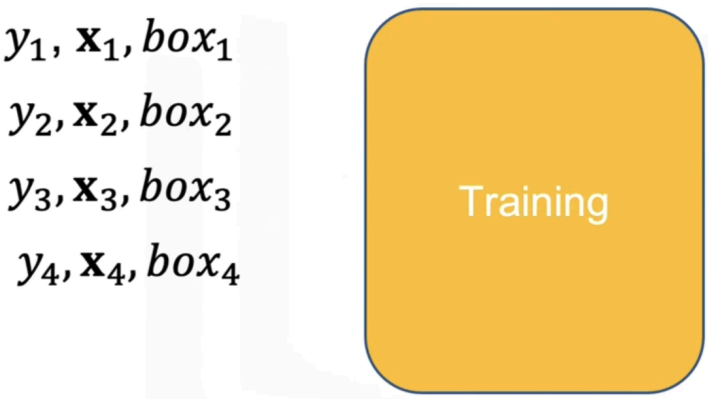


$$y_3, \mathbf{x}_3, box_3$$

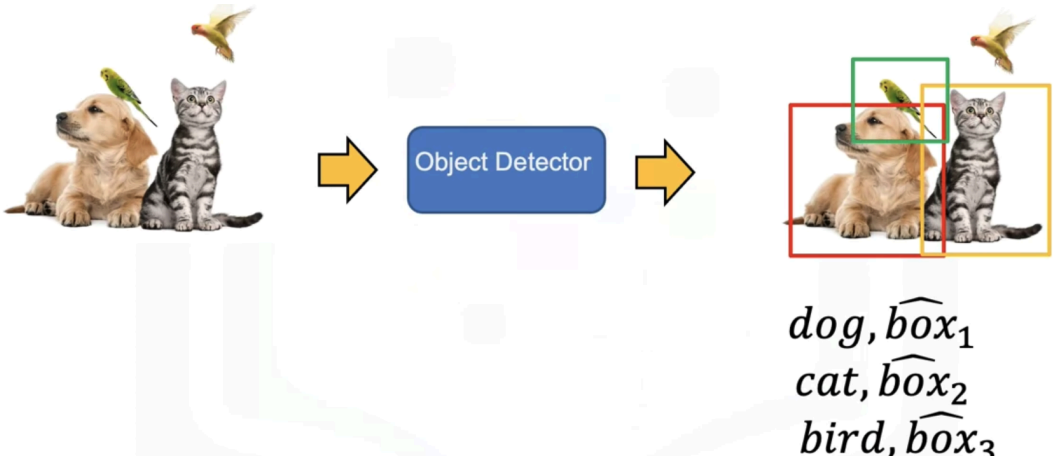


$$y_5, \mathbf{x}_5, box_5$$

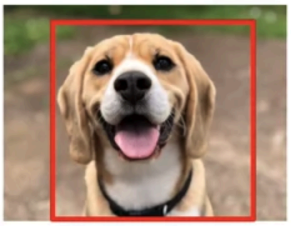
Similar to classification, we use the dataset to train the model, and we **include the box coordinates**.



The result is an object detector with **updated learning parameters**.



Object detection algorithms **provide** a **score** letting you know how confident the model prediction is.

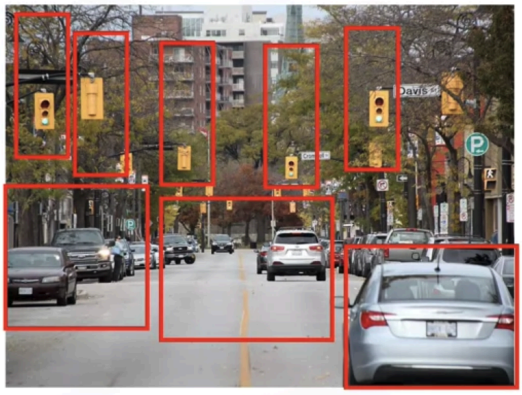
Score	0.5	0.99
$\hat{y}$	dog	dog
		

- **<u>Object Detection with Haar Cascade Classifier:</u>**

It is a **<u>machine learning method</u>** where a <u>cascade function</u> is **<u>trained on a large number of positive images</u>** which means that it **<u>includes the object</u>** we are trying to detect and **<u>negative images i.e the background.</u>**

Haar – Cascade Classifiers

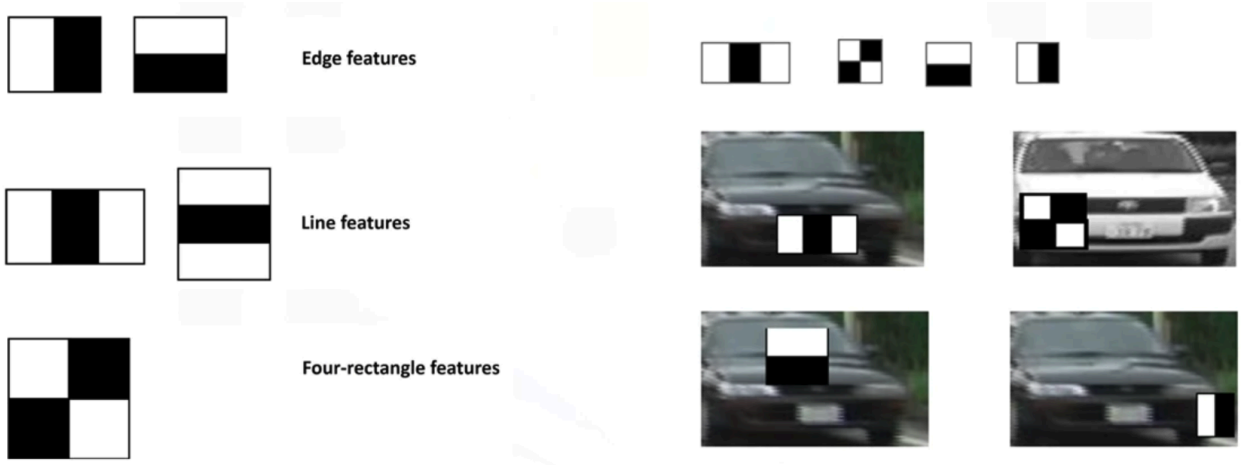
- Haar cascade was proposed by P. Viola and M. Jones in 2001
- It is a machine learning method
- Trained on both positive and negative images



- ## Haar Feature-Based Cascade Classifier:

1. It uses the idea of **the Harr Wavelets** in the Harr feature cascade. **Harr Wavelets** are **Convolution Kernels** used to extract features.

2. Harr Wavlets extracts information about (**Edges, Lines, and Diagonal Endges**):



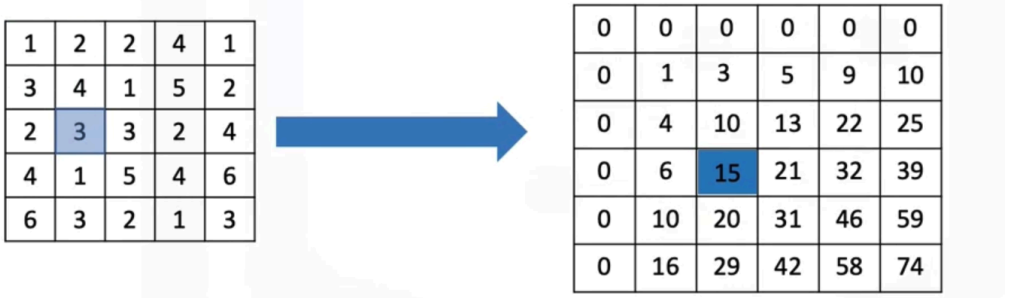
- ## The Harr Feature-Bassed Cascade Classifer Process:

1. The **Integral Image** is a concept that uses **the cumulative sum of pixels above** and uses **the left of the current pixel cell**. The concept takes in the pixels of an input image as

1	2	2	4	1
3	4	1	5	2
2	3	3	2	4
4	1	5	4	6
6	3	2	1	3

The sum of the green pixels:

- The Integral image concept



1	2	2	4	1
3	4	1	5	2
2	3	3	2	4
4	1	5	4	6
6	3	2	1	3

0	0	0	0	0	0
0	1	3	5	9	10
0	4	10	13	22	25
0	6	15	21	32	39
0	10	20	31	46	59
0	16	29	42	58	74

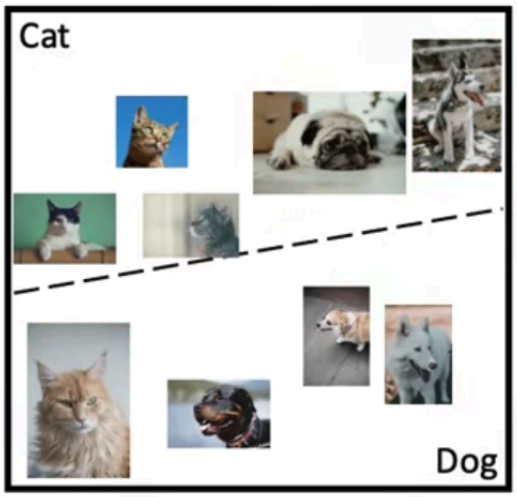
Integral image is a concept that uses the cumulative sum of pixels above and to the left of the current pixel cell

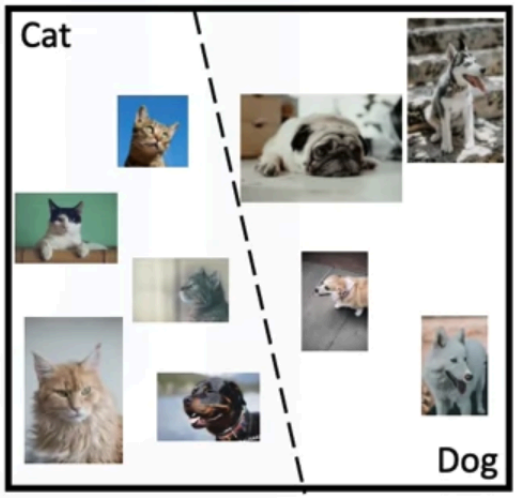
2. The algorithm **selects a few important features** from a large set to give highly efficient classifiers by employing the use of an **AdaBoost**, The idea is to **set weights** to both **classifiers** and **samples** in a way that forces classifiers to concentrate on observations that are difficult to correctly classify. Therefore, it selects only those features that help to improve the classifier's accuracy by constructing a strong classifier **which is a linear combination of weak classifiers**.

An AdaBoost classifier is used to reduce the number of features:

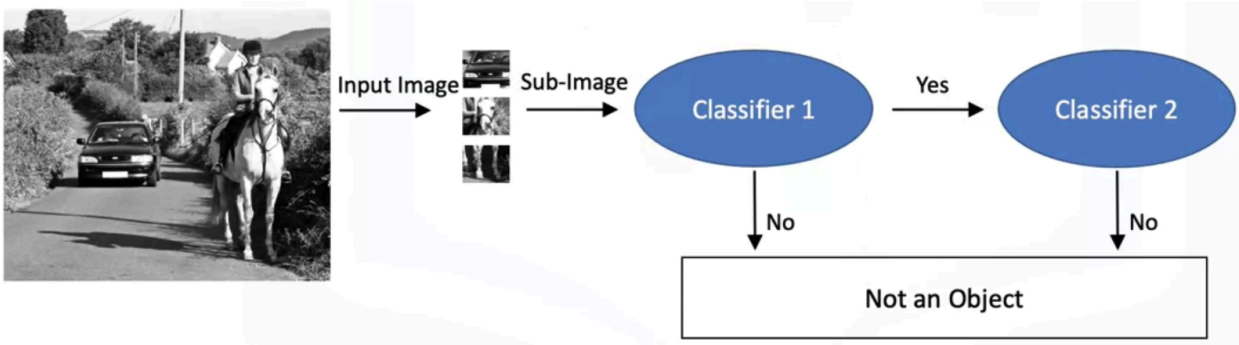
1. A weak classifier is made on top of the training data based on the weighted samples
2. It selects only those features that help to improve the classifier accuracy
3. AdaBoost cuts down the number of features significantly

The algorithm **selects a few important features** from a **large set to five highly efficient classifier by (AdaBoost)**. It minimized the number of errors by traying different linear classifier which build a strong classifier.



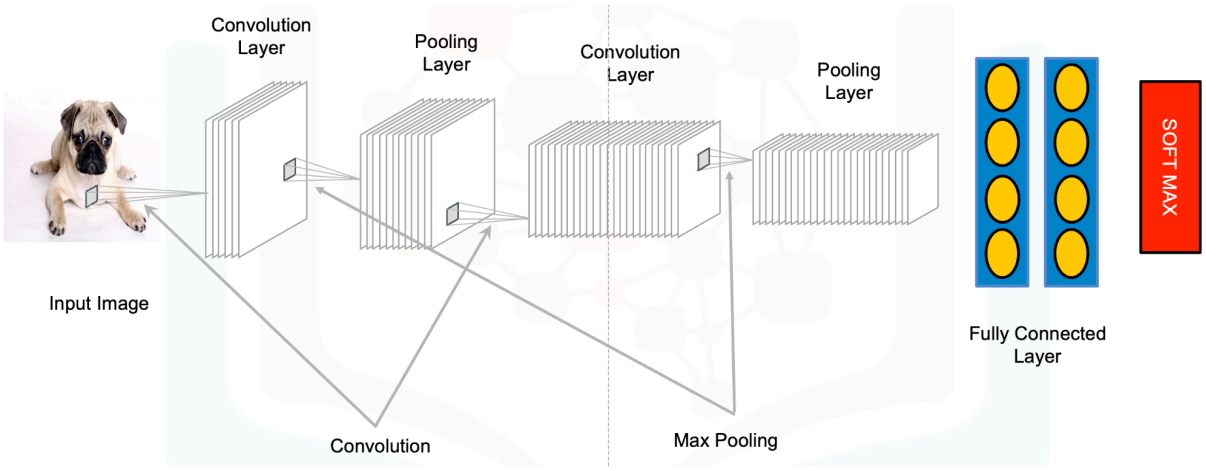


3. Cascades of Classifiers are **used**, this classifier groups sub-images from the input images in **stages**, and disregard any region that does not match the object it is trying to detect. The **classifier groups the features** into **multiple sub images**.

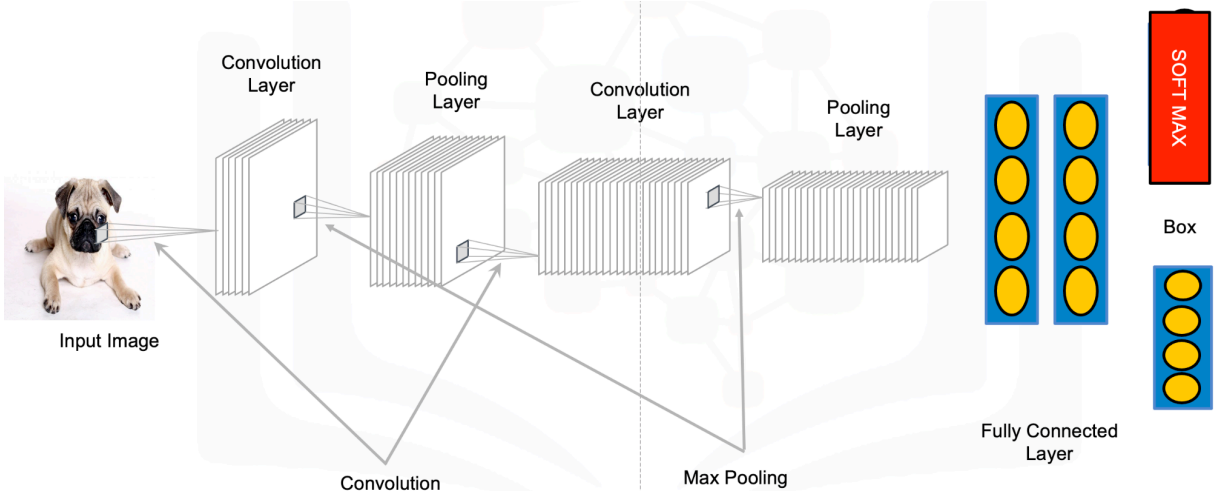


- ## Object Detection with Deep Learning:

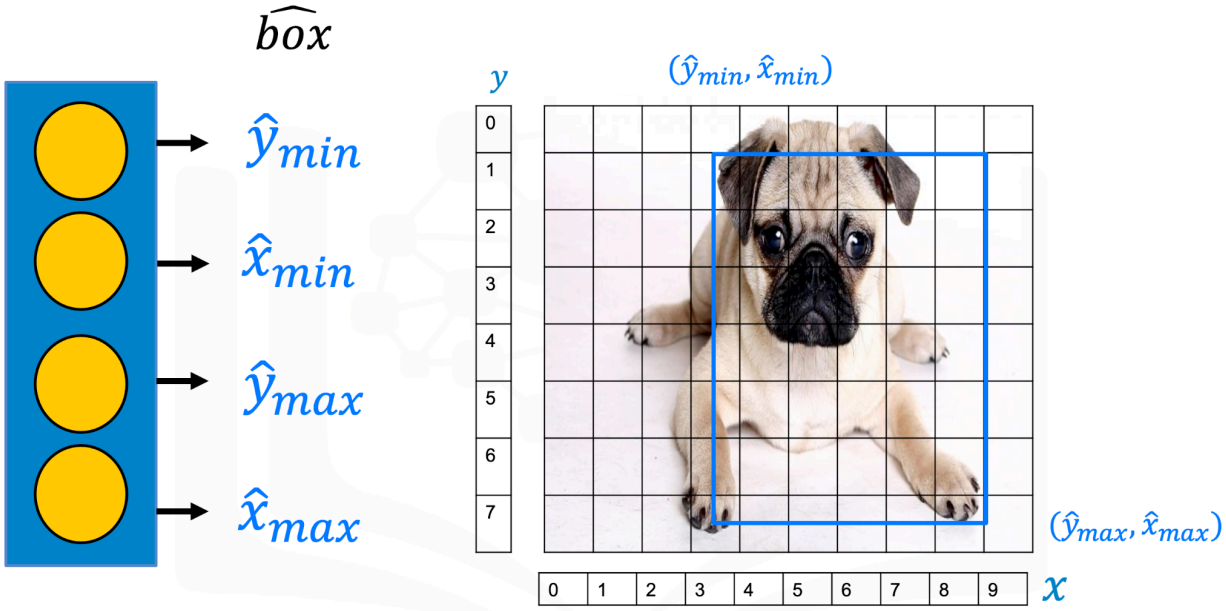
We can use a **sliding window** and a **CNN to detect an object**. We classify the image as a background or part of another class. We typically use the popular CNN Architectures pre-trained on ImageNet as shown here:



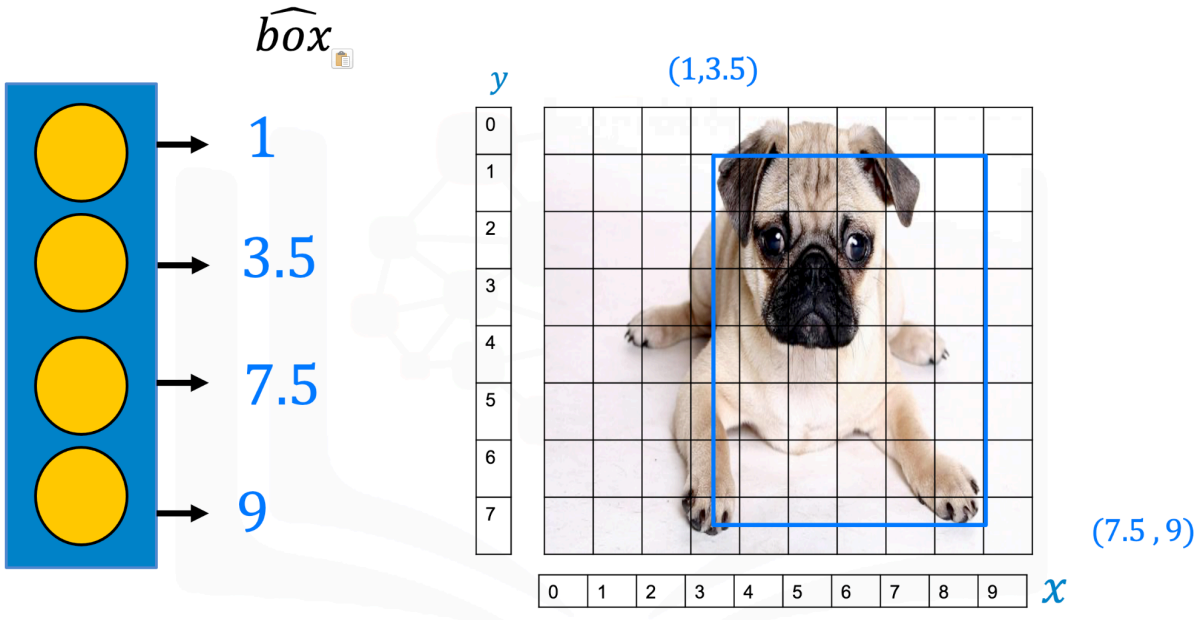
Where the number of neurons in the output softmax layer is **equal to the number of classes**. In many CNNs used for object detection, we **add four neurons to the output** layer to predict the **bounding box**, as shown in the following figure.



Each neuron outputs a different component of the box. This relationship is shown in Figure 3, with the corresponding bounding box around the object. We colour y hat and x hat to distinguish between the object class prediction y and x. To avoid confusion, unless explicitly referring to the coordinates system, we will use the term "box hat" to represent the output of these neurons.



Unlike classification, the output values of the neuron take on real numbers. Some possible values are shown in figure 4.



- ## Training for Object Detection:

Training in Object Detection has two objectives: we have to **determine the learnable parameters for the box** and we have to **determine the bounding boxes class**.

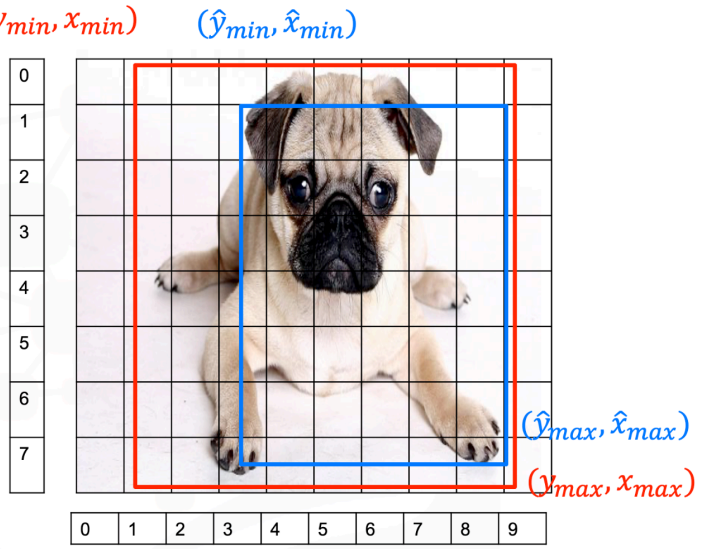
In order to determine the learnable parameters for the bounding box, we use the **L2 or squared loss**. This is used to find the **difference between real value predictions**. The L2 Loss Function **calculates squared differences** between the actual box value and the predicted box, as shown in figure 7, where we have the box and the L2 Loss for each coordinate of the box.

$$(y_{min} - \hat{y}_{min})^2$$

$$(y_{max} - \hat{y}_{max})^2$$

$$(x_{max} - \hat{x}_{max})^2$$

$$(x_{min} - \hat{x}_{min})^2$$



The **loss for each box** is given by:

$$\|box - \widehat{box}\|^2 = (y_{min} - \hat{y}_{min})^2 + (y_{max} - \hat{y}_{max})^2 + (x_{max} - \hat{x}_{max})^2 + (x_{min} - \hat{x}_{min})^2$$

Finally, to determine the **classification**, we **combine** the **L2 cost with the cross-entropy loss** in a Weighted Sum. This is called the **Multitask Loss**, which we use to **determine the cost**. We then use this to determine **all the parameters using gradient descent** to minimize the cost.

Multitask Loss = L2 Loss + Cross entropy

- **Types of Object Detection:**

Sliding window techniques are slow. Fortunatly, there are two major types of object detection that speed up the process. Region-based object detection breaks up the image into regions and performs a prediction, while Single-Stage uses the entire image.

Region-Based Convolutional Neural Network (R-CNN) are usually more accurate but slower; they include R-CNN, fast RCNN and Faster RCNN.

Single-Stage methods are faster but less accurate and include techniques like Single Shot Detection (SSD) and You Only Look Once (YOLO) .