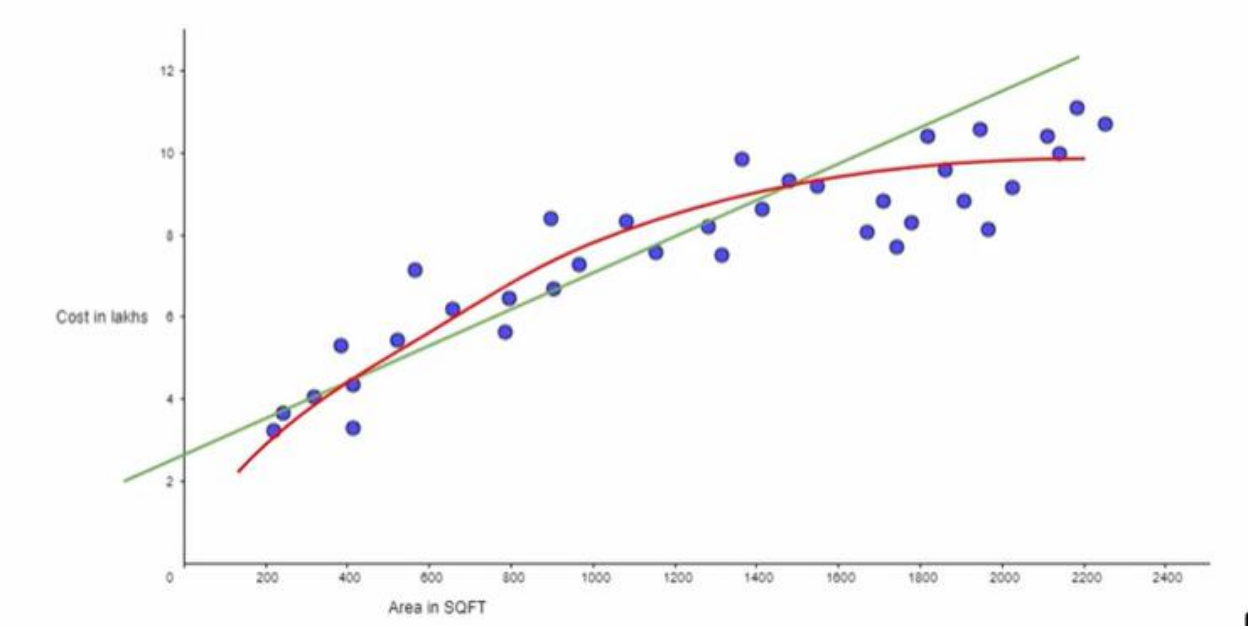
Supervised Learning

In supervised learning we are given a dataset and already know what our correct output should look like. With the understanding that there is a relatiionship between the input and output. The learning algorithm learns to map the input to the output.

Learning algorithm is basically learning the relationship between the x axis and the y axis or between the dependent variables and the independent variables and map them out.

Supervised learning problems divided into 2 kind of problems

1. Regression



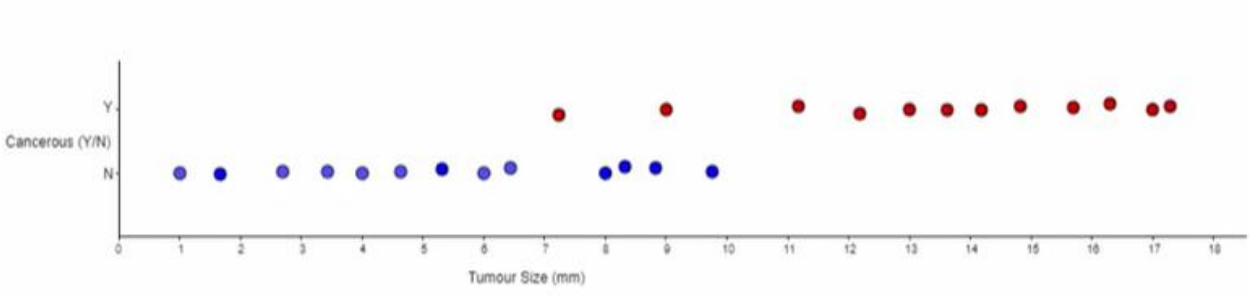
In here above as you can see , the data is plotted according to the cost and the area. The green color line is the called the Linear regression line (Linear best fit line) and the red color line is the Polynomial regression line.

We can find any cost if we know the area of the land likewise. So basically in regression you have a continuous flow of data or continuous data eventhough we plot few but the line plotted can show how it varies with the higher area than 2400 according to the above axis.

Regression problem – Attempting to anticipate results within a continous output(values in a range or range of values -> type of real data type or float or simply values could be in any number of decimal point or continuous) which means that we are mapping input vairables to a continuous function(Means the learning algorithm, acts as a continuous function because there

can be continous amount of outputs which are possible, the model itself a function which outputs continuous variables.).

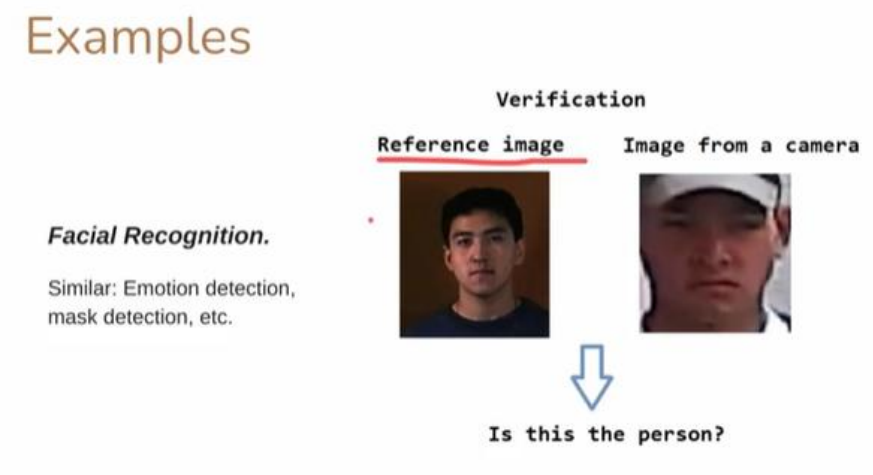
2. Classification



There is no continuous data flow or simply descrete data which in the above case is Yes or No based on the tumor size. It could be any type but its descrete.

Classification task – we are attempting to predict the outcomes in a discrete format(Yes or no or 1 or 0 or simply no continuous values or fixed.). In other words, we are attempting to group the input variables into discrete groups.

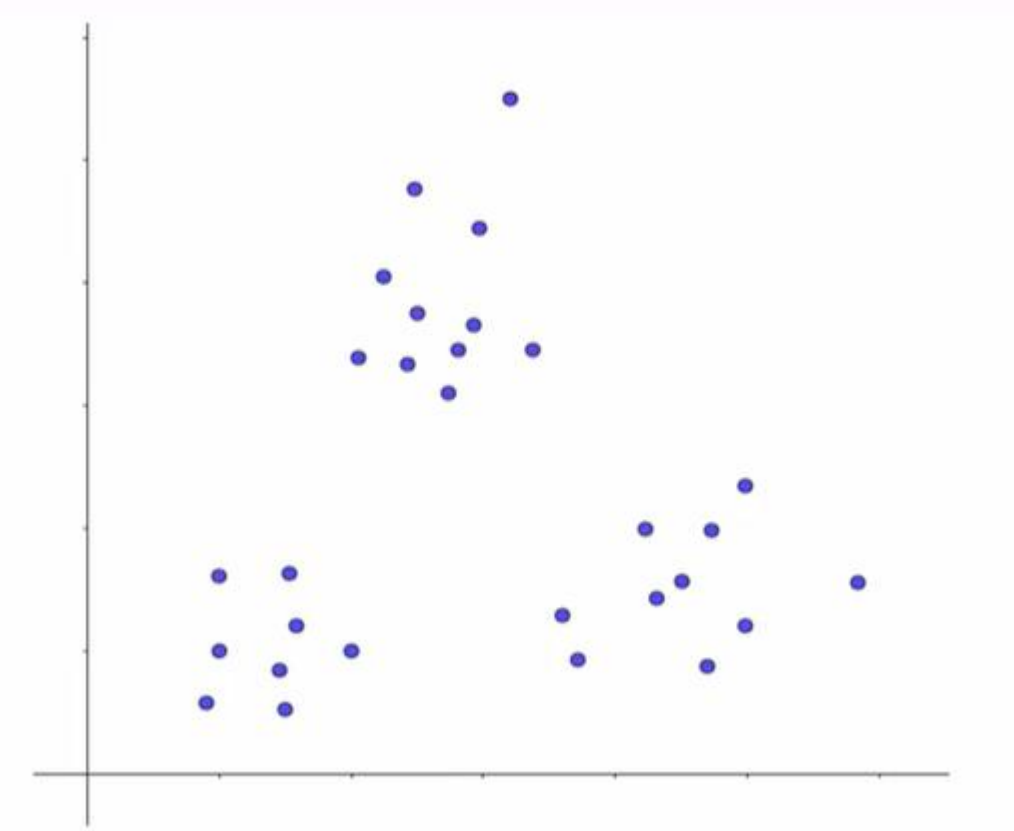
Examples of supervised Learning.



UNSUPERVISED LEARNING

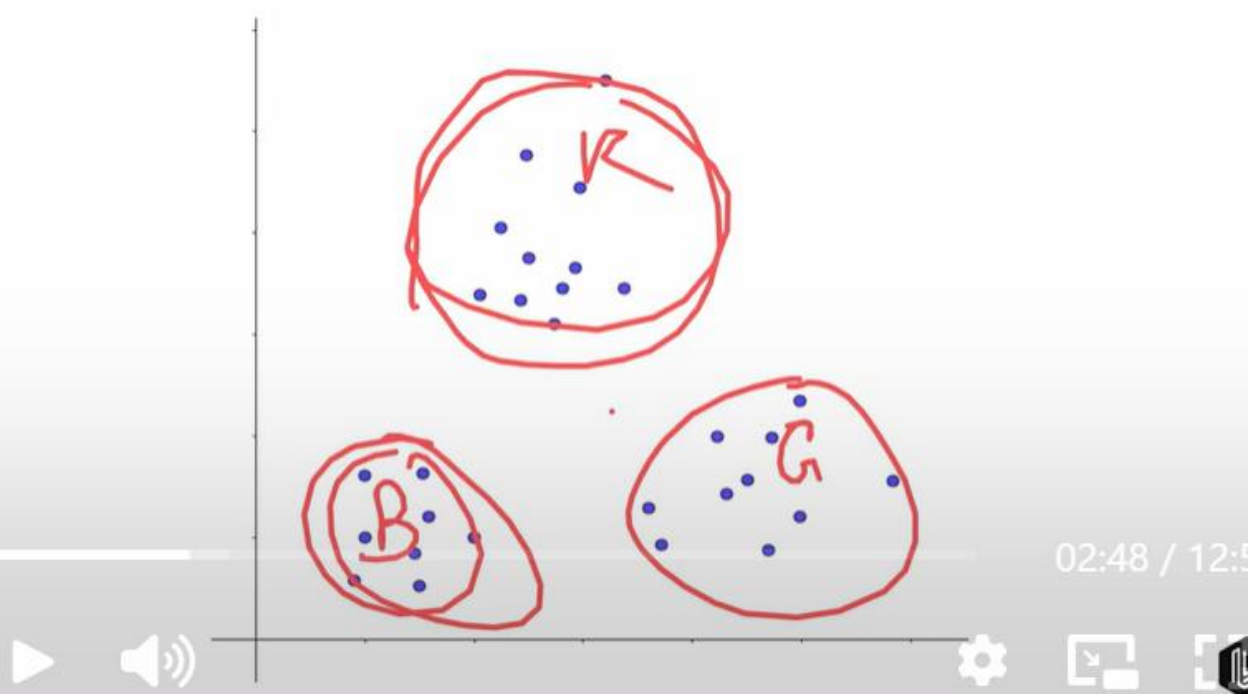
The other variety of machine learning.

EX:



If there is no labelling or difference between the sections as shown above , so in this case the purpose of our algorithm is to simplify this data into groups and maybe identify some zones here. No labelling or advice is given any guidance or labelling into what or where the data is.

Making sure the algorithm find its own. To simplify the above data 3 sections or zones, we are not given information that we have these three zones . Have to classify these sections.



Say as shown above , the we have no clue whether there are sections or likewise but however , since its not just small amount of data in real world or could be billions of gigabytes of data you cant just open and have a look directly therefore its hard to classify so we expect an algorithm to do the classification , so it shud be able to detect if there are 3 zones like red , blue and green.

DEFINITION : Allows us to approach problems with little to no idea what our results should look like. (might have the idea where has come from or might not have any idea about the data). So what we are expecting is to derive structure from the data where we are unaware of how data points affecting each other and how they are grouping. So there is no labelling or could be labelled as like above red , blue , green (no idea) , so we need an algorithm to detect is there are labels like red , blue or green.

So with unsupervised learning there is no feedback based on the prediction. Model just gives what it feels.

GOAL : The goal of unsupervised learning as previously mentioned is to explore and understand the underlying structure of the data without any pre-exisiting knowledge of what the output should look like (No idea about the output basically).

02 Common cases in Unsupervised Learning

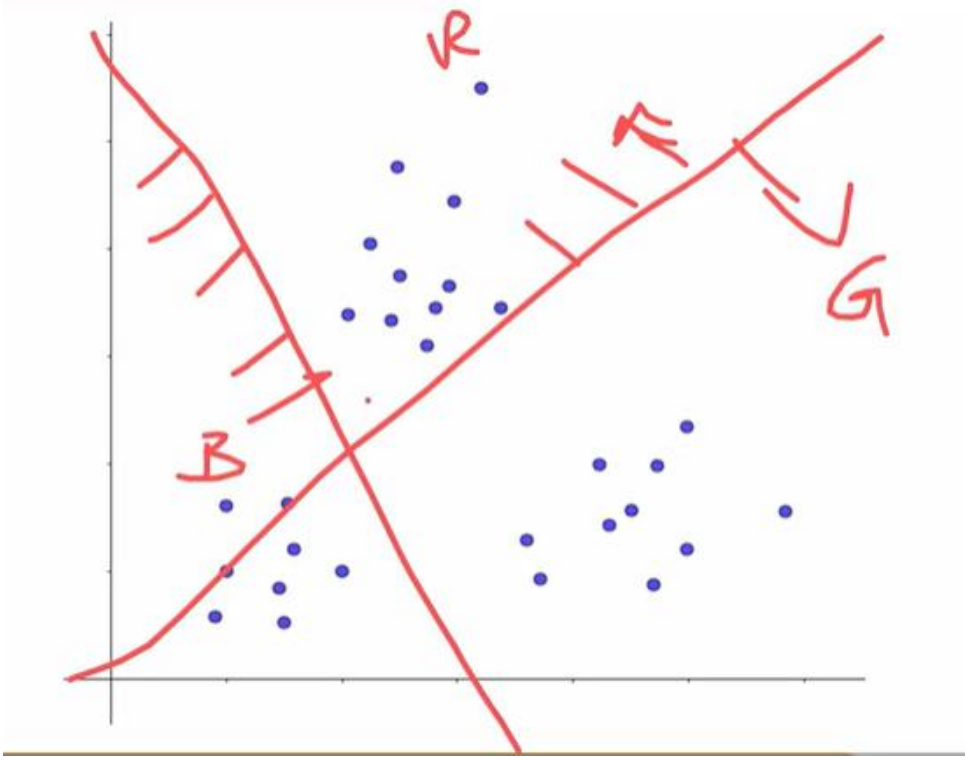
Clustering - This is same as the example which showed before. So clustering is basically the algorithm just group data points or similar data points together where each group is called clusters. But it varies from algorithm to algorithm.

Objective of clustering is to partition the data into groups so that data points within the same group and much simmilar to each other than those in other groups.

Dimensionally Reduction – Reduces the number of features or variables in the data while preserving its important characteristics.

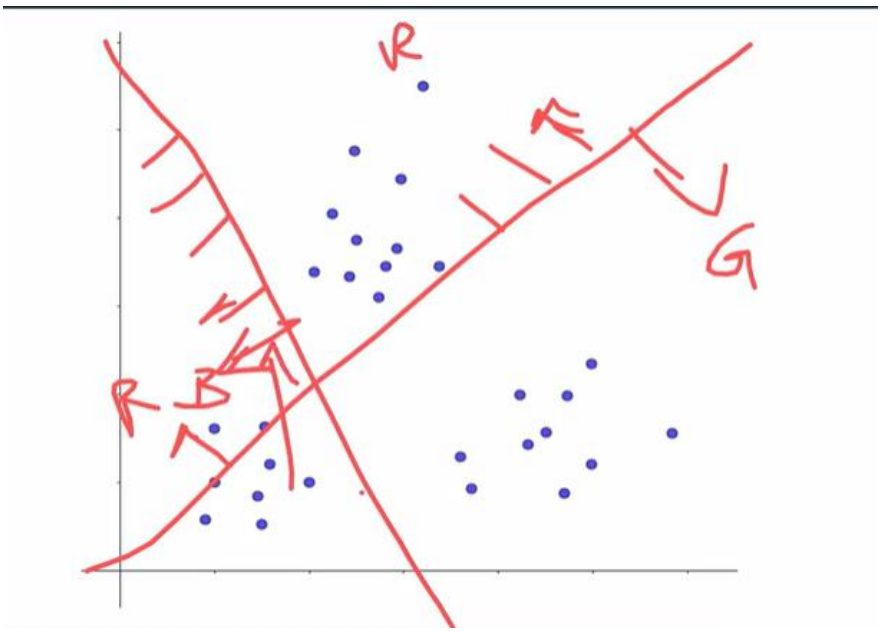
CLUSTERING EXAMPLE :

How clustering works , example one approach of clustering (there are many ways but to explain the concept of clustering)

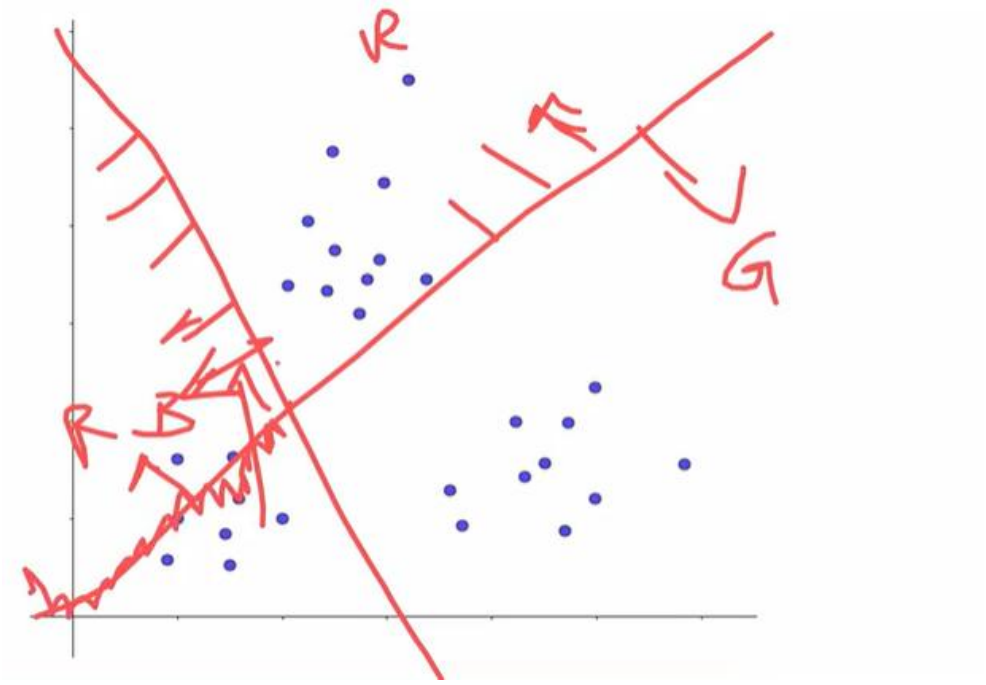


In here , we are dividing the regions like shown above and accordingly form clusters. We can also add condition of saying some mutual exclusivity , for example in above : red and blue cannot be together.

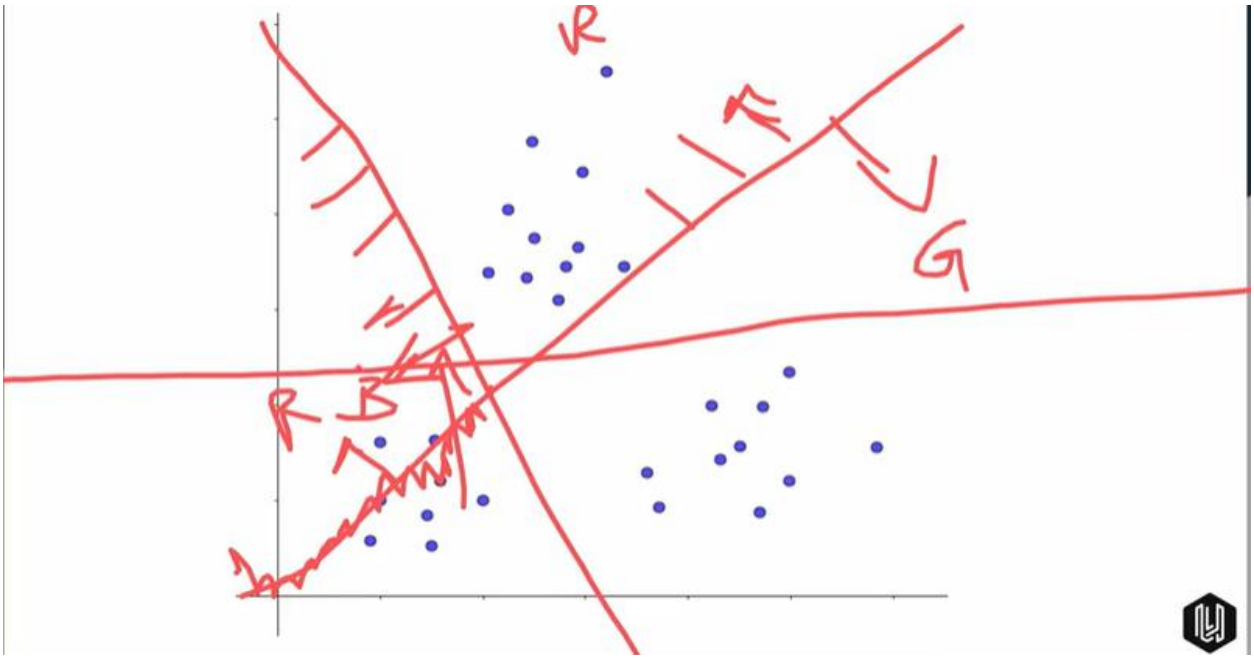
So there is some sort of metric which prevents you f rom happening and gives priority to blue.



So if it gives more priority to blue then it will adjust the clustering accordingly (Remember the intial lines are not permant cause we dont know what is the expected output so this is clustering , figuring our priorities and patterns of data and form clusters on the similar data)



There the bottom line or before intersection the line segement will be part of blue. Likwise it will find and divide accordingly. There will be supporting lines like this

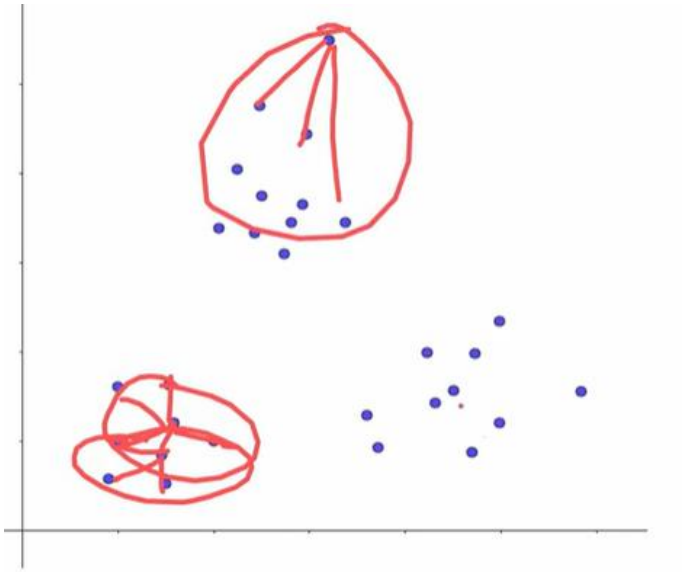


So it will help to differentiate between. (Like inequilities learned)

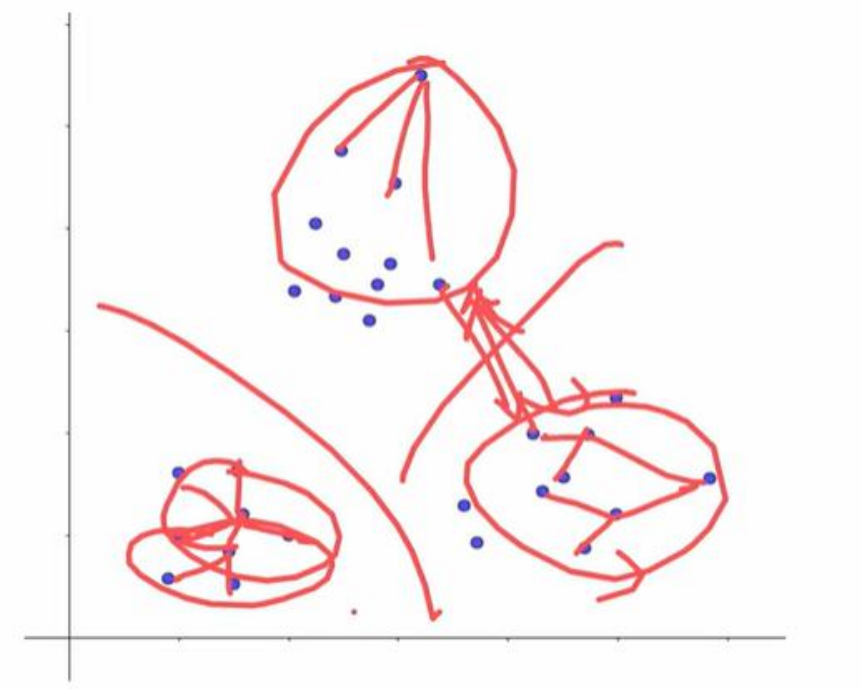
Above lines are called **DESCISION BOUNDARIES in this manner.** (Different context have different meaning of this word but in clustering , the lines are called decision boundaries.)

To be more specific , when having multiple decision boundaries of sorts , this is actually known as Tree or decision trees.

Another approach is , selecting one data point and finding the ones near that specific data point. or simply that specific point is looking for the neighbours and those neighbours together forms a community.



Once communities are made , they know that other commnuties or other points in different communnites are far from the current position.

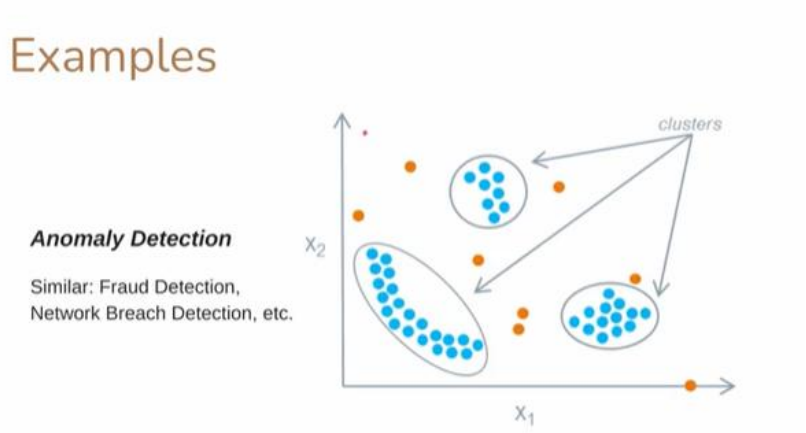


In that manner , they keep a seperation between them.

This appoach for looking for neighbours and forming a community is called as KNN

Likewise, there are different ways of approaching this clustering.

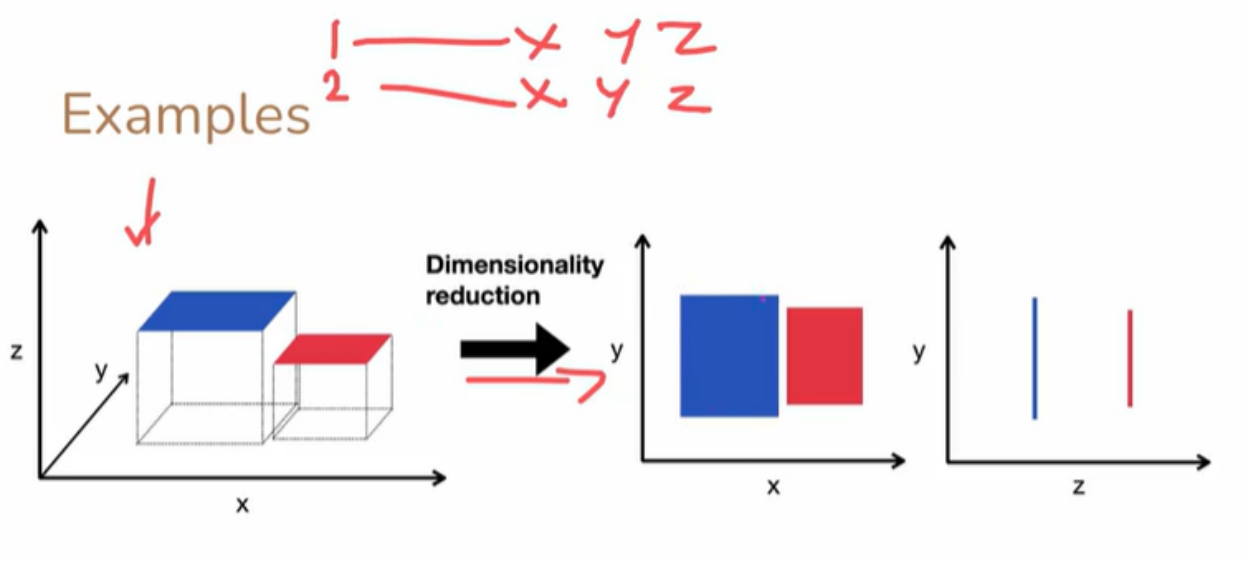
Examples of Unsupervised Learning :



If there are points which couldnt be able to find their neighbours (Above example is using knn), so they have their own individual communities (Orange dots) where datapoint = 1 or could be 2

likewise. These orange dots are called ANOMALIES or left over data (not in clusters). These are frauds according to above example etc.

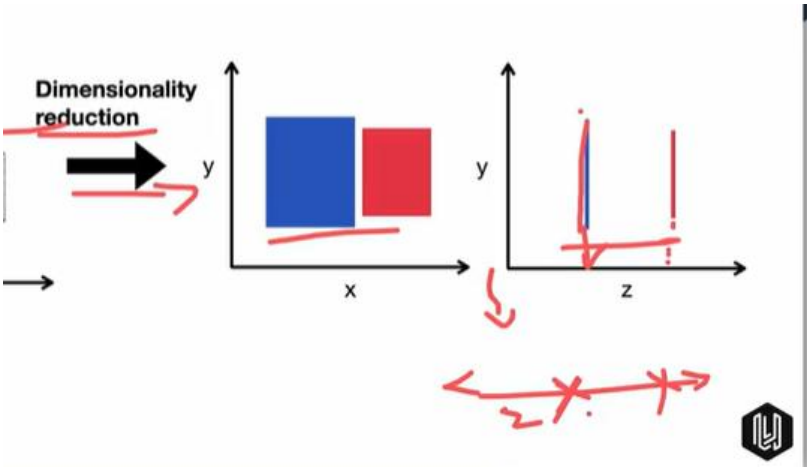
DIMENSIONAL REDUCTION EXAMPLE:



For each row -> x value , y value and z value

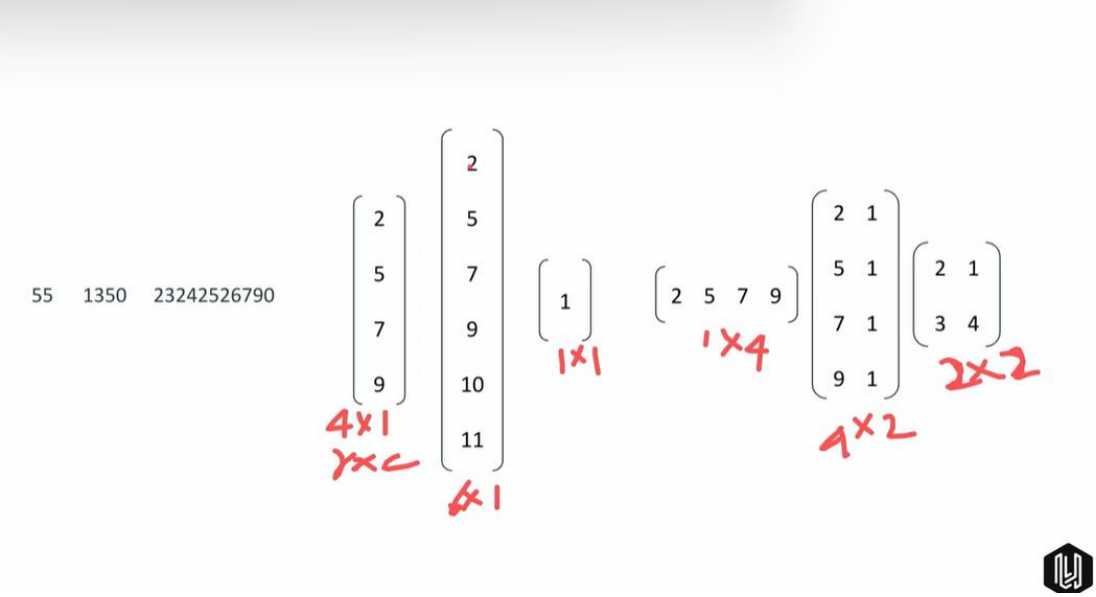
In here , dimentional reduction figures out a way of reducing it. As shown in the after arrow. First is basically after removing z and second is after removing x.

Likewise we can even reduce more like having only Z



However , it can affect the results dure to the fact thar we loast the magnitude of the line.

VECTORS AND MATRIX

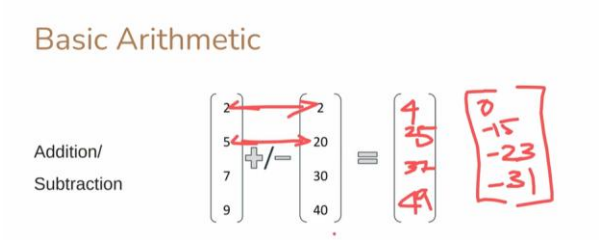


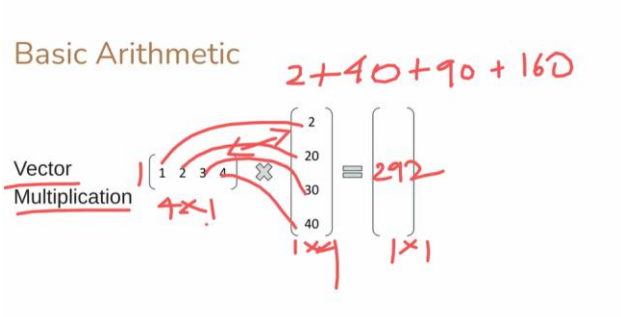
First is scalers

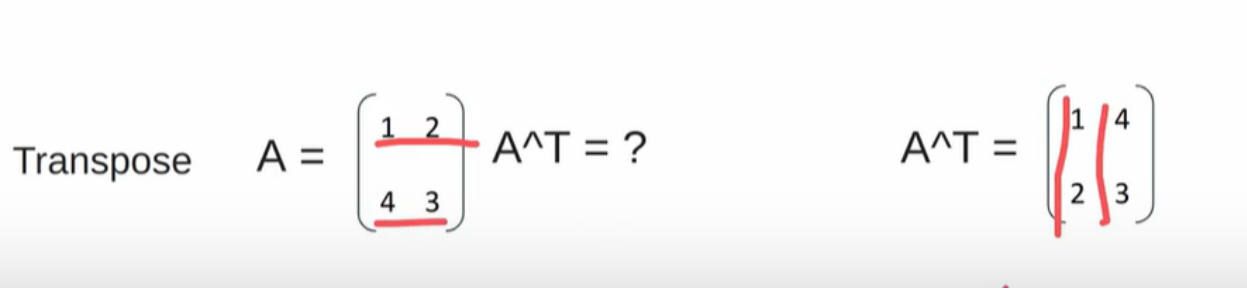
Second is Vectors

Third is Matri

-> There is no much difference between vectors and matrix but however , usually vectors are 1 dimentsional matrices could be column or row but only 1 dimentional. (A vector is a one-dimensional array of numbers,)







Transpose is basically A to the power T or A^T

MODULE 01 WORKSHEET

```
#### **What is Numpy?**
NumPy is the fundamental package for scientific computing in Python. It is a
Python library that provides a multidimensional array object, various derived
objects (such as masked arrays and matrices), and an assortment of routines for
fast operations on arrays, including mathematical, logical, shape manipulation,
sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic
statistical operations, random simulation and much more.

There are several important differences between NumPy arrays and the standard
Python sequences(lists):

*   NumPy arrays have a fixed size at creation, unlike Python lists (which can
grow dynamically). Changing the size of an ndarray will create a new array and
delete the original.
*   The elements in a NumPy array are all required to be of the same data type,
and thus will be the same size in memory. The exception: one can have arrays of
(Python, including NumPy) objects, thereby allowing for arrays of different sized
elements.
*    NumPy arrays facilitate advanced mathematical and other types of operations
on large numbers of data. Typically, such operations are executed more
efficiently and with less code than is possible using Python’s built-in
sequences.
*    A growing plethora of scientific and mathematical Python-based packages are
using NumPy arrays; though these typically support Python-sequence input, they
convert such input to NumPy arrays prior to processing, and they often output
NumPy arrays. Numpy is preferred due to its performance in computation.
```

Addition

Adding numpy vectors is as simple as performing regular addition in Python.

Code (Python)

```python
# TASK: Add vector [2, 2, 2] with n1 from earlier

n1 + np.array([2,2,2])

# Expected output: [24, 35, 46]
```

Markdown

Subtraction

Subtracting numpy vectors is also as simple as performing regular subtraction in Python

Code (Python)

```python
# TASK: Subtract vector [2, 3, 4] with n1 from earlier

n1 - np.array([2,3,4])

# Expected output: [20, 30, 40]
```

Scalar Multiplication/ Element-wise multiplication

This is also very easy to perform. Regular Python is enough or numpy also comes with functions to perform the same

https://numpy.org/doc/stable/reference/generated/numpy.multiply.html#numpy-multiply

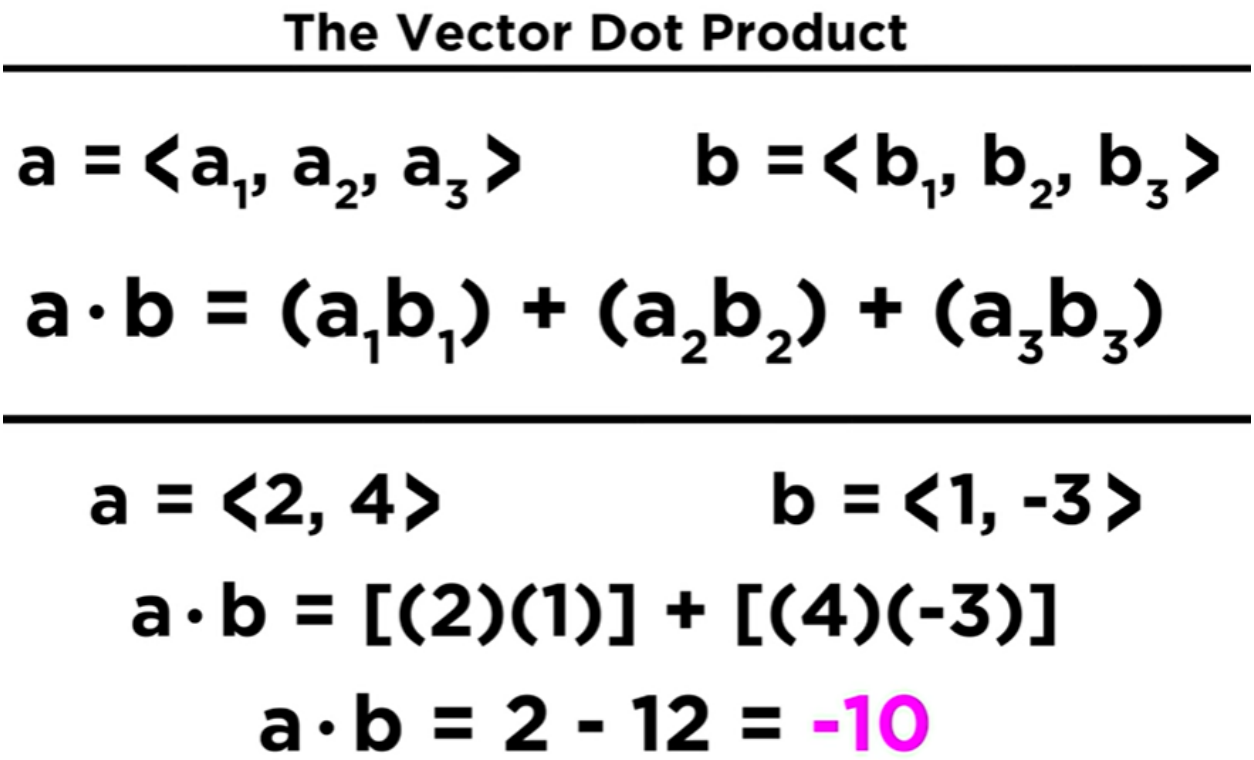
Code (Python)

```python
# TASK: Multiply n1 with 25

25 * n1

# Expected output: [550,  825, 1100]
```

```
Code (Python)

1  # TASK: perform element-wise multiplication of n1 and [11, 12, 13]. Use numpy's multiply function
2
3  np.multiply(n1, np.array([11, 12, 13]))
4
5  # Expected output: [242, 396, 572]
```

In here multiply n1 = [22,33,44] with the vector [11,12,13]

VECTOR MULTIPLICATION



This gives us a scaler or only a value (Vector dot Vector = scaler.)

1. n1 reshaped to (3 × 1)

Let's assume:

```ini
n1 = [22, 33, 44]
```

(Those numbers MUST be your original ones because the output matches these.)

After reshape:

```lua
n1.reshape(3,1) =
[[22],
 [33],
 [44]]
```

+ Ask anything

```
# TASK: perform vector multiplication of n1 and transpose of n1. Use numpy techniques for transpose

np.dot(n1, n1.T)

# Expected output: 3059
```

Multiplying n1 with n1's transpose using vector multiplication.

Q) Multiply v1 and v1 inverse (shud get identitiy matrix , however since computer cant store some exact value it gives precisions like 1.0000000001 likewise , therefore we use round to round the number , after rounding we get the identity matrix)

```python
import numpy as np

v1 = np.array([1, 3, 5,
               7, 11, 13,
               17, 19, 23]).reshape(3,3)

result = np.dot(v1, np.linalg.inv(v1)).round()

print(result)
```

```python
v1 = np.array([1, 3, 5,
               7, 11, 13,
               17, 19, 23]).reshape(3,3)
```

So:

```ini
v1 =
[[ 1   3   5 ]
 [ 7  11  13 ]
 [17  19  23 ]]
```

```python
v1_inv = np.linalg.inv(v1)
```

This gives a 3×3 matrix such that:

```java
v1 × v1_inv = identity matrix
```

But not exactly, because of floating-point precision errors.

Result will be something like:

```lua
[[ 1.00000000e+00 -2.22044605e-16  1.22124533e-15]
 [ etc… ]]
```

Small decimal noise happens because computers can't store perfect fractions.

✅ Step 3: Multiply v1 × inverse(v1)

```python
np.dot(v1, np.linalg.inv(v1))
```

This ideally should give:

This ideally should give:

```lua
[[1, 0, 0],
 [0, 1, 0],
 [0, 0, 1]]
```

But you get:

```lua
[[ 1.00000000e+00 -2.22044605e-16  1.22124533e-15],
 [ ... ]]
```

Those tiny values like `-2.22e-16` are basically **0**.

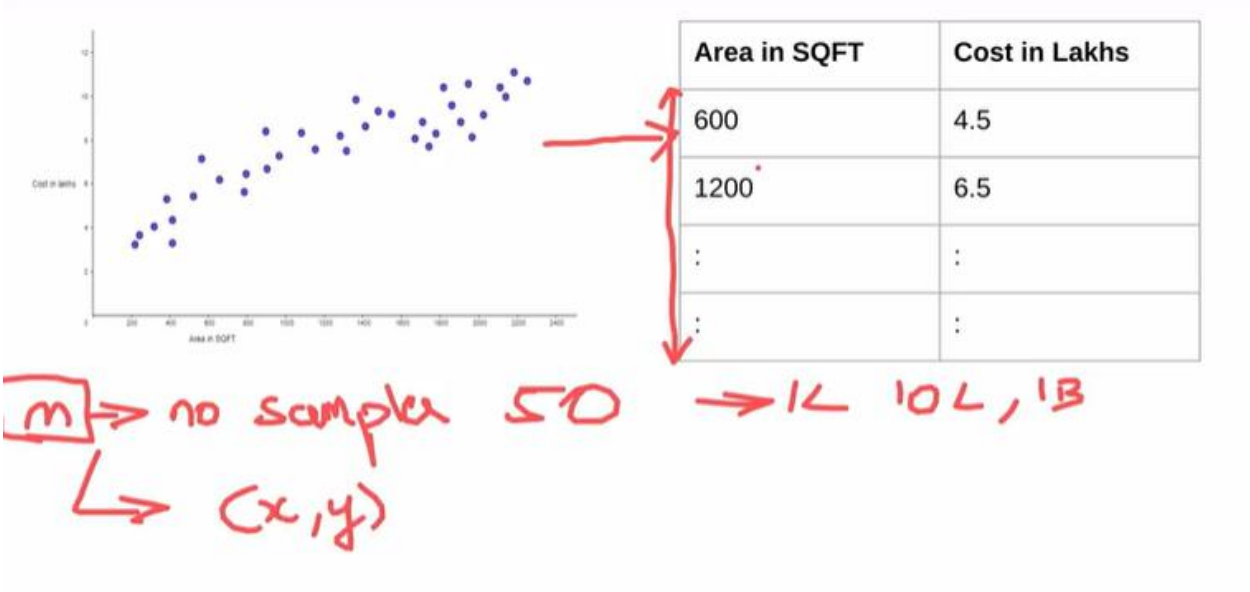
✅ Step 4: Fix floating errors using round()

```python
np.dot(v1, np.linalg.inv(v1)).round()
```

Final identity matrix:

```lua
[[ 1. -0.  0.]
 [ 0.  1.  0.]
 [-0. -0.  1.]]
```

Linear Regression

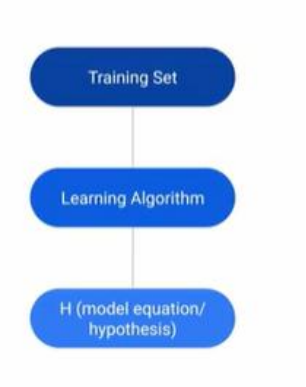


So in ML , we use the letter ‘m’ to represent the number of training samples or data (rows). Each sample have (x,y) where y is dependent variable and x is the independent variable.

Likewise “i^th” sample is ;

So if 50^{th} sample (x^50 , y^50).

CONCEPT OF TRAINING

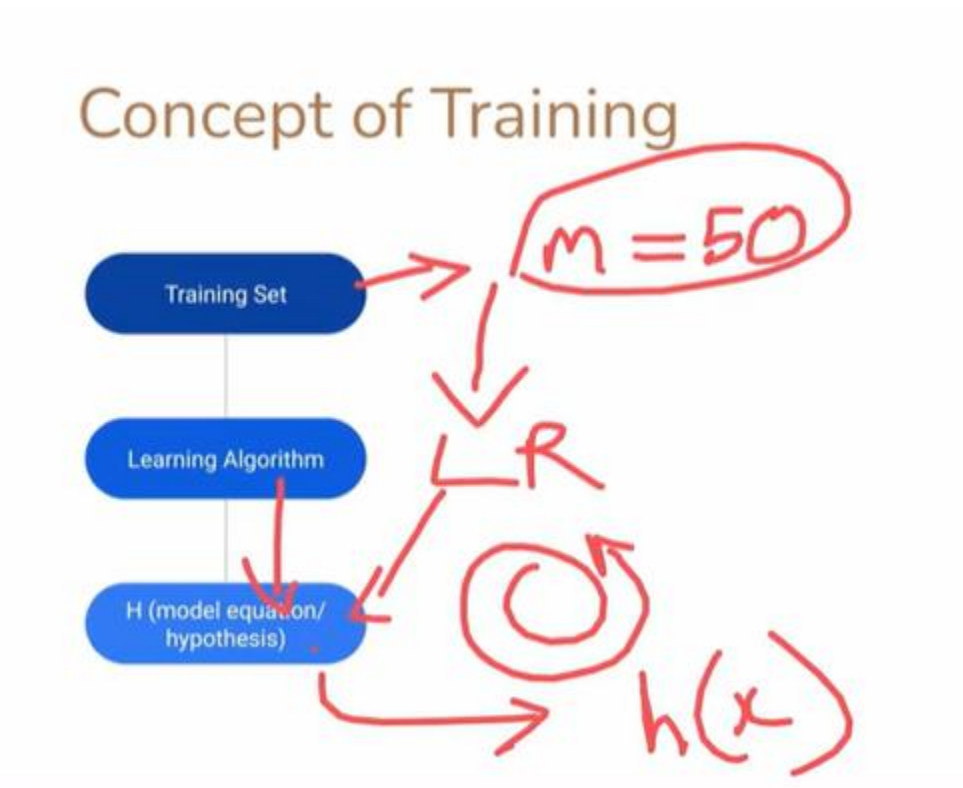


Training set is basically the data used to train or ‘m’

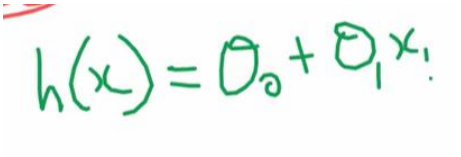
Selecting Learning algorithm – linear regression

Within learning algorithm or every learning algorithm has an equation or hypothesis and that is the actual equation which gives us the output.

So we have trainign set , then goes into the learning algorithm , then learnign algorithm iteratively goes through the model equation or hypothesis and at end of trainging which we have an hypothesis which will represented h(x). Using this we can get predictions.

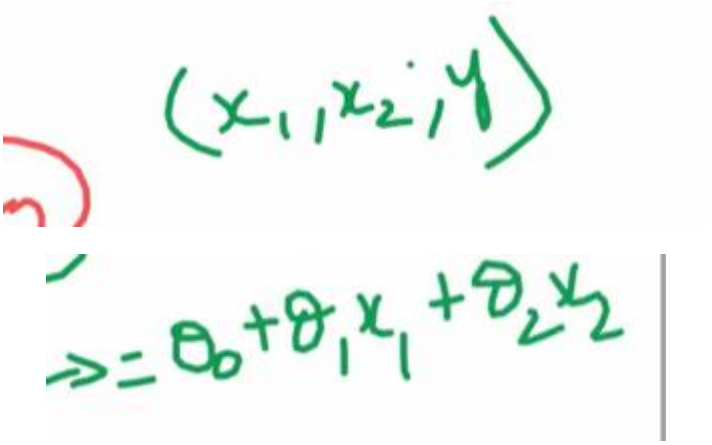


Therefore in Linear Regression , the hypothesis



X1 is optional, we have only 1 X so we can just write X.

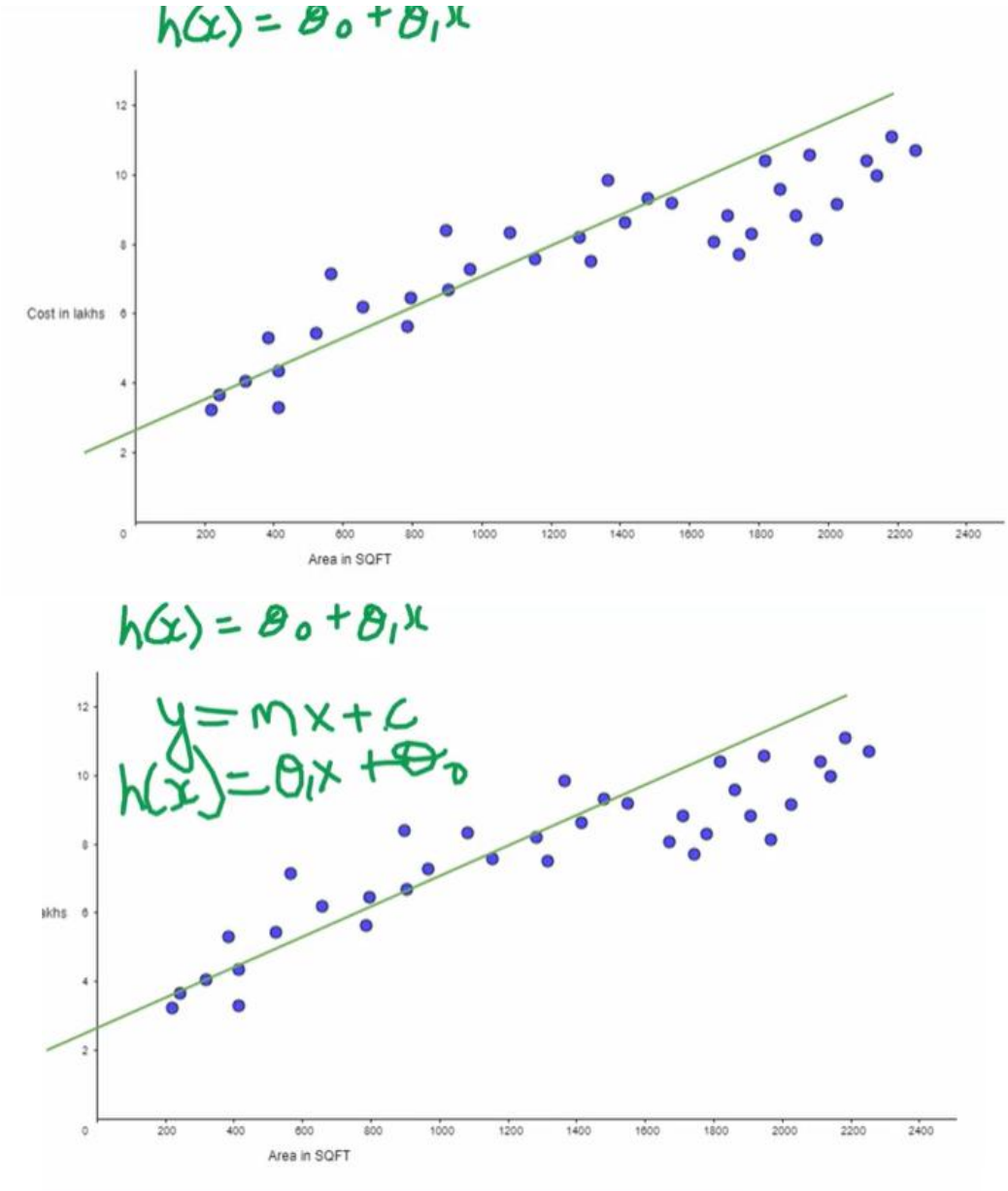
X = features , so lets say if two inputs vary the price , like bedrooms and square feet then we have 2 X values



So we are combining these are asking for an output.

Theta is paramters. These parameters of a model define how the hypothesis is going to be/output or h(x).

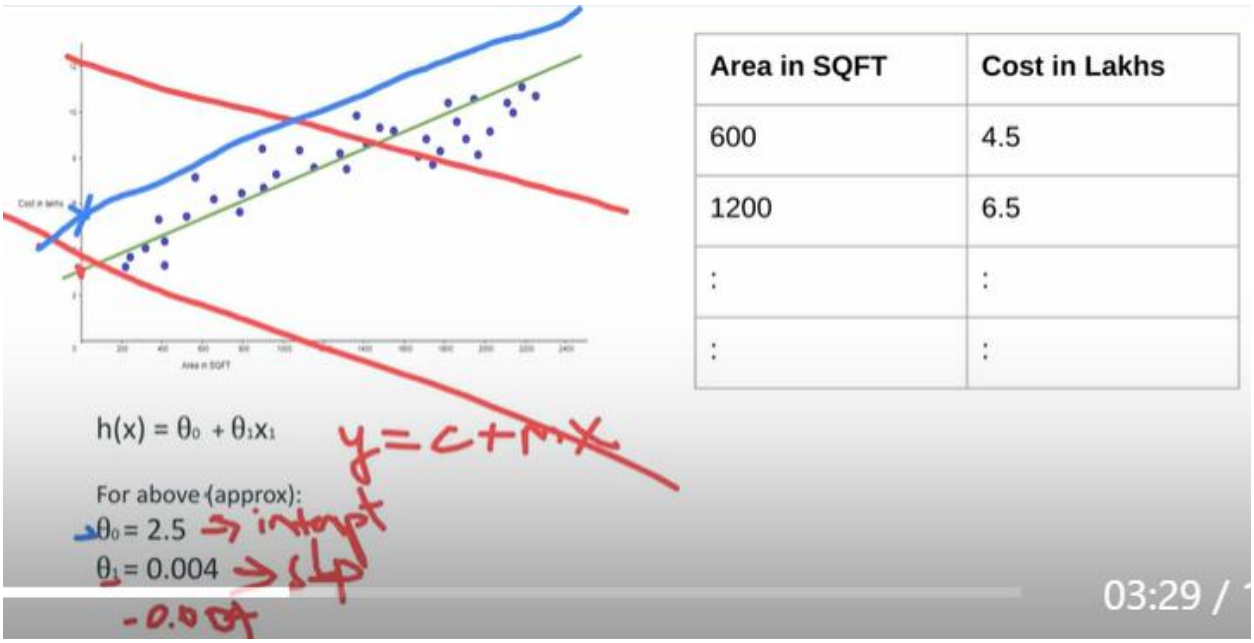
Ex: the below graph is after iterative steps and have out hypothesis.



Therefore gradient of the line is Theta1 , and y intercept is the Theta0

Cost Function

EX:



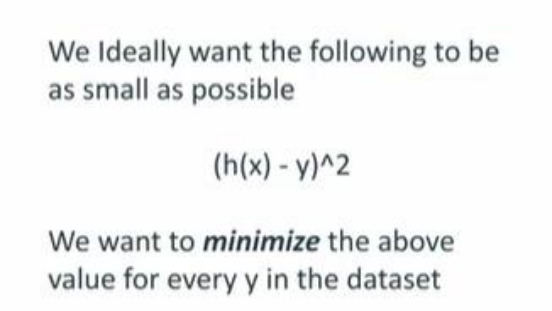
When we are starting a learning algorithm , there is an infinite variety of lines available under certain sence. So we need to find good values thetas so we get close value to the green line or the best fit line.

COST

Measures how well the models prediction matches the actual data by calculating the error between them.

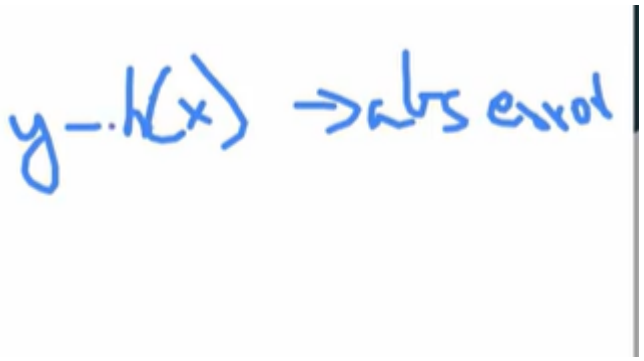
So lower the cost = more closer to the correct line or actual data.

We want to minimze the cost as possible



A**lways output a postive even if y is less than h(x) due to the square.**

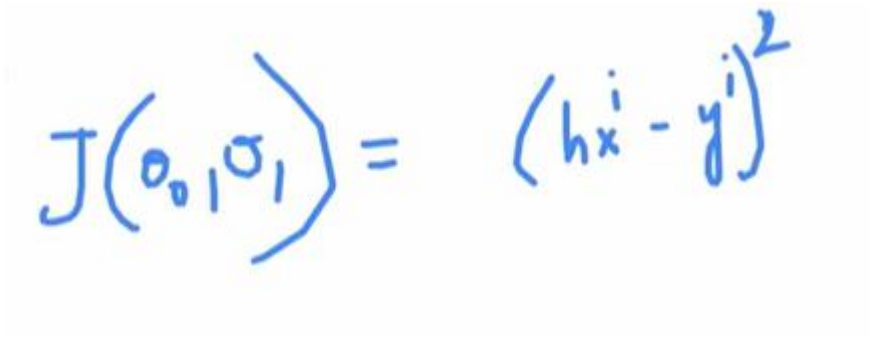
The output of this is called **Squared Error.**



The above or " y – h(x)" gives you the absolute error , or the difference between. Its y – h(x) not other way round because , we are finding how far I was away from the actual value or the current how far from the actual value. (True value – prediction.)

FORMULA , we are going to minimize a function that function is a cost function and it is shown below.

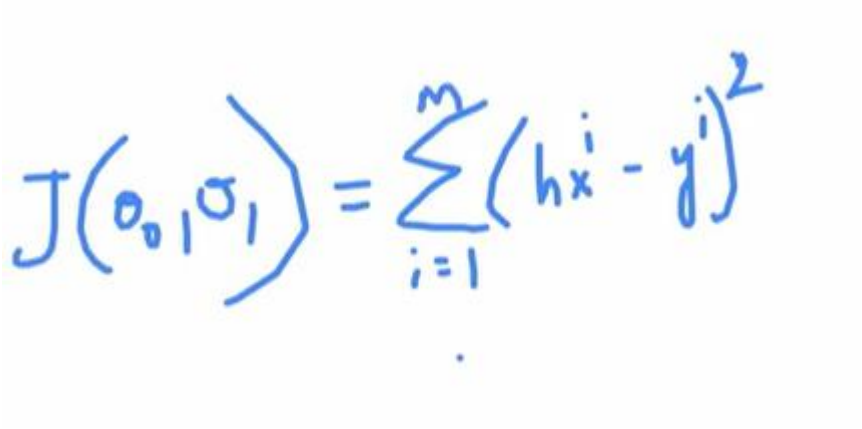
Step 1:



So cost function is represented J.

The above part of the formula gives you the cost function of the "ith sample"

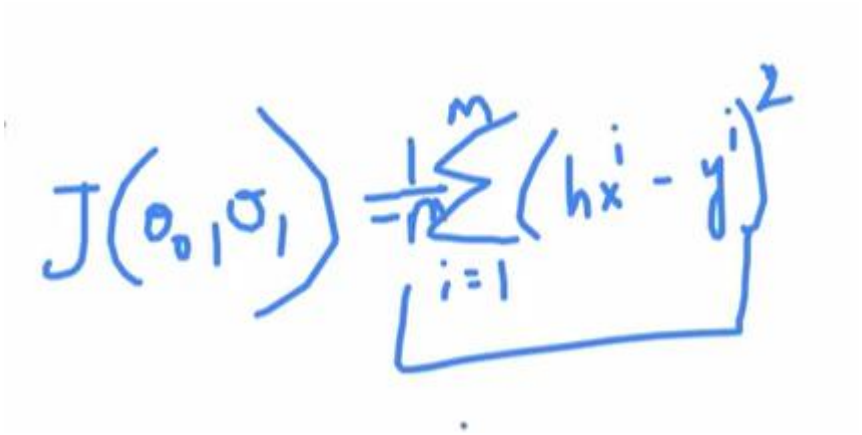
STEP 2 :



Then we take the sumation of the all samples likewise , so the sign is sigma and from i=1 to the total number of training samples m.

FINAL STEP:

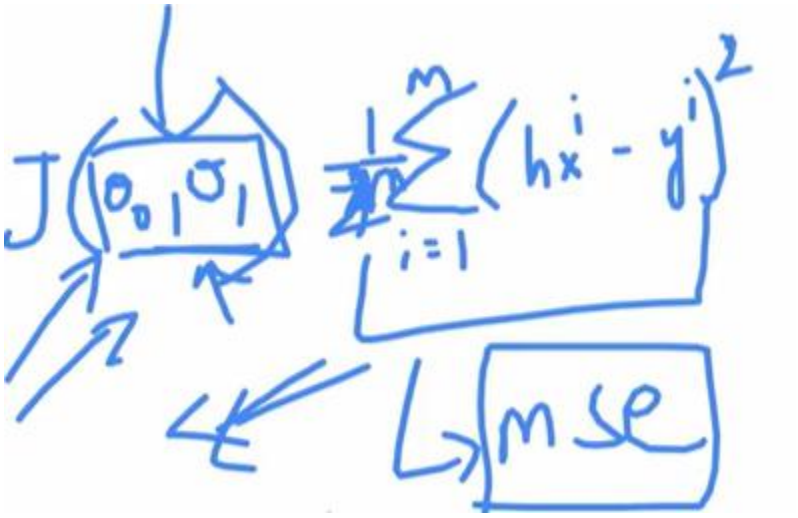
Divide the whole by 1/m



So the output or the part underlined , is called the “Mean Squared Error (MSE)” , simply average of the squared error.

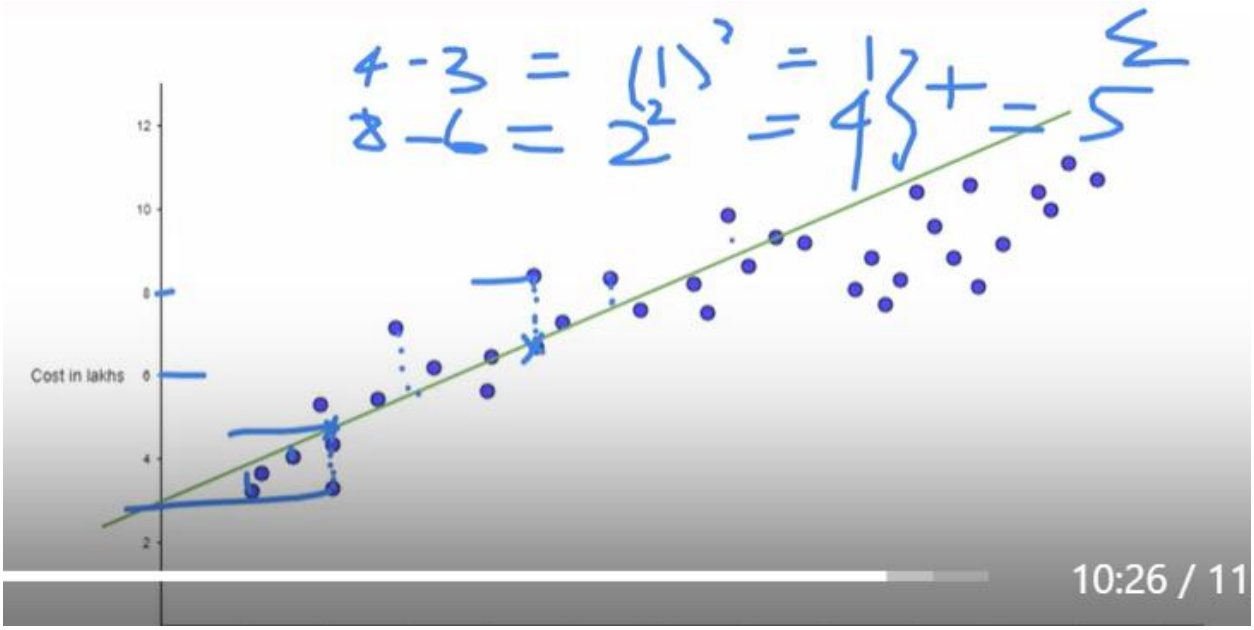
1/m - can vary depending on on the aid with computations. Essentially the same as 1/m others because it just divides all of the values with the ¼ additionally or likewise and when u get the mean and see the cost function , the difference is the same.

So we are basically minimizing the mean squared error obtained by using certain set of paramters or theta. Minimization occyrs by changing the set of parameters, so for every set of paramters there will be cost assiociated.

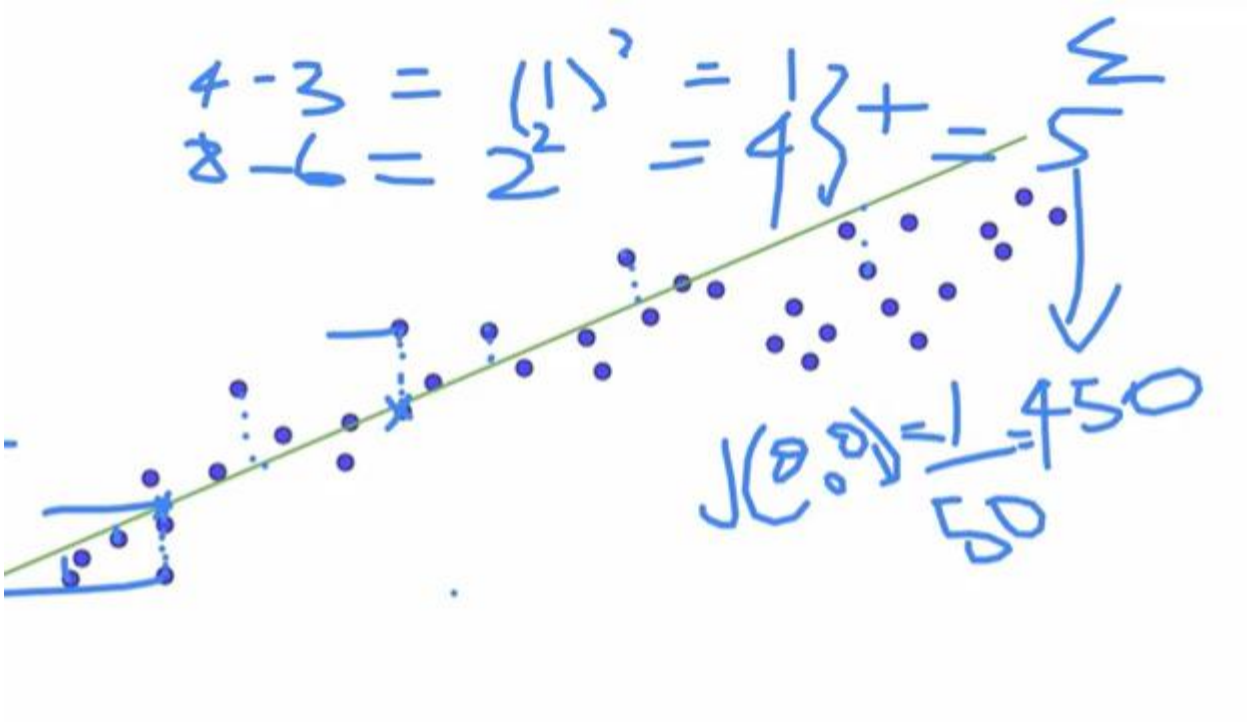


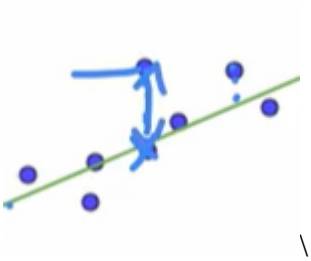
Constantly putting theta values so we get the lowest mean square error.

Ex:



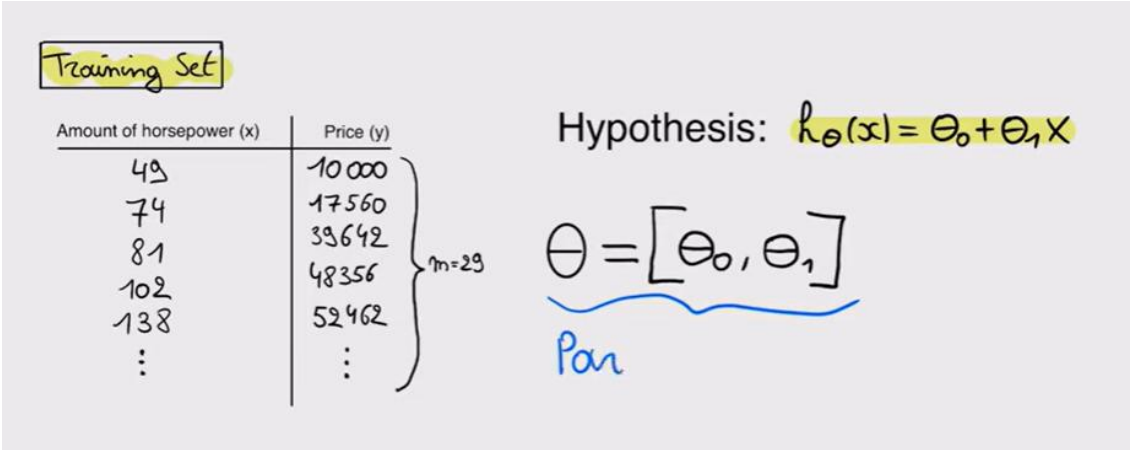
If we take the below point , we take the initial cost and then we find the best fit line cost from that point, and then we subtract and square it to the get cost. Then we take the summation likewise with the other data points then we multiply with 1/m in this case





The lines drawn like these are called **residual lines. Objective is to reduce these residual amount.**

Example calculation.



Cost Function :

$J(\theta_0\ \theta_1) = 1/m\ \Sigma\ ^{m}_{i=1}(h(x)-y)^2$

This is the cost function , so basically , we need θ_0 and θ_1 to find h(x) using the equation of linear regression for example in this instance $h(x) = \theta_0 + \theta_1 x$, where θ_0 is the y value for that perticular x value. (actual value) and θ1 is the gradient of the linear equation. Using the h(x) we calculated , we can find the precision or the error in between by subtracting from the actual value or the y for that relevant x value. And we square to get a positive value.

Likewise we continue for all other data points or values and finally get the summation of all the values (m samples) and calculate the mean and multiply the relevant coefficient accordinly to get the cost.

So for example as shown in the table

Assuming the gradient or $\theta_1 = 500$ and y intercept $\theta_0 = 0$

1. for x=48 , y= 10000

h_θ(49)= 0 + 500(48) = 24000

Therefore error;

Error = (24000-10000)^2 = 196,000,000

2. for x=74,y=17569

h_θ(74)= 0 + 500(74) = 37000

Therefore error;

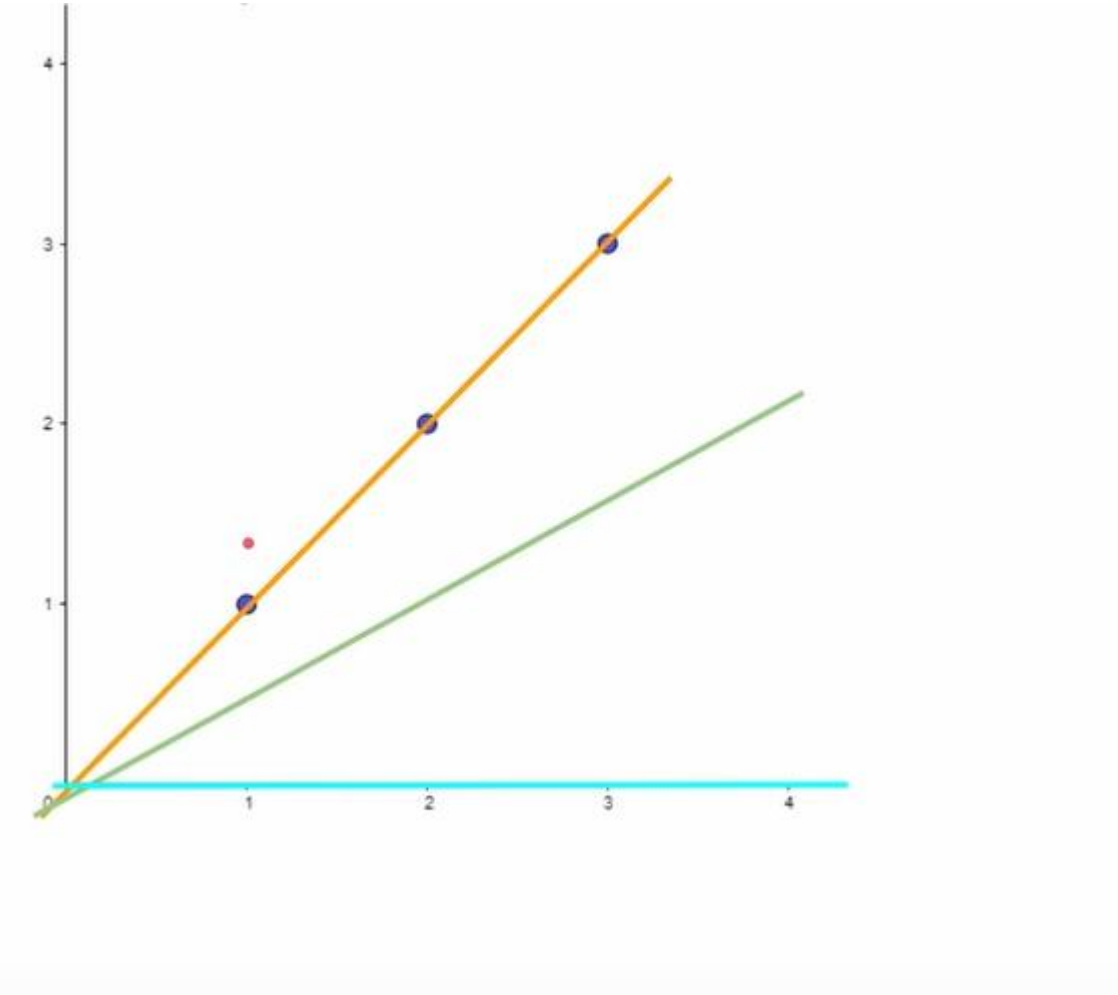
Error = (37000-10000)^2 = 377,344,400

Likewise we calculate the error or distance between the actual and the data points for all the points and then finally we

$J(\theta_0, \theta_1) = (1/10) \times$ [sum of all squared errors]

We get the final mean cost....

Ex: using a curve how to calculate the cost.



Assume that , h(x) is the y axis and X values is the x axis.

Blue dots = data points

Yellow line = line passes through data points

Likewise we have green and blue lines.

So initially , yellow , blue and green lines as you can see they dont have an y intercept , hence their θ_0 is 0 , hence the equation of

Yellow Line is = h(x) = **θ1 x**

The vertical axis shows both the actual y-values (dots) and the predicted h(x) values (lines).

So , if we are calculating the cost

Axes Explanation

- **X-axis (horizontal):** This represents the **input values** (x)
- **Y-axis (vertical):** This represents **both**:
  - The **actual values (y)** from your training data (the green dots)
  - The **predicted values h(x)** from your hypothesis (the lines)
- Basically y values of training data actual y values or the green dots, where the y values of the predicted values is hypothesis values h(x)

Cost = 0 for Yellow line (Because all the datapoints is on the yellow line which leads to no difference.)

Gradient = (2-1)/(2-1) = 1

$\theta_1 = 1$ (using gradient)

For each point:

1. **Point (1, 1):**
   a. h(1) = 1 × 1 = 1 (for predicted values)
   b. Error = h(1) - y = 1 - 1 = 0 (predicted value - actual data point value)
   c. Squared error = 0^2 = **0**
2. **Point (2, 2):**
   a. h(2) = 1 × 2 = 2
   b. Error = h(2) - y = 2 - 2 = 0
   c. Squared error = 0^2 = **0**
3. **Point (3, 3):**
   a. h(3) = 1 × 3 = 3
   b. Error = h(3) - y = 3 - 3 = 0
   c. Squared error = 0^2 = **0**

Total cost: $J(\theta) = (1/6) \times (0 + 0 + 0)$ = **0**

For Green line ;

Gradient = 2-1 / 4 –2 = 0.5

Therefore

$\theta_1 = 0.5$

So , h(x) = 0.5X

Therefore

Now cost;

For (1 , 0.5)

So, h(1) = 0.5

Error = h(1) - 1 = 0.5-1 = -0.5 which means the difference between is 0.5

Squared error = 0.5^2 = 0.25

For (2,1)

So, h(2) = 2*0.5 = 1

Error = h(2) - 2 = -1 , therefore difference is 1 (Remember h(2) is the y value for the x=2 which is 1 , and then 2 is the value of the actual y value of the data point)

Sqaured error = 1^2 = 1

For (3,1.5)

So, h(3) = 3*0.5 = 1.5

Error = h(3) - 3 = -1.5

Squared Error = 1.5^2 = 2.25

M=3 (3 data points)

Total cost = (0.25+1+2.25) * 1/3 = 1.2

For Blue line;

Gradient = 0

$\theta_1 = 0$

Therefore h(x) = 0

For (1,0)

So, h(1) = 0

Error = 0-1 = 1

Squared Error = 1^2 = 1

For (2,0)

So, h(2) = 0

Error = 0 – 2 = -2

Squared Error = 4

For (3,0)

So , h(3) = 0

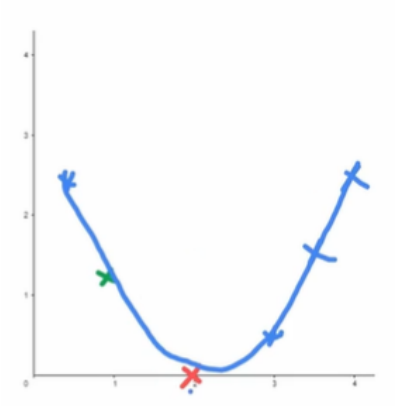
Error = 0 –3 = 3

Squared Error = 9

Cost = 1/3 * (9+4+1) = 14/3 = 4.67

Likewise if we find costs of more data points , we can conclude that , when **θ1** changes the , **cost also changes.**

If we plot a graph -> θ1 vs cost



Cost Function with Only θ_1 (No Intercept)

Given:

- Training data: (1, 3), (2, 5), (3, 7)
- Hypothesis: **$h(x) = \theta_1 \times x$** (no θ_0, so intercept = 0)
- Cost function: **$J(\theta_1) = (1/2m)\ \Sigma[h(x^i) - y^i]^2$** where m = 3

Step 1: Write Out the Full Equation

$J(\theta_1) = (1/6)[(\theta_1 \cdot 1 - 3)^2 + (\theta_1 \cdot 2 - 5)^2 + (\theta_1 \cdot 3 - 7)^2]$

Step 2: Expand Each Squared Term

Term 1: $(\theta_1 - 3)^2$
$= \theta_1^2 - 6\theta_1 + 9$

Term 2: $(2\theta_1 - 5)^2$
$= 4\theta_1^2 - 20\theta_1 + 25$

Term 3: $(3\theta_1 - 7)^2$
$= 9\theta_1^2 - 42\theta_1 + 49$

Step 3: Sum All Terms

Sum $= \theta_1^2 + 4\theta_1^2 + 9\theta_1^2 - 6\theta_1 - 20\theta_1 - 42\theta_1 + 9 + 25 + 49$

Sum = **$14\theta_1^2 - 68\theta_1 + 83$**

But remember m = 3 (3 data points) so therefore 2m gives 6. We need 2m because we are dividing the mean sum calculated by 2m.(6)

Step 4: Divide by 2m = 6

$J(\theta_1) = (14\theta_1^2 - 68\theta_1 + 83) / 6$

$J(\theta_1) = (7/3)\theta_1^2 - (34/3)\theta_1 + 83/6$

Or in decimal form:

$J(\theta_1) = 2.333\theta_1^2 - 11.333\theta_1 + 13.833$

Step 5: General Form

$J(\theta_1) = A\theta_1^2 + B\theta_1 + C$

Where:

- **$A = 7/3 \approx 2.333$** (coefficient of θ_1^2)
- **$B = -34/3 \approx -11.333$** (coefficient of θ_1)
- **$C = 83/6 \approx 13.833$** (constant term)

Step 6: Why Is This a Parabola?

The equation **$J(\theta_1) = A\theta_1^2 + B\theta_1 + C$** is a **quadratic function**.

- Since **$A > 0$** ($A = 2.333 > 0$), the parabola **opens upward** (U-shape)
- This guarantees **one minimum point**

Step 7: Find the Minimum (Optional) -> First derivative = 0 (gradient =0)

To find the minimum, take the derivative and set it to zero:

$dJ/d\theta_1 = 2A\theta_1 + B = 0$

$\theta_1 = -B / (2A) = -(-34/3) / (2 \times 7/3) = (34/3) / (14/3) = 34/14 = 17/7 \approx 2.43$

At $\theta_1 \approx 2.43$, the cost is minimized!

Verify:

- $h(1) = 2.43 \times 1 = 2.43$ (actual: 3)
- $h(2) = 2.43 \times 2 = 4.86$ (actual: 5)
- $h(3) = 2.43 \times 3 = 7.29$ (actual: 7)

Pretty close fit! ✓

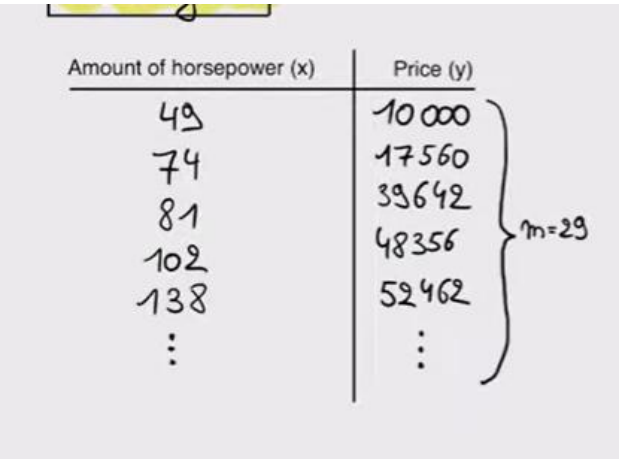
Summary:

$J(\theta_1) = 2.333\theta_1^2 - 11.333\theta_1 + 13.833$

This is a **parabola** (U-shape) because of the **θ_1^2 term**, with minimum at **$\theta_1 \approx 2.43$**.

WHAT IF THERE IS A Y INTECEPT OR θ_0 ?

EX:



If the line doesnt pass through origin , then we must follow the below steps.

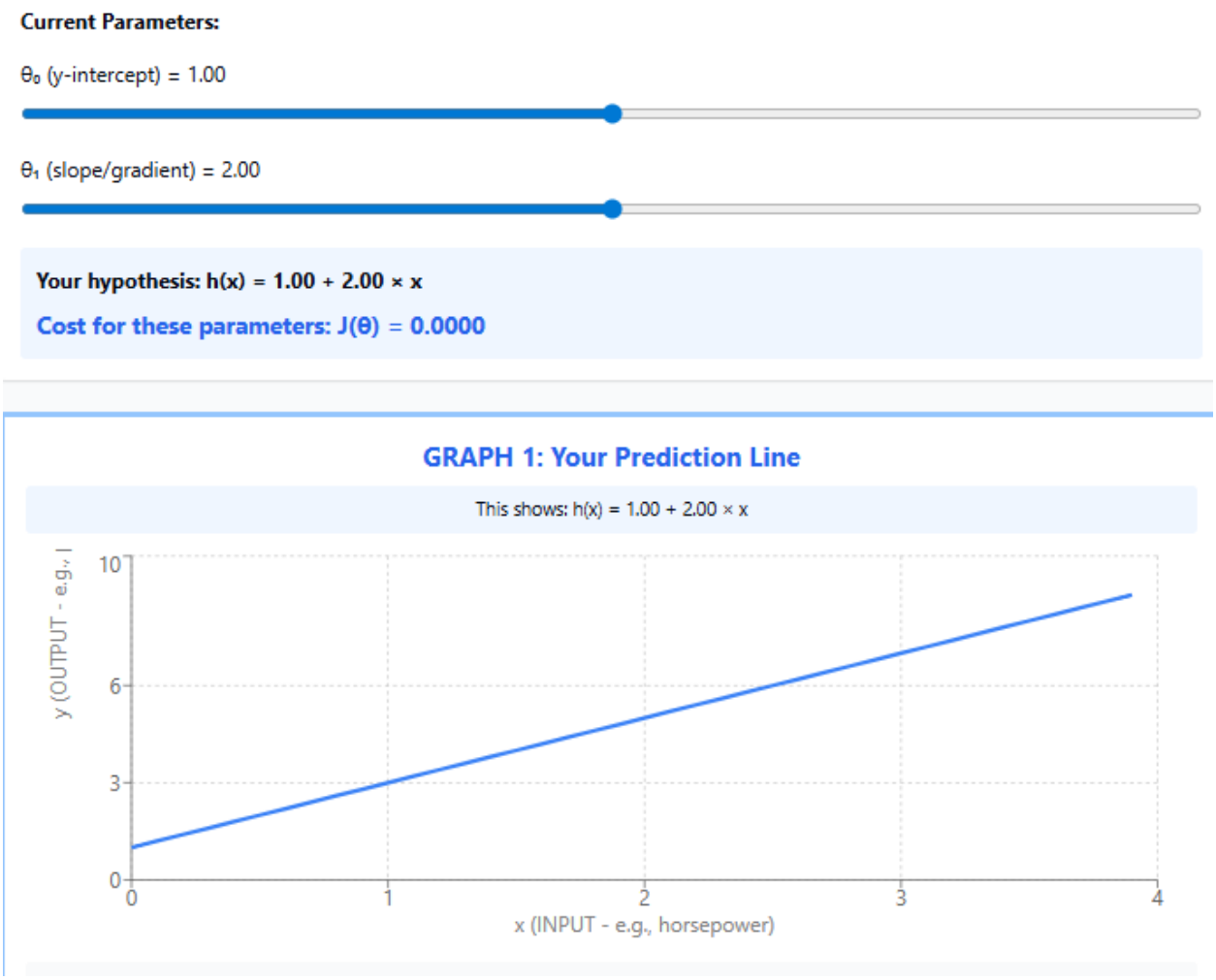
Graph 1: The LINE (your hypothesis)

This shows your **prediction model**:

- **X-axis** = input (horsepower, in your original example)
- **Y-axis** = output (price)
- **The line** = $h(x) = \theta_0 + \theta_1 \times x$
    - θ_0 = y-intercept (where line crosses y-axis)
    - θ_1 = slope (gradient)

The below graph in next page.

- **What this shows:**
  Green dots = actual data (x, y)
  Blue line = your predictions
  θ_0 = where line crosses Y-axis (y-intercept)
  θ_1 = steepness of line (slope)
- **This is for MAKING PREDICTIONS!**



Graph 2: The COST FUNCTION (how good is your line?)

This is a **completely different graph** that measures how GOOD your θ values are:

For 1 parameter (θ_1 only):

- **X-axis** = different values of θ_1 you could try
- **Y-axis** = Cost $J(\theta_1)$ - how bad the fit is
- This graph tells you: "If I pick this θ_1 value, what's my cost?"

For 2 parameters (θ_0 and θ_1):

- **X-axis** = different values of θ_0 you could try
- **Y-axis** = different values of θ_1 you could try
- **Z-axis** = Cost $J(\theta_0, \theta_1)$ - how bad the fit is

- This graph tells you: "If I pick this (θ_0, θ_1) combination, what's my cost?"

Shows: $J(\theta_0, \theta_1)$

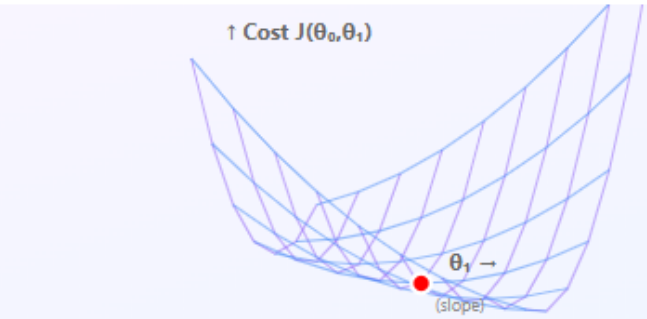
X-axis: θ_0 values

Y-axis: θ_1 values

Z-axis: Cost (error)

GRAPH 2: Cost Function

This shows: $J(\theta_0, \theta_1)$ = cost for each (θ_0, θ_1) pair



What this shows:

- NOT the same as Graph 1!
- X-axis = different θ_0 values to try
- Y-axis = different θ_1 values to try
- Z-axis (height) = Cost/error
- Red dot = your current (θ_0, θ_1) position

This is for FINDING BEST θ values!

Proof using the Formula for the shape of the curve

$$J(\theta_0, \theta_1) = (1/2m)\ \Sigma[\theta_0 + \theta_1 x^i - y^i]^2$$

Let's use our data: (1,3), (2,5), (3,7) where m = 3

Step 1: Expand the Equation

$J(\theta_0, \theta_1) = (1/6)[(\theta_0 + \theta_1 \cdot 1 - 3)^2 + (\theta_0 + \theta_1 \cdot 2 - 5)^2 + (\theta_0 + \theta_1 \cdot 3 - 7)^2]$

Let me expand each squared term:

Term 1: $(\theta_0 + \theta_1 - 3)^2 = \theta_0^2 + \theta_1^2 + 9 + 2\theta_0\theta_1 - 6\theta_0 - 6\theta_1$

Term 2: $(\theta_0 + 2\theta_1 - 5)^2 = \theta_0^2 + 4\theta_1^2 + 25 + 4\theta_0\theta_1 - 10\theta_0 - 20\theta_1$

Term 3: $(\theta_0 + 3\theta_1 - 7)^2 = \theta_0^2 + 9\theta_1^2 + 49 + 6\theta_0\theta_1 - 14\theta_0 - 42\theta_1$

Step 2: Add All Terms Together

Sum = $3\theta_0^2 + 14\theta_1^2 + 12\theta_0\theta_1 - 30\theta_0 - 68\theta_1 + 83$

Step 3: Divide by 2m = 6

$J(\theta_0, \theta_1) = (1/2)\theta_0^2 + (7/3)\theta_1^2 + 2\theta_0\theta_1 - 5\theta_0 - (34/3)\theta_1 + 83/6$

This can be written as:

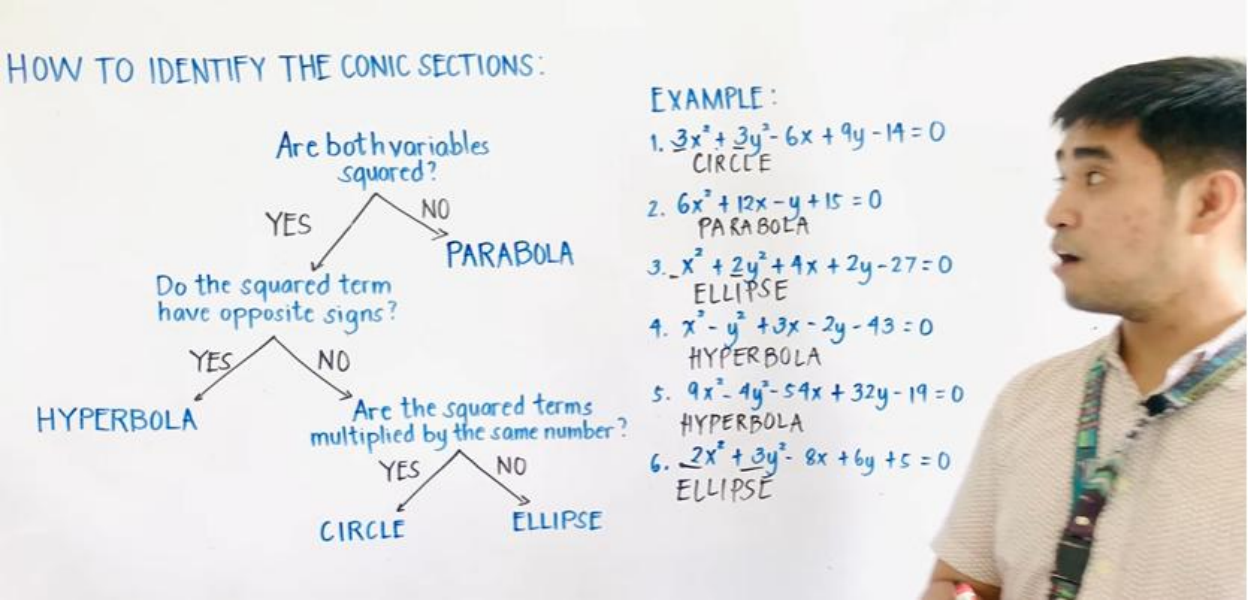
$J(\theta_0, \theta_1) = A\theta_0^2 + B\theta_1^2 + C\theta_0\theta_1 + D\theta_0 + E\theta_1 + F$

Coefficient of θ_0^2 = +

Coefficient of θ_1^2 = +

θ_0^2 and θ_1^2 values ½ and 7/3 are not multiplied by same value , hence the shape is Ellipse. (See below guide to have a look how to get the shapes.)

How to identify the shape using the equation:



Gradient Descent

Having a Cost function J, a function of parameter/theta

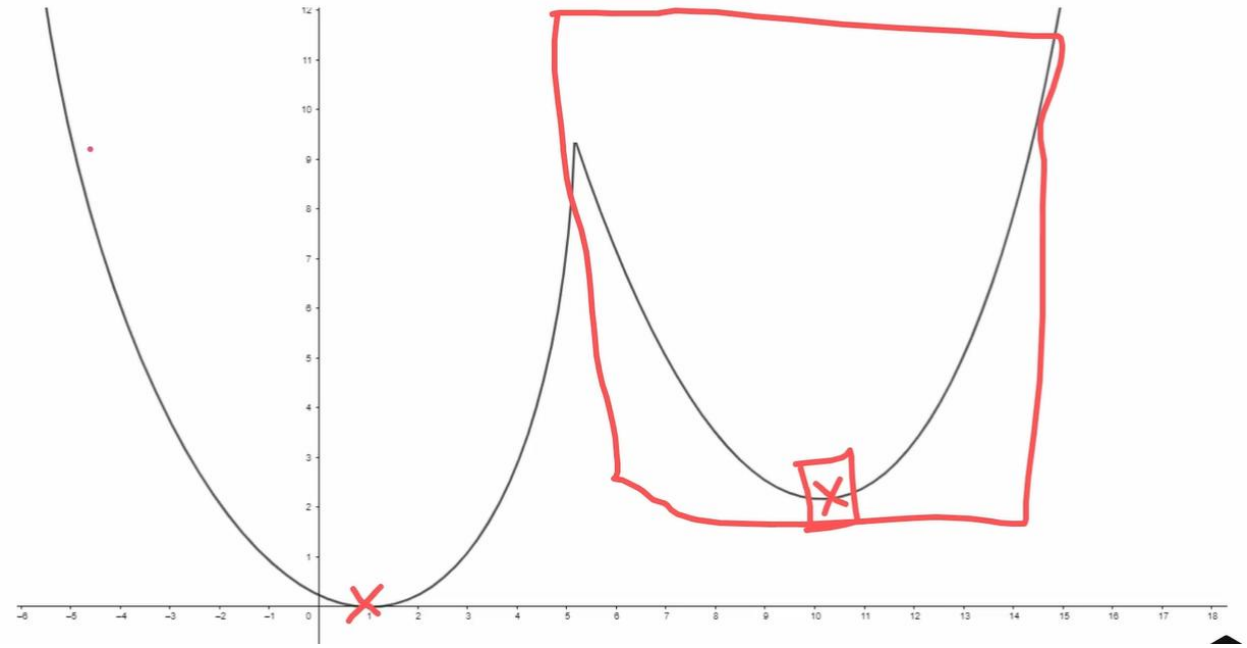
We want to minimize this function J

-> Start with some random value for the parameter(s)

-> Keep changing parameter/s to minimize the function J.

-> The changing continues with the objective that we manage to find the minima for the cost function.

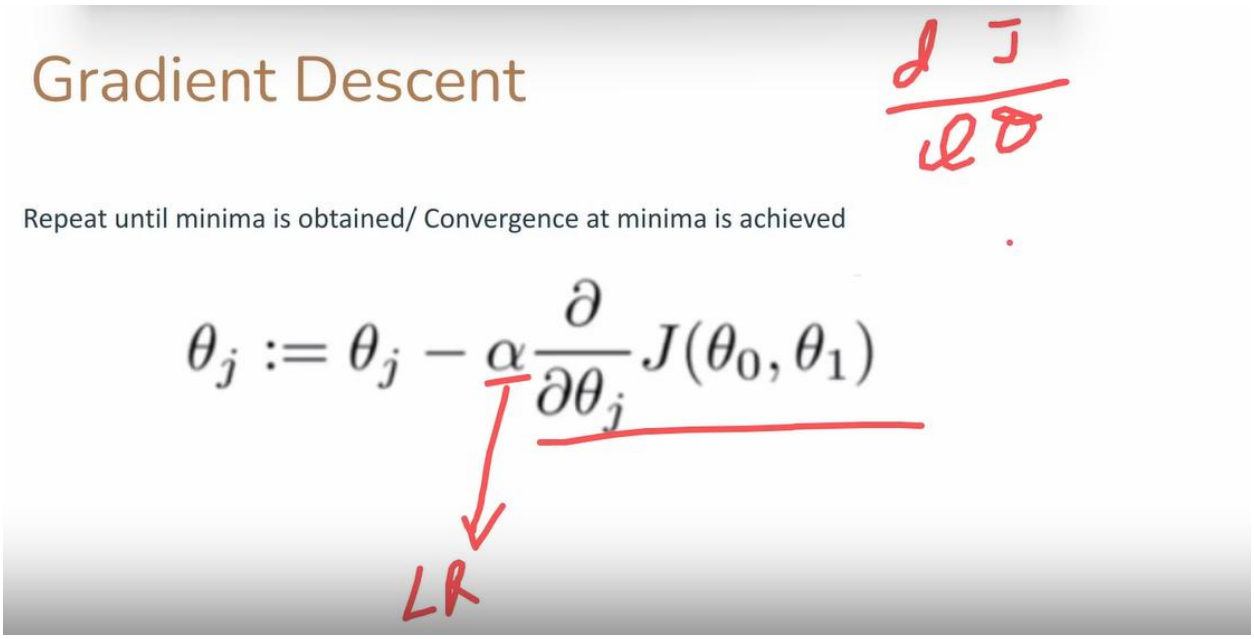
Ex:



If we have a graph like this, as you can see the cross of the bigger parabola is the global minima and the one with squared is local minima or based on that segemnt it is the minima.

Minima of the entire graph or scope , is the minima of the big parabola.

You choose a decent set of paramters and come likewise try to primaly converge at the global minima if not the next best minima hoping the minima next is itelf enough to manage. Basically we have many minimas and someone of the local minimas can come similar or very very small difference with the global minima , so then we can take them into accoount as well , however objective is to find the global minima.



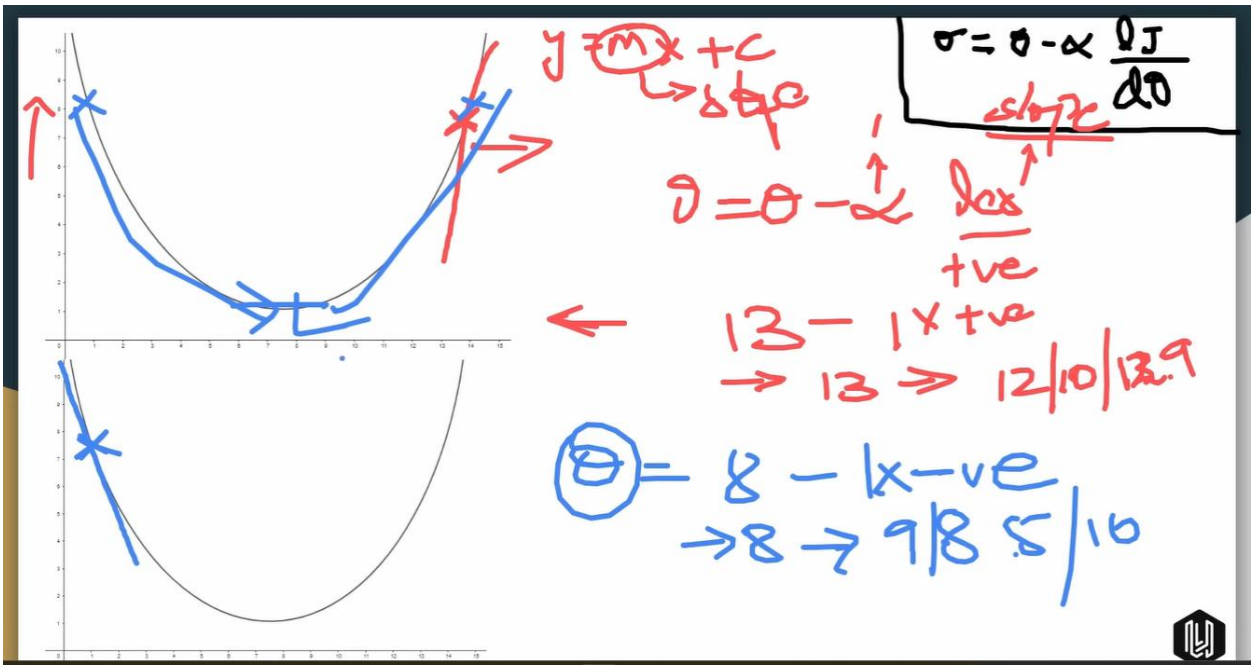
So here is the convergence at minima is achieved. So as you can see , we keep on updating our theta by using a certain value .

Alpha is LR or learning rate. A constant which is initiated at the start of gradient descent.

The other part of the alpha , is the partial derative of the cost function respective to theta or the paramters theta. Its partial derivative because we cannot do a complete derivative due to other paramters as well , so we only can do partial derivative.

This derivative is basically small change , so we simply subtract a small change in cost from the theta.

Learning Rate



The gradient of the curve or the slope is the derivative. We can find the derivative by drawing a tangent to the point on the curve. As we can see ,the first curve with red cross marked, the tangent have a positive gradient. So

$\theta j = \theta i$ - derivative(costfunction)/theta * ALPHA

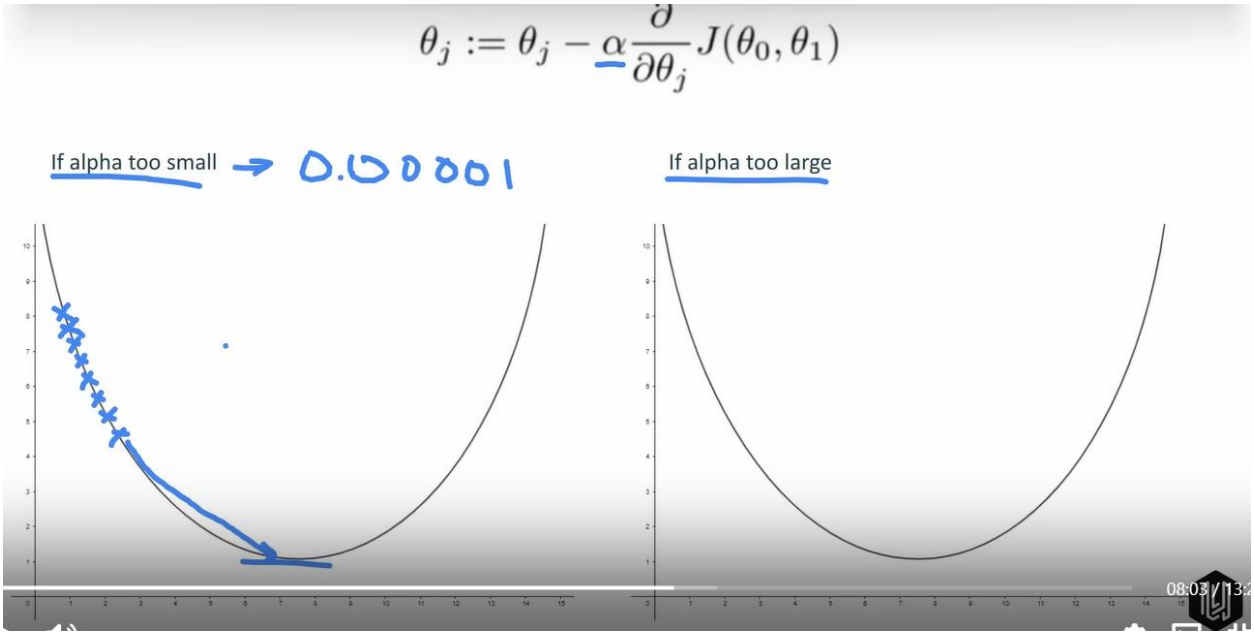
So basically alpha is the slope , θ_1 is the y intercept of the line. The output of the equation gives you a value which is less than the current theta value because we simply subtract a positive value.

However in the second curve , the gradient or the small change in cost function respective to theta is negative hence , it gives a positive increment to the current theta.

The slopes or slope and these outputs gives you guidence and more information where is the minima is or when the gradient is 0. Gives how theta values updated depending on where we exists on the curve.

The above is to get or understand the direction is taken.

What happend if ALPHA too small and too long?

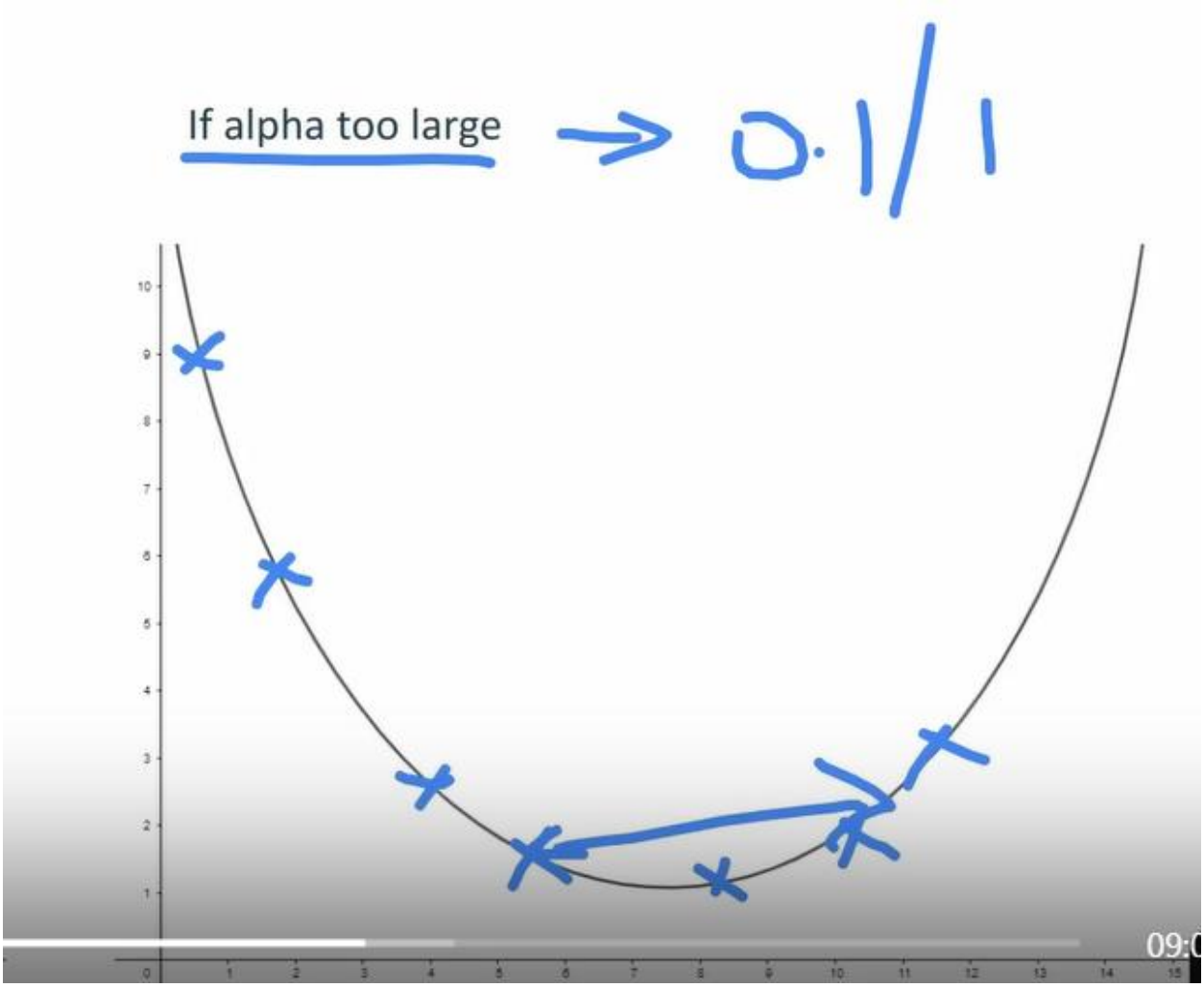


WHEN ALPHA IS TOO SMALL.

The slope is weighted by the alpha , so if the alpha is too low , then the multipler is low , hence the change is low so update itself is very small.

So it will check with small changes always and constantly , and eventually it will reach the minima.

WHEN ALPHA IS TOO LARGE



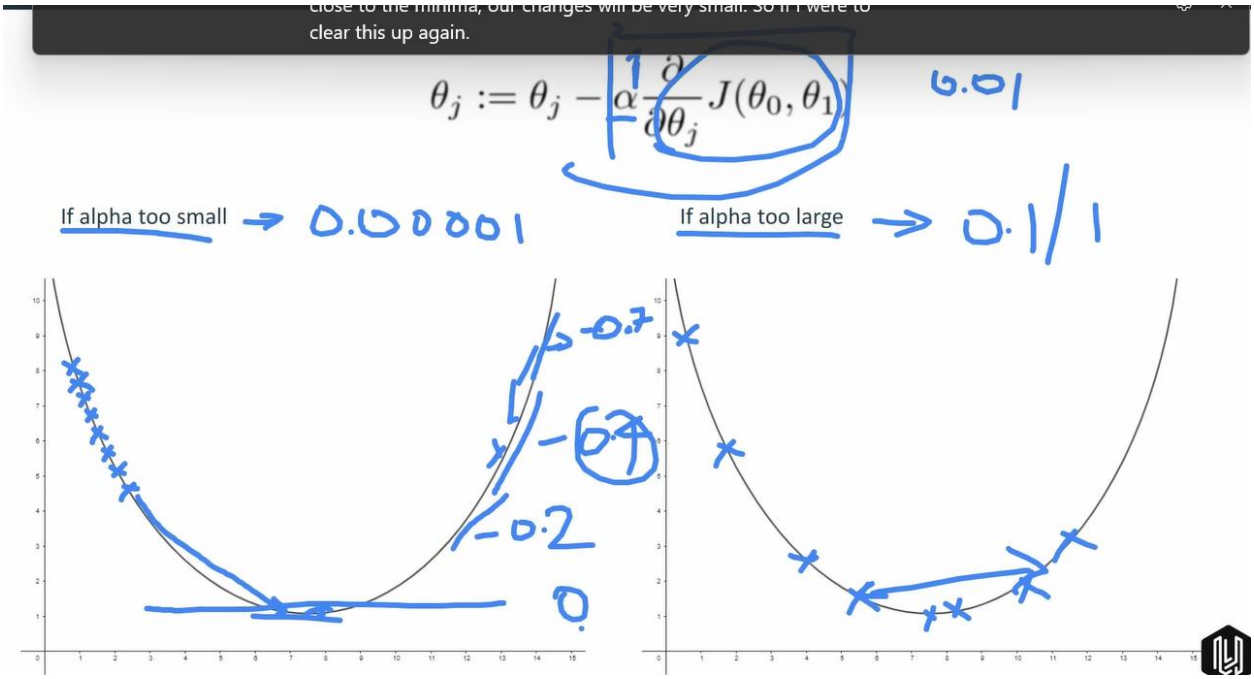
So now channge is very big , hence it will reach minima quickly , however due to the big jumps or changes it might mistakenly or might skip the minima and keep jumping in the directions due to the sign changing. It might not reach the minima but keep jumping in the directions due to the big jumping. So simply I meant was

Initially big jumps from left side towards right

Mistakenly skip the minima and hence the gradient is now positive

So it comes back again to find the minima

a cost function is what a model tries to minimize, and the gradient descent equation is the method it uses to achieve that minimization. The cost function provides the target, while gradient descent provides the step-by-step instructions for reaching it



So as you can see, the if we take the value of alpha too small , so it will keep on updating the slope until we get 0.

a zero gradient indicates that the algorithm has successfully found the global minimum of the cost function. This is the ideal outcome, as it represents the optimal set of parameters for the model

Even in the alpha too small , it will keep on jumping on the minima so we will be approximately at the minima.

Alpha controls THE RATE LEARNING OF THE MODEL. Since alpha controls the gradient descent is functioning and how your model is going to learn, it also known as Hyperparamter.

Theta is the property of the model or the Weight of the model. They define the model.

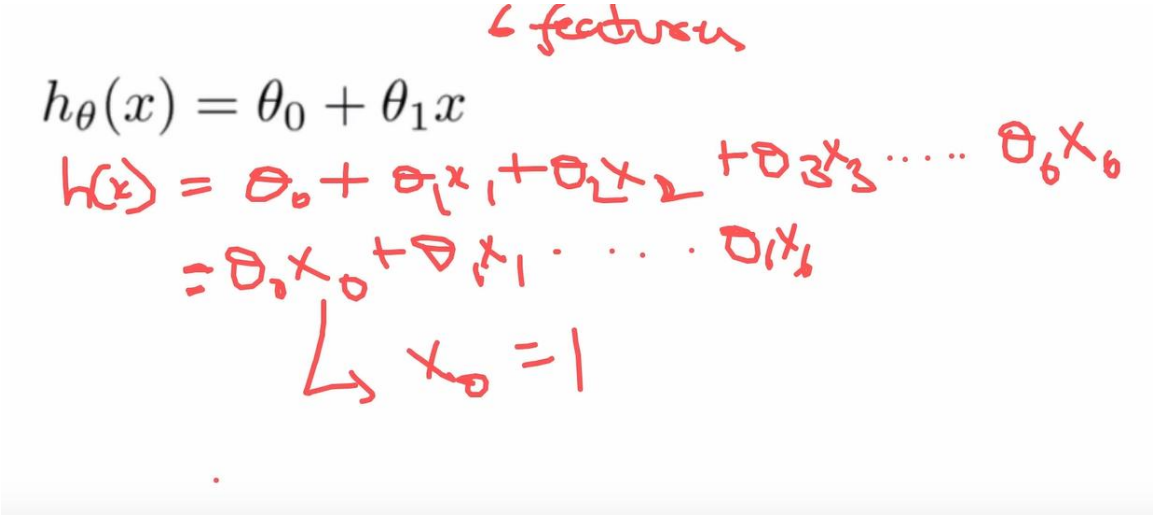
Alpha defines how the model is going to be built

Hyper paramters always control your hyper paramters or parameters which control your parameter learning itself. That's why they are hyper parameters.

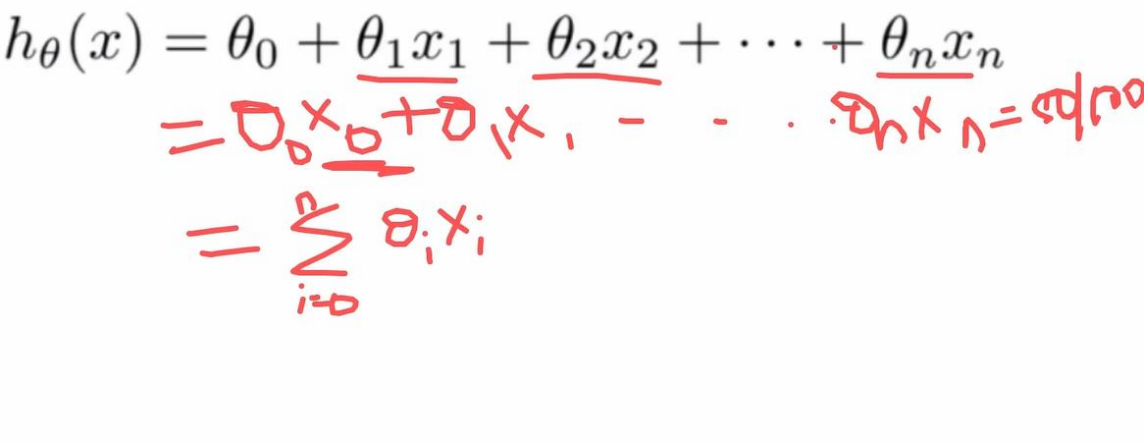
MULTIVARIATE LINEAR REGRESSION

Area in SQFT	No of Rooms	Verandah	Age of House (years)	Patio	Carpet Area in SQFT	Cost in Lakhs
600	2	Yes	12	Yes	495	4.5
1200	3	No	7	Yes	950	6.5

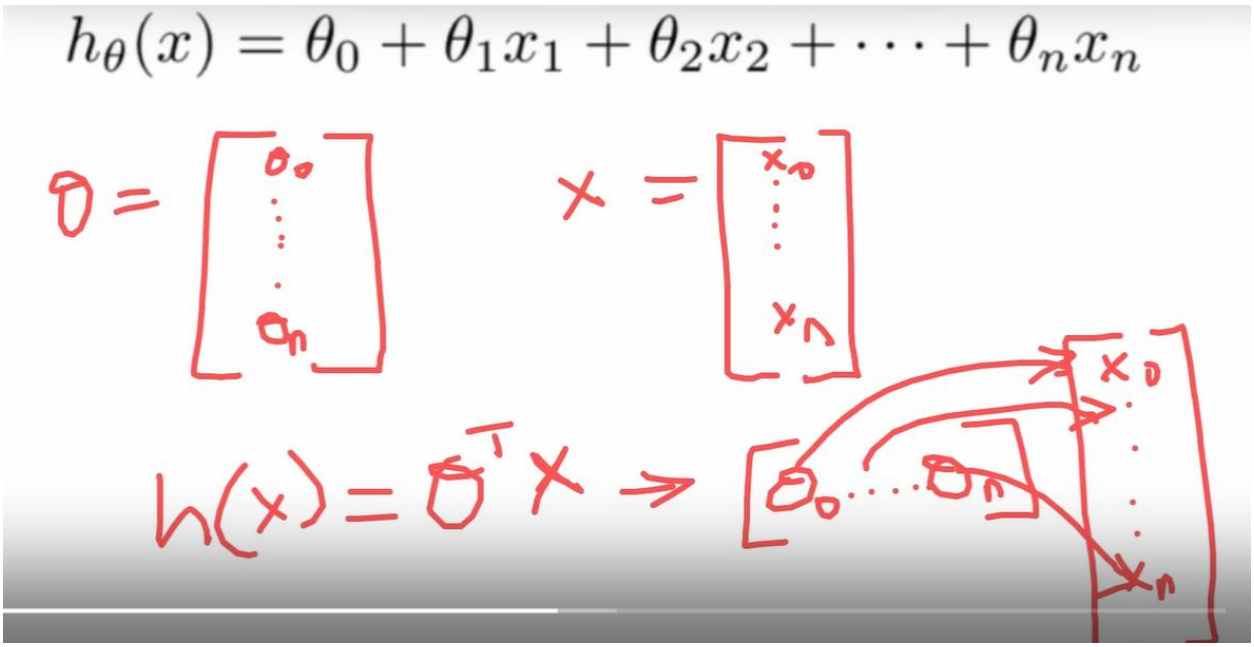
Lets say we have 6 features and , each feature is alternating the cost in lakhs



So for all 6 features the hypothesis is this. Where X0 is 1 since its just theta0 for first term



However , keep taking the summation like this is tiresome hence we can use Vectorization

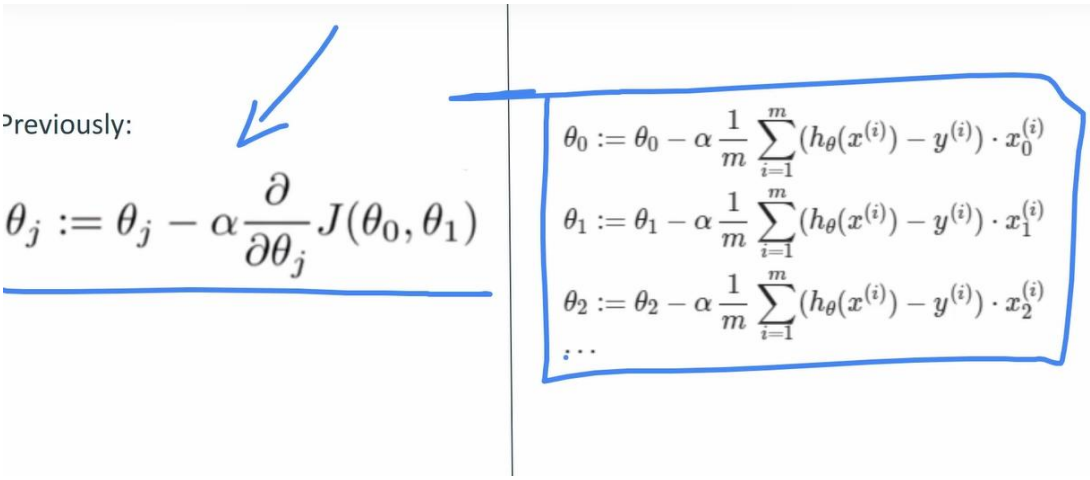


Theta transpose vector multiplied by X. This can be run in just one computation

Where if we are runnign the previous , ex: say havign data of n=1000, then we must run the for loop for 1000 times and keep taking the sum. This takes some time.

GRADIENT DESCENT

In gradient descent we have to actually update the paramters



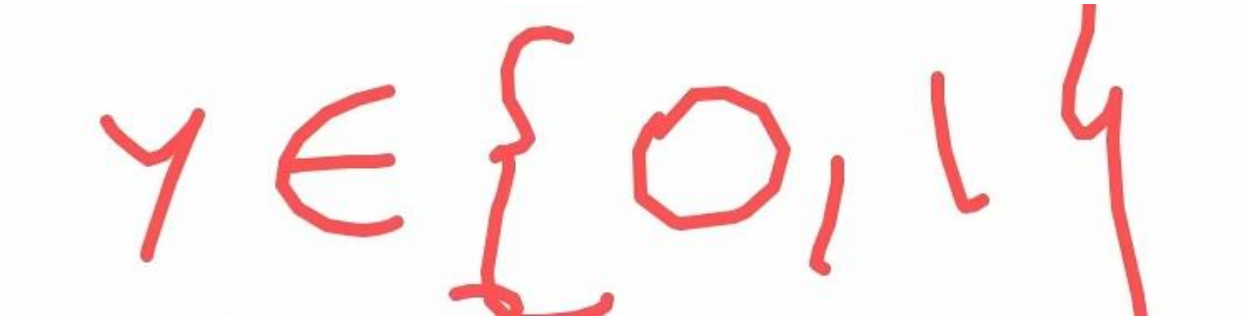
Actually the previous one is for gradient descent for linear but for multi variate regression. So after every iteration, in multi regression we will calculate our cost, take partial derivative of the cost and we will directly update our theta value.(update all theta values once not one theta in one iteration). Does this iteratively until all the thets manage to fit into our equation.

LOGITIC REGRESSION EXAMPLES

- Primary exisits for CLASSIFICATION PURPOSES.
- Logistic focusing on the type of algorithm and the classification if for classification problems. Classification is type of Logistic regression which will be explained later.

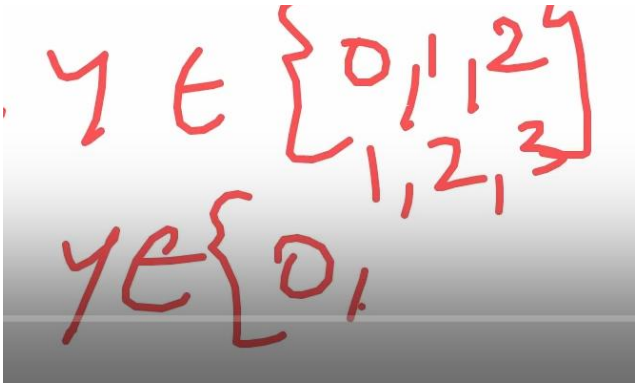
- Emails: Spam/ Not Spam
- Manufacturing: OK/ Not OK
- Medical Imaging: Cancerous Tumor/ Non-Cancerous Tumor
- Dogs: Labrador/ Beagle/ Siberian Husky
- Fruits: Apple/ Banana/ Orange

Therefore it can be ,



Where 0 is not ok a nd 1 is ok.

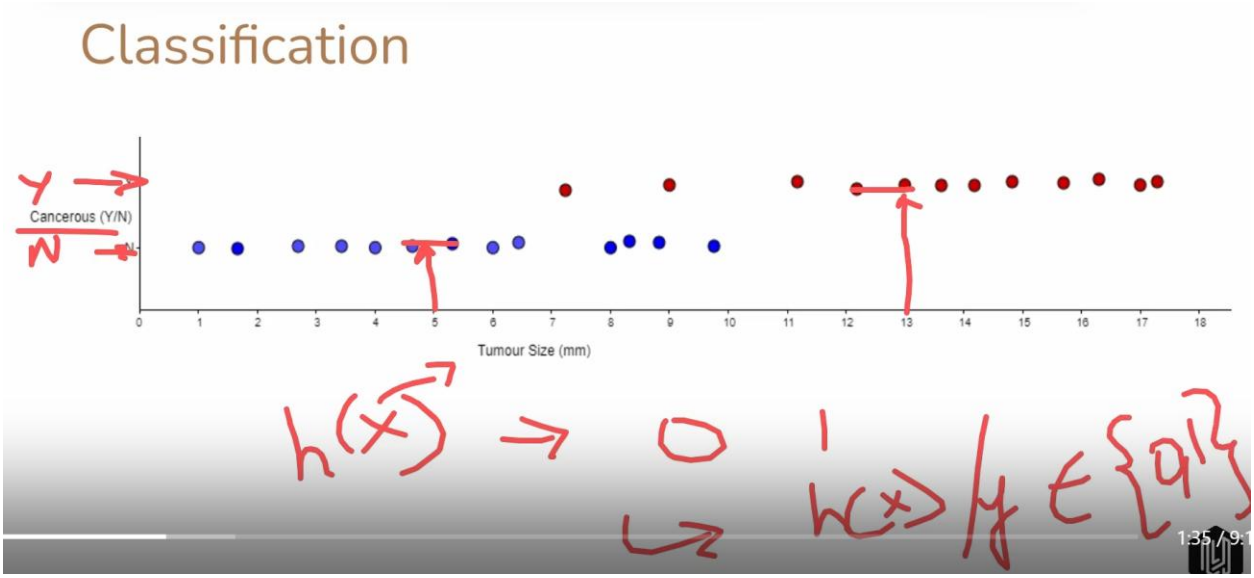
- For fruits it could be 1,2,3 or 0,1,2



- If there are 2 options then 0 and 1 or anyother number , then it is called “Regular Classification”
- If more than 2 , “Multi class classification”

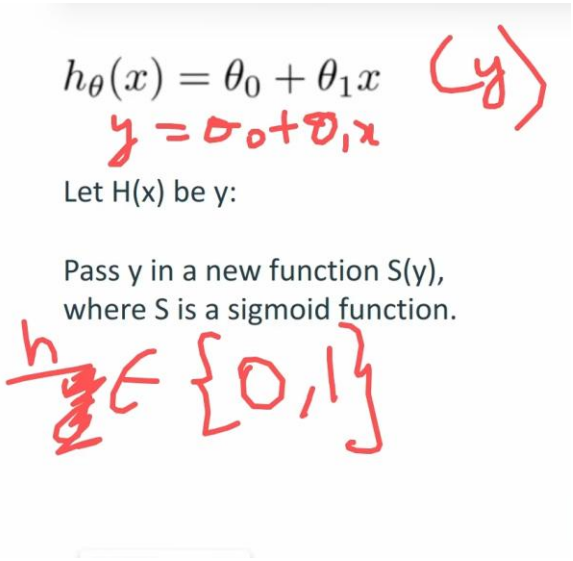
Hypothesis and Model Representation for Logistic Regression

- Hypothesis is a function or model that maps input data to predicted outputs



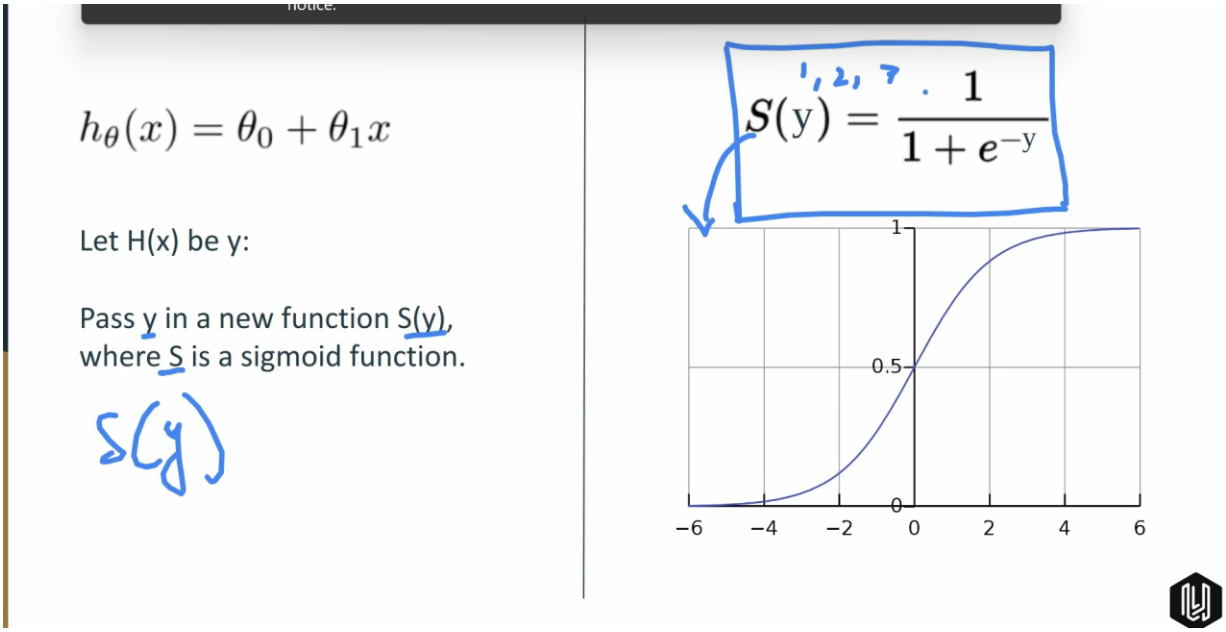
- Here **X** means input or the tumor size in this example , then 0 and 1 are the values where 0 is for non cancerous and 1 is for cancerous.
- Hence hypothesis belongs to 0 or 1

In here **S(y) is a sigmoid function.**

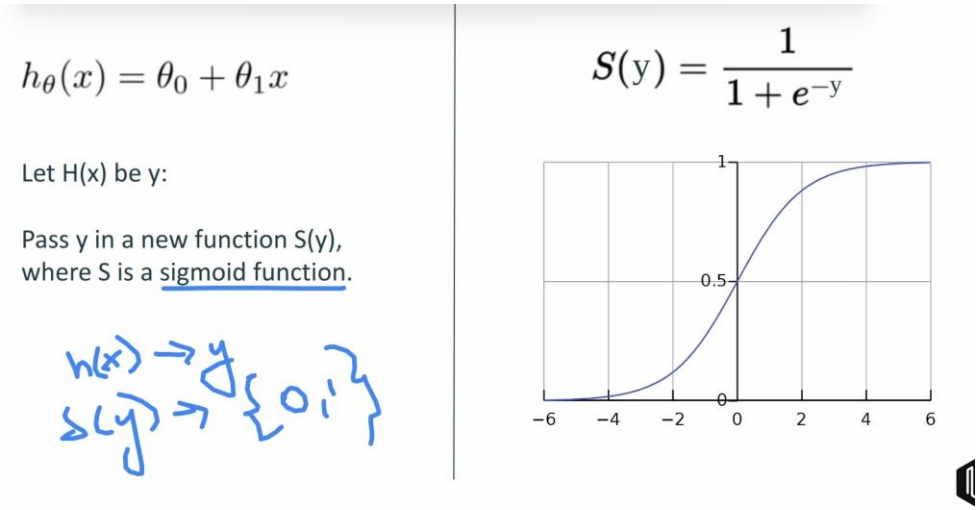


In here , hypothesis is to give an output of 0 to 1.

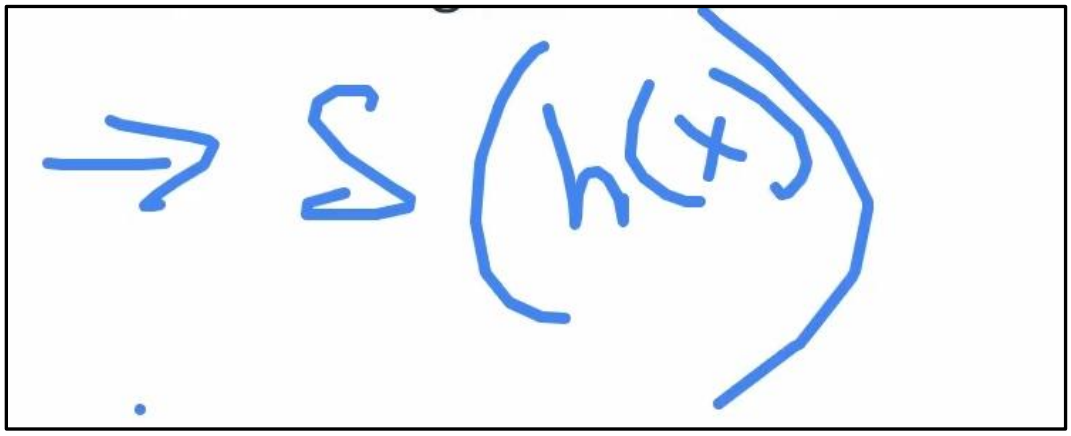
We pass the hypothesis to a new function **S(y)**

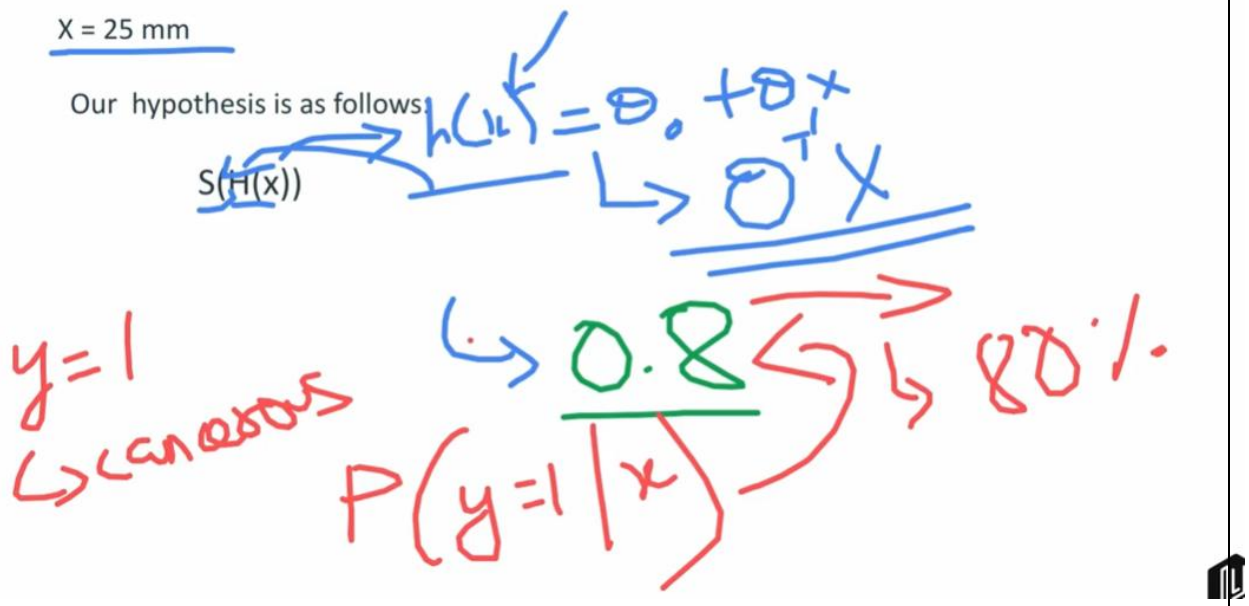


- Sigmoid function is also known as the logictic function and it has values rnaging and outputs ranging from 0 to 1.
- Touches 0 at infinity or even though u see it touches actually the y =0 or touches the x axis when the x value is (-) infinity.
- Likewise the touches 1 at positive infinity,
- Sigmoid function tends to 0 and 1 at negative and positive infinity. Not REACHING 1
- Sigmoid function enables to perform a logistic regression. So this is a logistic function and it is converting an output into a logistic



- The hypothesis h(x) = y (Output of the hypothesis)
- This y is used as parameter for the sigmoid function to give a value of 0 or 1.

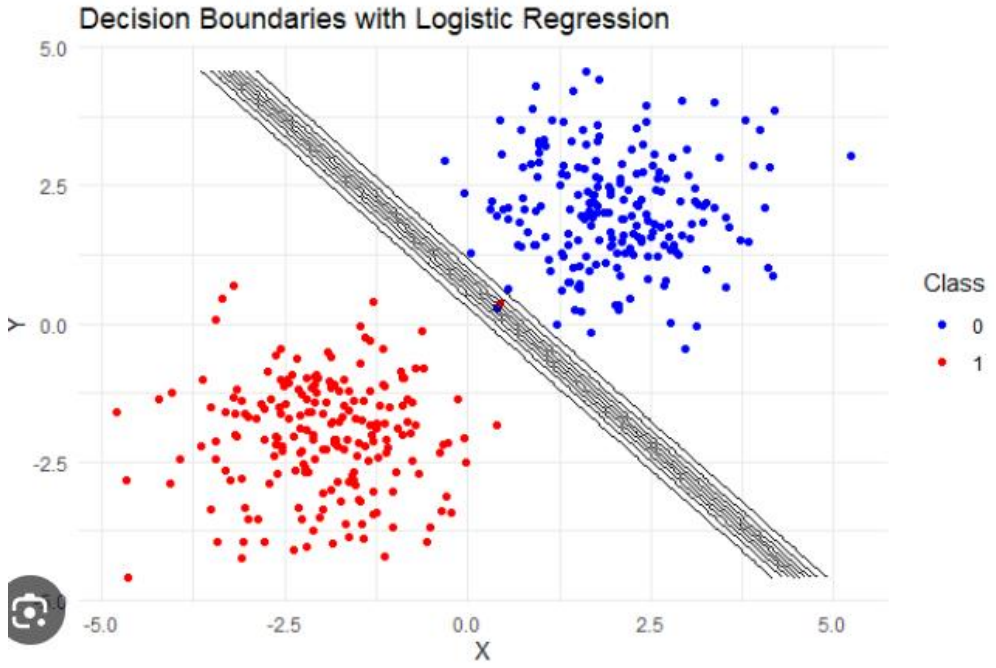


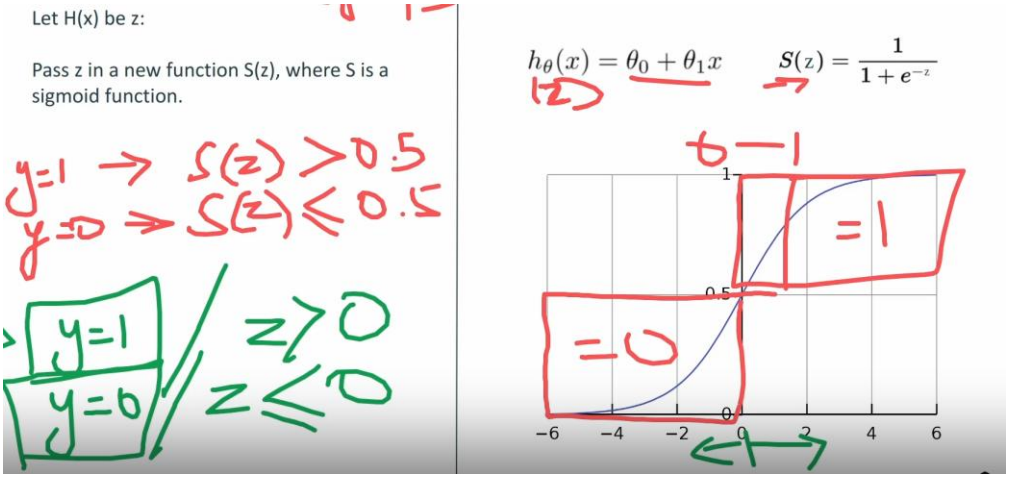


- Remember

DECISION BOUNDARIES FROM LOGISTIC REGRESSION PERSPECTIVE

- The dividing line or surface that separates different classes in a feature space, telling a model where to switch its prediction from one label to another





- We assum h(x) is Z , which is the hypothesis. We pass Z or value found by hypothesis to sigmoid function to get a probability.

- Remember if hypothesis is negative , gives a value <=0.5 and likewise if positive then >0.5. To prove this;

Example	z value	Method 1: z ⩾ 0?	Method 2: S(z)	S(z) ⩾ 0.5?	Prediction
1	z = 5	5 ⩾ 0? ✅ YES	S(5) = 0.993	0.993 ⩾ 0.5? ✅ YES	y = 1
2	z = 3	3 ⩾ 0? ✅ YES	S(3) = 0.953	0.953 ⩾ 0.5? ✅ YES	y = 1
3	z = 2	2 ⩾ 0? ✅ YES	S(2) = 0.881	0.881 ⩾ 0.5? ✅ YES	y = 1
4	z = 1	1 ⩾ 0? ✅ YES	S(1) = 0.731	0.731 ⩾ 0.5? ✅ YES	y = 1
5	z = 0.5	0.5 ⩾ 0? ✅ YES	S(0.5) = 0.622	0.622 ⩾ 0.5? ✅ YES	y = 1
6	z = 0	0 ⩾ 0? ✅ YES	S(0) = 0.500	0.500 ⩾ 0.5? ✅ YES	y = 1
7	z = -0.5	-0.5 ⩾ 0? ❌ NO	S(-0.5) = 0.378	0.378 ⩾ 0.5? ❌ NO	y = 0
8	z = -1	-1 ⩾ 0? ❌ NO	S(-1) = 0.269	0.269 ⩾ 0.5? ❌ NO	y = 0
9	z = -2	-2 ⩾ 0? ❌ NO	S(-2) = 0.119	0.119 ⩾ 0.5? ❌ NO	y = 0
10	z = -5	-5 ⩾ 0? ❌ NO	S(-5) = 0.007	0.007 ⩾ 0.5? ❌ NO	y = 0

SEE? Both methods ALWAYS give the same prediction!

As u can see if hypothesis is negative gives a value less than 0 , and sigmoid function gives value less than 0.5

The reason for comparison z with 0 and sigmoid function with 0.5 is because if we insert z=0 to the sigmoid function , it gives the answer as 0.5.

- Remember $h(x)=\theta_0 + \theta_1$, can be written as $\theta^T x$
- https://drive.google.com/file/d/1LTBc5JKuxnef3Mu7RIxuDkmbtNiOxSBp/view?usp=drive_link

Therefore , $\theta^T x > 0$ means y = 1

If we input the value of which $\theta^T x > 0$ then sigmoid value is greater than 0.5 hence it gives y=1

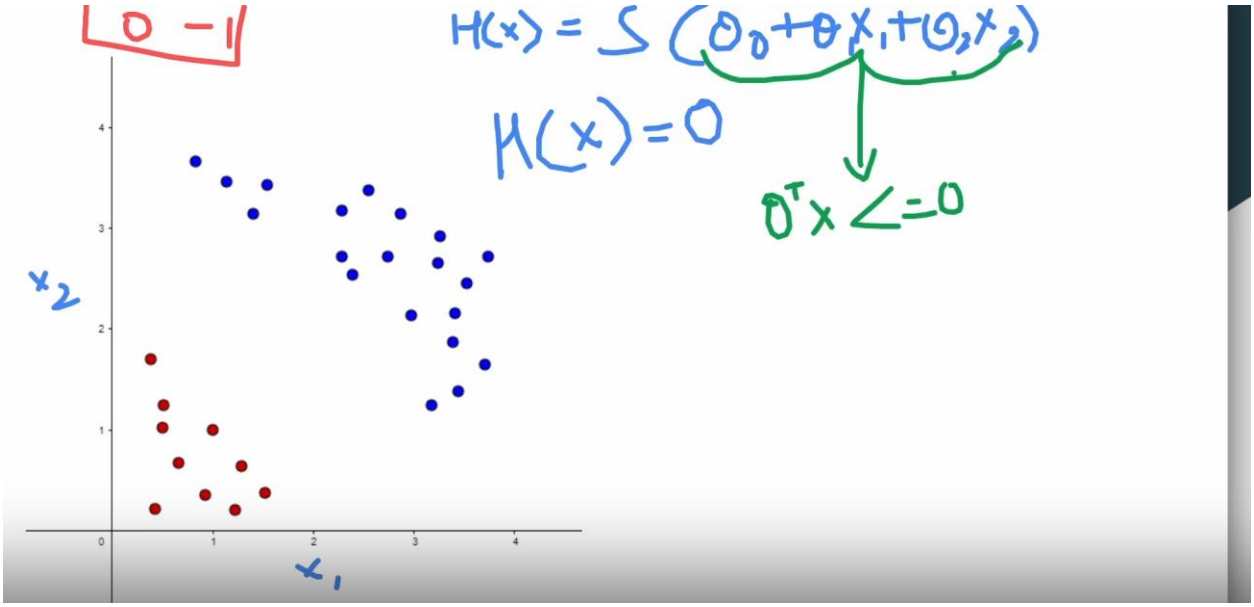
EX:

REMEMBER TO REFER HERE

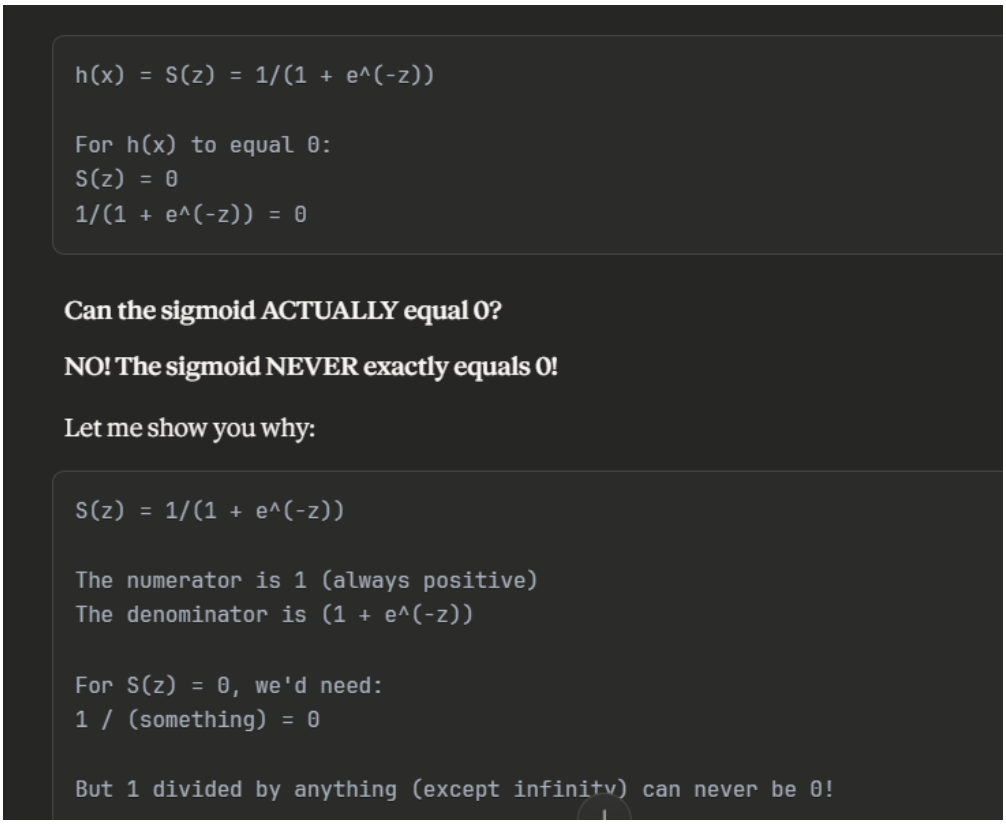
https://drive.google.com/file/d/1m_e22stcznD9TiTopfMkeSLWMKCDlrtM/view?usp=drive_link

For more explanation of the hypothesis why h(x) is not the same as Z and why h(x) is same as the output of the sigmoid function!!!!!

REMEMBER LOGISTIC REGRESSION IS FOR CLASSIFICATION PROBLEMS!



- Remember $z = \theta^T x$, and the value should be <=0 because when



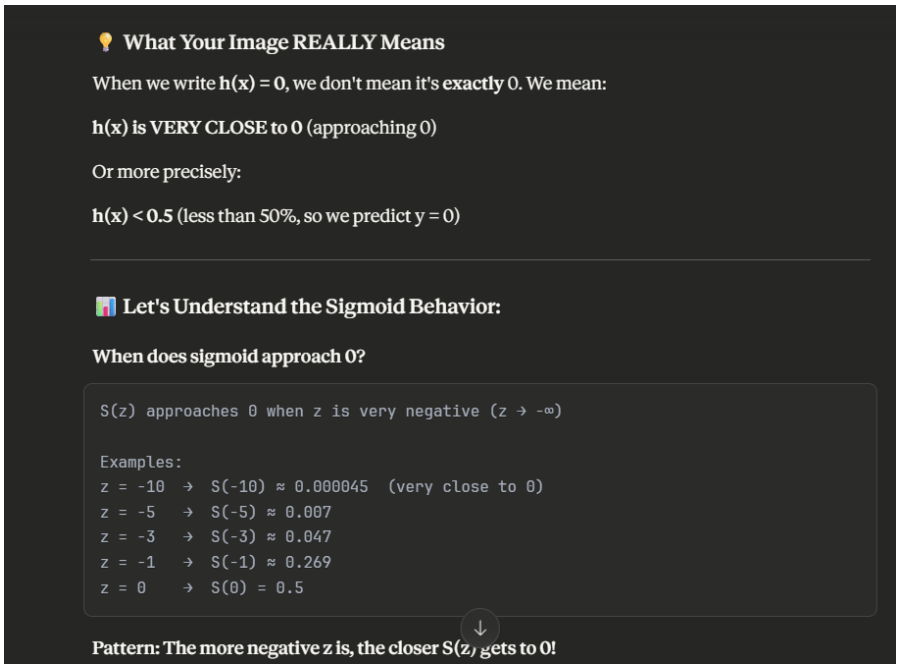
```
h(x) = S(z) = 1/(1 + e^(-z))

For h(x) to equal 0:
S(z) = 0
1/(1 + e^(-z)) = 0
```

Can the sigmoid ACTUALLY equal 0?

NO! The sigmoid NEVER exactly equals 0!

Let me show you why:

```
S(z) = 1/(1 + e^(-z))

The numerator is 1 (always positive)
The denominator is (1 + e^(-z))

For S(z) = 0, we'd need:
1 / (something) = 0

But 1 divided by anything (except infinity) can never be 0!
```



💡 **What Your Image REALLY Means**

When we write **h(x) = 0**, we don't mean it's **exactly** 0. We mean:

h(x) is VERY CLOSE to 0 (approaching 0)

Or more precisely:

h(x) < 0.5 (less than 50%, so we predict y = 0)

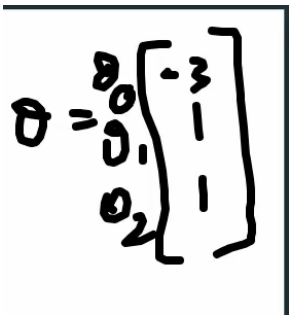
📊 **Let's Understand the Sigmoid Behavior:**

When does sigmoid approach 0?

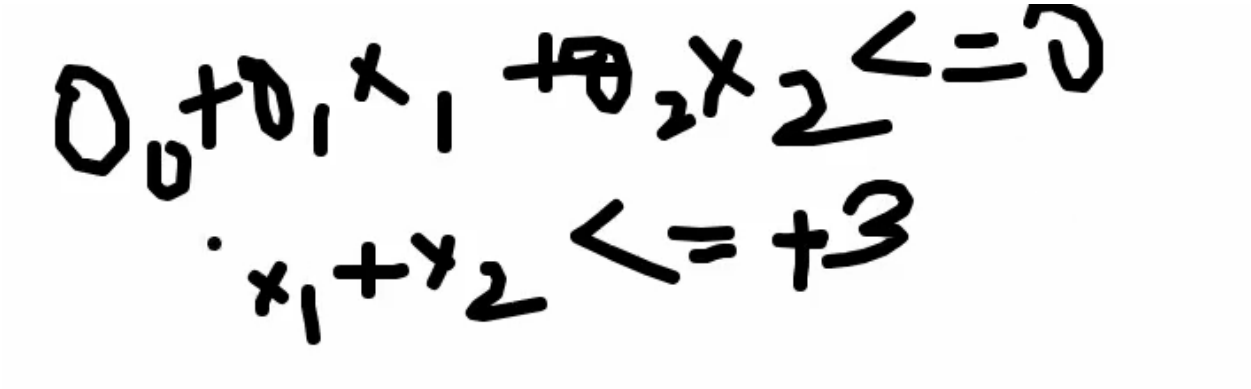
```
S(z) approaches 0 when z is very negative (z → -∞)

Examples:
z = -10  →  S(-10) ≈ 0.000045  (very close to 0)
z = -5   →  S(-5) ≈ 0.007
z = -3   →  S(-3) ≈ 0.047
z = -1   →  S(-1) ≈ 0.269
z = 0    →  S(0) = 0.5
```

Pattern: The more negative z is, the closer S(z) gets to 0!

Ex: Assuming



Therefore:



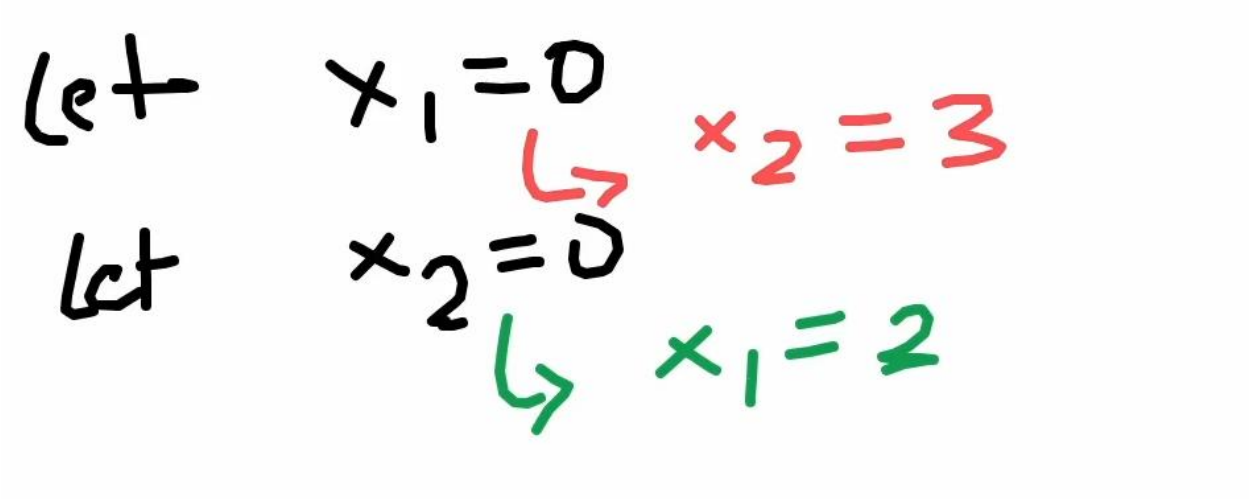
θ0 = -3

θ1=1

θ2=1

BELOW IS FOR THE GRAPH!!

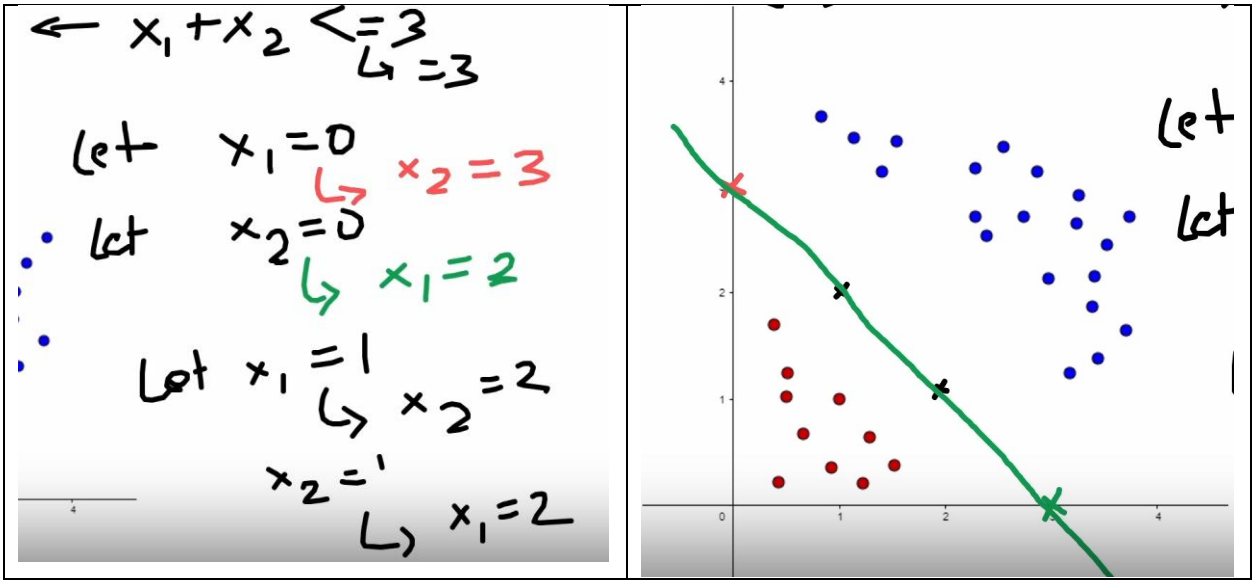
CASES WHEN = 0 OR EQUATION IS X1 + X2 = 3



- This means that if we take the case '=3' and set x1=0 then x2=3

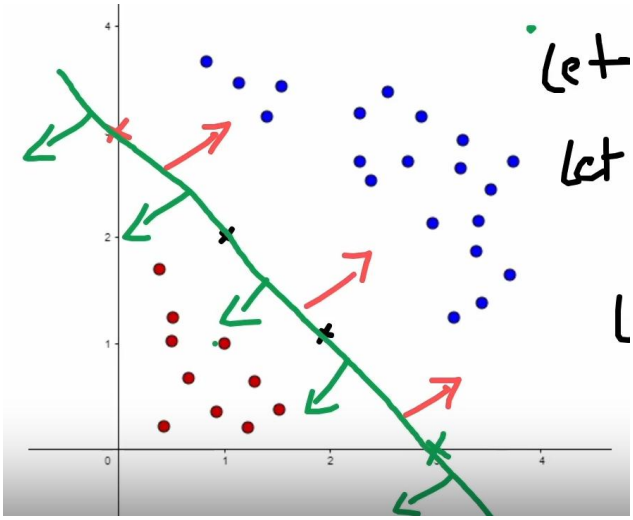
- Similarly for x2=0 gives the same answer

Likewise if we continue sample points and plot then we get:

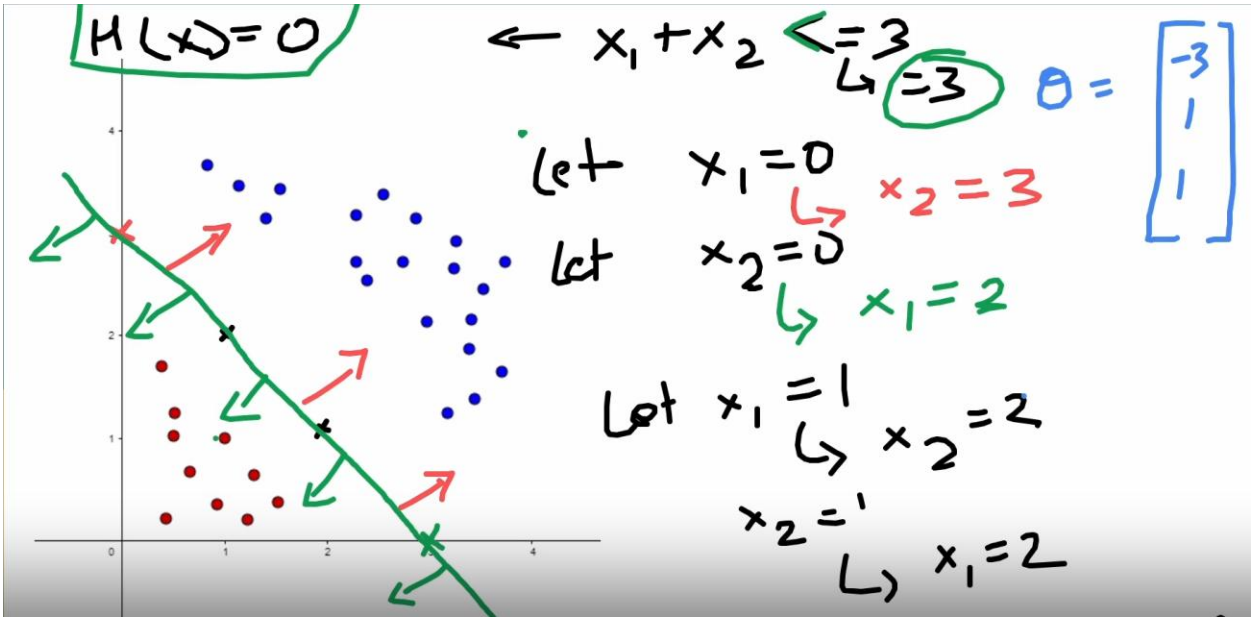


CASES WHEN < 0 OR EQUATION IS X1 + X2 < 3

- If we assume the point (4,4) which is above line it doesn't give the value < 3 or it gives 4+4=8
- But if we assume a point below the line which is say (1,1) then it gives value < 3 1+1=2
- Therefore , values less than the or less than side of the line gives values for h(x) = 0 or classified as h(x)=0 which is same as sigmoid = 0



Red arrows marked as or classified as h(x) = 1 and Green arrows classified as h(x) = 0



- Overall view.
- The theta values remember they are taken from TRAINING , NOT JUST SAMPLE DATA BUT FOR EXAMPLE WE TOOK.

LOGISTIC REGRESSION – COST FUNCTION

- Cost function is how far is the point from the actual value.

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

- Sigmoid function is a non linear function hence we cannot use the same cost function used from the linear regression because non linearity of the sigmoid function causes Mean squared errors and regular mean absolute errors to have very non convex shapes. Basially instead of giving the global minimas or best convex shapes it gives local minimas or less convex shape.

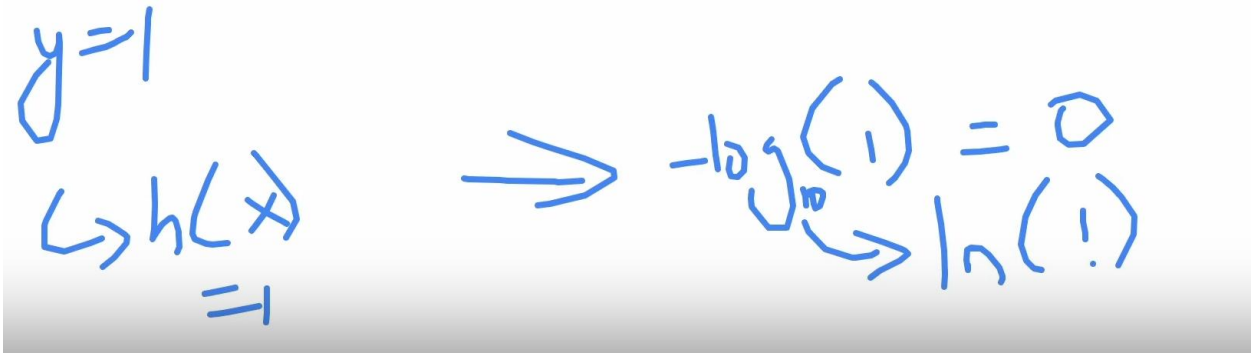
Problems with MSE in Logistic Regression:

- **Non-Convexity:** The sigmoid (S-shaped) function combined with the squared error results in a cost curve with many hills and valleys (local minima) instead of a single bowl shape (global minimum).
- **Gradient Descent Failure:** Gradient descent can get stuck in a local minimum, failing to find the optimal parameters (weights) for the model, unlike in linear regression where MSE is always convex.
- **Wavy/Erratic Behavior:** The cost function becomes non-sensical, with high costs for correct predictions and low costs for wrong ones in certain scenarios, making learning difficult.

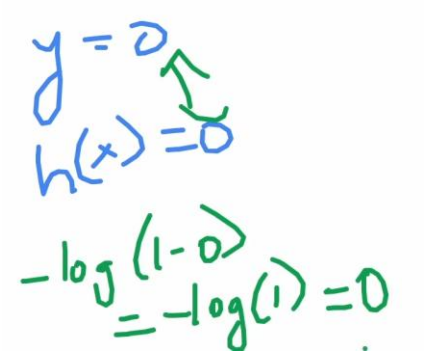
- If we use the same cost function here , left one is for a correct usual shape from linear regression, right one is the from the logistic regression , it will give a pattern like this which produces very non convex and very erratic shape. Hence, difficult for finding minima.

THEREFORE THIS IS THE NEW COST FUNCTION FOR LOGISITIC REGRESSION

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$



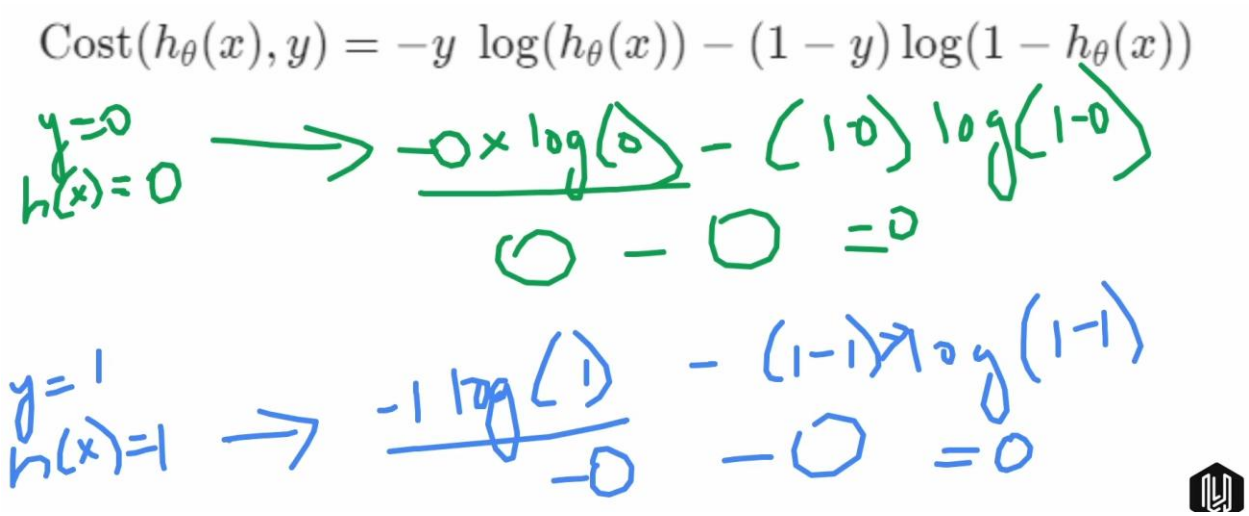
- As you can see , y=1, we use the first equation.
- When y=0 , use the second equation.



- Which means our predicted value is not different from the cost.

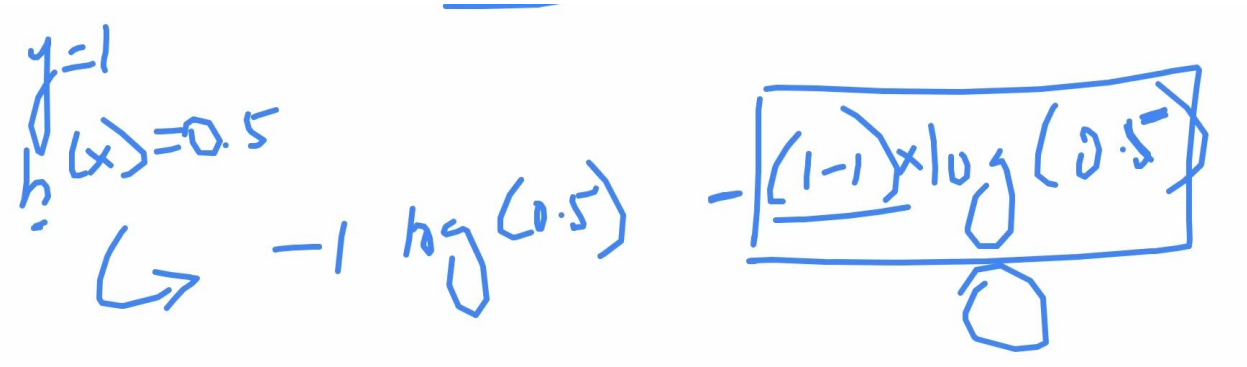
❖ We cant use two equations since the values vary and we must decide which function to use. However , there is an equation which can be used for both the above scenarios when y=0 and y=1

$$\text{Cost}(h_\theta(x), y) = -y \ \log(h_\theta(x)) - (1 - y) \log(1 - h_\theta(x))$$



❖ So basically we check if the predicted value is how far from the actual value. Or the cost.
❖ LOOK , as you can see, there is a pattern,
  - For any Y=1 or Y=0 , it will lead to one of the terms to 0.

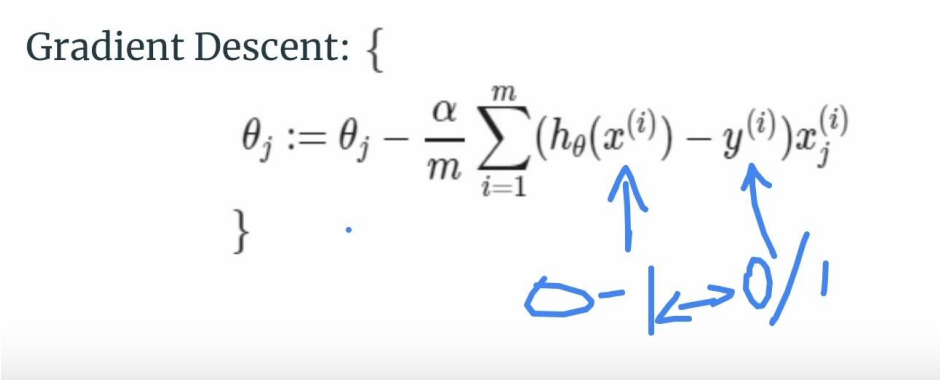
❖ In logistic regression, h(x) is indeed the predicted value, specifically the predicted probability that the output belongs to the positive class (y=1) for a given input x, obtained by passing the linear combination of features through the sigmoid function.
❖ The cost function (cross-entropy loss) then measures the error between this predicted probability h(x) and the actual binary outcome y (0 or 1).



- Actual value is 1 , assuming the hypothesis or the predicted value from the model is 0.5.

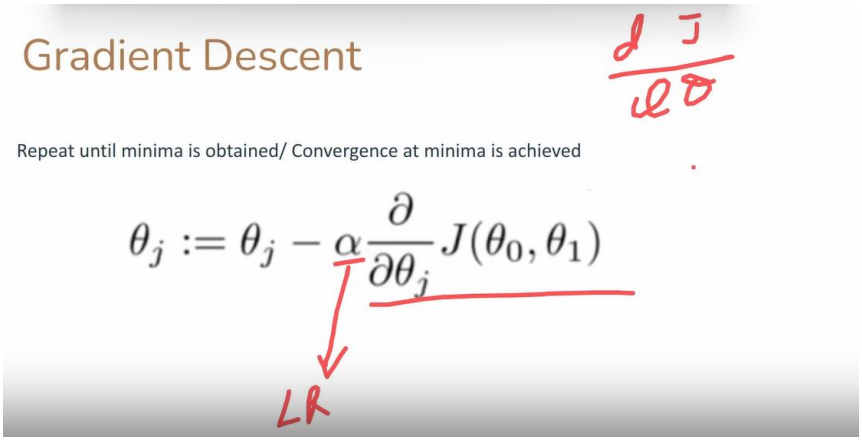
Eventhough our cost function is changed , our Gradient Descent equation is same as the logistic regression as well. Cause it might look different but meaning is the same.

Logistic regression



- In the linear regression gradient descent it also the similar but the second part is different.

Below is from the Linear regression



- As you can see , the theta0 and theta 1 values and the x^I – y^I gives the difference between actual value from the predicted value even if different formulas. And within the range of 0-1 for both.
- Its only that the hypothesis has changed internally but the hypothesis is changed because the actual values have also changed.

Step 1: Separate the two things clearly

Thing 1: Cost function

- Linear: $J(\theta) = \frac{1}{2m}\sum(h(x) - y)^2$
- Logistic: $J(\theta) = -\frac{1}{m}\sum[y\log(h(x)) + (1-y)\log(1-h(x))]$

✅ These are **different formulas**. No debate here.

Thing 2: Gradient

- Gradient = derivative of the cost function w.r.t. parameters
- After you do the math:
  - Linear: $\frac{1}{m}\sum(h(x) - y)x$
  - Logistic: $\frac{1}{m}\sum(h(x) - y)x$

🤯 Looks identical! But notice:

- $h(x)$ is **different for each model**
  - Linear: $h(x) = \theta^T x$
  - Logistic: $h(x) = \text{sigmoid}(\theta^T x)$

❖

Step 2: Why it "simplifies" for logistic

When you take the derivative of **log-loss**:

1. Apply chain rule: derivative of log-loss w.r.t sigmoid output
2. Multiply by derivative of sigmoid
3. Magic happens → the complicated terms **cancel out**

You are left with:

$$\text{gradient} = (h(x) - y) \cdot x$$

- This is the same structure as linear regression
- But $h(x)$ is **sigmoid output**, not linear output

So yes: **cost functions are different**, but the **gradient formula looks the same**.

Think of it like walking down two different hills:

- Linear hill = parabola
- Logistic hill = log-shaped curve

Gradient descent = “look at the slope, take a step down.”

- Step formula = same: `θ -= α * slope * input`
- Only the slope numbers differ because the hills are different

EX:

Step 1: Setup

Suppose we have **one feature** x and **one parameter** θ.

Data:

x	y
1	0
2	1

Learning rate $\alpha = 0.1$

Step 2: Linear Regression

Model: $h(x) = \theta x$

Initial weight: $\theta = 0.5$

Compute predictions:

- $x = 1$: $h(1) = 0.5 * 1 = 0.5$
- $x = 2$: $h(2) = 0.5 * 2 = 1.0$

Compute errors:

- error = $h - y$
- $x = 1$: 0.5 - 0 = 0.5
- $x = 2$: 1.0 - 1 = 0.0

Gradient (average error × x):

$$\frac{1}{m}\sum(h-y)x = \frac{(0.5*1 + 0.0*2)}{2} = 0.25$$

Update weight:

$\theta = \theta - \alpha * gradient = 0.5 - 0.1 * 0.25 = 0.475$ ✅

Step 3: Logistic Regression

Model: $h(x) = \sigma(\theta x) = \frac{1}{1+e^{-\theta x}}$

Initial weight: $\theta = 0.5$

Compute predictions (sigmoid):

- $x = 1$: $h(1) = 1/(1 + e^{-0.5*1}) \approx 0.622$
- $x = 2$: $h(2) = 1/(1 + e^{-0.5*2}) \approx 0.731$

Compute errors:

- error = $h - y$
- $x = 1$: 0.622 - 0 = 0.622
- $x = 2$: 0.731 - 1 = -0.269

Gradient (average error × x):

$$\frac{1}{m}\sum(h-y)x = \frac{(0.622*1 + -0.269*2)}{2} = 0.042$$

Update weight:

$\theta = \theta - \alpha * gradient = 0.5 - 0.1 * 0.042 \approx 0.496$ ✅

Step 4: Look at it carefully

- Both **gradients are calculated as** `(prediction - actual) * x`, just like linear regression.
- Difference: **prediction** is **linear** in linear regression, **sigmoid** in logistic regression.
- Update rule is **exactly the same structure**.

This is why your brain says:

> "Wait... the cost is different, the output is different, but why does the formula look the same?!"

Because the magic is: **sigmoid + log-loss derivative = prediction - actual**. That's why the update looks identical.

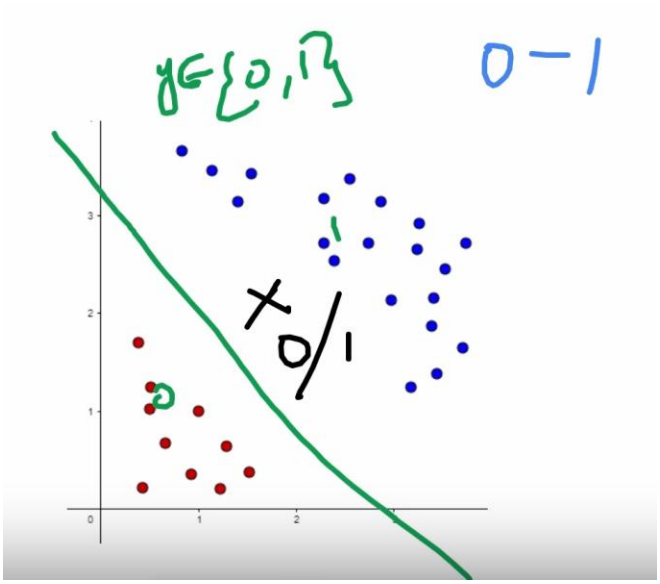
TL;DR with numbers

Model	prediction	error	gradient	θ update
Linear	0.5, 1.0	0.5, 0	0.25	0.475
Logistic	0.622, 0.731	0.622, -0.269	0.042	0.496

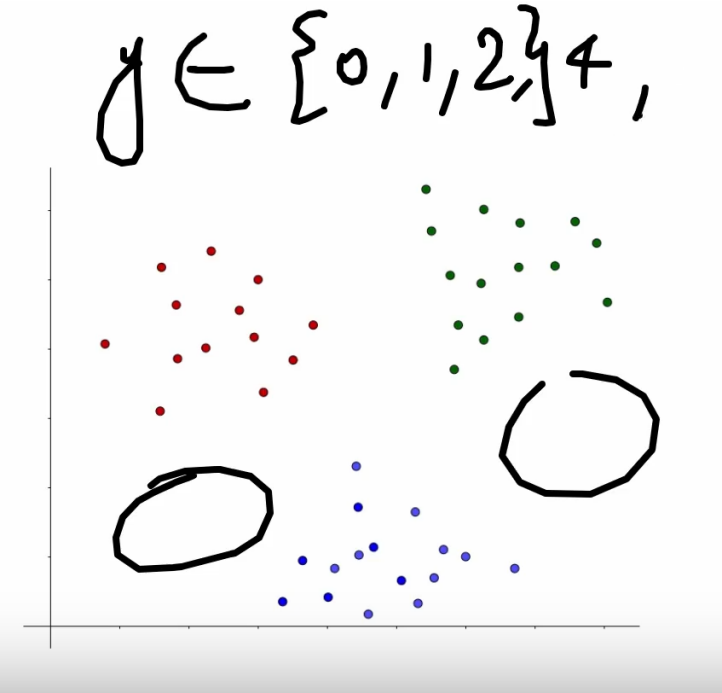
- Same **idea:** `θ -= α * error * x`
- Different **numbers** because predictions differ

LOGISTIC REGRESSION MULTICLASS CLASSIFICATION

This is binary class classification due to only 0,1

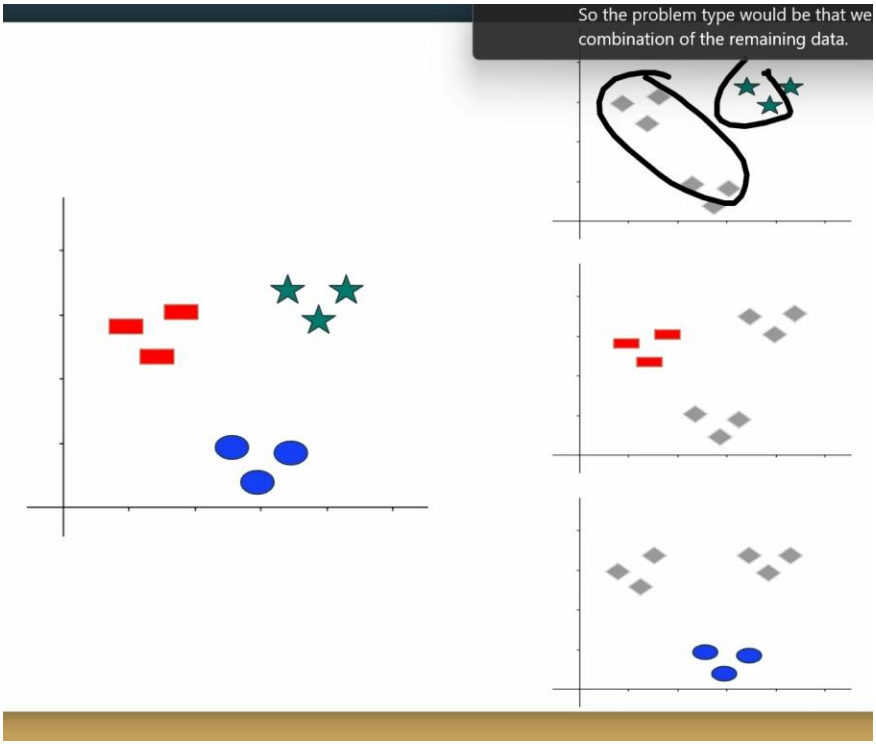


Multiclass classification



If 2 class = binary and if more than 2 then multiclass classification

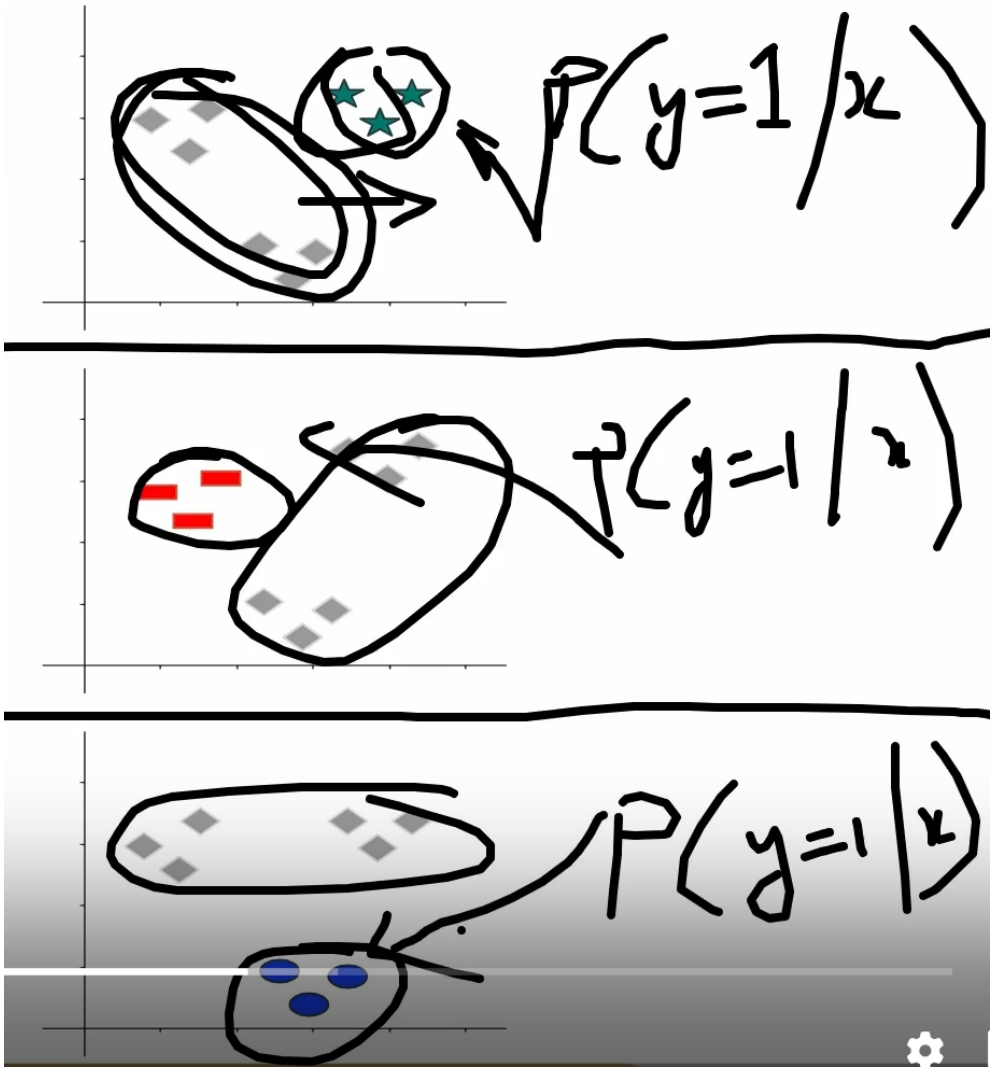
How to tackle multiclass classification?



We are perfoming 1 vs combination of other remaining data. Basically seperating them and perfoming 1 vs with others.

Perform groups seperately to others as well.

Likewise probability comes for 3 different scenarios :



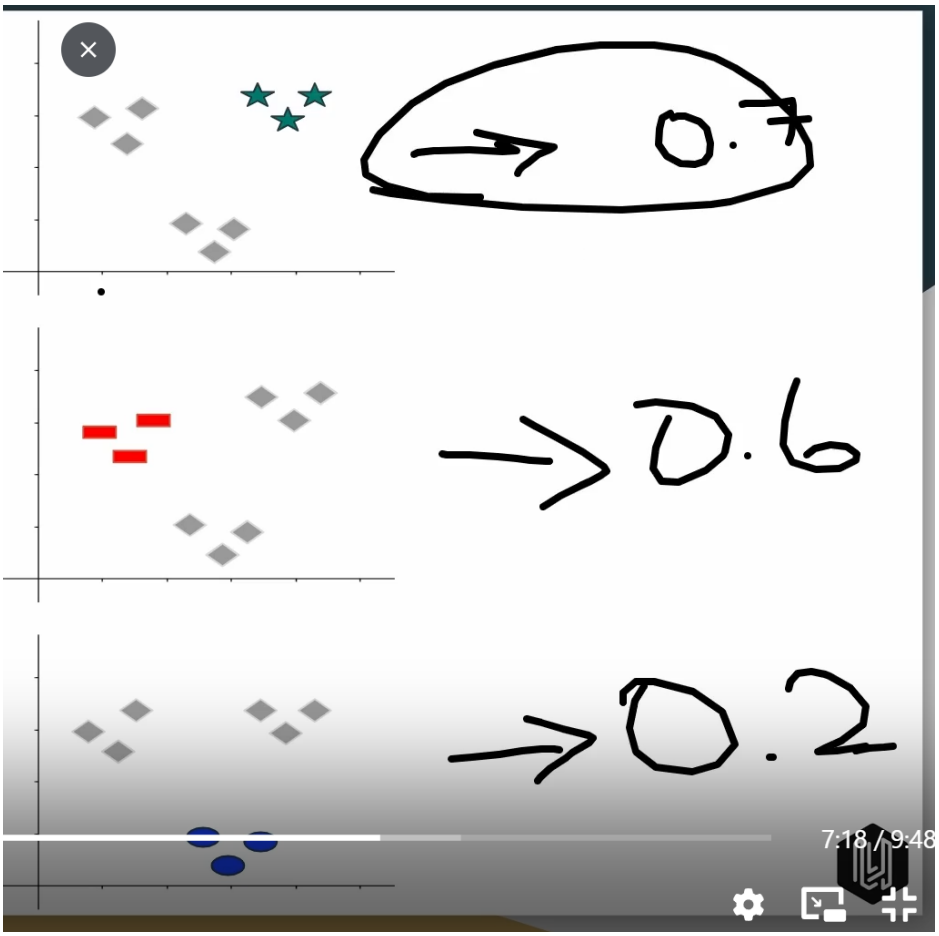
Each individual probability is responsible for the probability of each group its tackling.

- **Out of all choose the one** which gives the **maximum probability**

Ex:

Basically this is about a scenario where we get an unknown point like shown in first binary graph but in multiple class groups.

So we take the probability with the highest value which represents that the relevant value belongs to that particular group.



One-vs-Rest

- Train a Logistic Regression for each class in the dataset. Where h(x) is the probability for the class.
- Then for any given input for x, make a prediction for the class which has the maximum probability out of all the logistic regressions.

Underfitting and Overfitting

UNDERFITTING

Underfitting occurs when a model is too simplistic to capture the underlying patterns in the data, resulting in poor performance on both training and test datasets.

Characteristics:

- Misclassification of data points from multiple classes.
- Oversimplified decision boundaries that fail to effectively separate classes.
- High bias and low variance.
- Poor performance on both training and test dataset

- Occurs when a model is too simplistic to capture the underlying patterns in the data resulting in poor performance on both training and test datasets.
- Model is not trained well to perform.
- High bias and low variance

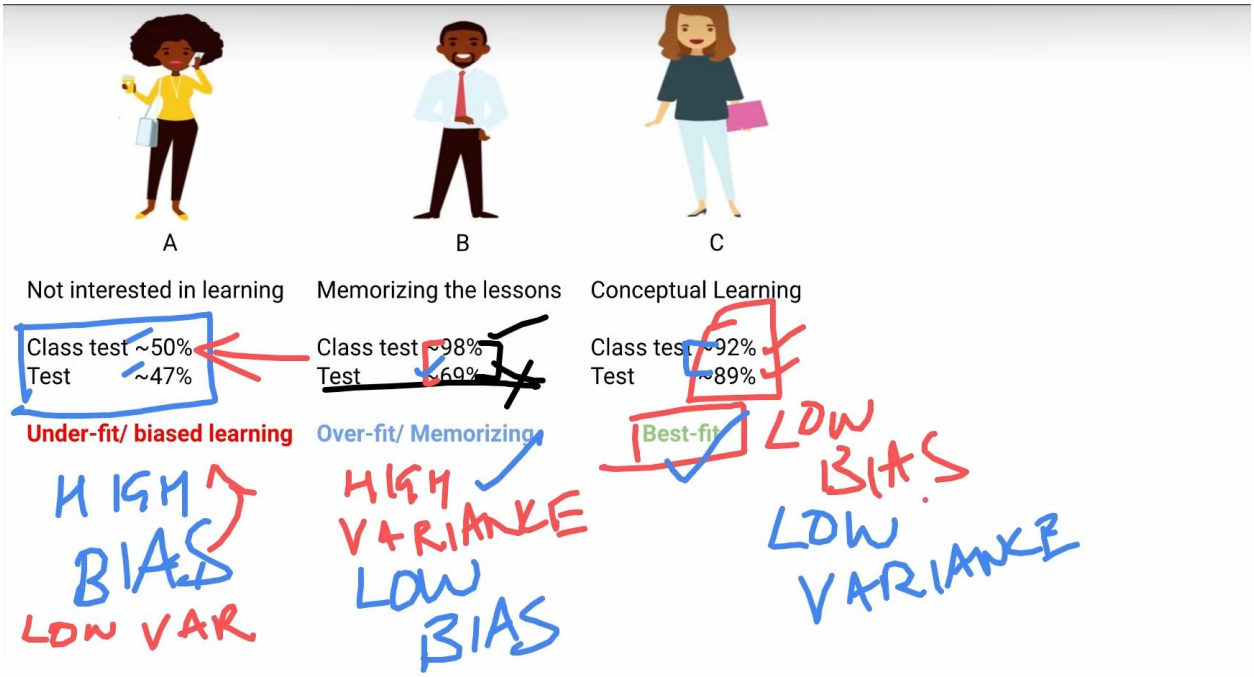
OVERFITTING

Overfitting occurs when a model captures noise or random fluctuations in the training data, resulting in excellent performance on the training dataset but poor generalization to unseen data.

Characteristics:

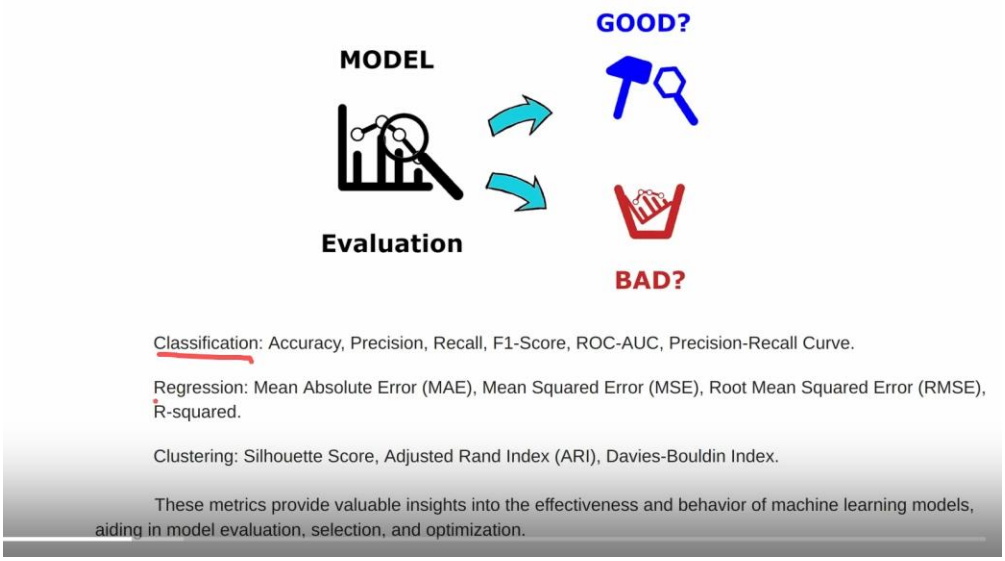
- High accuracy on the training data but significantly lower accuracy on the test data.
- Complex decision boundaries that fit the training data too closely, capturing noise or irrelevant patterns.
- Low bias and high variance.
- Sensitive to small fluctuations in the training data

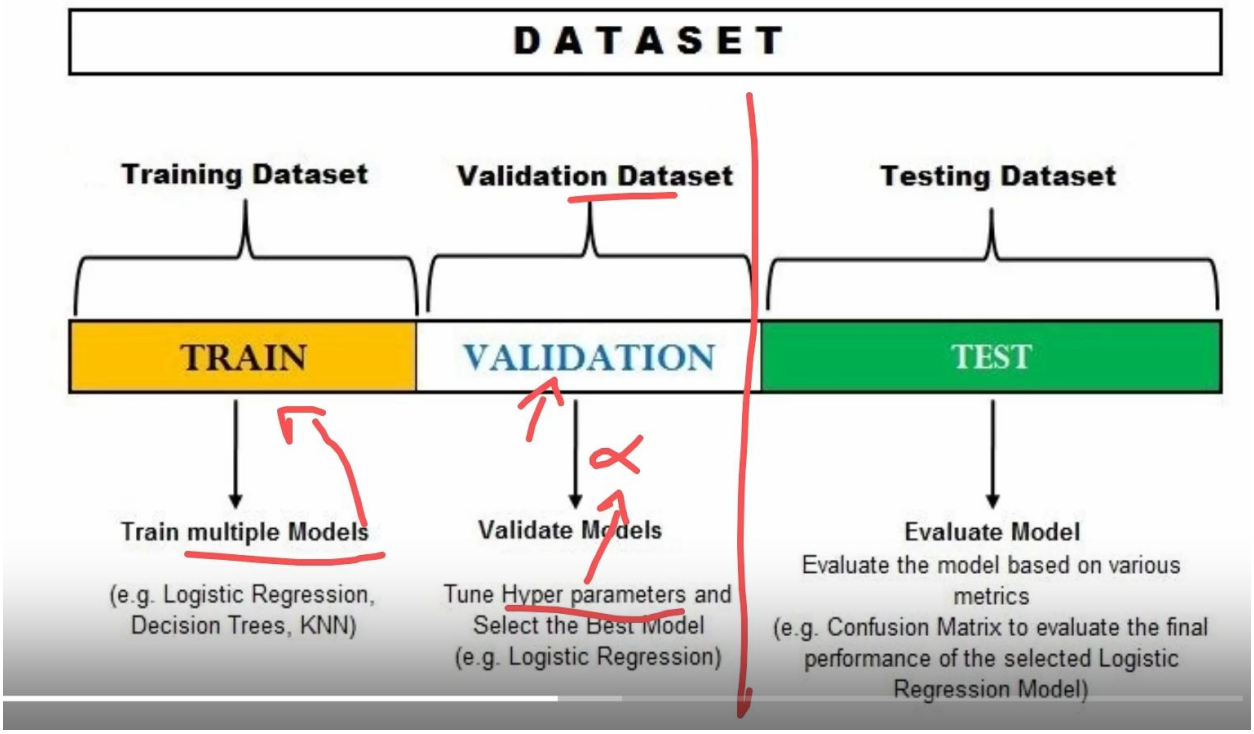
- Trying to capture everything
- Gives good performance on training data but from unseen data.
- If very low accurary -> high bias
- If significant drop between then high variance(basically standard devistion or spead)



Model evaluation

- We have a matrix called “performance matrix” to evaluate how the performance is.
- Vary from task to task:





- Hyper paramters = example: alpha

Regularization

What is Regularisation?

Regularization is a technique used in machine learning to prevent overfitting by adding a penalty term to the model's loss function. Its primary purpose is to discourage the model from learning overly complex patterns in the training data that may not generalize well to unseen data

Regularization helps to prevent the model from fitting the noise in the training data and instead focuses on capturing the underlying patterns.

- Is a technique used in ML , to prevent overfitting by adding a penalty to the model's loss function.
- Prevent Overfitting.
- Helps the model to generalize and learn the patterns which are noise or test data.

Model: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{m}\theta_j^2$$

- The underlined is the Regularization term which is controlled by a hyper parameter, LAMDA.
- We penzalize the cost so it promotes not to learn too much specific and to pinpoint parameters. (Prevent from overfitting)
- If LAMDA IS TOO HIGH , The penalty is strong, forcing the model's coefficients to be very small, possibly even zero. This is because they are input terms which

Case 1 : Normal rule (good)

- Small slopes allowed
- Medium slopes allowed
- Line can tilt enough

✅ Data trend is learned

Case 2 : Very strict rule (high penalty)

Now the rule becomes:

> "Almost no slope allowed."

So the model thinks:

- "If I tilt the line, I get punished."
- "Better keep the line almost flat."

Result:

- Line ignores the data
- Predicts almost the same value everywhere

❌ This is **underfitting**

Why does higher penalty cause underfitting? (plain words)

Because:

> **The model is not allowed to move enough to match the data.**

It is **too restricted**.

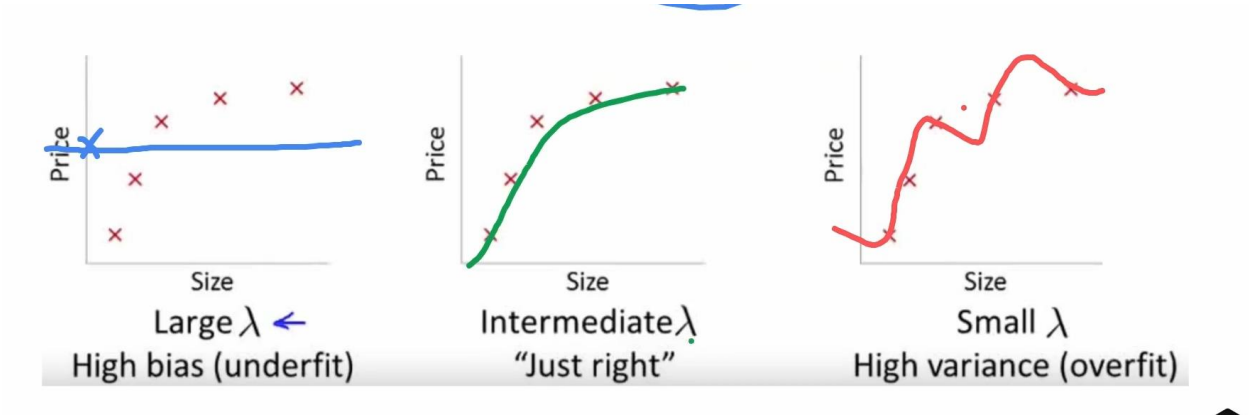
One daily-life analogy (this usually clicks)

Imagine learning to ride a bike:

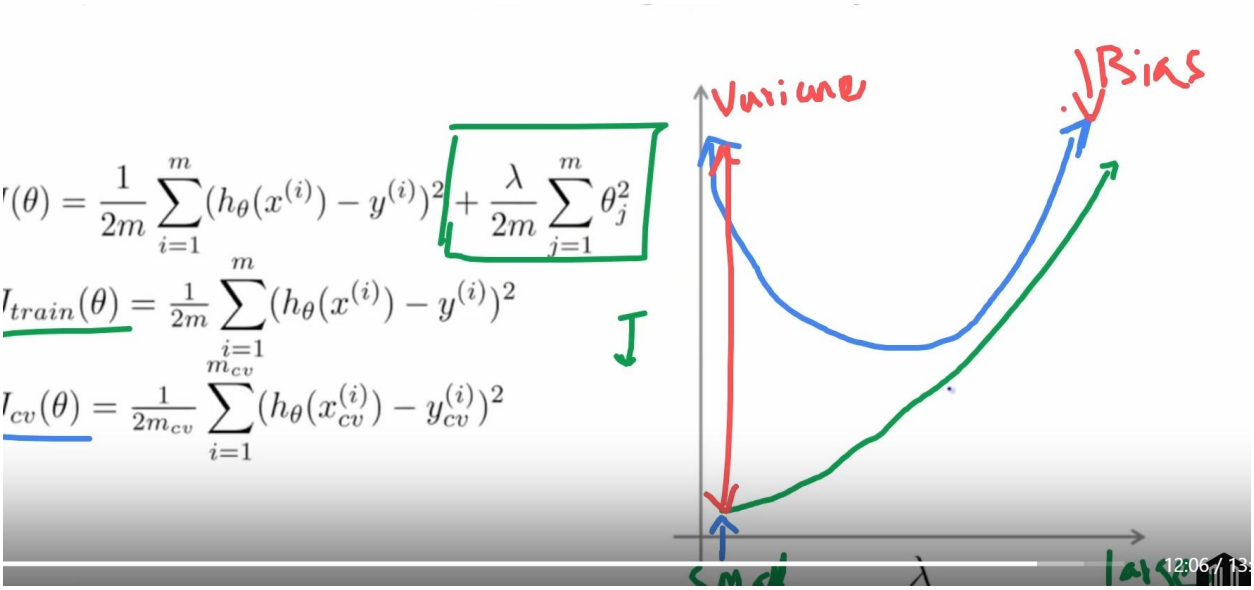
- No rules → you fall everywhere (overfitting)
- Some rules → you learn balance (good fit)
- **Too many rules → you can't move** (underfitting)

High regularization = too many rules.

- If the term is too low , then it almost same as the original cost function, so more trainng process is free to go to alter the parameters as it likes and hence leads to OVERFITTING.



Bias/Variance as a function of regularization parameter



What Lambda Really Does

You're right to be confused! Here's the key:

When λ is LARGE:

- The regularization term `(λ/2m) Σ θ_j²` becomes VERY IMPORTANT in the cost function
- To minimize the total cost J(θ), the model is **forced** to make θ values smaller
- Why? Because large θ values would create a HUGE penalty when multiplied by large λ
- **Result:** Small θ values → Simple, smooth model (blue curve)

When λ is SMALL (close to 0):

- The regularization term `(λ/2m) Σ θ_j²` becomes LESS IMPORTANT
- The model doesn't care much about the penalty
- So θ values can be large without much punishment
- **Result:** Large θ values allowed → Complex, wiggly model (green curve)

Think of it like a fine/penalty:

- **Large λ** = Expensive parking fine ($1000). You'll be VERY careful to keep your parameters small to avoid the huge cost!

- **Small λ** = Cheap parking fine ($1). You don't care much, so your parameters can grow large.

The Math:

If θ = 100 and:

- λ = 0.01 → penalty = 0.01 × 100² = 100 (small penalty, θ can stay large)
- λ = 10 → penalty = 10 × 100² = 100,000 (HUGE penalty, forces θ to shrink!)

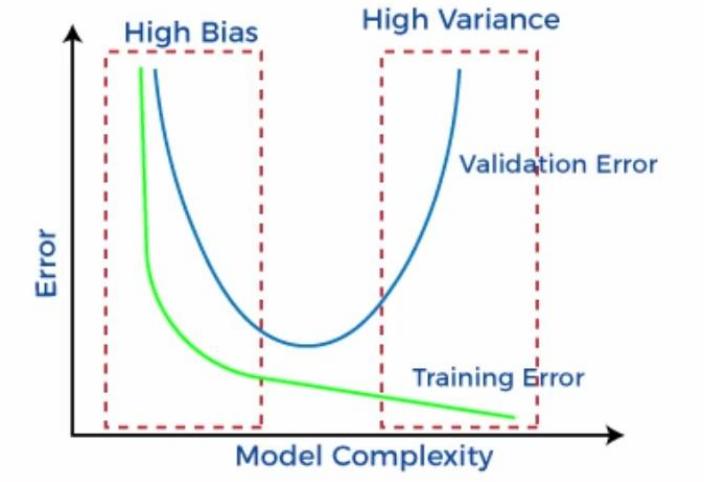
BIAS AND VARIANCE

BIAS

- Bias is the difference between the predicted and the true values. Also referred as the error made by the model.
- As Bias increases , or as the error by the model increases , model predicts less accurately on the training dataset. Higher bias refers to high error in training.
- When model not performing well in training data set then the errors is consistent throughout for others two data sets(Validation and the test) as well.

VARIANCE

- How sensitive/inconsistent your model is to different training data
- Variance = How spread out your predictions are across different training sets
- If you train model 100 times on slightly different data:
    - Low variance → predictions stay similar
    - High variance → predictions are all over the place



The Two Curves

Green Line = Training Error

What it shows: How well your model fits the training data

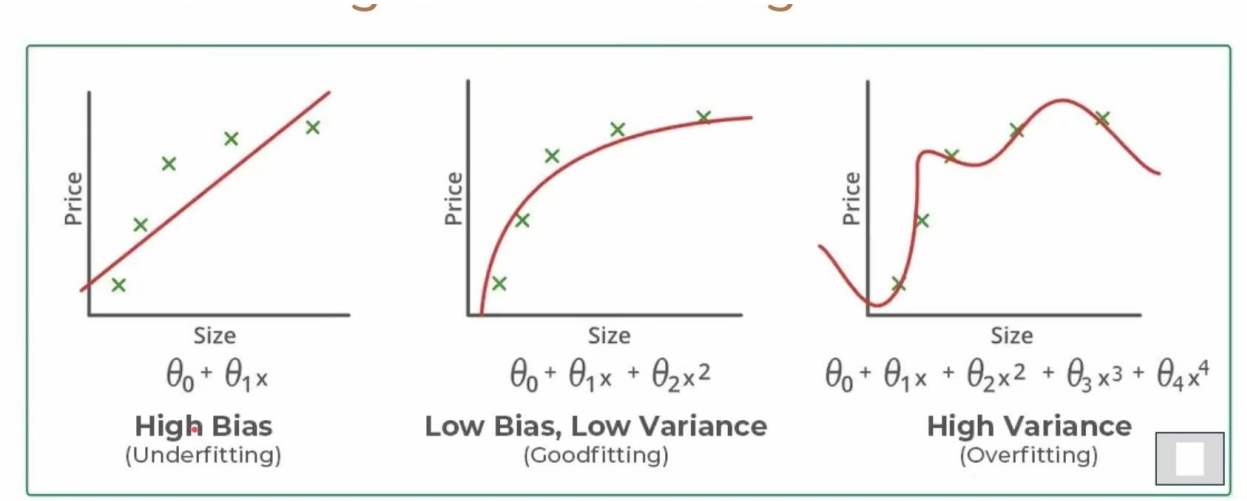
- **Left side (high):** Simple model can't even fit training data well
- **Keeps going down:** As model gets more complex, it fits training data better and better
- **Almost reaches zero:** Very complex models can memorize training data perfectly

Blue Line = Validation Error (Test Error)

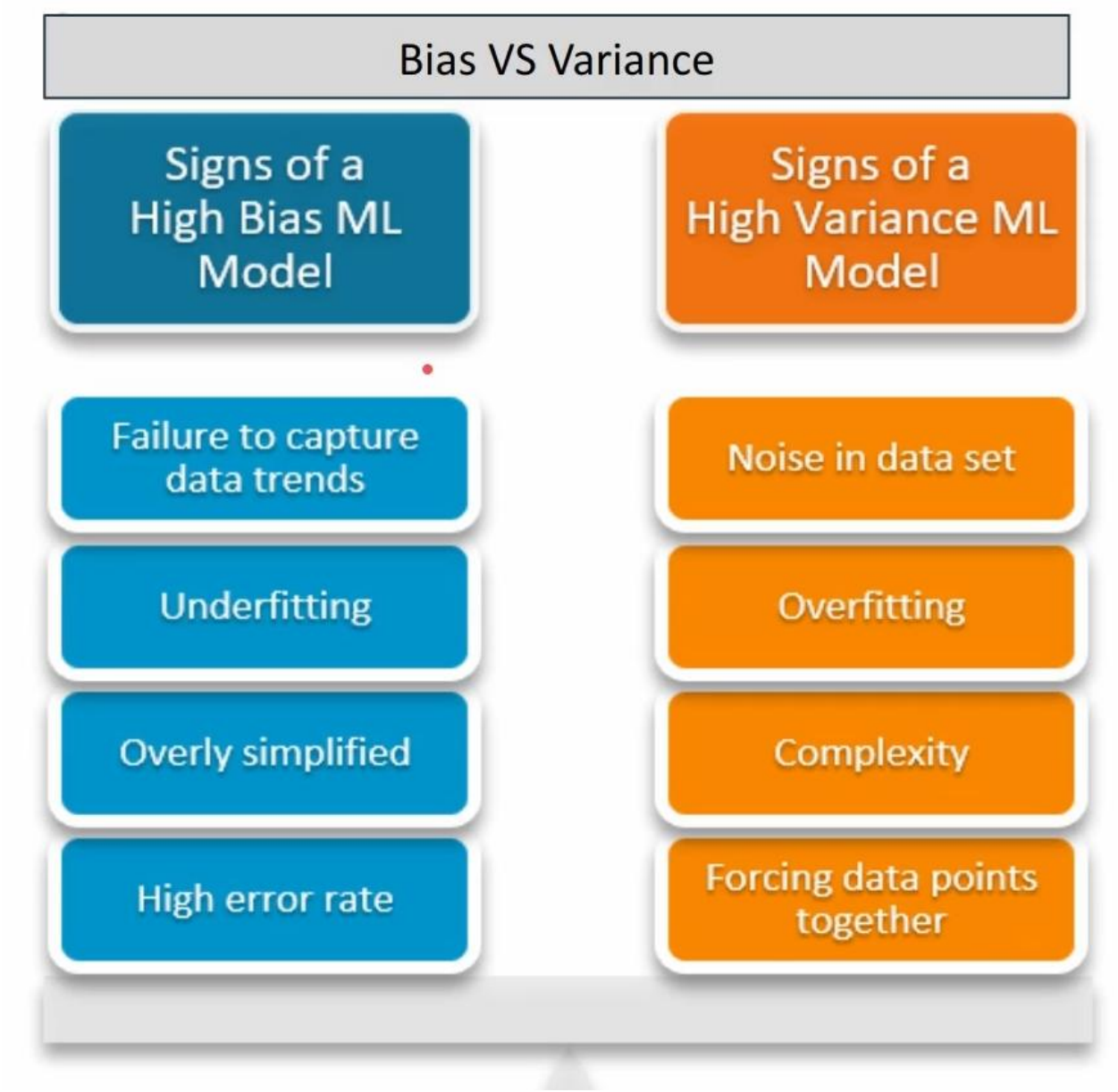
What it shows: How well your model performs on NEW, unseen data (the real test!)

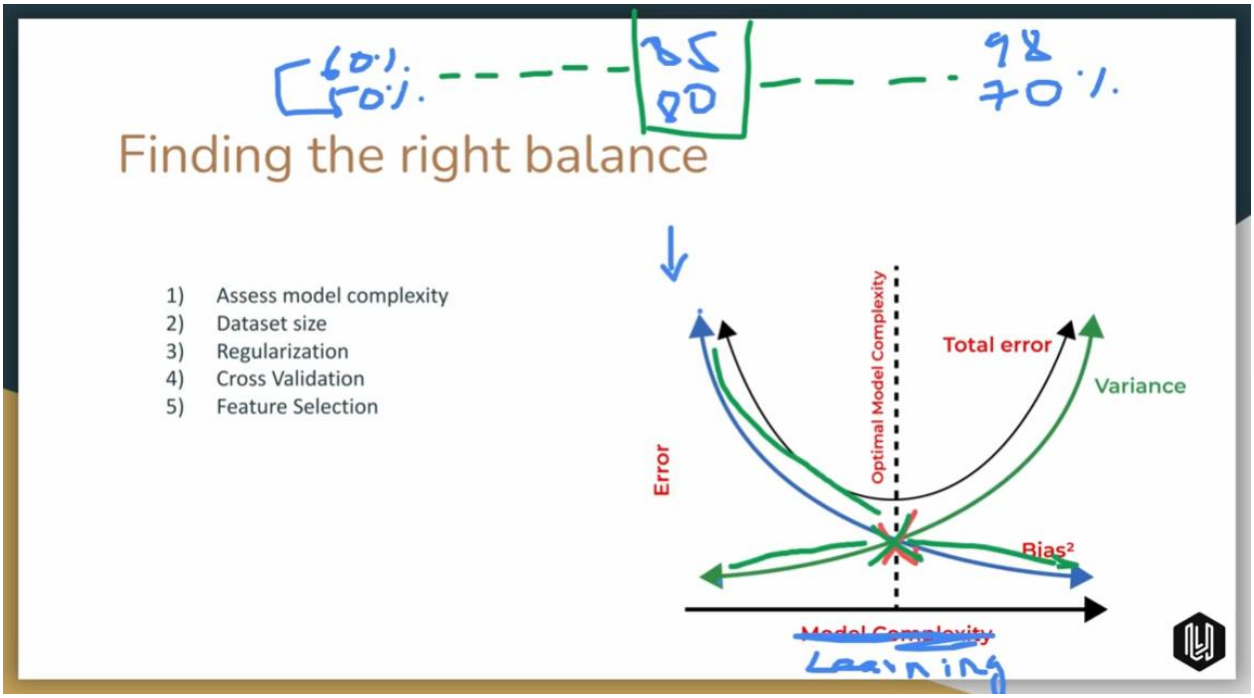
- **Left side (high):** Simple model performs poorly on new data too
- **Goes down first:** As complexity increases, it generalizes better
- **Bottom of the curve (SWEET SPOT!):** Best balance
- **Then goes UP:** Too complex = overfitting = bad on new data!

UNDERFITTING AND OVERFITTING



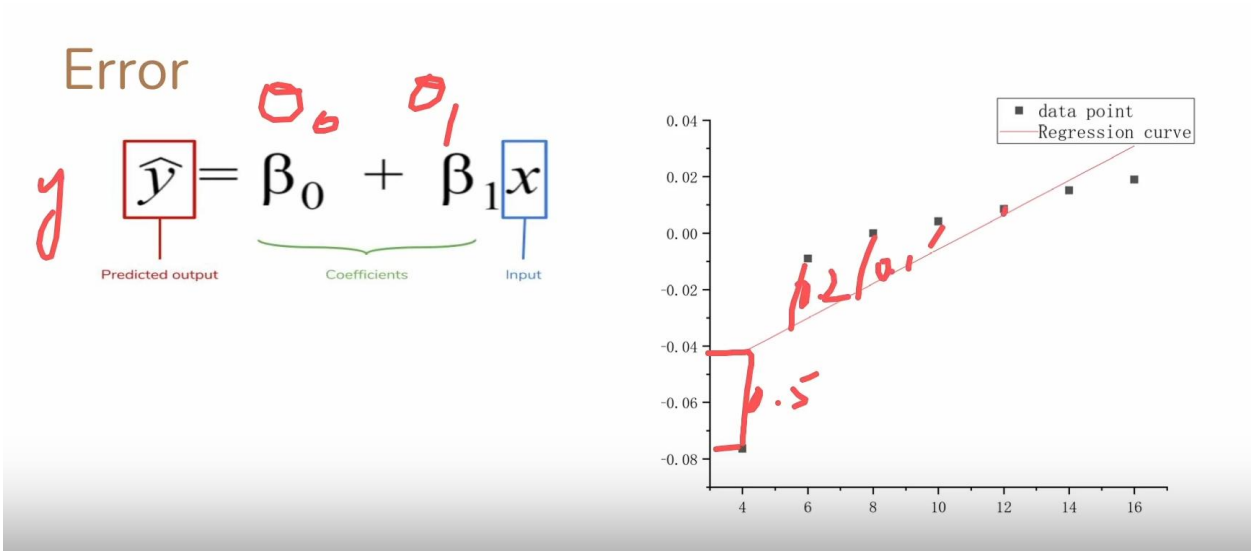
- Higher variance is really good but if there is a point far away or testing data then gives validation error because unable to generalize.


Bias VS Variance
Signs of a High Bias ML Model
Signs of a High Variance ML Model
Failure to capture data trends
Noise in data set
Underfitting
Overfitting
Overly simplified
Complexity
High error rate
Forcing data points together



1) SIMPLE MODELS = HIGH BIAS
2) DATASET SIZE = Increasing the size of the data set reduce bias and very small data increase variance because no enough data to learn.
3) Regularization = helps with reducing variance
4) Cross validation = having multiple hyper paramters and setups and trying each one out at once and seeing which one giving the best result. To reduce Variance
5) Feature selection = if we use lots of information of unnecessary leads to high variance.

Error Analysis



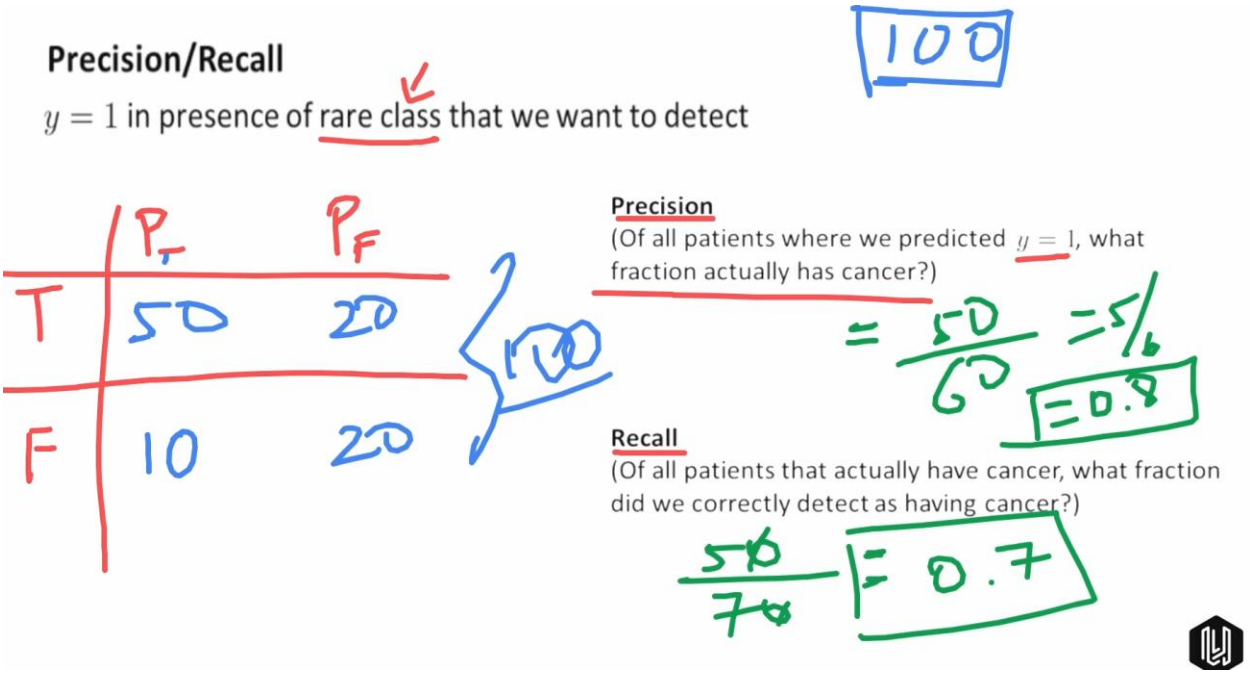
- After we get difference we sum them up or magintudes.

Precision

- From all the predicted cases what fraction is actually True.
- For instance : Of all the patients where we predicted y=1 what fraction is actually has cancer?

Recall

- From all the patients that actually have cancer what fraction did we correctly detect as having cancer?



F_1 Score (F score)

How to compare precision/recall numbers?

	Precision(P)	Recall (R)
Algorithm 1	0.5	0.4
Algorithm 2	0.7	0.1
Algorithm 3	0.02	1.0

$= 2 \times 0.5 \times 0.4 / 0.9$

0.44

$= 2 \times 0.7 \times 0.1 / 0.8$

$= 0.18$

$= 2 \times 0.02 \times 1 / 1.02$

0.04

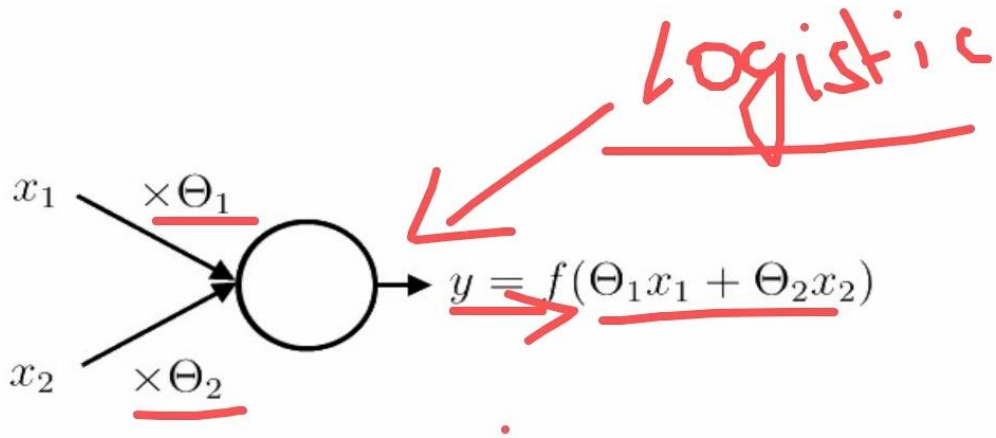
$= \frac{2 \times P \times R}{(P + R)}$

- The right side is the equation for the F1 SCORE CALCULATION.
- The last algorithm is worst
- However when we compare the values of P and R of algorithms 1 and 2 as you can see the precision is 0.7 is algorithm 2 however the F1 score is the final prediction.
- Higher the value , more balanced!

NEURAL NETWORK

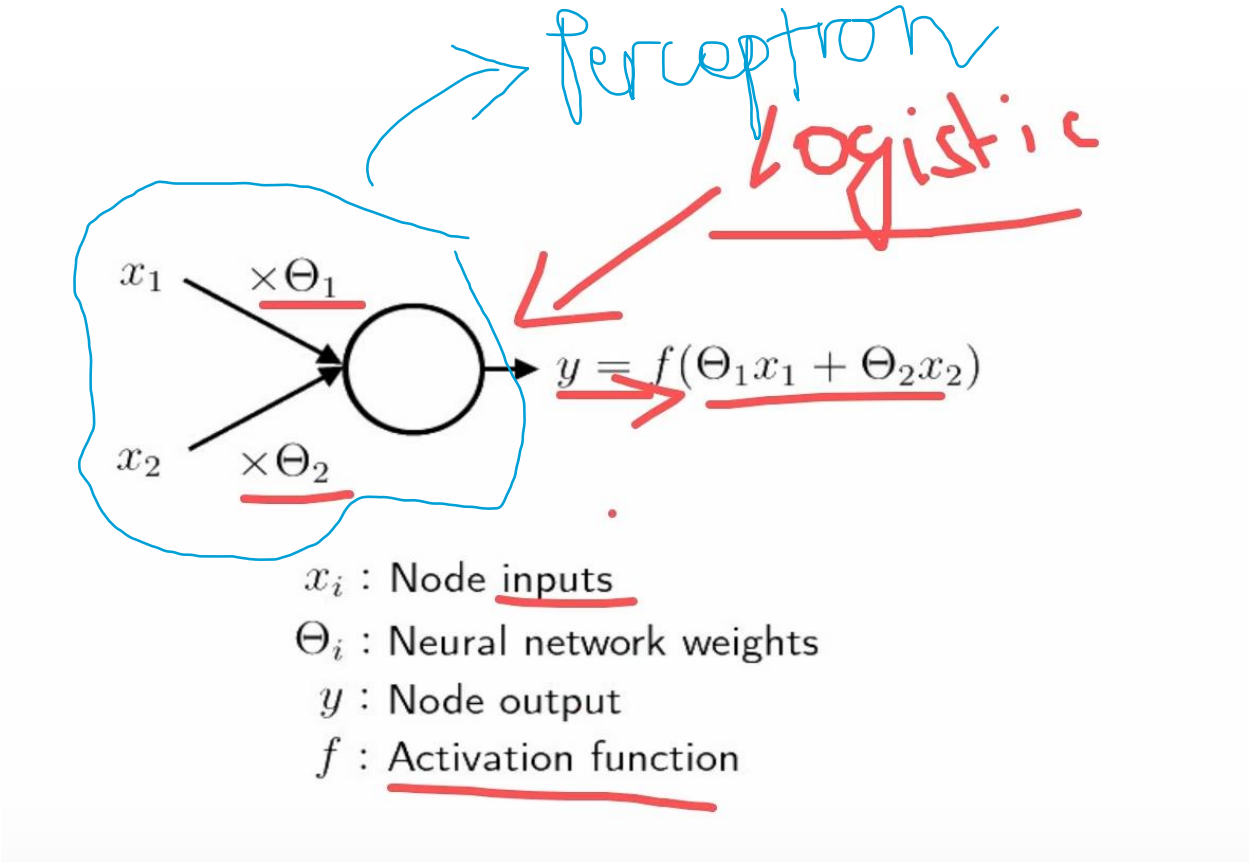
What is neural network

- Neural Networks are machine learning algorithms known to mimic the behavior of the human brain to solve complex data-driven problems.
- The input data is processed through different layers of artificial neurons stacked together to produce the desired output.
- From speech recognition and person recognition to healthcare and marketing, Neural Networks have been used in a varied set of domains.



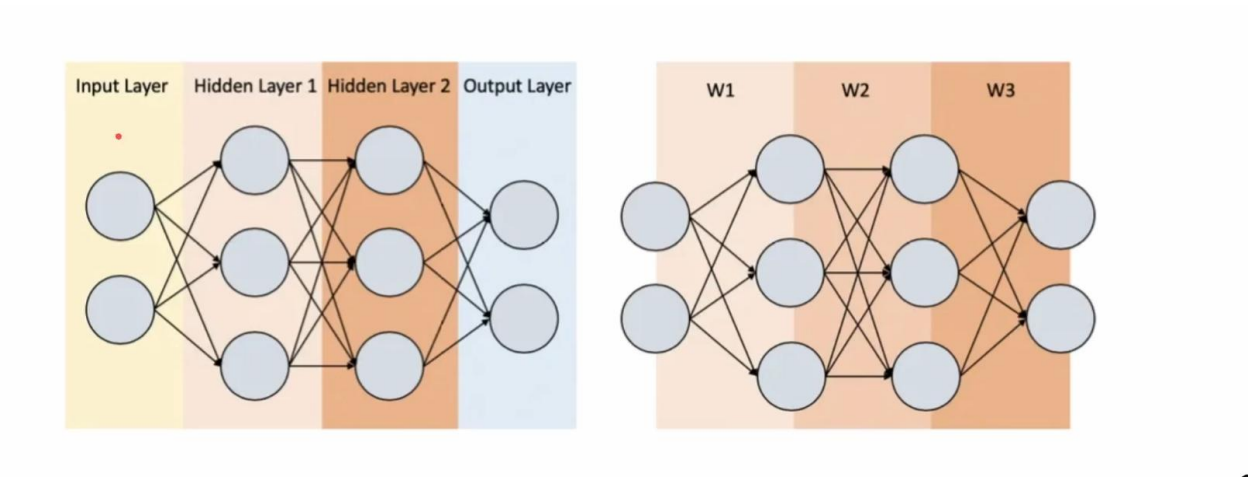
x_i : Node inputs
Θ_i : Neural network weights
y : Node output
f : Activation function

- Basically we have two inputs and then we multiplier with the weights and we take the result y and determines what to pass to t he next node.
- The equation is similar to a logistic regression equation. We know that for logistic regression we use linear regression equation and pass through sigmoid function. The sigmoid functions is kind of activation function.
- Activation function – activate or deactivate the data or pass the data.

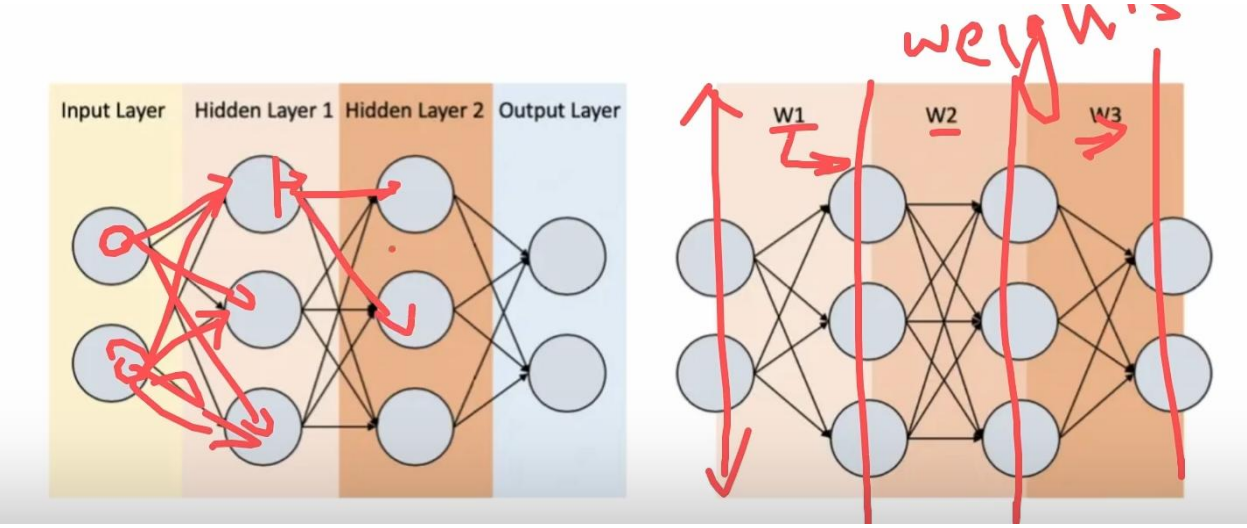


- This blue circled one is called a Perceptron.

Neural network architecture

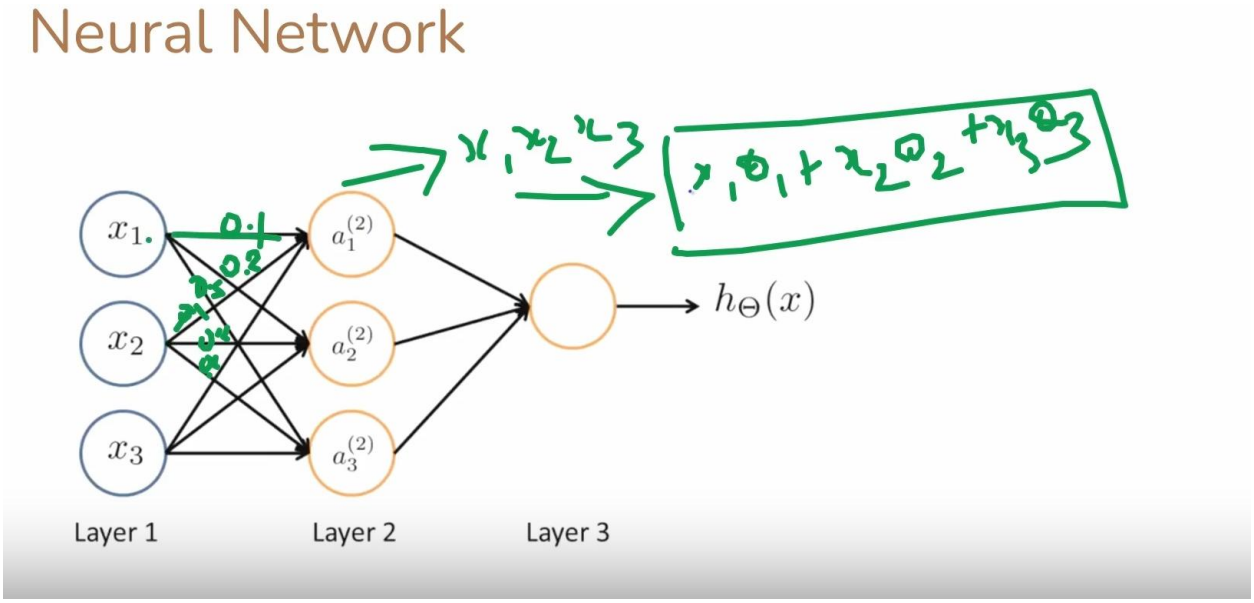


- These connections are just numbers and these numbers are WEIGHTS OF THE NEURAL NETWORKS.
- Each layer have each set of weights like W1,W2,W3
- **Input layer doesn't have any weights because its just the data itself no weight**



In here as you can see

- Each node traverse through all the other nodes
- The other nodes activation function decide which to pass and not
- Likewise it continues like a neural network
- Each layer has its own weight for computations , likewise traverse through till the output layer.



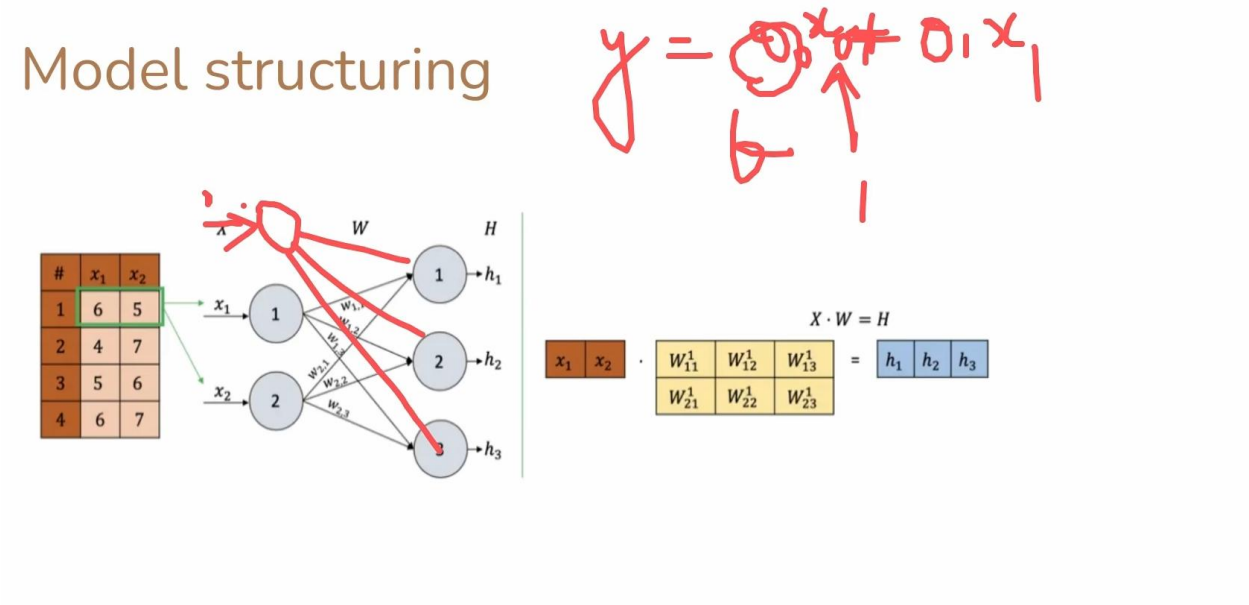
For example

- Assume that each node has the values of weights as 0.1 ,0.2,0.3
- So basically layer 1 is called the input layer because its just the data itself.

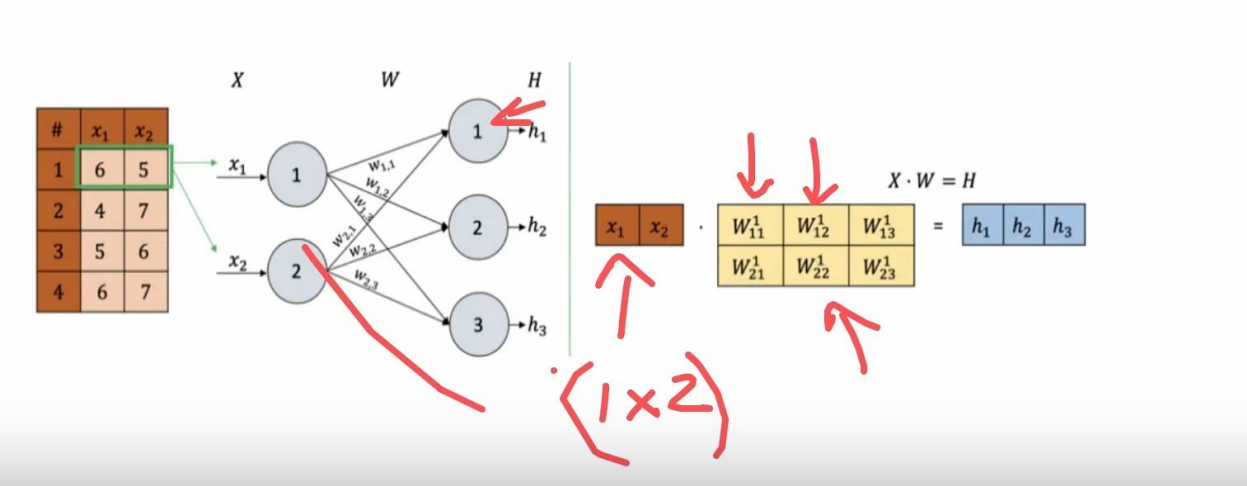
- Then each node in layer 2 receives data of all the three nodes in the input layer which is X1,X2,X3.
- Each node connection has its weight.
- So to determine the total weight we calculate
    - X1θ1 + X2θ2 + X3θ3
      Where each θ is basically the weight.
    - This is similar to $\boldsymbol{\theta^T X}$
    - Then result of $\boldsymbol{\theta^T X}$ is passed to the **sigmoid function** which is the activation function.

Representation aspects of neural network

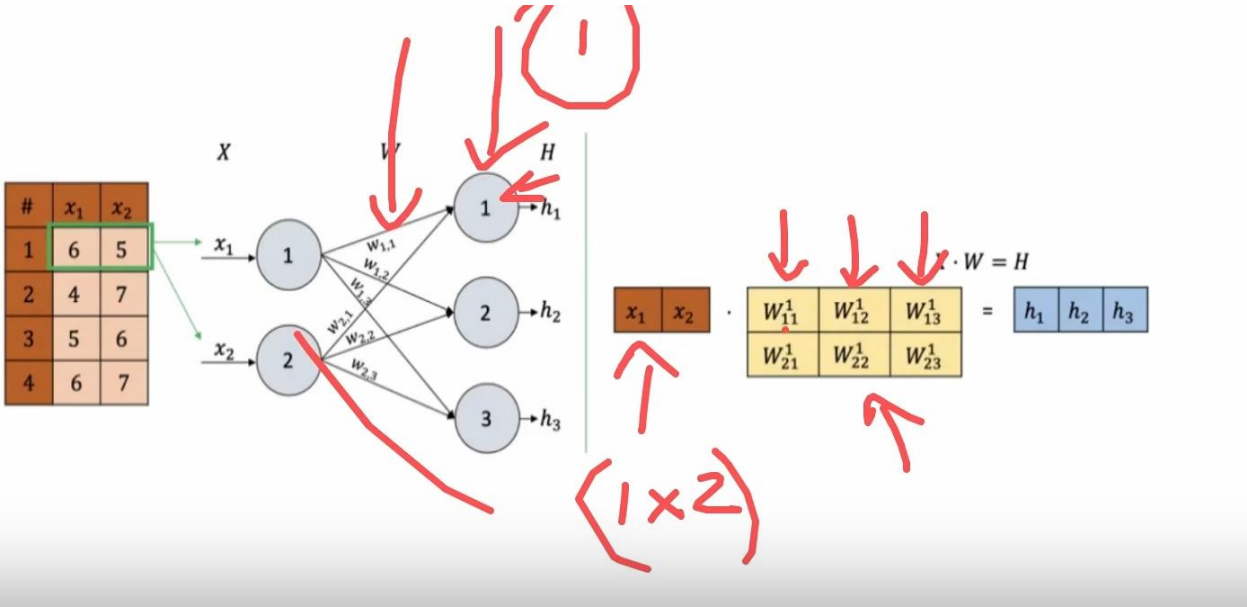
- In most of the network architectures the bias node will be also present which doesn't do anything but bias.
- For instance in the linear regression the line is y = θ0 X0 + θ1X1 , where we omit the X0 because X0 is basically 1.
- Likewise here we can have another node like below



- The node table is like above shown, where x1 and x2 are the input features
- We have two components which is the computation and the activation component.

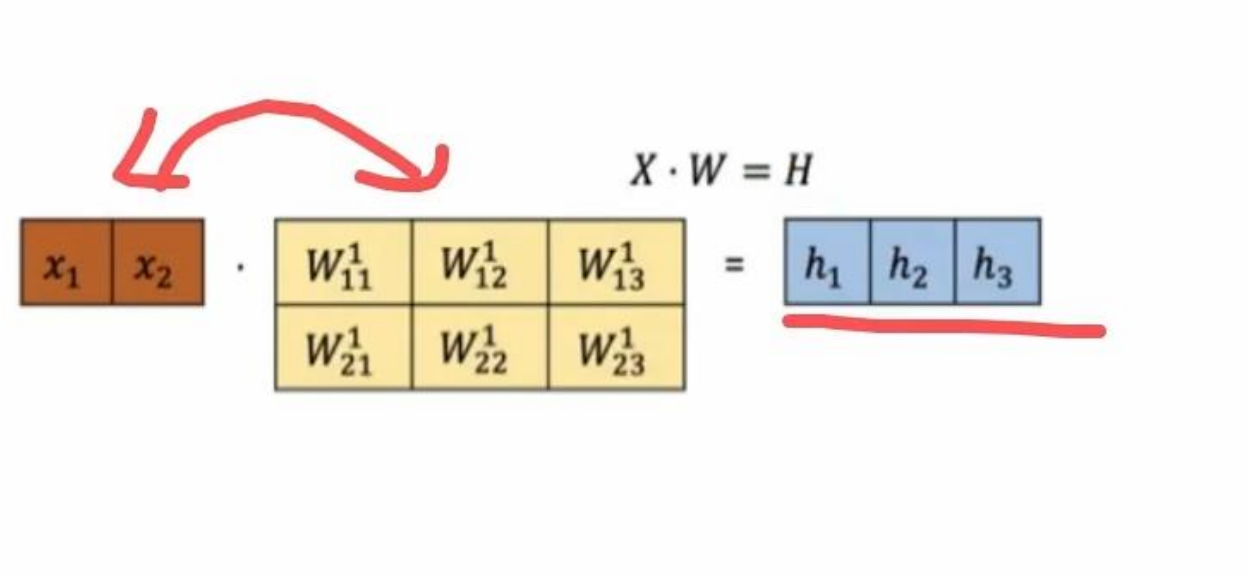


- As you can see we can represent as vectors
  - First is the input features which can be represented by 1 x 2 vector
  - Second is the individual weights of each layer. Like each column represents the weights coming to each node for instance, the first column all weights coming to the first node. This is 2 x 3 vector
  - Therefore final vector is size of 1 x 3

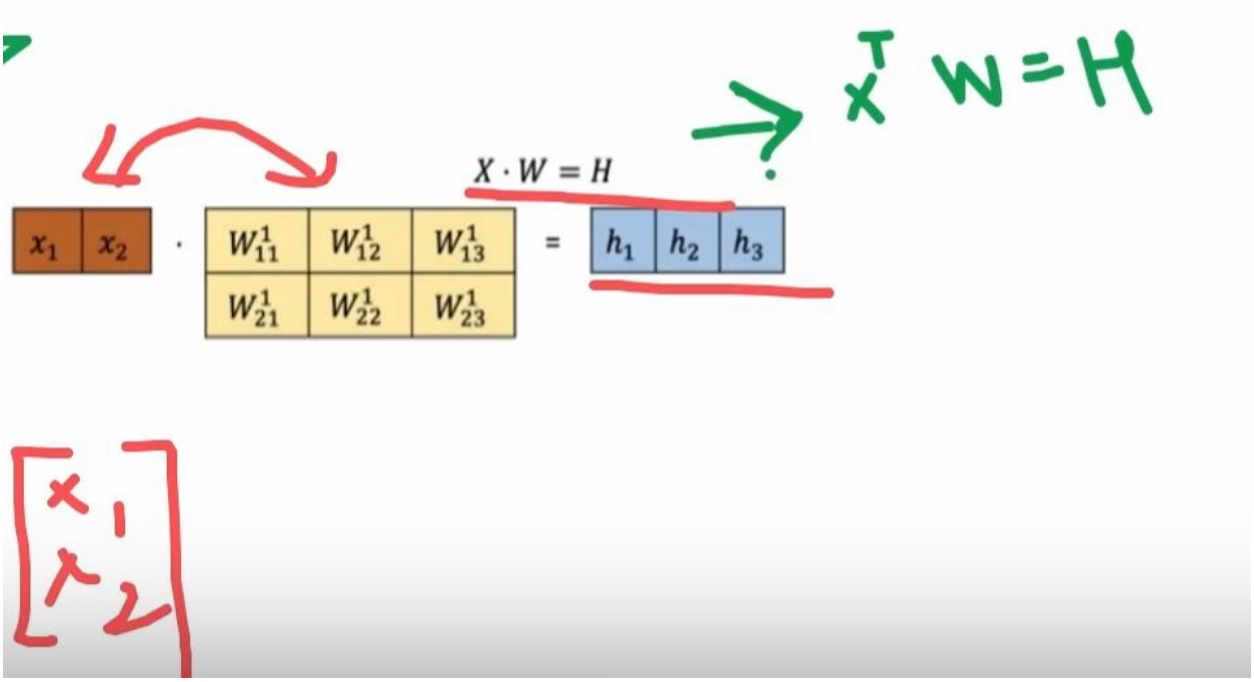


- As you can see here , for each W the to the power number of the subscript which is shown 1 , is basicallyt he layer. As we all know input layer is just data nothing else , so we don't consider it as layer 1.

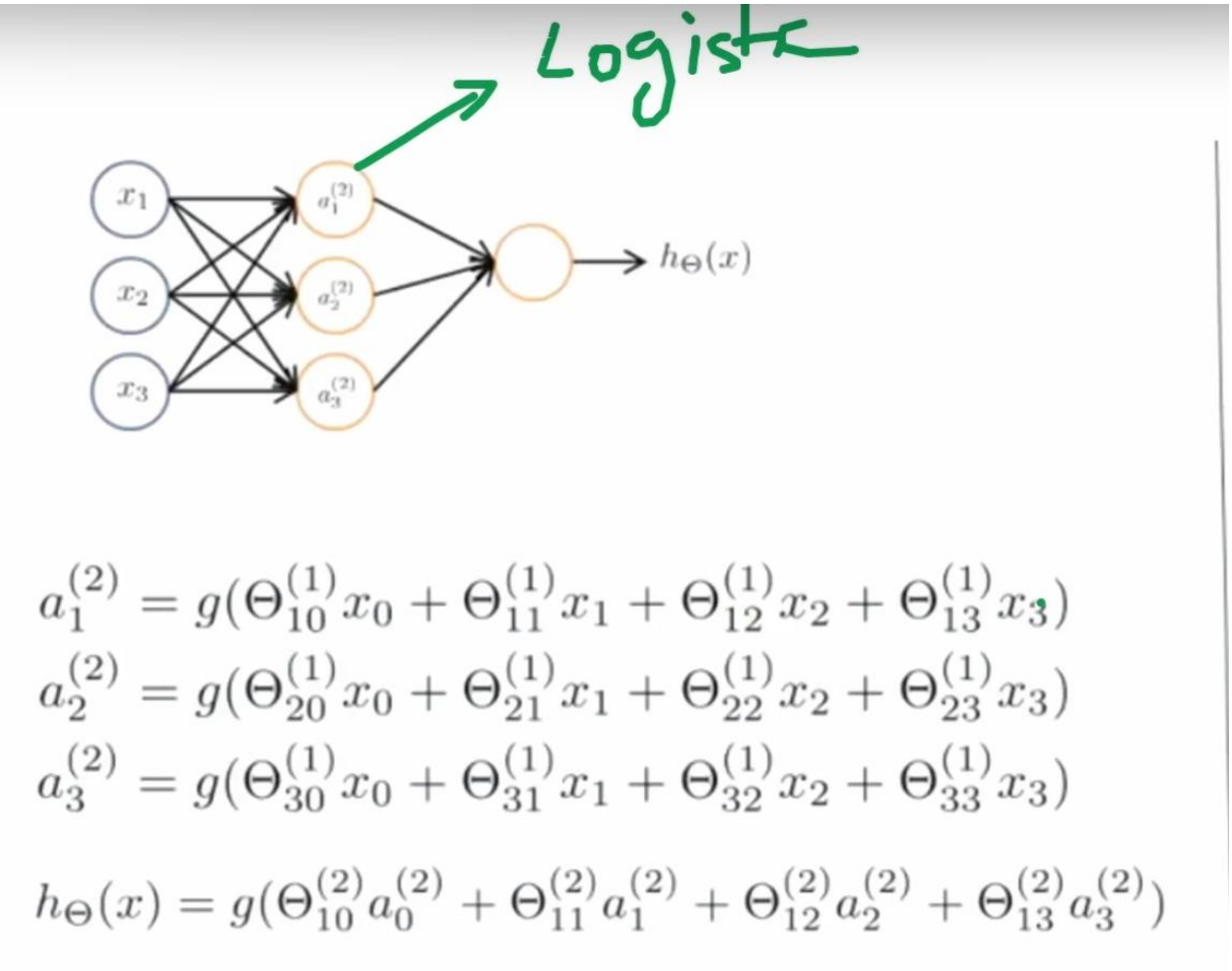
- And the down number basically means the start node and the end node or from which nodes between for instance W^{1}_{11} represents the weights coming to the node 1 in the layer 1 and then number 11 represents from node 1 in the previous layer and the current node number 1.



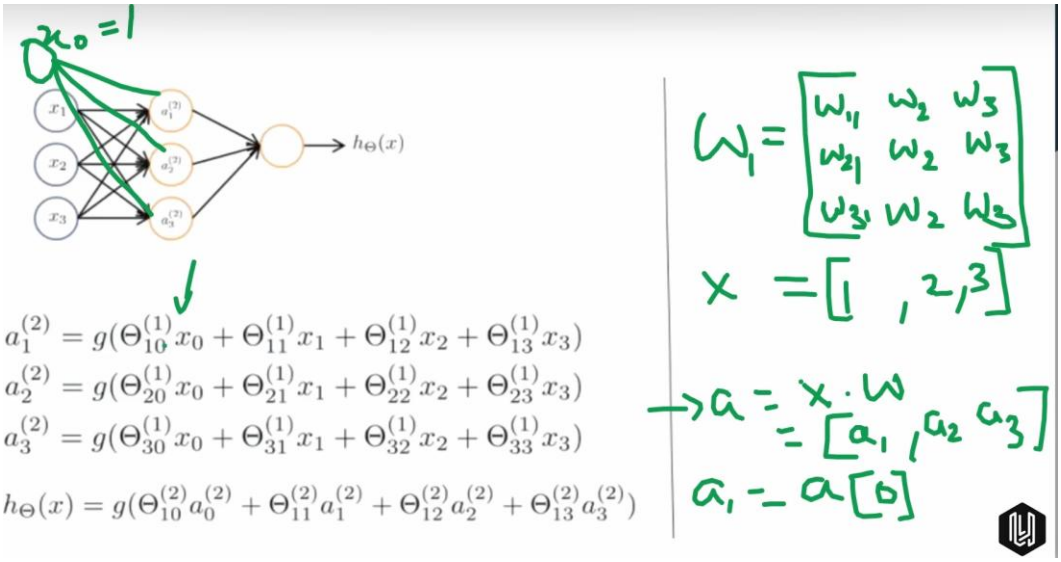
- Final result is basically a dot product between the weights and the input features or X.



Assume each node is a literall LOGISITIC REGRESSION MODEL.



- Activation functions is sigmoid!



- Remember dot product returns a scaler but , activation vector is basically

Neural Network Activation Vector - Dot Product Explanation

The Confusion:

Dot product returns a **scalar** (single number), so why is the activation **a** a vector?

The Answer:

We perform **multiple dot products** — one for each neuron in the layer.

How It Works:

Individual Neurons (Scalars):

Each neuron computes one dot product:

- **Neuron 1:** $a_1^{(2)} = g(\Theta_{10}x_0 + \Theta_{11}x_1 + \Theta_{12}x_2 + \Theta_{13}x_3) \rightarrow$ **one scalar**
- **Neuron 2:** $a_2^{(2)} = g(\Theta_{20}x_0 + \Theta_{21}x_1 + \Theta_{22}x_2 + \Theta_{23}x_3) \rightarrow$ **one scalar**
- **Neuron 3:** $a_3^{(2)} = g(\Theta_{30}x_0 + \Theta_{31}x_1 + \Theta_{32}x_2 + \Theta_{33}x_3) \rightarrow$ **one scalar**

Combine Into Vector:

All these scalars together form the **activation vector:**

$\mathbf{a^{(2)} = [a_1^{(2)}, a_2^{(2)}, a_3^{(2)}]}$ — a vector of 3 numbers

Combine Into Vector:

All these scalars together form the **activation vector:**

$\mathbf{a^{(2)} = [a_1^{(2)}, a_2^{(2)}, a_3^{(2)}]}$ — a vector of 3 numbers

Example with Numbers:

Weights:

```
W = [1 2 3]
    [4 5 6]
    [7 8 9]
```

Input: x = [2, 3, 1]

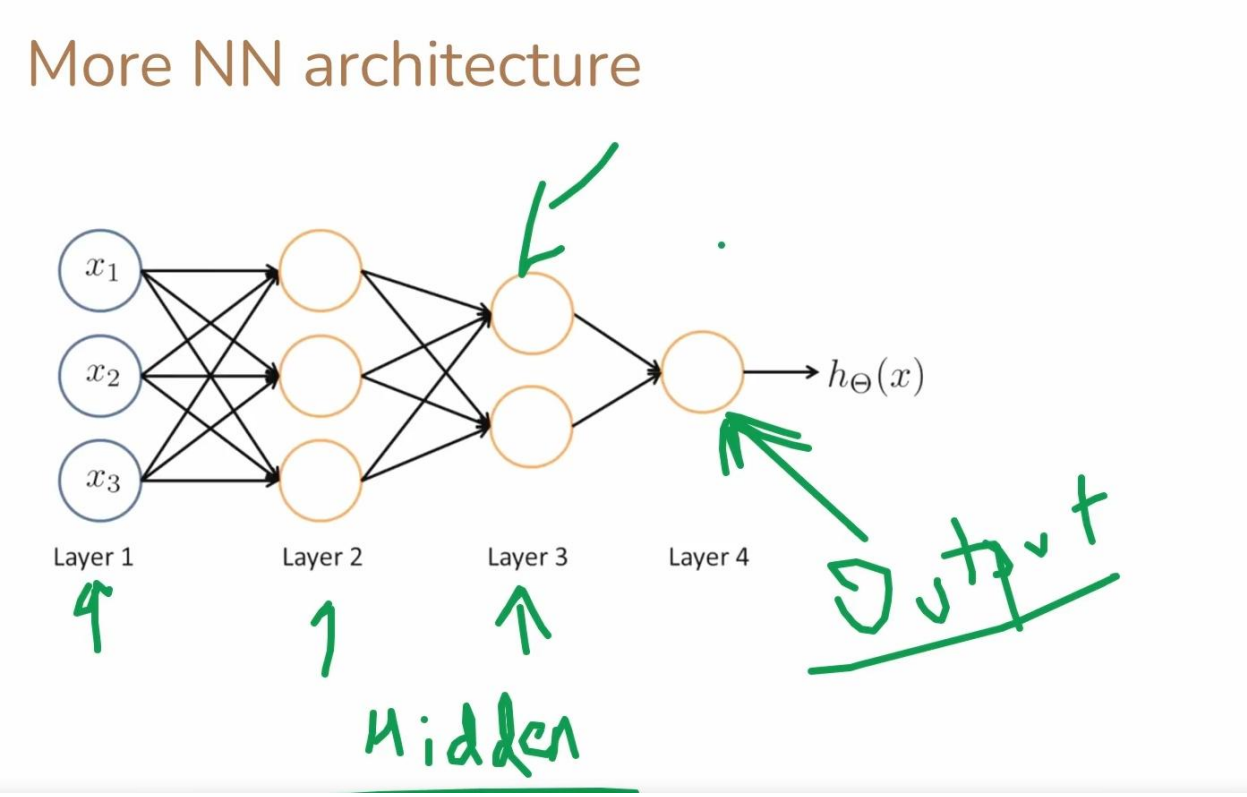
Calculations:

- Neuron 1: 2×1 + 3×4 + 1×7 = **21**
- Neuron 2: 2×2 + 3×5 + 1×8 = **27**
- Neuron 3: 2×3 + 3×6 + 1×9 = **33**

Activation vector: a = [21, 27, 33]

Then apply activation function: **a = [g(21), g(27), g(33)]**

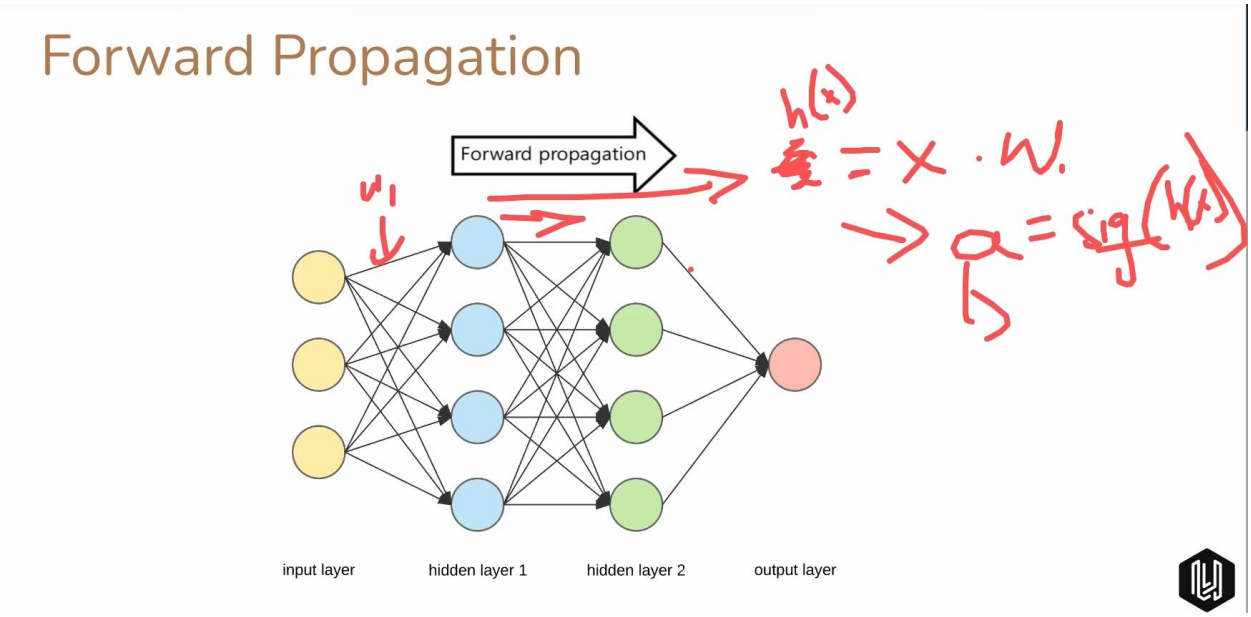
- Remember g is the sigmoid function.
- Then this activation vector's sigmoid result will be input to the next layer. Likewise we continue until the last or output layer.

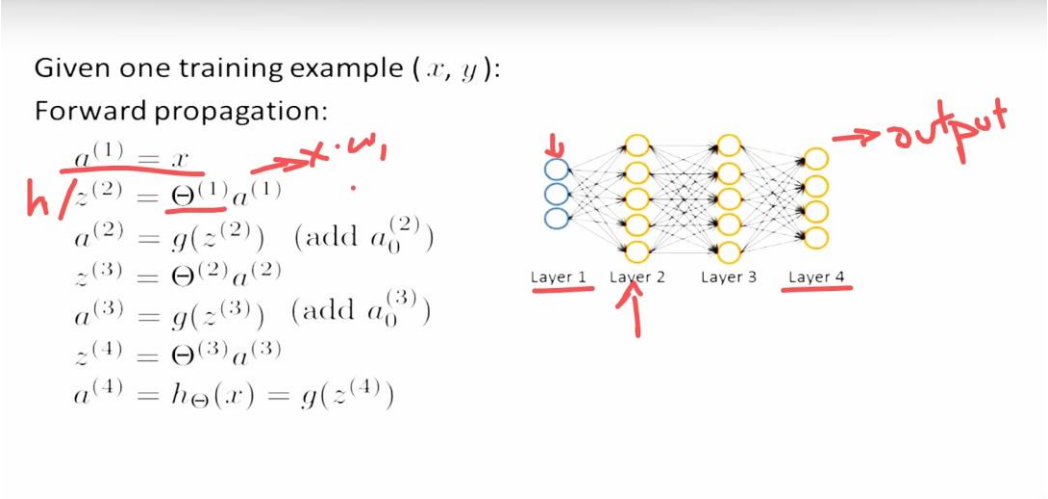


- All the yellow nodes are the actual nodes because they contain the both the components which is the computation component and the activation component. Whereas, the layer 1 is just the input layer or just data itself.

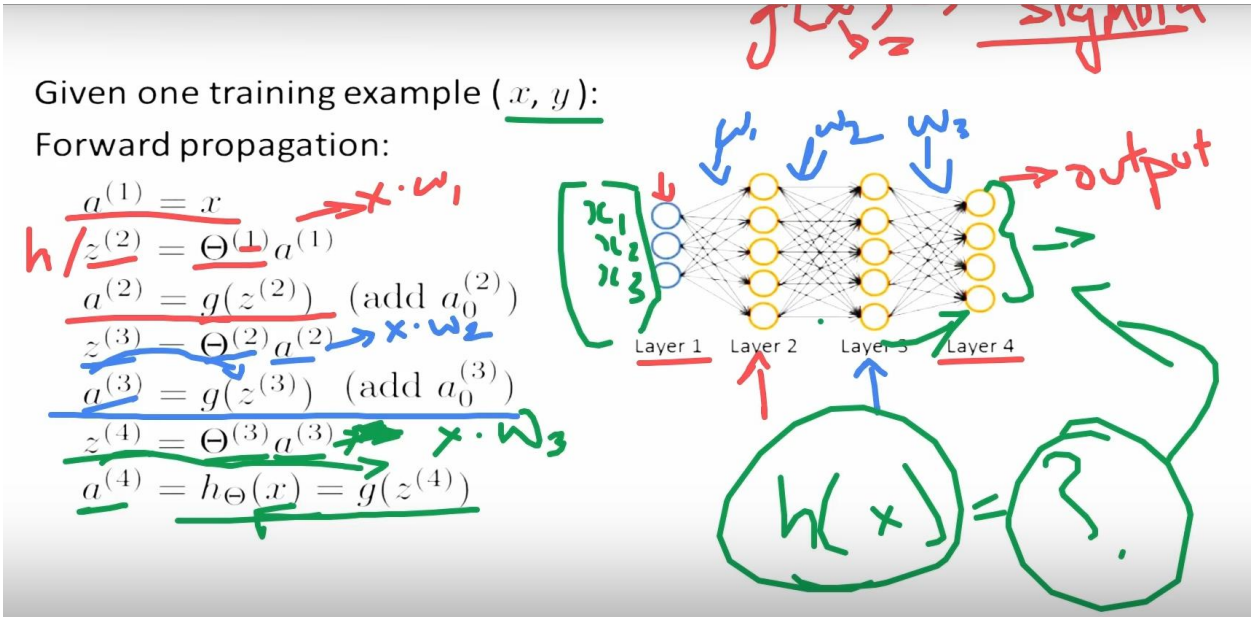
Neural network learning process.

Forward propagation





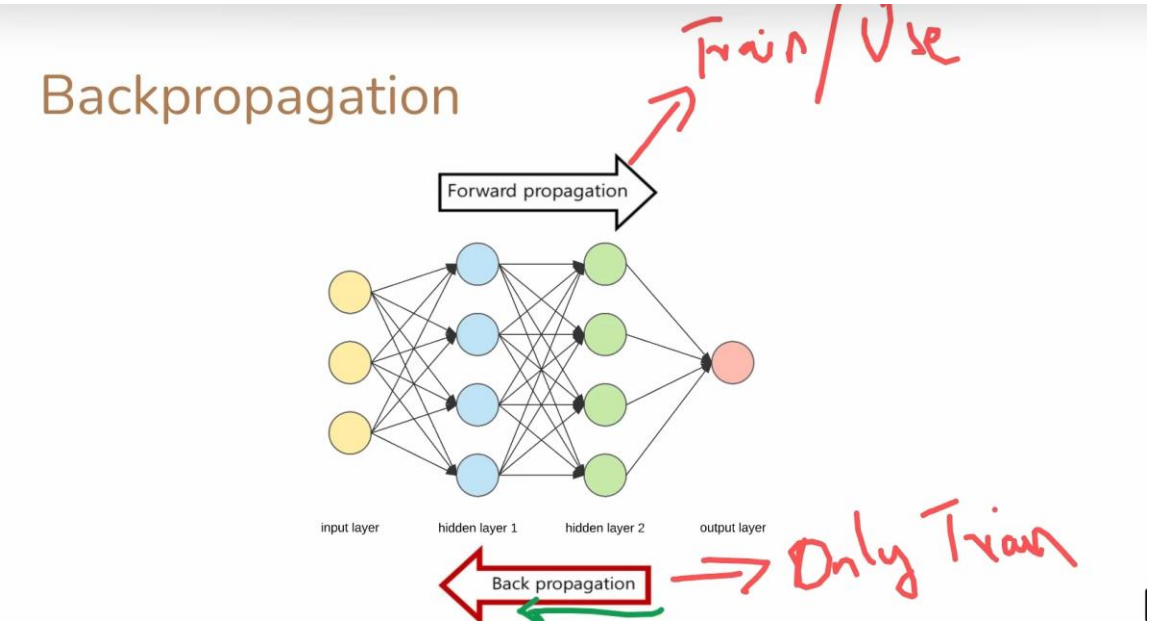
- Z is same as h in previous slide we wrote and also theta is the X1W1
- Likewise we continue.



- Same as before!

BACK PROPAGATION

- This propagation is used to perform the actual learning process or to train the model but the forward propagation is used to just run the neural network nothing else.



- Difference =
    - in forward propogation we pass the input to get the out or basically we pass X to get Y values

- Where as in the backward propagation we pass the errors or propagates the error in the neural network in reverse direction.
  - Starts from the output node, because we know we get an error once we got the output
  - Then we can propagate to the other nodes. Not propagate to the input layer but till the hidden layer 1
  - For training we need propagation to get the error or output then back propagation to learning and training.

Gradient computation: Backpropagation algorithm

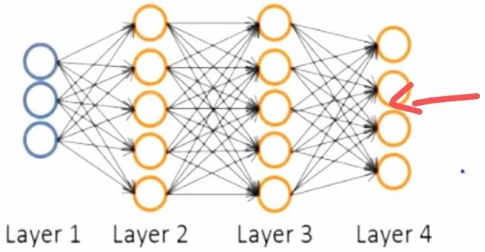
Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l.

For each output unit (layer L = 4)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$$

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$



Where,

$$g'(z^{(l)}) = a^{(l)} .* (1 - a^{(l)})$$

.* is element-wise multiplication

- Error (delta) vector of node j in the layer l.

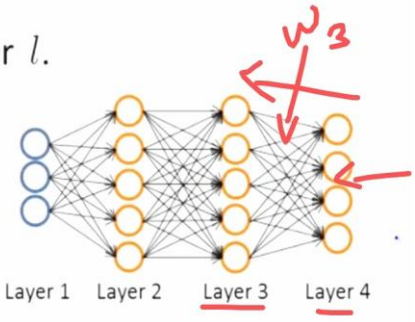
Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l.

For each output unit (layer L = 4)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$$

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

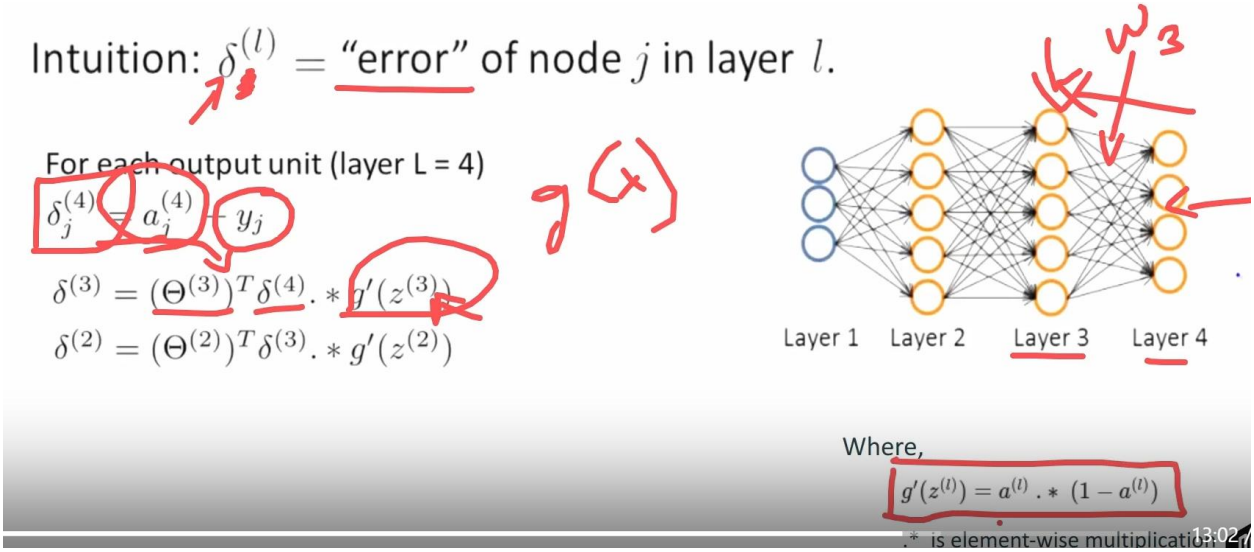


Where,

$$g'(z^{(l)}) = a^{(l)} .* (1 - a^{(l)})$$

.* is element-wise multiplication

- Here what we do as you can see we multiply theta which is also known as the Weight , Theta 3 means W3 , we multiply W3 with error in the forward layer which is layer 4 and then



- Remember we need to multiply with g' inorder to know the room of error made by that particular node. For instance say

g'(z) tells us: "How much does the sigmoid output change when z changes?"

The math formula is: **g'(z) = g(z) × (1 - g(z))**

Or simpler: **g'(z) = a × (1 - a)** where **a = g(z)** (the output).

Let me calculate it:

Output (a)	g'(z) = a(1-a)	Meaning
0.01	0.01 × 0.99 = **0.0099**	Output barely changes
0.3	0.3 × 0.7 = **0.21**	Output changes moderately
0.5	0.5 × 0.5 = **0.25**	Output changes the MOST
0.7	0.7 × 0.3 = **0.21**	Output changes moderately
0.99	0.99 × 0.01 = **0.0099**	Output barely changes

Step 5: Why Do We Use g'(z) in Backpropagation?

When the network is learning, it needs to know: **"Should I adjust this neuron's weights?"**

Let's say the output layer says: **"Hey, I have an error of -0.2"**

Now we go back to Layer 3. It has 2 neurons:

Neuron A:

- Output = 0.5
- $g'(z) = 0.5 \times 0.5 = \mathbf{0.25}$

Neuron B:

- Output = 0.99
- $g'(z) = 0.99 \times 0.01 = \mathbf{0.0099}$

Step 6: Simple Analogy

Imagine you're a teacher grading students on a test:

Student A (score = 50%):

- Has lots of room to improve
- You focus on helping them → **give them more feedback**

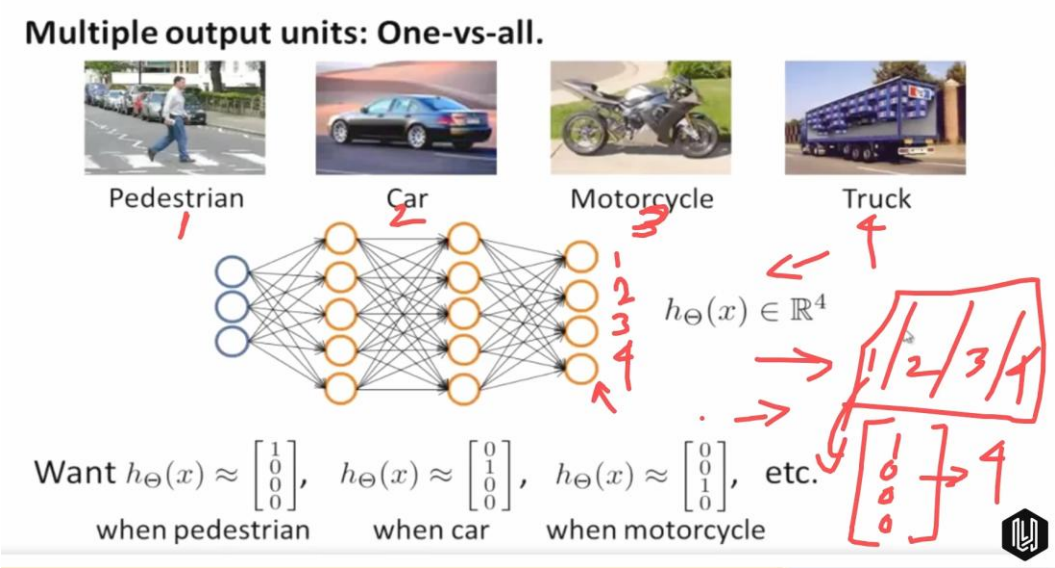
Student B (score = 99%):

- Already near perfect
- Can't improve much more → **less feedback needed**

g'(z) is like measuring "how much room for improvement" each neuron has!

- If **g(z) = sigmoid(z)**, then **g'(z)** tells us: **"How much does the output change when the input changes slightly?"**

NEURAL NETWORK IN CLASSIFICATION



TRAINING PHASE

This is where the network **learns** to recognize different objects.

Step 1: Prepare Training Data

We collect thousands of labeled images:

Image	Input (x)	Label (y)
🚶 Pedestrian photo	$x^{(1)}$ = [pixel values]	[1, 0, 0, 0]
🚗 Car photo	$x^{(2)}$ = [pixel values]	[0, 1, 0, 0]
🏍️ Motorcycle photo	$x^{(3)}$ = [pixel values]	[0, 0, 1, 0]
🚚 Truck photo	$x^{(4)}$ = [pixel values]	[0, 0, 0, 1]
... 996 more images	...	...

Total: 1000 images (250 of each type)

Step 2: Initialize Network with Random Weights

At the start, the network knows nothing. All weights are random.

```
Weights: [random numbers]
```

If you show it a car image, it might output:

```
h_θ(x) = [0.23, 0.18, 0.31, 0.28]  ← Random garbage!
```

Step 3: Train with Example 1 - A Car Image

Show the network a car:

Input:

```
x = [0.8, 0.2, 0.7, 0.3, 0.9, ...]  (pixels of red Toyota)
```

What we want (label):

```
y = [0, 1, 0, 0]  ← "This is a car"
```

What the network predicts (initially garbage):

```
h_θ(x) = [0.23, 0.18, 0.31, 0.28]
```

Calculate Error:

```
Error = Prediction - Target
      = [0.23, 0.18, 0.31, 0.28] - [0, 1, 0, 0]
      = [0.23, -0.82, 0.31, 0.28]
```

What this means:

- Neuron 1 (pedestrian): predicted 0.23, should be 0 → **Too high! Reduce it**
- Neuron 2 (car): predicted 0.18, should be 1 → **Way too low! Increase it!**
- Neuron 3 (motorcycle): predicted 0.31, should be 0 → **Too high! Reduce it**
- Neuron 4 (truck): predicted 0.28, should be 0 → **Too high! Reduce it**

Adjust Weights (Backpropagation):

The network uses backpropagation to adjust all weights slightly to reduce this error.

```
Old weights → Adjust → New weights
```

Step 4: Train with Example 2 - A Pedestrian Image

Show the network a pedestrian:

Input:

```
x = [0.1, 0.5, 0.2, 0.8, 0.4, ...]  (pixels of person walking)
```

What we want:

```
y = [1, 0, 0, 0]  ← "This is a pedestrian"
```

What the network predicts (after one adjustment):

```
h_θ(x) = [0.35, 0.22, 0.25, 0.18]
```

Calculate Error:

```
Error = [0.35, 0.22, 0.25, 0.18] - [1, 0, 0, 0]
      = [-0.65, 0.22, 0.25, 0.18]
```

What this means:

- Neuron 1 (pedestrian): predicted 0.35, should be 1 → **Too low! Increase it!**
- Neuron 2 (car): predicted 0.22, should be 0 → **Too high! Reduce it**
- Neuron 3 (motorcycle): predicted 0.25, should be 0 → **Too high! Reduce it**
- Neuron 4 (truck): predicted 0.18, should be 0 → **Too high! Reduce it**

Adjust Weights Again:

Weights get adjusted to reduce this error.

Step 5: Repeat for ALL Training Examples

We repeat this process for **all 1000 images**, going through the dataset **many times** (epochs).

After 1 epoch (1000 images):

- Network is slightly better

After 10 epochs (10,000 images shown):

- Network is getting good

After 100 epochs (100,000 images shown):

- Network is well-trained!

Step 6: Check Training Progress

After training, let's test on a **training image** (one it has seen):

Show it a car from training set:

```
Input: Car image
Prediction: h_θ(x) = [0.01, 0.97, 0.02, 0.00]
Target: [0, 1, 0, 0]
```

Much better! Now it correctly identifies cars with 97% confidence.

TESTING PHASE

Now we test the trained network with **brand new images** it has never seen before.

Test Example 1: Clear Car Image

Input: A new red Honda (not in training set)

```
x = [0.75, 0.19, 0.68, 0.33, 0.91, ...]  (pixel values)
```

Forward Pass Through Network:

Step 1: Input Layer

```
x goes into the network
```

Step 2: Hidden Layers

```
Layer 1 → [weighted sum + activation]
Layer 2 → [weighted sum + activation]
Layer 3 → [weighted sum + activation]
```

Step 3: Output Layer (4 neurons)

Each output neuron computes:

Neuron 1 (Pedestrian):

```
z₁ = weights₁ · hidden_layer_output
a₁ = sigmoid(z₁) = 0.02
```

Neuron 2 (Car):

```
z₂ = weights₂ · hidden_layer_output
a₂ = sigmoid(z₂) = 0.94
```

Neuron 3 (Motorcycle):

```
z₃ = weights₃ · hidden_layer_output
a₃ = sigmoid(z₃) = 0.03
```

Neuron 4 (Truck):

```
z₄ = weights₄ · hidden_layer_output
a₄ = sigmoid(z₄) = 0.01
```

Collect Outputs:

```
h_θ(x) = [0.02, 0.94, 0.03, 0.01]
```

Make Prediction:

Step 1: Find highest value

```
0.94 at position 2
```

Step 2: Map to class

```
Position 2 = Car
```

Step 3: Final result

```
Prediction: CAR
Confidence: 94%
```

Output to user: "This is a CAR (94% confident)" ✅

Test Example 2: Blurry Motorcycle Image

Input: A foggy, distant motorcycle photo

```
x = [0.42, 0.55, 0.38, 0.61, 0.29, ...]  (blurry pixels)
```

Forward Pass:

Same process as before, but with these pixel values.

Output:

```
h_θ(x) = [0.18, 0.29, 0.32, 0.21]
```

Make Prediction:

Step 1: Find highest value

```
0.32 at position 3
```

Step 2: Map to class

```
Position 3 = Motorcycle
```

Step 3: Check confidence

```
Confidence: 32% (LOW!)
```

Step 4: Decision

```
"Prediction: Motorcycle, but only 32% confident.
⚠ WARNING: Low confidence! Image unclear. Manual review recommended."
```

Test Example 3: Street Scene (Car + Pedestrian)

Input: Busy street with both car and person

```
x = [0.62, 0.81, 0.44, 0.73, 0.55, ...]
```

Forward Pass:

Network processes the image.

Output:

```
h_θ(x) = [0.78, 0.82, 0.06, 0.04]
```

Make Prediction:

Step 1: Find highest value

```
0.82 at position 2
```

Standard approach: "It's a CAR"

But look at all values:

```
Pedestrian: 0.78 (HIGH!)
Car: 0.82 (HIGH!)
Motorcycle: 0.06 (low)
Truck: 0.04 (low)
```

Better interpretation:

```
"ALERT: Both CAR (82%) and PEDESTRIAN (78%) detected!
🚨 Multiple objects in scene!"
```

For self-driving car: This is critical safety information!

Test Example 4: Truck Image

Input: Large delivery truck

```
x = [0.91, 0.23, 0.67, 0.45, 0.88, ...]
```

Forward Pass:

Network processes.

Output:

```
h_θ(x) = [0.01, 0.03, 0.02, 0.94]
```

Make Prediction:

Step 1: Find highest

```
0.94 at position 4
```

Step 2: Map to class

```
Position 4 = Truck
```

Step 3: Result

```
Prediction: TRUCK
Confidence: 94%
```

Output: "This is a TRUCK (94% confident)" ✅

Summary of Testing:

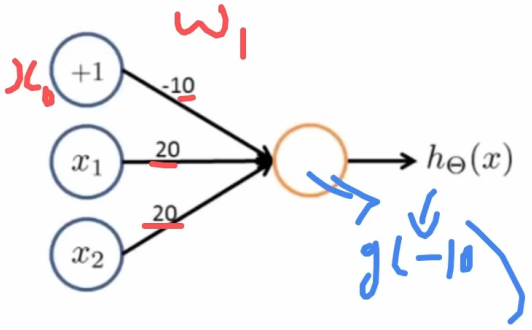
Test Image	Output Vector	Prediction	Confidence	Action
Clear car	[0.02, 0.94, 0.03, 0.01]	Car	94%	Accept ✅
Blurry motorcycle	[0.18, 0.29, 0.32, 0.21]	Motorcycle	32%	Review ⚠️
Car + Pedestrian	[0.78, 0.82, 0.06, 0.04]	Both!	82%, 78%	Alert 🚨
Truck	[0.01, 0.03, 0.02, 0.94]	Truck	94%	Accept ✅

Key Differences:

Aspect	Training Phase	Testing Phase
Goal	Learn to recognize	Make predictions
Input	Known images + labels	New unknown images
Output	Predictions (to calculate error)	Final predictions only
Error	Yes, calculate and adjust	No error (no labels)
Weights	Constantly changing	Fixed (no changes)
Process	Forward + Backward + Update	Forward only

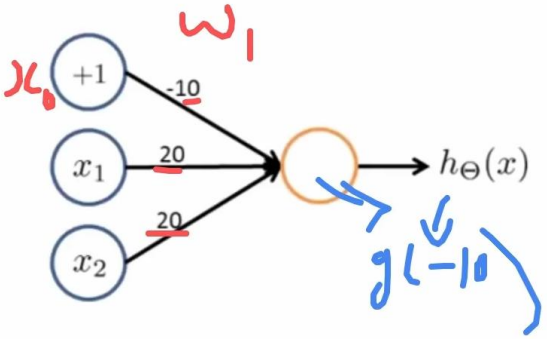
UNDERSTANDING THE WEIGHTS

Example: OR function



x_1	x_2	$h_\Theta(x)$
0	0	0
0	1	
1	0	
1	1	

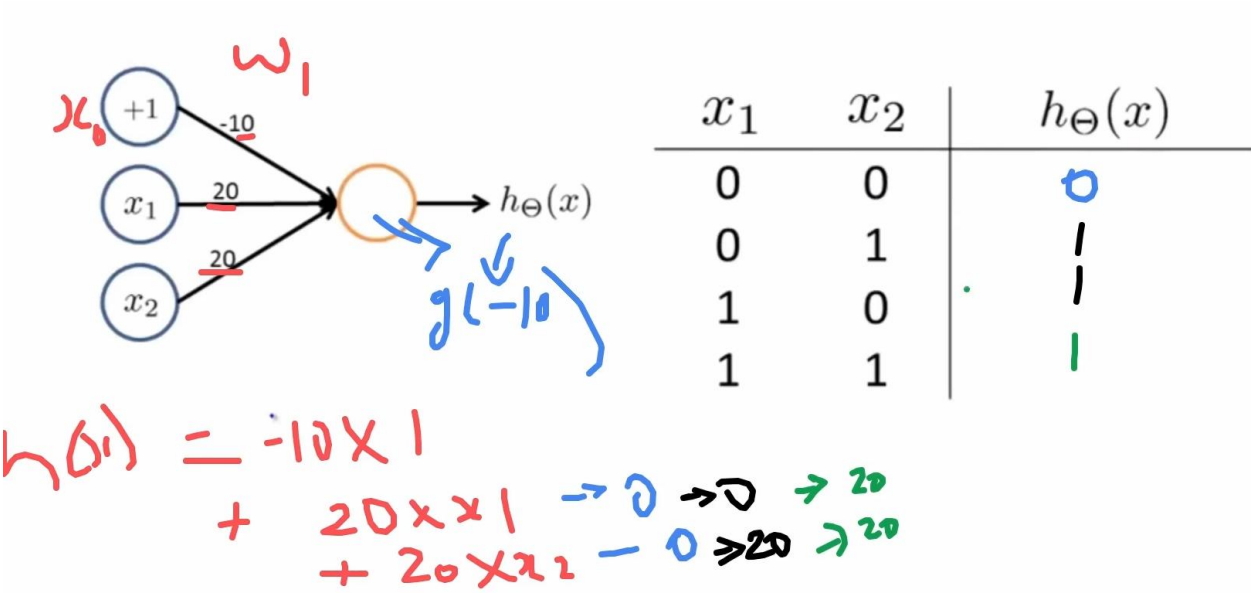
- The computation is calculated and since in the first case X1 AND X2 are 0 therefore total weight is only from the base which is 1 * -10 = -10 and the sigmoid of the high negative gives output nearly 0 , hence output is 0



x_1	x_2	$h_\Theta(x)$
0	0	0
0	1	1
1	0	
1	1	

- Remember X2 is 1 in the next case so then 20 * 1 = 20 and then add up with -10 which gives = 10 , hence gives 1 since more positive.

Likwise for others :



x_1	x_2	$h_\Theta(x)$
0	0	0
0	1	1
1	0	1
1	1	1

TENSORFLOW

What is Tensorflow?

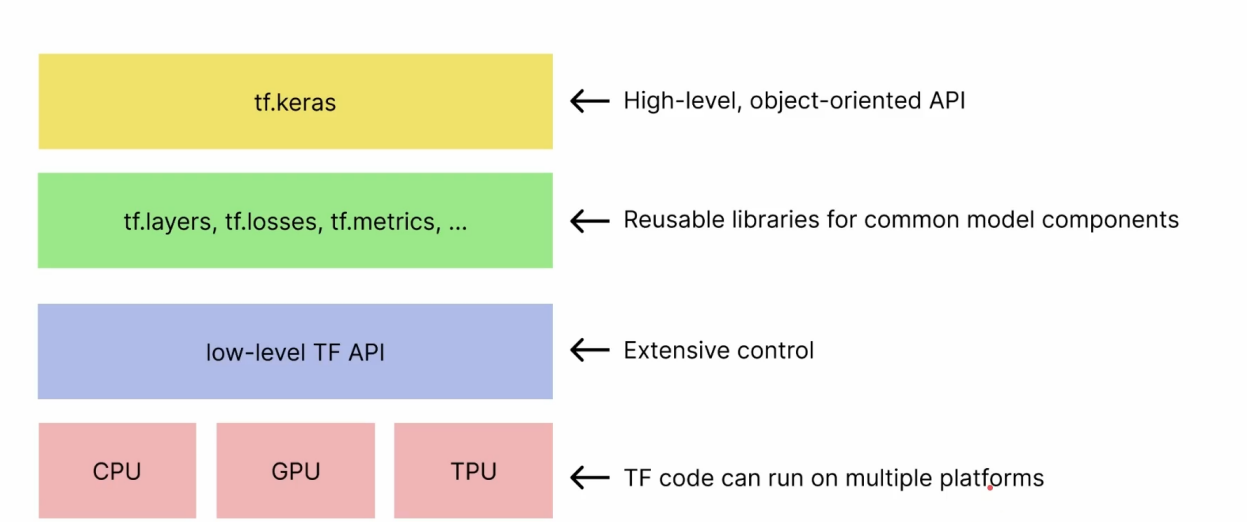
TensorFlow is an open source, end-to-end platform for Machine Learning, under Google. Tensorflow makes it easy for beginners and experts to create machine learning models for desktop, mobile, web, and cloud.

- TensorFlow makes it easy to create ML models that can run in any environment.
- Supports Multiple Dataset management and import tools. Even has own datasets loaded in the framework.
- It is highly flexible to use. Has abstracted APIs to quickly build models but also has Core API provision for very fine-tuned development.
- TensorFlow has a large community of developers and contributors, extensive documentation, and numerous tutorials and resources available online.

- Keras is an API designed for human beings, not machines.
- Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs
- It minimizes the number of user actions required for common use cases, and it provides clear & actionable error messages.
- Keras also gives the highest priority to crafting great documentation and developer guides.
- Keras has the low-level flexibility to implement arbitrary research ideas while offering optional high-level convenience features to speed up experimentation cycles.

- "What is the difference between steering wheel and the car. When should i use one over the other".

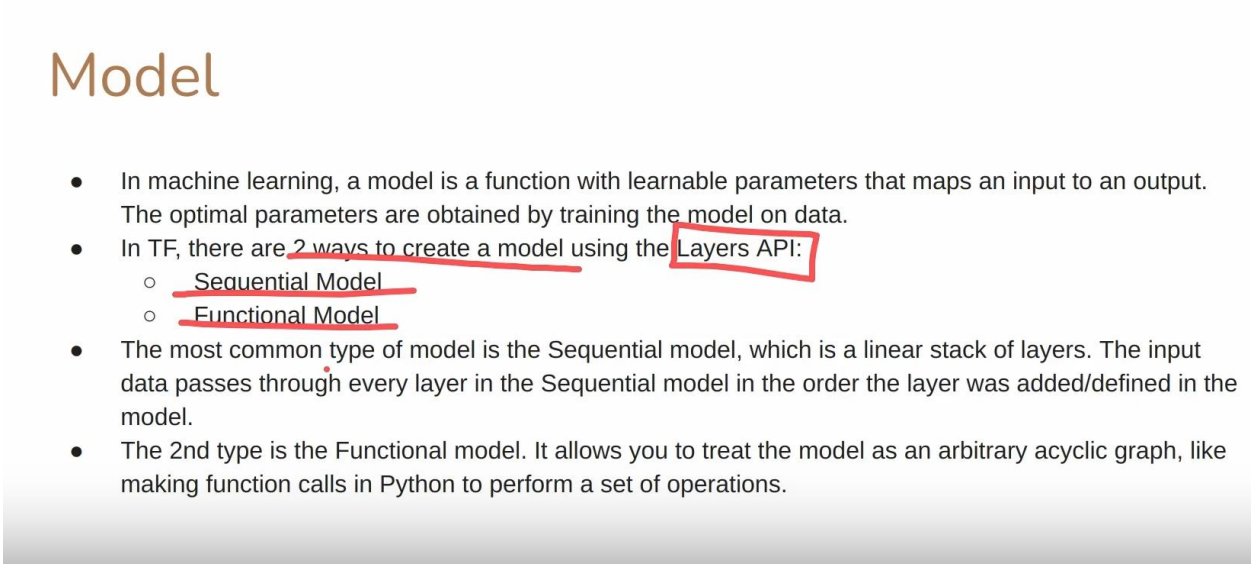
- TensorFlow is a low level library. Keras is higher level library that is build on top of TensorFlow (Keras used to be able to run on other low level libraries but i believe it is in a past). When you use Keras you are using TensorFlow indirectly.
- So if you are building something simple use Keras. If you need more control over your model you will need to use TensorFlow. Start with Keras it is very intuitive and simple library but very powerful.



BASICS IN TENSORFLOW

Tensors

- Tensors are multi-dimensional arrays with a uniform type (called a dtype)
- Tensors are kind of like np.arrays for understanding purposes. You can have vectors and matrices in Tensors, they behave a bit differently than numpy vectors/matrices.
- All tensors are immutable like Python numbers and strings: you can never update the contents of a tensor, only create a new one.
- Tensors can be directly stored on GPU/TPUs when being created. Any operations using Tensors also support using the GPU/TPU
- Few terms:
  - **Shape**: The number of elements of each of the axes of a tensor.
  - **Rank**: Number of tensor axes. A scalar has rank 0, a vector has rank 1, a matrix is rank 2.
  - **Axis** or **Dimension**: A particular dimension of a tensor.
  - **Size**: The total number of items in the tensor.



- Sequential model – usually linear stack of layers.

1. **Sequential Model** - This is the simpler, more straightforward approach. Think of it like stacking LEGO blocks in a straight line. You add one layer on top of another in order:
    - Input goes into Layer 1
    - Layer 1's output goes into Layer 2
    - Layer 2's output goes into Layer 3
    - And so on...

    It's called "sequential" because data flows sequentially through each layer in the exact order you defined them.

2. **Functional Model** - This is more flexible and powerful. Instead of just stacking layers linearly, you can create complex architectures with branches, multiple inputs/outputs, or skip connections. It's like building a graph where layers can connect in any way you want, not just straight through. Think of it as being able to have layers that split, merge, or skip ahead.

When to use which?

- Sequential: Great for simple, straightforward networks (most beginner projects)
- Functional: When you need something more complex, like ResNet architectures with skip connections, or models with multiple inputs/outputs

Layers

- Layers are the basic building blocks of neural networks.
- A layer consists of a tensor-in tensor-out computation function (the layer's method, activation, etc) and some state, held in TensorFlow variables (the layer's weights).
- Layers maintain a state, updated when the layer receives data during training, and stored in the layers' properties
- A model is built using a combination of layers such as InputLayer(), Dense(), RNN(), etc.
- You can use model.summary() to actually view all your layers assembled into the model, their shapes and parameter counts.

What are Layers? Layers are the fundamental building blocks of neural networks. Every neural network you build is made up of these layers stacked or connected together.

What's inside a Layer? Each layer has two main components:

1. **Computation function** (tensor-in, tensor-out):
    - This includes the layer's method (like matrix multiplication), activation function (like ReLU, sigmoid), and other operations
    - It takes a tensor as input and produces a tensor as output
2. **State** (the layer's weights):
    - These are the learnable parameters stored in TensorFlow variables
    - These weights get updated during training to make the model better

Key Points:

- **Layers maintain state** - The weights/parameters are stored in the layer's properties and get updated as the model learns from data during training
- **Models are combinations of layers** - You build a complete model by combining different types of layers like:
    - `InputLayer()` - defines what shape data comes in
    - `Dense()` - fully connected layer
    - `RNN()` - recurrent layer for sequences
    - etc.
- **You can inspect your model** - Use `model.summary()` to see:
    - All the layers in your model
    - The shape of data flowing through each layer
    - How many parameters (weights) each layer has

Sequential Model

```
# Define Sequential model with 3 layers
model = keras.Sequential(
    [
        layers.Dense(2, activation="relu", name="layer1"),
        layers.Dense(3, activation="relu", name="layer2"),
        layers.Dense(4, name="layer3"),
    ]
)
```

model = keras.Sequential([...])

- You're creating a Sequential model by calling keras.Sequential()
- Everything inside the square brackets [...] is a list of layers that will be stacked in order

layers.Dense(2, activation="relu", name="layer1")

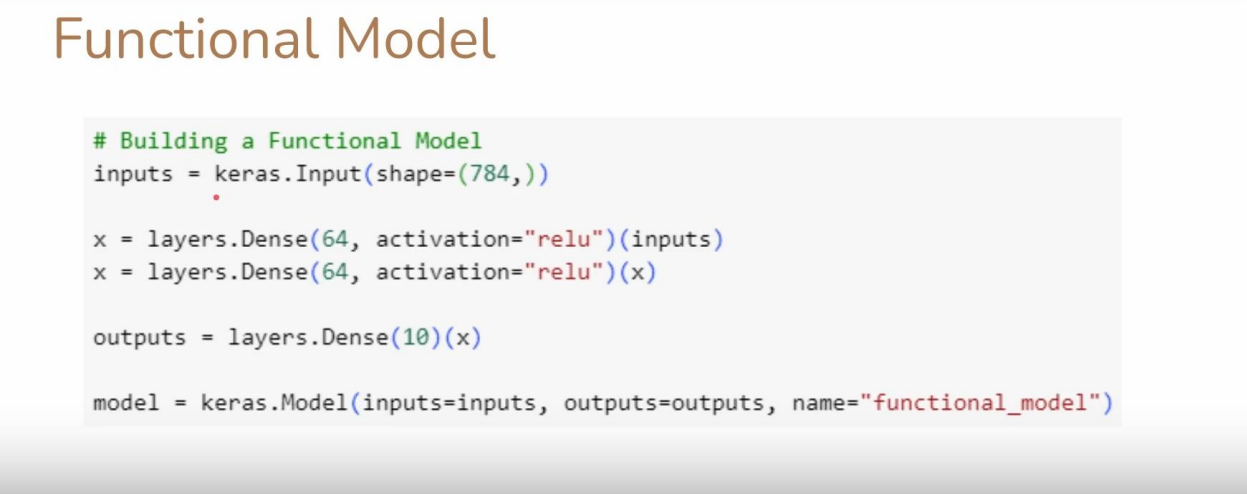
- This creates the FIRST layer
- Dense means it's a fully connected layer (every neuron connects to every neuron in the next layer)
- The number 2 means this layer has 2 neurons/units
- activation="relu" means after doing the math, apply the ReLU activation function (turns negative values to 0)
- name="layer1" is just a label so you can identify this layer later

layers.Dense(3, activation="relu", name="layer2")

- This is the SECOND layer
- Has 3 neurons
- Also uses ReLU activation
- Data from layer1 automatically flows into this layer

layers.Dense(4, name="layer3")

- This is the THIRD and final layer (output layer)
- Has 4 neurons
- Notice there's NO activation specified here! That's common for output layers, depending on your task
- Could be a classification task with 4 classes, for example. So the data flow is: Input → Layer1(2 neurons) → Layer2(3 neurons) → Layer3(4 neurons) → Output



inputs = keras.Input(shape=(784,))

- You're creating an INPUT layer explicitly
- shape=(784,) means your input data has 784 features (like a flattened 28x28 image = 784 pixels)
- The comma after 784 makes it a tuple in Python
- This creates a placeholder for where data will enter the model

x = layers.Dense(64, activation="relu")(inputs)

- Creates a Dense layer with 64 neurons and ReLU activation
- The (inputs) part at the end is KEY - you're calling the layer like a function and passing in the inputs
- This connects the Dense layer to receive data from the inputs
- The result is stored in variable x (this is a tensor, the output of this layer)

x = layers.Dense(64, activation="relu")(x)

- Creates ANOTHER Dense layer with 64 neurons and ReLU
- This time you pass in x (the previous layer's output)
- The output overwrites x, so x now holds the output of this second Dense layer
- This is how you chain layers together manually

outputs = layers.Dense(10)(x)

- Creates the final output layer with 10 neurons (probably for 10 classes in classification)
- Takes x (the previous layer) as input
- Stores the result in outputs variable
- No activation here either (you'd add softmax during compilation for classification)

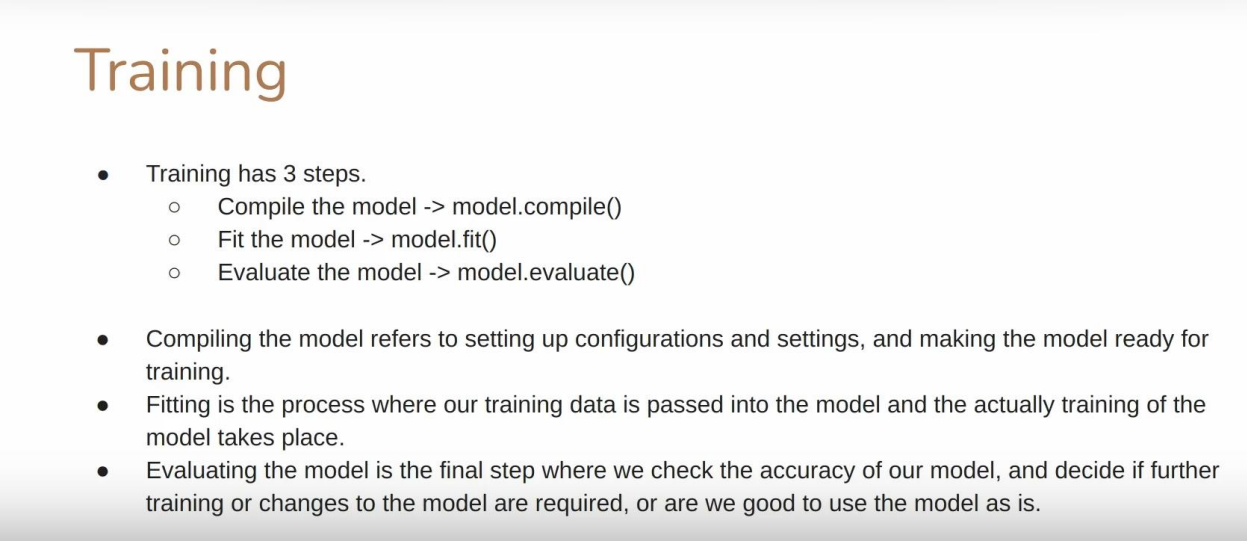
model = keras.Model(inputs=inputs, outputs=outputs, name="functional_model")

- NOW you actually create the model!
- You tell it: "the model starts at inputs and ends at outputs"
- Keras traces the path from inputs to outputs through all the layers you connected
- name="functional_model" is just a label for the whole model

So the data flow is: inputs(784) → Dense(64) → Dense(64) → Dense(10) → outputs

KEY DIFFERENCE IN THE CODE:

- **Sequential:** You just list the layers, and Keras automatically chains them
- **Functional:** You manually connect each layer by calling it like a function with the previous layer as input, then explicitly tell Keras where the model starts (inputs) and ends (outputs)



ACTIVATION FUNCTIONS

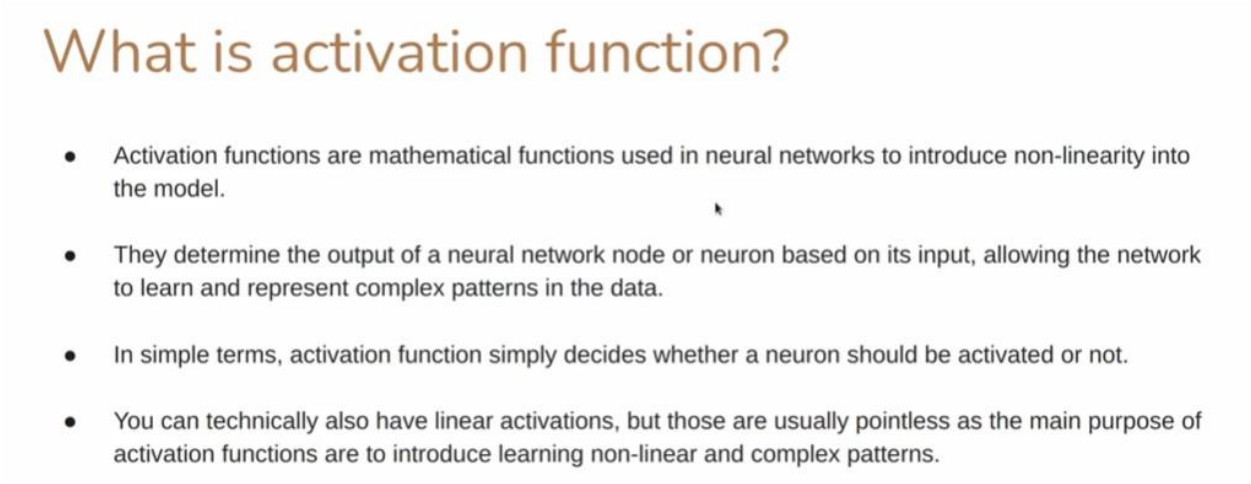
What They Do

Think of activation functions as **decision-makers** in a neural network.

They take a number (input) and transform it into another number (output) in a specific way. This transformation helps the network learn complex patterns instead of just straight lines.

Why we need them: Without activation functions, no matter how many layers you stack, your neural network would just be doing fancy addition and multiplication - it could only learn straight-line relationships.

Activation functions add the "twist" that lets networks learn curves, patterns, and complex decisions.

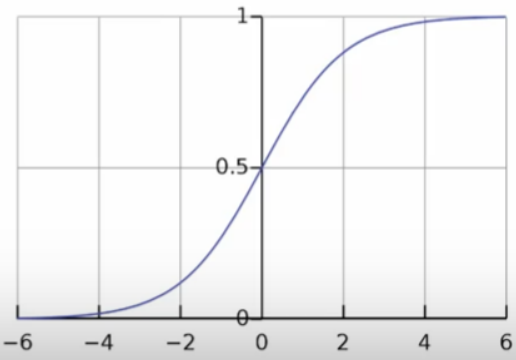


- Having non linear and complex patterns help to learn the model with other data.

Sigmoid

$$S(x) = \frac{1}{1 + e^{-x}}$$

- Output ranges from 0 to 1.
- Output is 0 at -infinity while output is 1 at +infinity
- Most common function used to introduce non-linearity (like seen in Logistic Regression)



1. Sigmoid

Formula: S(x) = 1/(1+e^(-x))

What it does: Squishes any number into a range between 0 and 1.

- Big negative numbers → close to 0
- Big positive numbers → close to 1
- Zero → 0.5

Why use it:

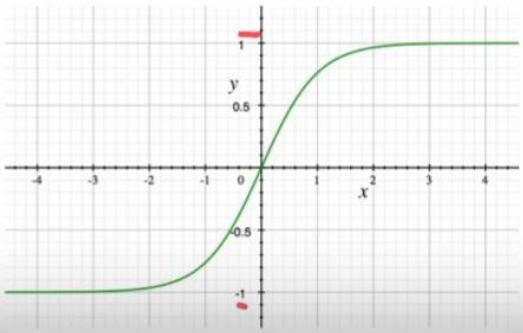
- Great for **probability outputs** (like "is this a cat? 70% sure")
- Used in **logistic regression**
- Good for the **final layer** when you need a yes/no probability

Downsides:

- **Vanishing gradient problem** - when values are very large or small, learning becomes super slow
- Outputs aren't centered around zero (always positive)

TanH

$$f(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$



- Output ranges from -1 to 1.
- Output is -1 at -infinity while output is 1 at +infinity
- TanH has slightly larger gradients and hence has a stronger effect in training.

- Prefered over sigmoid.
- Similar to sigmoid but in **range -1 to 1** where as sigmoid is 0-1
- Larger gradients and make the training more faster and efficient.
- Has an negative effect! If neuron is negative it has a pulling effect or otherside.

2. TanH (Hyperbolic Tangent)

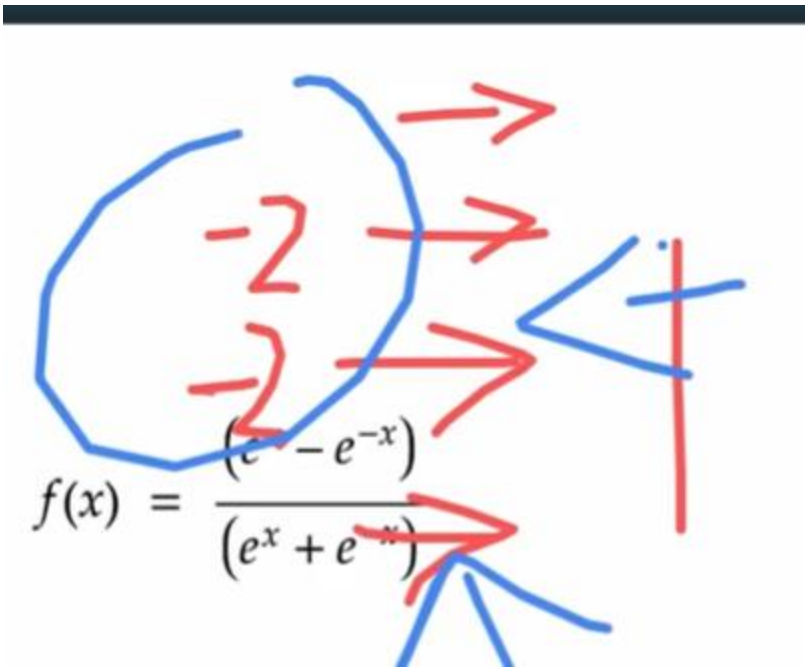
Formula: f(x) = (e^x - e^(-x))/(e^x + e^(-x))

What it does: Similar to sigmoid but squishes numbers between -1 and 1 instead.

Why use it:

- **Better than sigmoid** for hidden layers
- **Zero-centered** - outputs can be negative or positive
- **Stronger gradients** - learns faster than sigmoid
- Good when you want outputs that can push neurons in both directions (positive/negative)

When to use: Hidden layers in older neural networks (though ReLU is more common now)

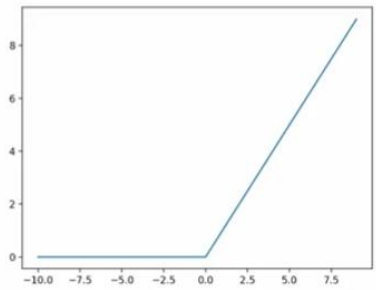


ReLU

(Rectified Linear Unit)

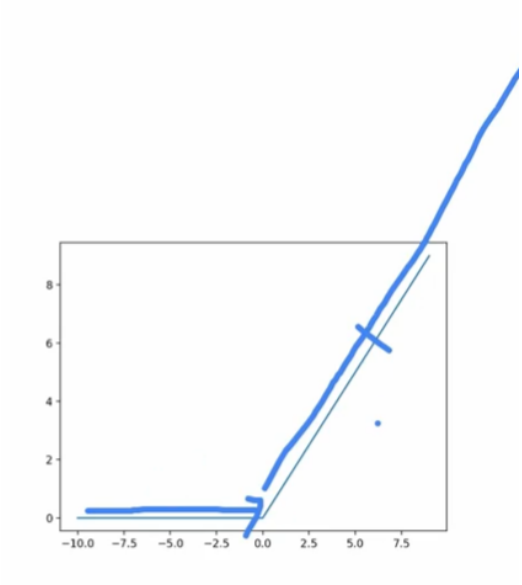
ReLU(x) = max(0,x)

- Output ranges from 0 to +infinity
- Simply, defined as the positive part of its input.
- Most actively used activation function in Neural Networks



- ACTUAL GOTO FUNCTION!
- Rectified L Unit.
- Max function between 0 and X.
- If X is positive it will use , if negative then 0.

- Most active and mostly used.
- After certain threshold , neurons fire , if stimulus is given some amount , it will fire till that much.



3. ReLU (Rectified Linear Unit) ⭐ THE GO-TO CHOICE

Formula: ReLU(x) = max(0, x)

What it does: Super simple!

- If input is **positive** → keep it as is
- If input is **negative** → make it 0

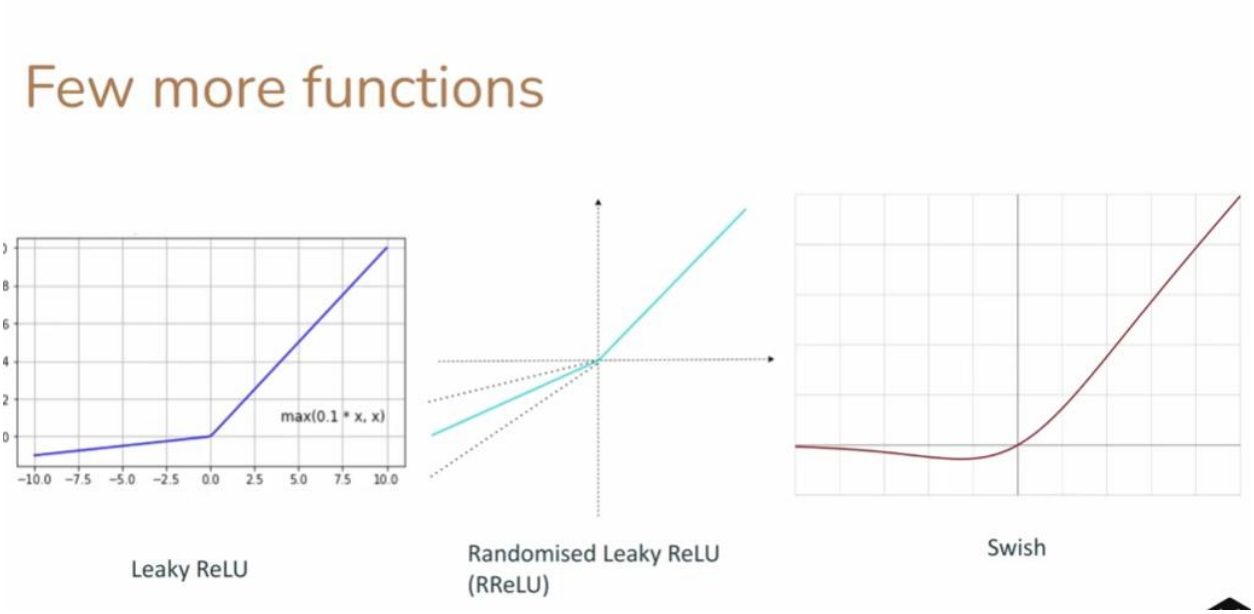
Why it's THE BEST:

- **Fast to compute** - just a simple comparison
- **No vanishing gradient** for positive values
- **Sparse activation** - many neurons output 0, making the network efficient
- **Most widely used** in modern deep learning (CNNs, etc.)

Real-world analogy: Like a neuron in your brain - it only "fires" (activates) after a certain threshold. Below threshold = nothing happens (0). Above threshold = it responds proportionally.

Downside:

- **Dying ReLU problem** - if a neuron always gets negative inputs, it stays at 0 forever and stops learning



4. Leaky ReLU

Formula: $f(x) = \max(0.01x, x)$

What it does: Like ReLU but gives a small slope to negative values instead of making them completely 0.

Why use it:

- **Fixes the dying ReLU problem**
- Negative inputs still have a tiny gradient (0.01x instead of 0)
- Neurons can "recover" if they go negative

When to use: When you notice many neurons dying during training with regular ReLU

5. Randomized Leaky ReLU (RReLU)

What it does: Like Leaky ReLU but the slope for negative values is **random** (changes during training)

Why use it:

- Acts as a form of **regularization** (prevents overfitting)
- The randomness helps the model generalize better

6. Swish

Formula: $f(x) = x \times \text{sigmoid}(x)$

What it does: A smooth, non-linear function developed by Google

Why use it:

- **Smooth everywhere** (no sharp corners like ReLU)
- Often performs **better than ReLU** in very deep networks
- Self-gated (it modulates itself)

When to use: State-of-the-art deep networks where you want maximum performance

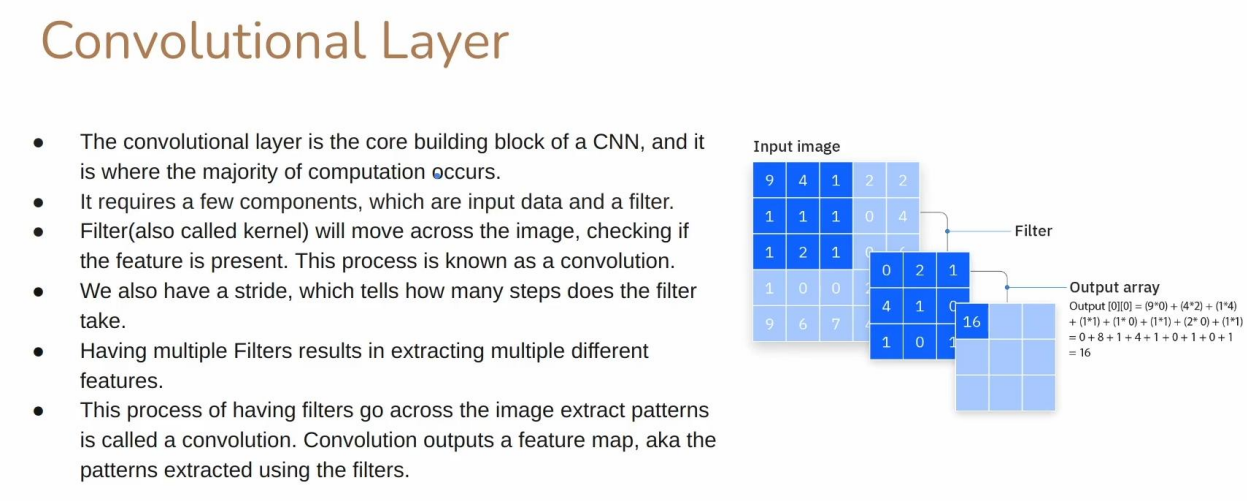
CNNs (Convolutional neural networks)

What are CNNs?

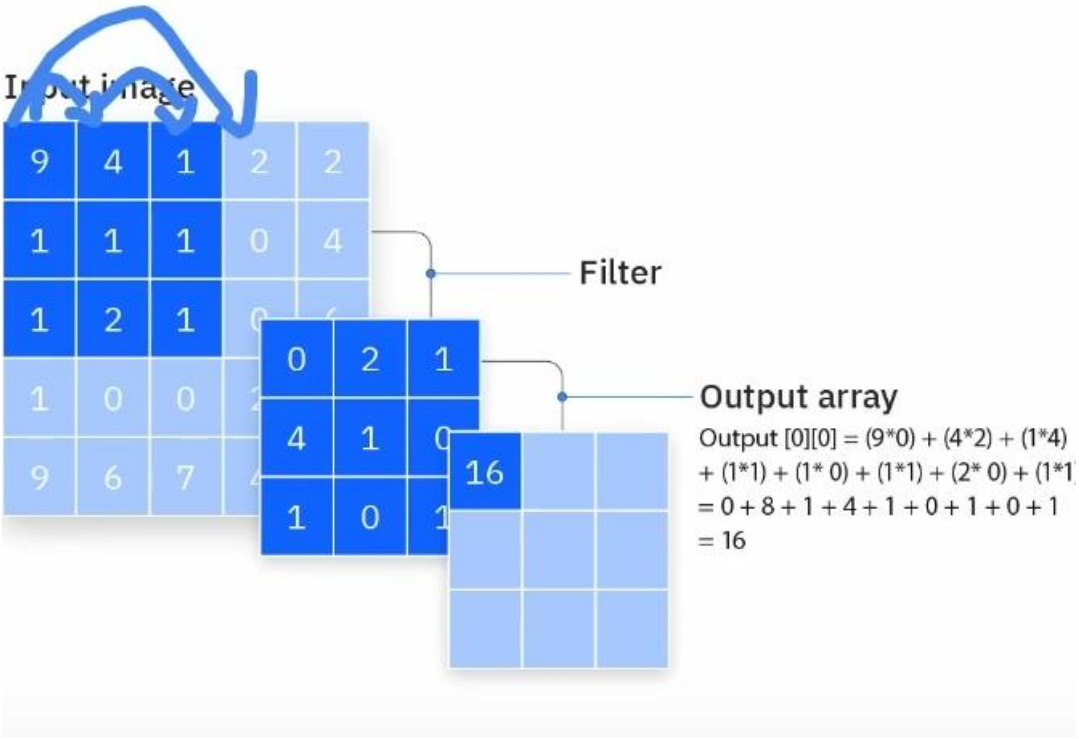
- Convolutional neural networks use three-dimensional data for image classification and object recognition tasks.
- Convolutional neural networks provide an approach to image classification and object recognition tasks, leveraging principles matrix multiplication concepts, to identify patterns within an image.
- They have three main types of layers, which are:
  - Convolutional layer
  - Pooling layer
  - Fully-connected (FC) / Dense layer

- 3 dimentional means RGB colors for each pixel.

Convolutional Layer

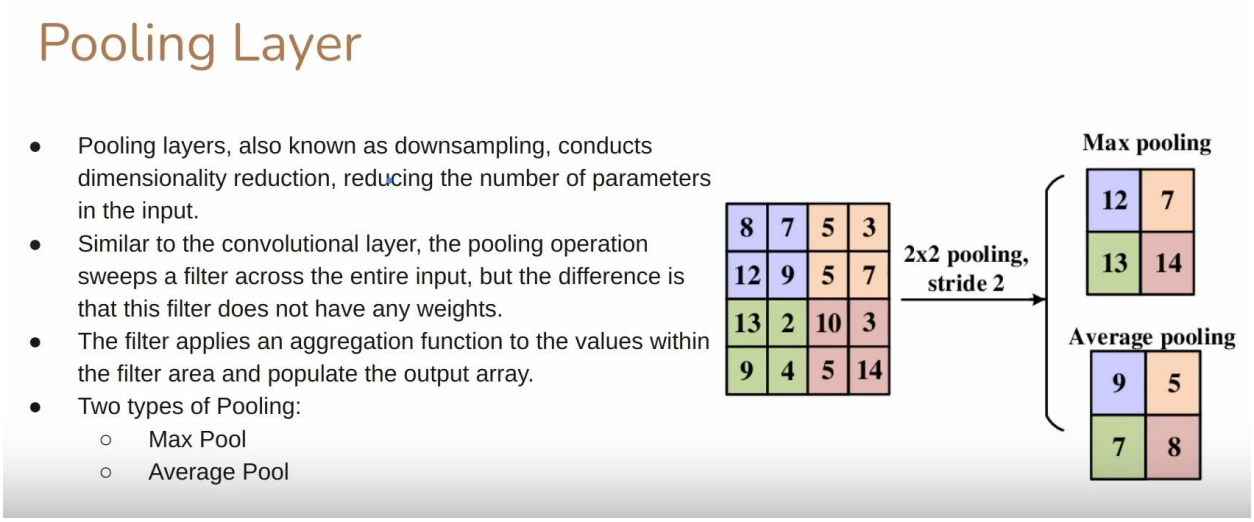


- A filter is also a kernal actually a vector or a window or slide which moves across the image. And perform the computation to receive the output and see if a feature is present.
- Stide say how many steps does this filter move. The filter move like one cell at a time or jumps two cells at a time.

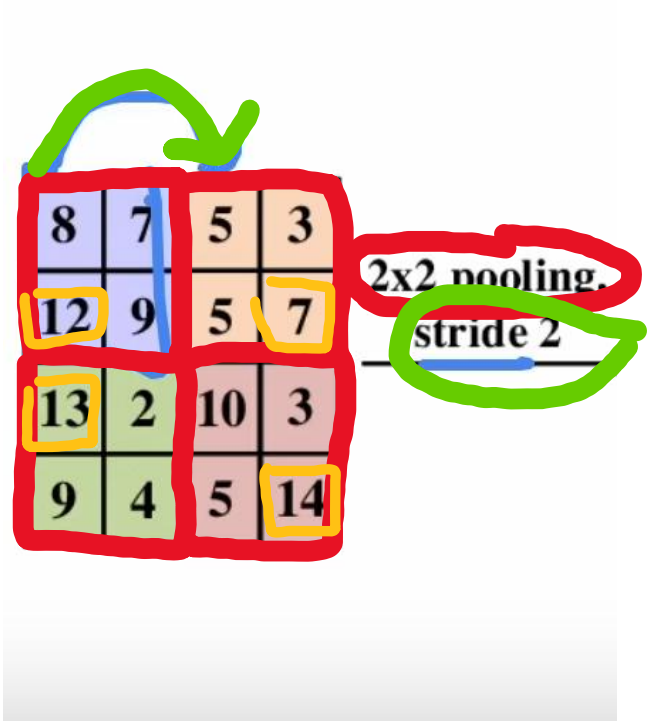


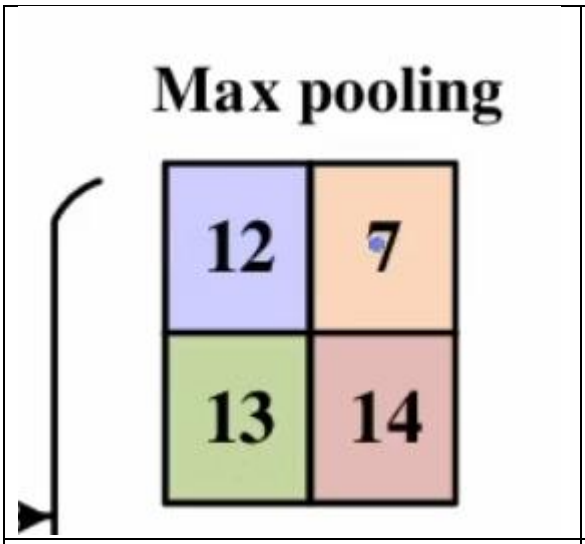
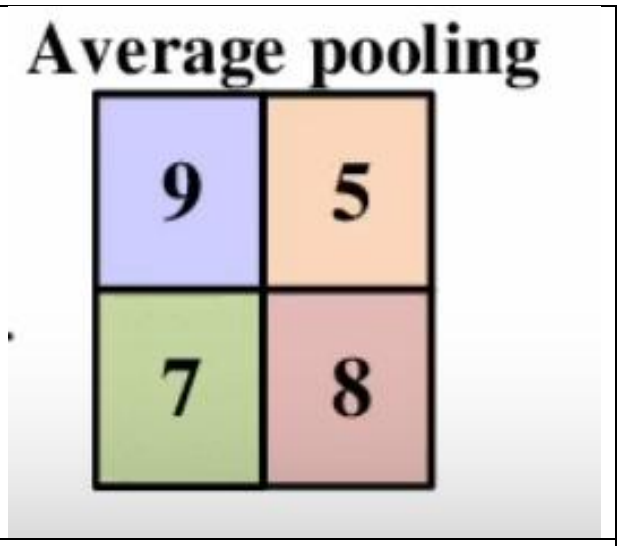
- Having multiple filters results in extracting multiple features.

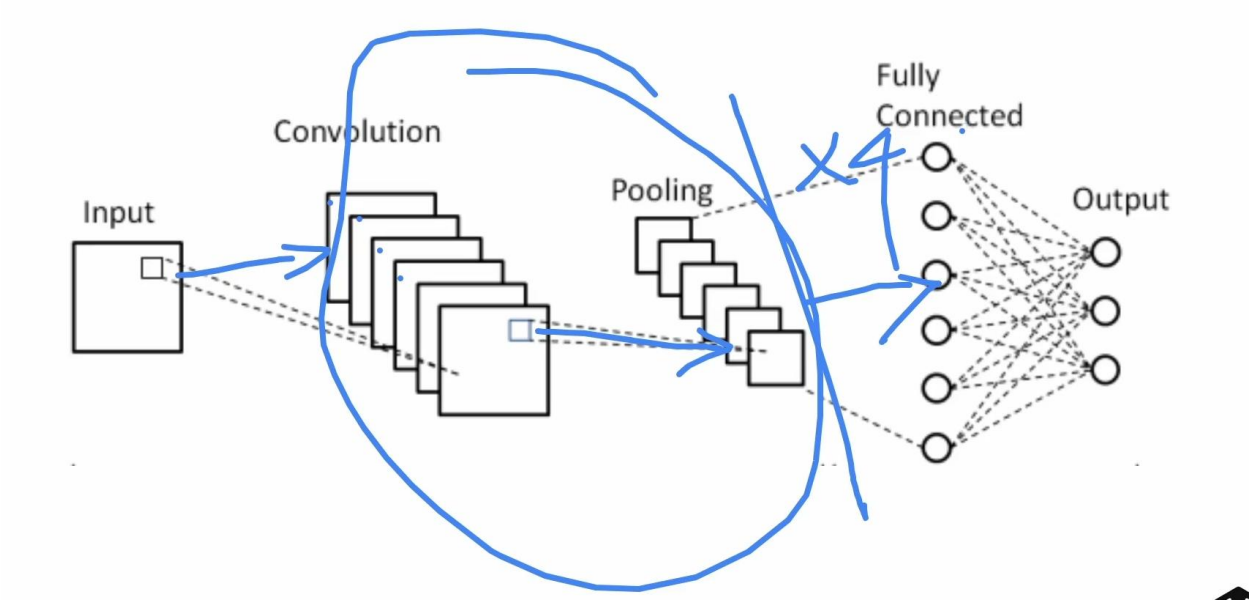
Pooling Layers



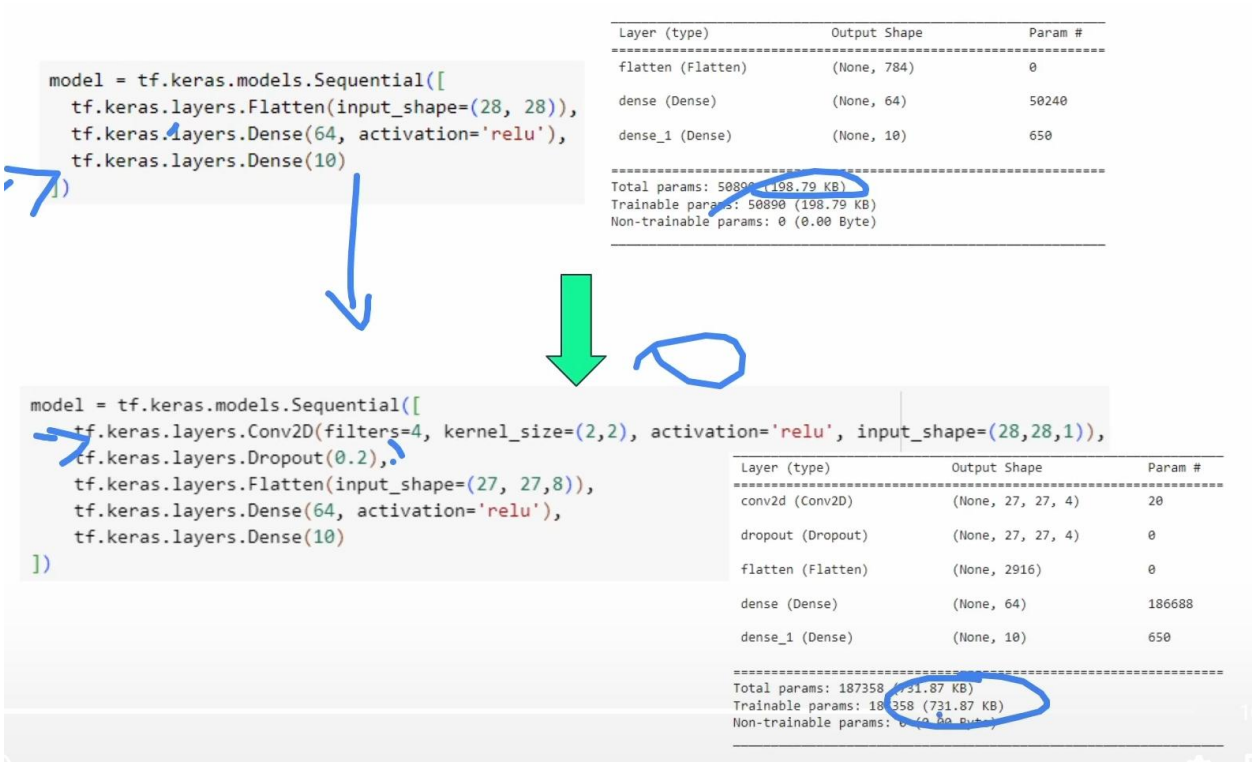
- Strink input vector size. Here we also have filter but filter does not have any weight or confirgaration to the filter and stride.
  - Max pooling is basically picking the highest or max number of each region so its 12,7,13,14



	
Maximum from all the pools	Average of each pool



- Filter is 2x2 matrix , number of filters = 4
- Dropout means randomly skips out a couple of outputs. If convolutional layer is outputting 25 values, the dropout will randomly skip 20% of the value. (turns 20% of the values to 0)



```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(64, activation='relu'),
  tf.keras.layers.Dense(10)
])
```

```
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0

dense (Dense)                (None, 64)                50240

dense_1 (Dense)              (None, 10)                650

=================================================================
Total params: 50890 (198.79 KB)
Trainable params: 50890 (198.79 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(filters=4, kernel_size=(2,2), activation='relu', input_shape=(28,28,1)),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Flatten(input_shape=(27, 27,8)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10)
])
```

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 27, 27, 4)         20

dropout (Dropout)            (None, 27, 27, 4)         0

flatten (Flatten)            (None, 2916)              0

dense (Dense)                (None, 64)                186688

dense_1 (Dense)              (None, 10)                650

=================================================================
Total params: 187358 (731.87 KB)
Trainable params: 187358 (731.87 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

- Used for high input values not just small input value.

Transfer Learning

What is Transfer Learning?

- Transfer learning is the reuse of a pre-trained model on a new problem.
- Transfer Learning can train deep neural networks with comparatively little data.
- This is very useful in the data science field since most real-world problems typically do not have millions of labeled data points to train such complex models.
- With transfer learning, we basically try to exploit what has been learned in one task to improve generalization in another. We transfer the weights that a network has learned at “task A” to a new “task B.”

Advantages

- Training time
- Better performance with less data
- Not having to design an architecture from scratch

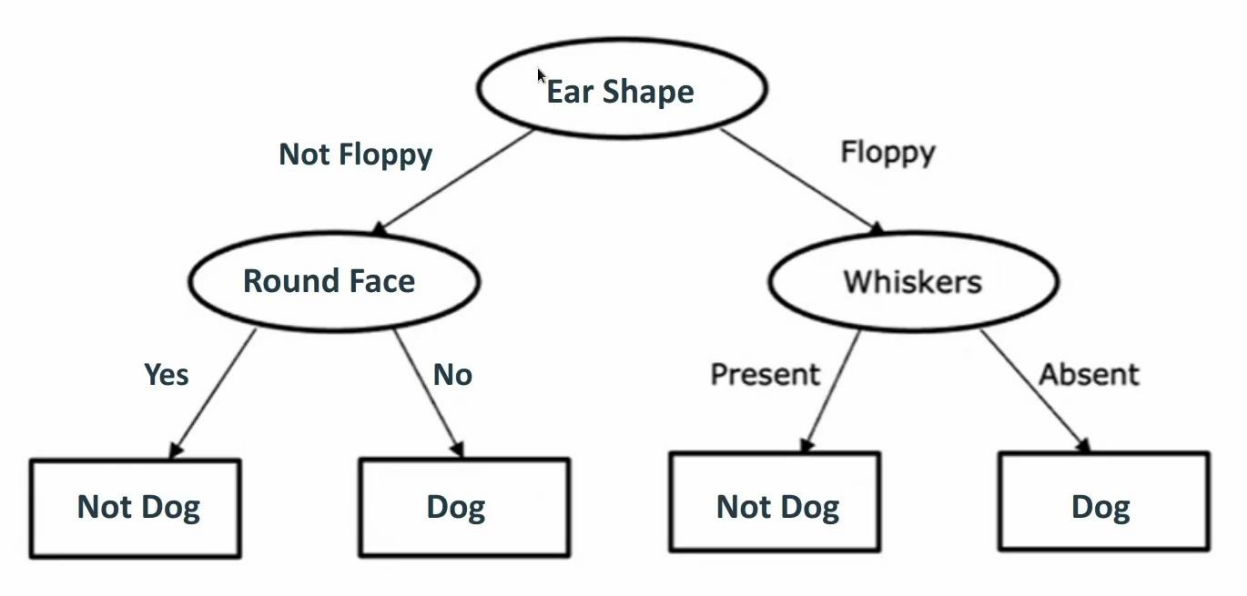
Few example scenarios for Transfer Learning

- Use a model trained to detect dogs, and retrain it a bit using new images to detect wild animals like wolves and foxes.
- Use a face detection model, and train with new data to create a expression detection model
- Use a model architecture of an animal image classification model and train your own model for car image classification

DECISION TREES

Ear type	Round Face	Whiskers	Is Dog
Not Floppy	No	No	1
Floppy	No	Yes	0
Not Floppy	Yes	Yes	0
Floppy	Yes	No	1
Not Floppy	No	No	1
Not Floppy	No	No	1
Not Floppy	Yes	Yes	0
Floppy	Yes	No	1

- A flow diagram with conditons to classify whether a Dog or Not Dog.



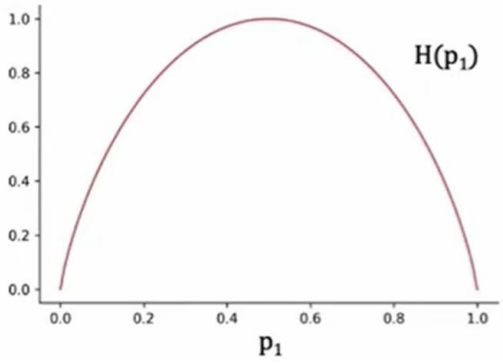
- An acylic tree because it goes only from top to bottom not other way.(Similar to tree data structure.)

Learning Process

Entropy

- A measure of disorder or impurity in a node, or how much variance the data has.

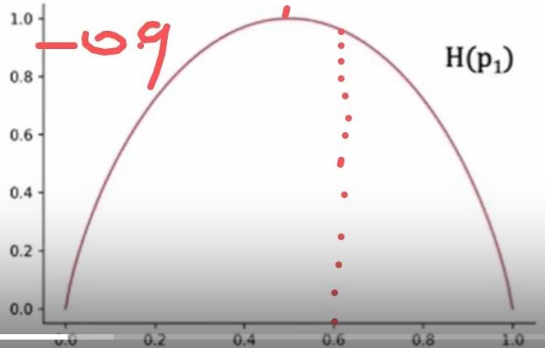
$$-p_1\ log_2(p_1) - (1 - p_1)log_2(1 - p_1)$$



Entropy

- A measure of disorder or impurity in a node, or how much variance the data has.

$$-p_1\ log_2(p_1) - (1 - p_1)log_2(1 - p_1)$$



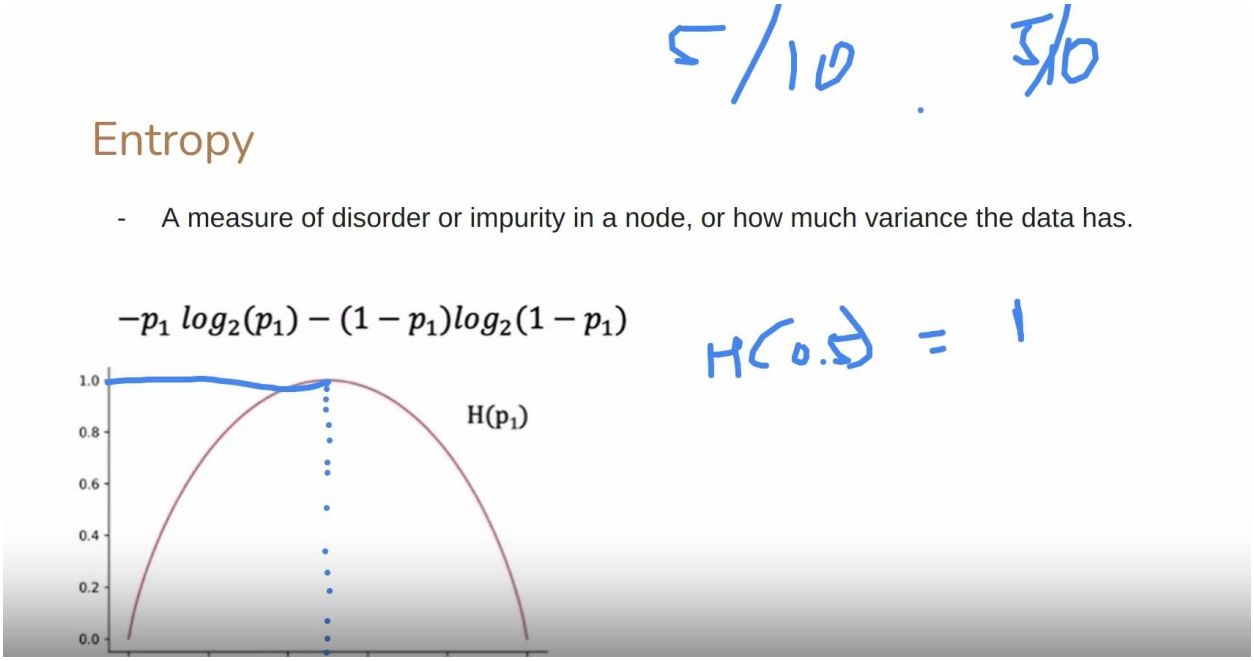
H(p)

$p_1 = 3/5$

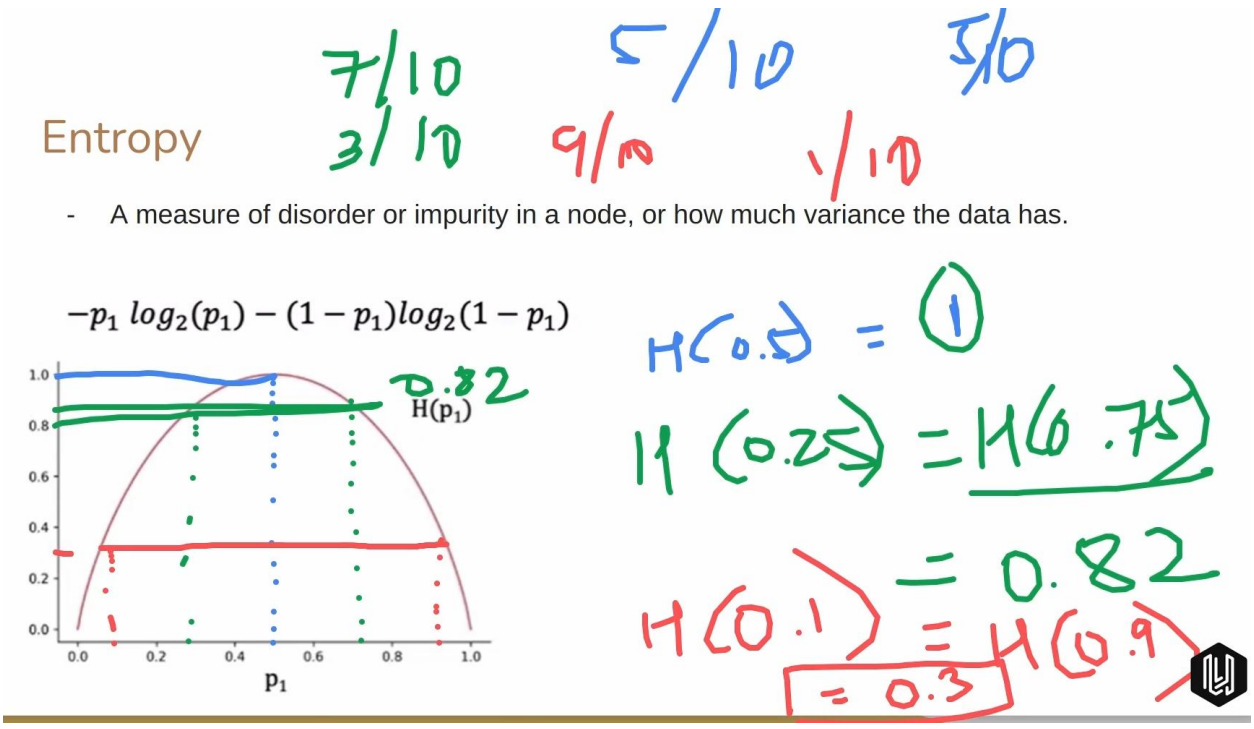
$= 0.6$

- If out of 5 samples 3 are the match then u would check and get the entropy

- If the entropy is 0.5 that is the point where it's the most impure this is because we cant split(Can't split YES OR NO!). Hence we can take 0.5 as the value leading to entropy of 1.



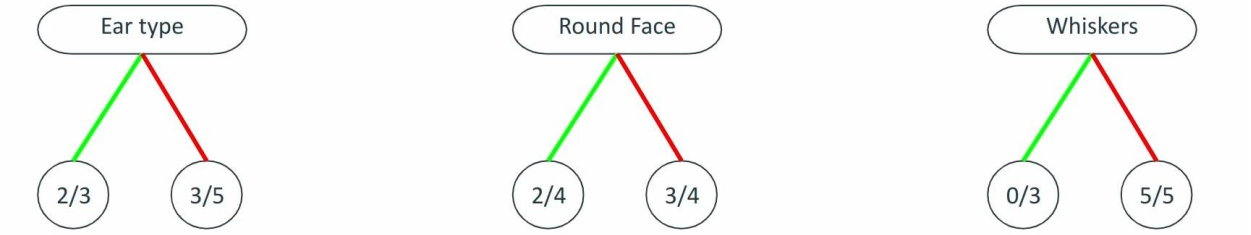
- Blue means 5 samples true and 5 are false
- Green means 7 samples true and 3 samples false
- Red means 9 samples are true and 1 sample is false.



Information Gain

- As the entropy or impurity decreases the information gain increases. In decision tree training, the reduction of entropy/impurities is called as Information gain.
- **Root** node is basically the previous node or the parent node of the current node.
- **W = Weight**
- **H = Entropy**

$$H(p_1^{root}) - \left(w^{left} H\left(p_1^{left}\right) + w^{right} H\left(p_1^{right}\right)\right)$$

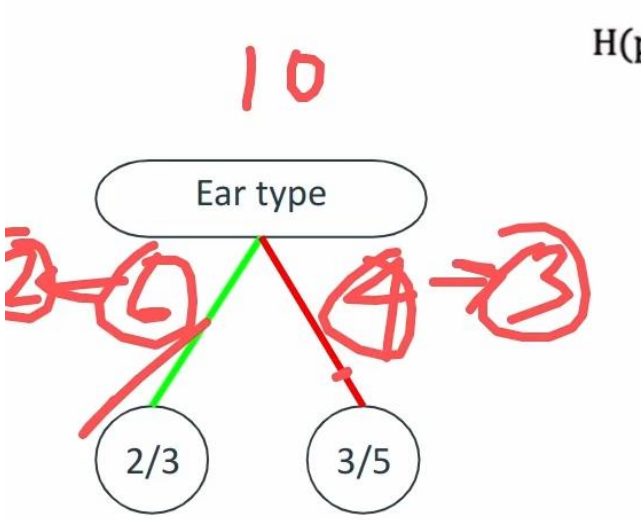


IG = 1 - (⅜ * 0.92 + ⅝ * 0.97) = 0.05

IG = 1 - (½ * 1 + ½ * 0.81) = 0.1

IG = 1 - (⅜ * 0 + ⅝ * 0) =

- As you can see in the first example:
  - 2/3 and 3/5 are the weights however remember Weight is basically the total not belonging to that specific class!!!



- For example if we have 10 samples , 6 samples are green and 4 samples are red. But actually out of the green 6 samples only 2 belong to the green class and in the red out off 4 only 3 belong to the red.
- Entropy means how many actually belong to the respective class not the total!
- Values 2/3 and 3/5 are the entropy values! Not the Weights.

For ear type

The Weights (3/8 and 5/8):

Weights = (number of samples in that branch) / (total samples)

- Total samples at parent = **8 dogs**
- Left branch has **3 dogs** → weight = **3/8**
- Right branch has **5 dogs** → weight = **5/8**

The weights represent **what fraction of the data went to each branch.**

The Entropy Values (0.92 and 0.97):

These are calculated using the **purity within each leaf node:**

Left branch (2/3):

- 3 dogs total
- 2 are "Dog", 1 is "Not Dog"
- Probability of "Dog" = 2/3
- Probability of "Not Dog" = 1/3

Entropy formula: $H = -p_1\log_2(p_1) - p_2\log_2(p_2)$

$H = -(2/3)\log_2(2/3) - (1/3)\log_2(1/3) = \mathbf{0.918 \approx 0.92}$

Right branch (3/5):

- 5 dogs total
- 3 are "Dog", 2 are "Not Dog"
- Probability of "Dog" = 3/5 = 0.6
- Probability of "Not Dog" = 2/5 = 0.4

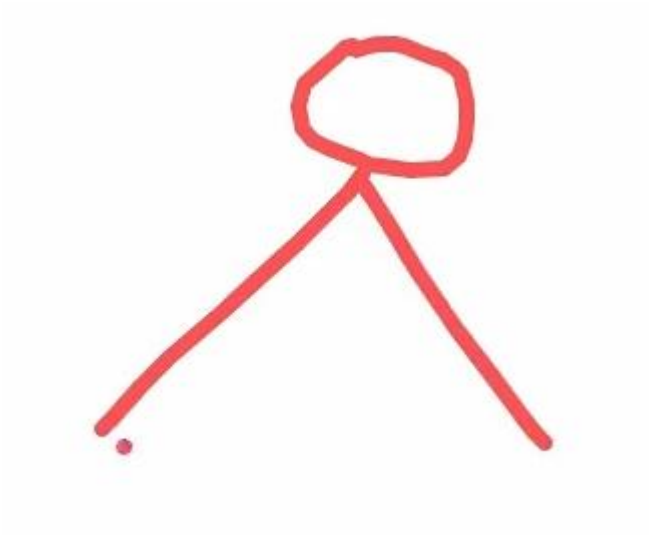
$H = -(3/5)\log_2(3/5) - (2/5)\log_2(2/5) = \mathbf{0.971 \approx 0.97}$

$IG = 1 - (3/8 \times 0.92 + 5/8 \times 0.97) = 0.05$

PROCESS OF MAKING A DECISION TREE

- Start with all examples at the root node.
- Calculate information gain for all possible features, and pick the one with the highest information gain.
- Split dataset according to selected feature, and create left and right branches of the tree.
- Keep repeating splitting process until stopping criteria is met:
    - When a node obtained is 100% pure
    - When splitting a node will result in the tree exceeding a preset maximum depth.
    - Information gain from additional splits is less than threshold.
    - When number of examples in a node is below a threshold.

- Do Information gains for all the features and the highest value or highest information gain is the root of the decision tree.
- Split the dataset according to the selected feature and create branches



- Likwise branches created with new nodes or features and splitted.

1. Start with all examples at the root node

- Begin with all 8 dogs at the top
- Parent entropy = 1.0 (4 "Dog", 4 "Not Dog")

---

2. Calculate information gain for all possible features

Test each feature:

- **Ear type:** IG = 0.05
- **Round Face:** IG = 0.1
- **Whiskers:** IG = 0 (no information gained!)

Pick the highest IG → Round Face wins! (IG = 0.1)

3. Split dataset according to selected feature

Split by "Round Face":

- **Left branch (Yes)**: some dogs go here
- **Right branch (No)**: other dogs go here

Create two new child nodes.

---

4. Keep repeating until stopping criteria:

○ **When a node is 100% pure**

- Example: Whiskers → Absent gives **5/5** (all "Dog")
- This is a **leaf node**, stop splitting!

○ **Maximum depth exceeded**

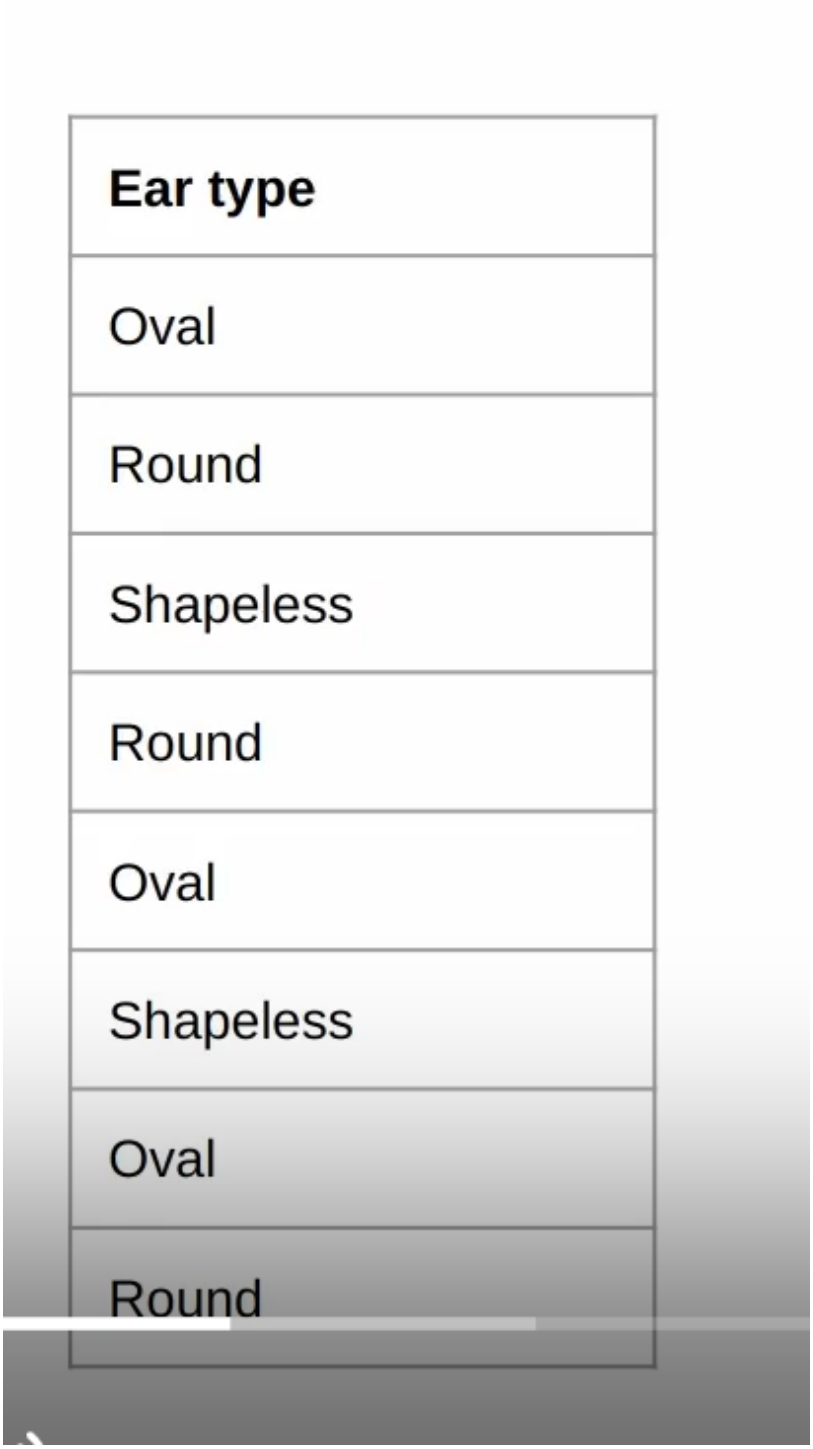
- If you set max_depth = 3, stop at level 3
- Prevents overfitting

○ **Information gain < threshold**

- Example: If IG = 0.001, too small to be useful
- Stop splitting, not worth it

Feature Handling

- Previously we had only single category like yes or no or Ears floppy or no likewise , However if we had:



Ear type
Oval
Round
Shapeless
Round
Oval
Shapeless
Oval
Round

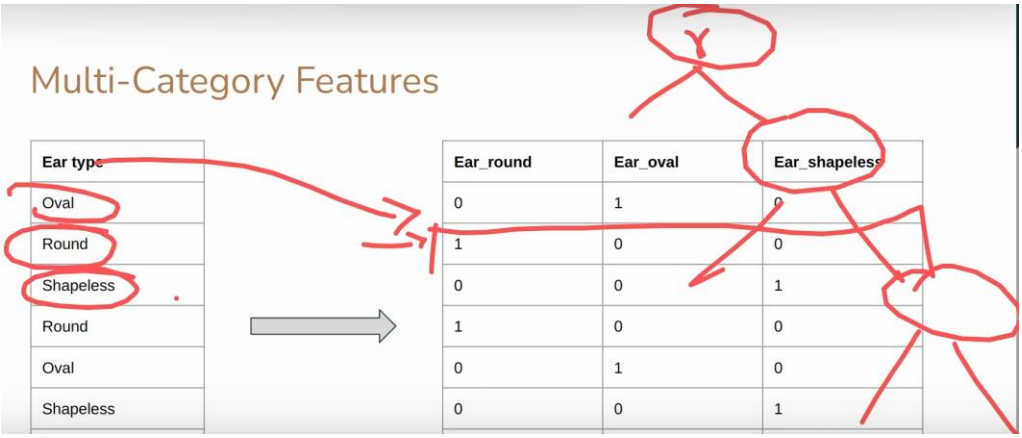
Multi Category Features

- Remember we cant split a node to multiple branches!!
- Only 2 branches so then how are we going to split?
- So what we do is we basically convert the individual category into a feature itself.
- Known as Hot encording. We convert them to binary.

Multi-Category Features

Ear type
Oval
Round
Shapeless
Round
Oval
Shapeless
Oval
Round

Ear_round	Ear_oval	Ear_shapeless
0	1	0
1	0	0
0	0	1
1	0	0
0	1	0
0	0	1
0	1	0
1	0	0



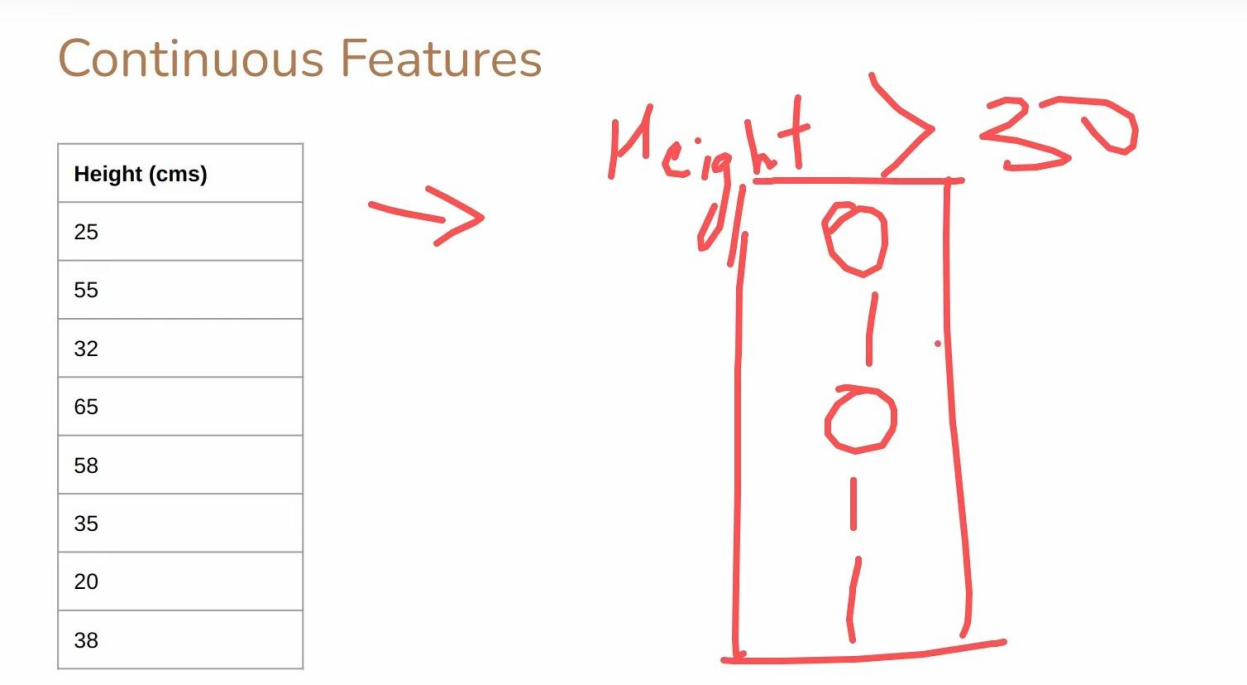
- Like multiple If/Else statements.
- If oval
    - If Round:
    - Else:
    - If Shapeless:
    - Else:

Continuous Features

- If its not categorical that means it’s a measured value.

Height (cms)
25
55
32
65
58
35
20
38

- We could set a threshold like: **Height > 30**



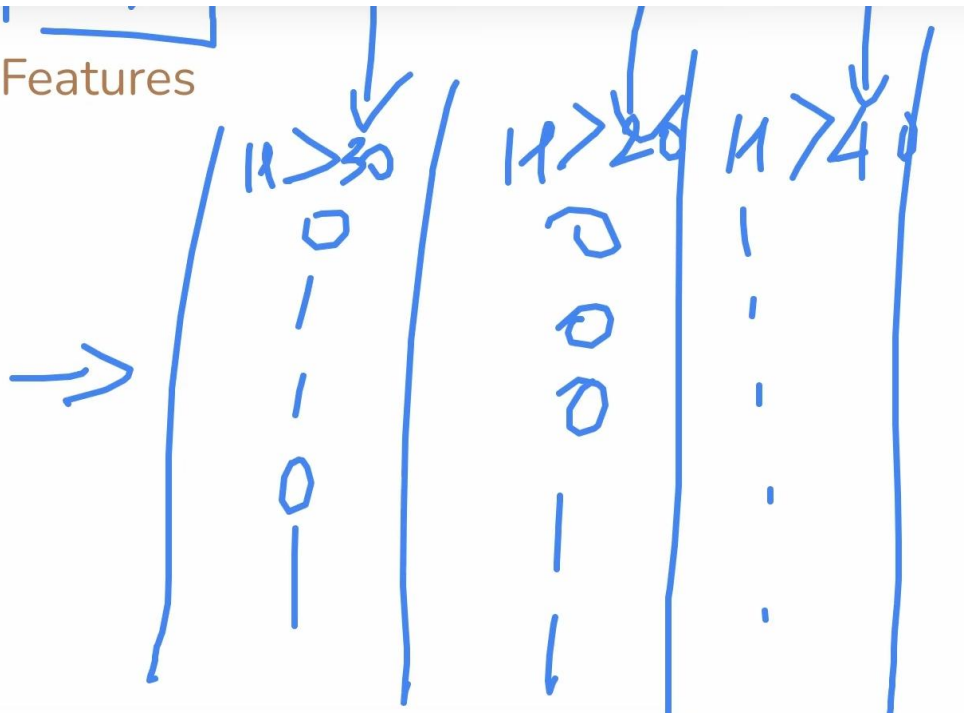
But how to get the threshold value 30?

It is done by basically like in this manner:

- **Initally we do heights with other values like 20,30,40 etc and the one with the highest information gain is taken!!**

Continuous Features

Height (cms)
25
55
32
65
58
35
20
38



- In rare cases we use multiple thresholds.

Ensemble Modelling

Ensemble Modelling - Simple Explanation

Think of it like asking for advice from multiple experts instead of just one!

What is it?

Combining **multiple models** (like decision trees) to make **better predictions** than any single model alone.

The Key Idea: "Wisdom of the Crowd"

- **Weak learners** = models that are only slightly better than random guessing
- When you combine many weak learners, they **balance out each other's mistakes**
- Result: A **strong learner** that's much more accurate!

Real-World Analogy:

Imagine you want to predict if it will rain:

- **Friend 1** (Decision Tree 1): Looks at clouds → says "Yes"
- **Friend 2** (Decision Tree 2): Looks at humidity → says "No"
- **Friend 3** (Decision Tree 3): Looks at wind → says "Yes"

Ensemble vote: 2 Yes, 1 No → Prediction: Rain! ☔

One friend might be wrong, but **together they're more reliable.**

Benefits:

✅ **Better accuracy** - multiple perspectives
✅ **Better generalization** - works well on new data
✅ **Reduces overfitting** - individual model mistakes cancel out

Random Forest

Random Forest

- Random Forest is an ensemble learning method for both classification and regression tasks.
- Multiple decision trees are constructed during the training process.
- The output is the mode of classes (classification) or mean prediction (regression) of individual trees.
- Functions under the concept of "bagging" (Bootstrap Aggregating). Hence, also called bagged decision trees.

- Multiple decision trees are contructed during the training. The output is the mode of the classes or the mean of the prediction of all the trees.

The Key Differences:

1. Ensemble Learning (General Concept)

- **What:** Combine multiple models (any type!)
- **Models can be:** Decision trees, neural networks, SVMs, anything!
- **How to combine:** Voting, averaging, stacking, etc.

Example:

- Combine 1 Decision Tree + 1 Neural Network + 1 SVM → Ensemble

---

2. Random Forest (Specific Method)

- **What:** Combine ONLY decision trees
- **Uses bagging** (bootstrap + aggregating)
- **Trees are trained independently** in parallel
- **Each tree sees different random data**

BAGGING

- The process or concept used to identify or learn.

The Ball Example - Complete Walkthrough

Your Original Dataset (The Bag):

You have **8 balls** representing your training data:

Bag: 🔴 🔵 🟢 🟡 🟠 🟣 ⚫ ⚪

Each ball represents **one training example** (like one dog with its features).

PART 1: BOOTSTRAP (Creating Samples)

What Does "WITH REPLACEMENT" Mean?

Let me show you the difference:

❌ **WITHOUT Replacement (Normal Picking):**

Imagine picking balls for a game:

1. Pick 🔴 → Remove it from bag
2. Pick 🔵 → Remove it from bag
3. Pick 🟢 → Remove it from bag
4. ...continue until bag is empty

Result: You get ALL 8 balls, each ball **exactly once**

🔴 🔵 🟢 🟡 🟠 🟣 ⚫ ⚪ ← Always the same 8 balls!

Problem: Every tree would see the **EXACT SAME DATA** → No diversity!

✅ **WITH Replacement (Bagging Way):**

Now imagine this process:

Round 1 - Creating Sample for Tree 1:

1. **Reach into bag** → Pick 🔴 → **PUT** 🔴 **BACK** → Write down "🔴"
2. **Reach into bag** → Pick 🟢 → **PUT** 🟢 **BACK** → Write down "🟢"
3. **Reach into bag** → Pick 🔴 → **PUT** 🔴 **BACK** → Write down "🔴" (red AGAIN!)
4. **Reach into bag** → Pick 🟡 → **PUT** 🟡 **BACK** → Write down "🟡"
5. **Reach into bag** → Pick ⚪ → **PUT** ⚪ **BACK** → Write down "⚪"
6. **Reach into bag** → Pick ⚪ → **PUT** ⚪ **BACK** → Write down "⚪" (white AGAIN!)
7. **Reach into bag** → Pick 🟠 → **PUT** 🟠 **BACK** → Write down "🟠"
8. **Reach into bag** → Pick 🔵 → **PUT** 🔵 **BACK** → Write down "🔵"

Tree 1's Sample: 🔴 🟢 🔴 🟡 ⚪ ⚪ 🟠 🔵

Notice:

- 🔴 appears **2 times** (because we put it back!)
- ⚪ appears **2 times**
- 🟣 and ⚫ **never appeared** (bad luck, they weren't picked)

Round 2 - Creating Sample for Tree 2:

The bag STILL has all 8 balls (we always put them back!):

Bag: 🔴 🔵 🟢 🟡 🟠 🟣 ⚫ ⚪

1. Pick 🔵 → PUT BACK → "🔵"
2. Pick 🔵 → PUT BACK → "🔵" (blue again!)
3. Pick 🟡 → PUT BACK → "🟡"
4. Pick 🟣 → PUT BACK → "🟣"
5. Pick 🟠 → PUT BACK → "🟠"
6. Pick 🔴 → PUT BACK → "🔴"
7. Pick 🟢 → PUT BACK → "🟢"
8. Pick ⚪ → PUT BACK → "⚪"

Tree 2's Sample: 🔵 🔵 🟡 🟣 🟠 🔴 🟢 ⚪

Notice:

- 🔵 appears **2 times**
- ⚫ **never appeared** this time

Round 3 - Creating Sample for Tree 3:

1. Pick 🟢 → PUT BACK → "🟢"
2. Pick 🔴 → PUT BACK → "🔴"
3. Pick 🟣 → PUT BACK → "🟣"
4. Pick 🟡 → PUT BACK → "🟡"
5. Pick ⚪ → PUT BACK → "⚪"
6. Pick ⚪ → PUT BACK → "⚪"
7. Pick 🟠 → PUT BACK → "🟠"
8. Pick 🟠 → PUT BACK → "🟠"

Tree 3's Sample: 🟢 🔴 🟣 🟡 ⚪ ⚪ 🟠 🟠

Notice:

- ⚪ appears **2 times**
- 🟠 appears **2 times**
- 🔵 and ⚫ **never appeared**

Why "With Replacement" is CRUCIAL:

Reason: Each tree sees **DIFFERENT data!**

- **Tree 1** thinks 🔴 is very important (saw it twice)
- **Tree 2** thinks 🔵 is very important (saw it twice)
- **Tree 3** thinks 🟠 and ⚪ are very important (saw them twice)

Each tree learns DIFFERENT patterns → This creates **diversity!**

PART 2: TRAINING

Now we train each tree on its bootstrap sample:

- **Tree 1** is trained on: 🔴 🟢 🔴 🟡 ⚪ ⚪ 🟠 🔵
- **Tree 2** is trained on: 🔵 🔵 🟡 🟣 🟠 🔴 🟢 ⚪
- **Tree 3** is trained on: 🟢 🔴 🟣 🟡 ⚪ ⚪ 🟠 🟠

Each tree builds different decision rules because they saw different data!

PART 3: AGGREGATING (Combining Predictions)

Now let's say a **NEW ball** appears: 🔶 (orange diamond)

We want to classify it: **"Is it similar to 🟠 (orange circle) or 🔴 (red circle)?"**

Step-by-Step Aggregation:

Step 1: Ask Each Tree

Tree 1's prediction:

- Tree 1 saw 🟠 once and 🔴 twice
- Tree 1 thinks: "This 🔶 looks more like 🔴"
- **Prediction: RED family** 🔴

Tree 2's prediction:

- Tree 2 saw 🟠 once and 🔴 once
- Tree 2 thinks: "This 🔶 looks more like 🟠"
- **Prediction: ORANGE family** 🟠

Tree 3's prediction:

- Tree 3 saw 🟠 twice and 🔴 once
- Tree 3 thinks: "This 🔶 is definitely like 🟠"
- **Prediction: ORANGE family** 🟠

Step 2: Vote (For Classification)

Count the votes:

Prediction	Votes
RED family 🔴	1 vote (Tree 1)
ORANGE family 🟠	2 votes (Tree 2, Tree 3)

Majority wins!

Final Prediction: ORANGE family 🟠 ✅

Reasoning Why This Works:

- **Tree 1** might be **wrong** because it saw too much red by chance
- **Trees 2 and 3** both agree on orange
- The **group decision** is more reliable than any single tree!

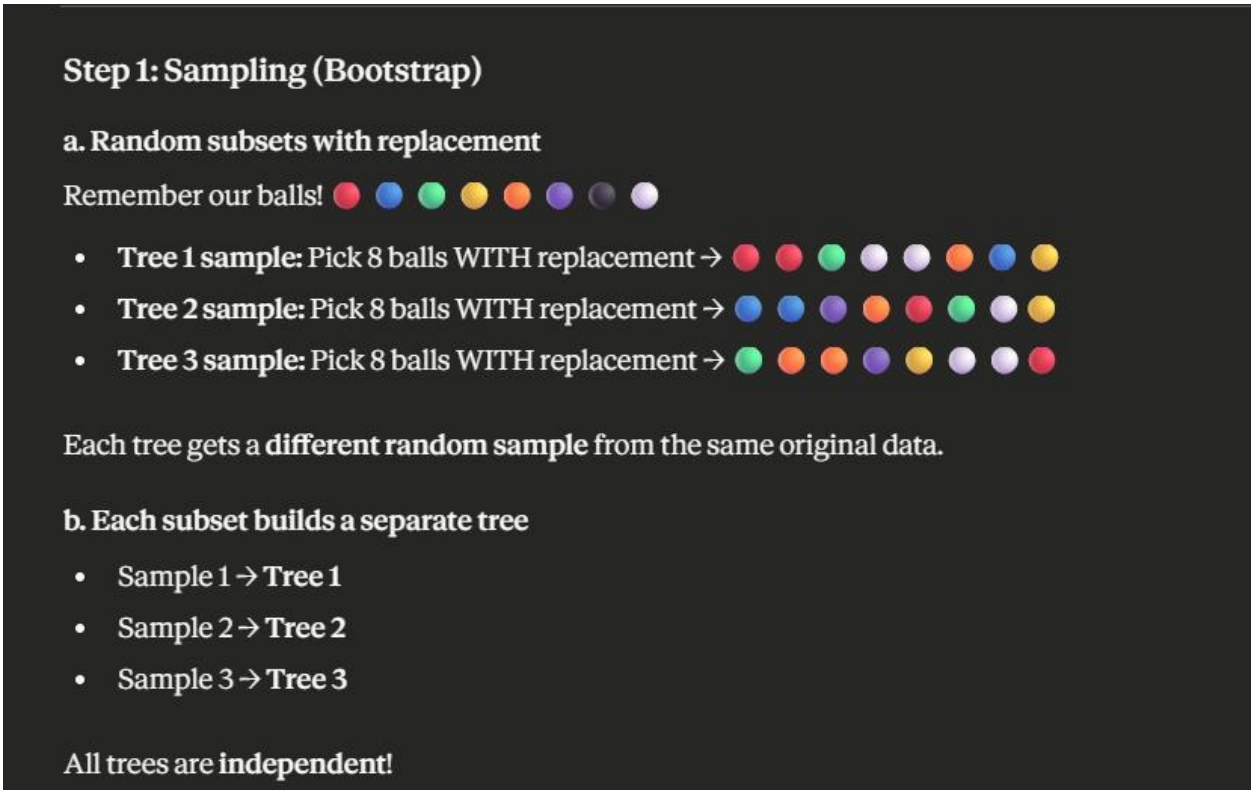
It's like a jury: One person might be biased, but the majority vote is usually more accurate.

- so basically we use with replacement to learn more patterns and diversity the algorithm

The Process

1. **Sampling:**
   a. Random subsets of the training data are sampled with replacement technique.
   b. Each subset is used to build a separate decision tree in the forest.
2. **Build Decision Trees:**
   a. Decision trees are constructed independently from each other for better diversity.
   b. Each tree is grown to the largest extent possible without applying pruning techniques.
3. **Random Feature Selection:**
   a. At each split in the tree, a random subset of features is considered for splitting.
   b. Helps in reducing correlation between the individual decision trees.
4. **Voting/Averaging:**
   a. For classification, each tree votes for a class, and the majority vote is taken as the final result.
   b. For regression, the average of outputs from all trees is calculated to give the final prediction.

- How our previous example is divided here:



Step 2: Build Decision Trees

a. Trees are independent (diversity!)

Each tree is built **separately** without knowing what other trees are doing.

Why? So they learn **different patterns**!

b. Grown to largest extent (no pruning)

Normal Decision Tree: Stop early or prune to avoid overfitting **Random Forest Tree:** Let it grow FULL size!

Why can we do this?

- Individual trees might **overfit** (memorize training data)
- But when you **combine many overfitted trees**, the overfitting **cancels out!**

Example:

- Tree 1 overfits pattern X
- Tree 2 overfits pattern Y
- Tree 3 overfits pattern Z
- **Average:** Balanced, no overfitting! ✅

Step 3: Random Feature Selection ⭐ (Extra Randomness!)

This is an **EXTRA layer** of diversity on top of bagging!

a. At each split, random subset of features

Let's say your dog data has these features:

- Ear Shape
- Round Face
- Whiskers
- Tail Length
- Fur Color

Normal Decision Tree:

- At each split, consider **ALL 5 features**
- Pick the best one

Random Forest:

- At each split, **randomly pick 2-3 features** (not all 5!)
- Pick the best from **only those random features**

Example at a Split:

Tree 1 at split node:

- Randomly selected features: {Ear Shape, Whiskers}
- Calculate IG for both
- Pick best between these 2

Tree 2 at the SAME position:

- Randomly selected features: {Round Face, Tail Length}
- Calculate IG for both
- Pick best between these 2

Result: Even if trees saw the same data, they'd make **different decisions**!

---

b. Reduces correlation between trees

Without random feature selection:

- All trees might pick "Ear Shape" first (if it has highest IG)
- Trees become **too similar** → less diversity

With random feature selection:

- Tree 1 might pick "Ear Shape" first
- Tree 2 might pick "Whiskers" first (because Ear Shape wasn't available)
- Tree 3 might pick "Round Face" first

More diversity → Better predictions! 🎯

Step 4: Voting/Averaging (Aggregation)

Now you have 100 trained trees. Time to make predictions!

a. Classification (Majority Vote)

New dog appears: 🐕

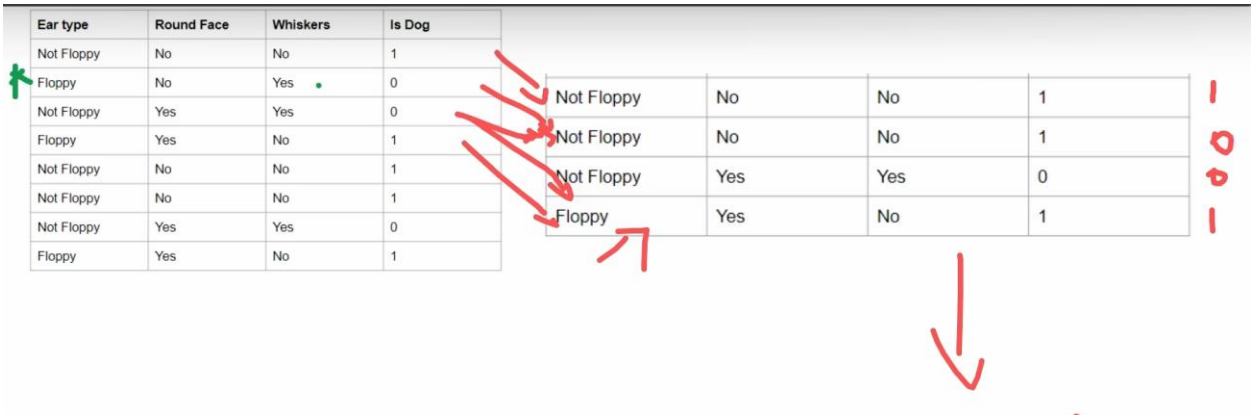
- Tree 1: "Dog" ✅
- Tree 2: "Dog" ✅
- Tree 3: "Not Dog" ❌
- Tree 4: "Dog" ✅
- ...
- Tree 100: "Dog" ✅

Count votes:

- "Dog": 85 votes
- "Not Dog": 15 votes

Final Prediction: Dog (majority wins!) 🐕

XGBOOST



Ear type	Round Face	Whiskers	Is Dog
Not Floppy	No	No	1
Floppy	No	Yes	0
Not Floppy	Yes	Yes	0
Floppy	Yes	No	1
Not Floppy	No	No	1
Not Floppy	No	No	1
Not Floppy	Yes	Yes	0
Floppy	Yes	No	1

Not Floppy	No	No	1
Not Floppy	No	No	1
Not Floppy	Yes	Yes	0
Floppy	Yes	No	1

1 0 0 1

- If the original tree didn't predict correctly like in the second one which is marked as 0. Therefore we assign a BIAS MARKER for the INCORRECT ONE!

- Which means every time the model starts to predict it is shown first or giving a preference to this record making more frequently sampled.
- Therefore, in the next time there is more probability of appearing many times.

XGBoost (eXtreme Gradient Boosting)

- Open Source implementation of Boosted Trees. (Other open-source ex: LightGBM, CatBoost)
- Fast to train and efficient implementation.
- Has great defaults and handling for default split criteria and splits stopping criteria.
- Built-in regularization methods as defaults.
- A very highly competitive and performant algorithm.

Using XGBoost

https://xgboost.readthedocs.io/en/stable/get_started.html#python

```python
from xgboost import XGBClassifier

model = XGBClassifier()

model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

```python
from xgboost import XGBRegressor

model = XGBRegressor()

model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

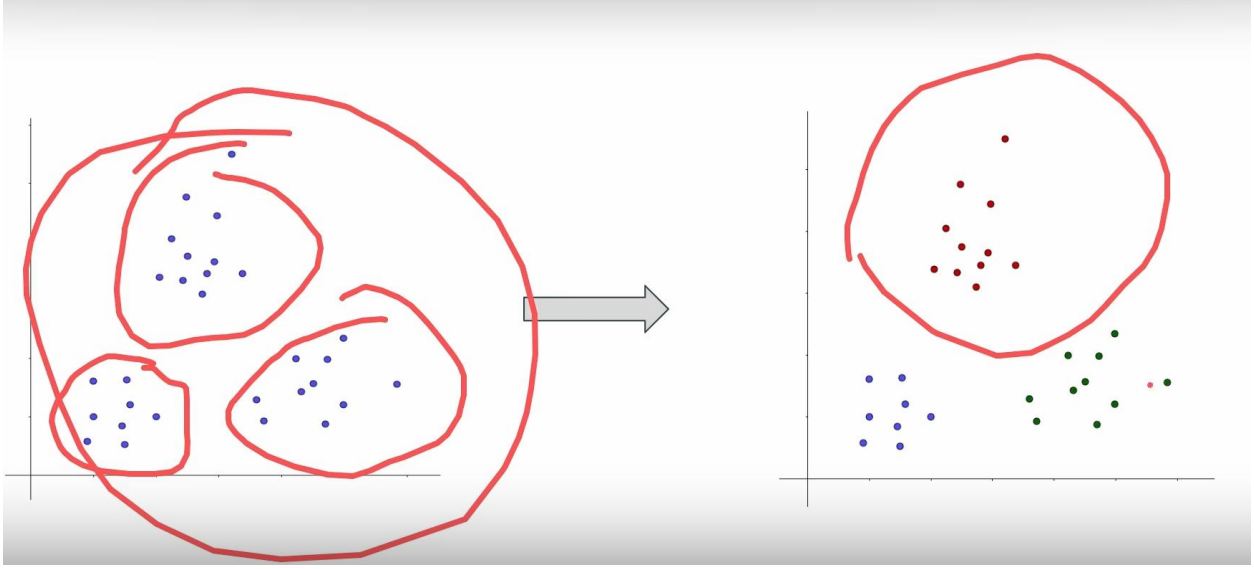
K-means Clustering

Clustering

- Grouping of data of similar points together based on their characteristics.

- Clustering is a machine learning technique used to group similar data points together based on certain characteristics.
- The objective is to partition the data into groups so that data points within the same group are more similar to each other than to those in other groups.
- Examples: Customer segmentation, image recognition, etc.

- Data is single group at beginning and the model or the algorithm does classify the points according to their characteristics.



K-means Clustering

- K-Means Clustering is an algorithm that divides data into K distinct clusters based on distance to the centroid of a cluster.
- It allows us to cluster the data into different groups and a convenient way to discover the categories of groups in the unlabeled dataset on its own without the need for any training.
- The algorithm takes the unlabeled dataset as input, divides the dataset into k-number of clusters, and repeats the process until it does not find the best clusters. The value of k should be predetermined in this algorithm.

In clustering, you have **NO labels:**

- Just data points: 
- You want to **discover natural groups**

---

2. K = Number of Clusters

You must decide K in advance!

Example: K = 3 means "divide data into 3 groups"

---

3. Centroid = Center of a Cluster

The **average position** of all points in that cluster.

Think of it as the "home base" for each group.

Main Purpose: To GROUP Similar Things TogetherThe centroid helps us decide which group each data point belongs to.

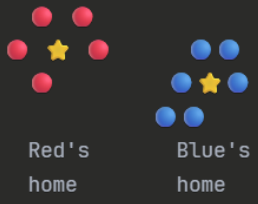
How It Works:The Rule:

"Each data point belongs to the group whose center (centroid) is CLOSEST to it"

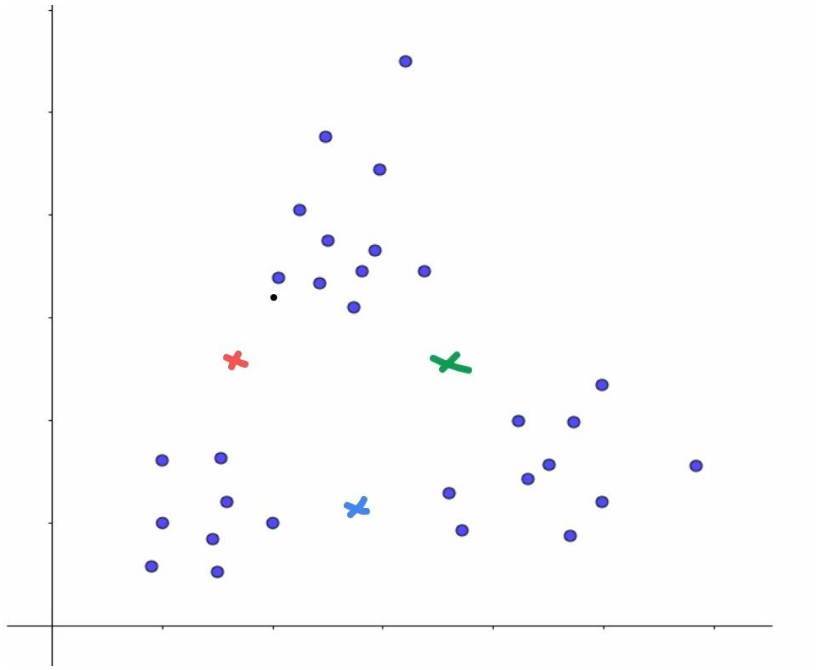
In K-Means Context:

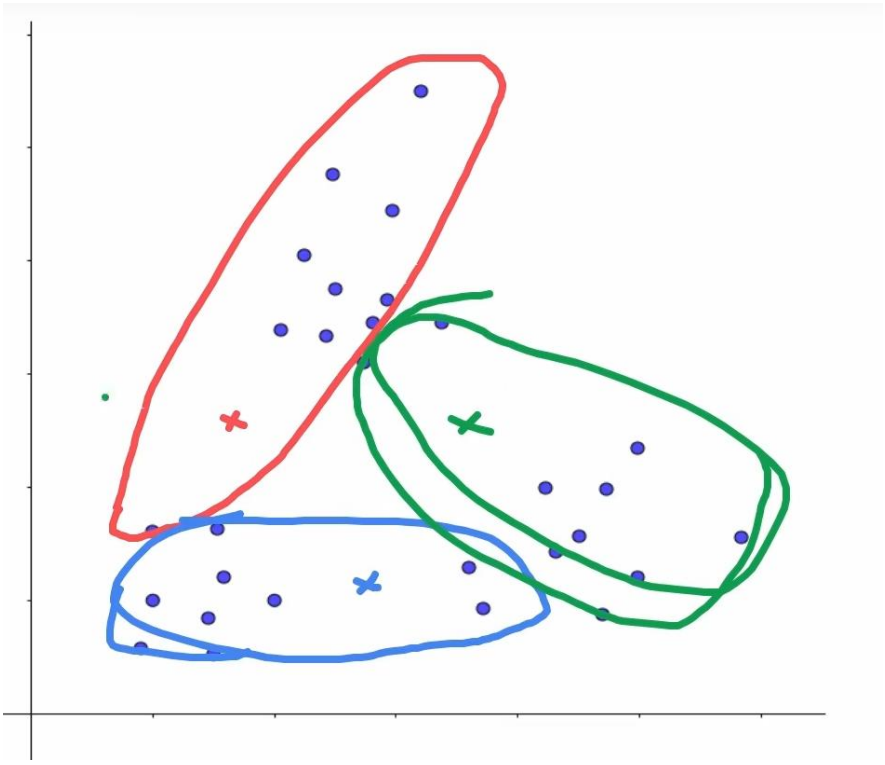
When you have data points scattered around:

The algorithm finds the **home base (centroid)** for each color group:

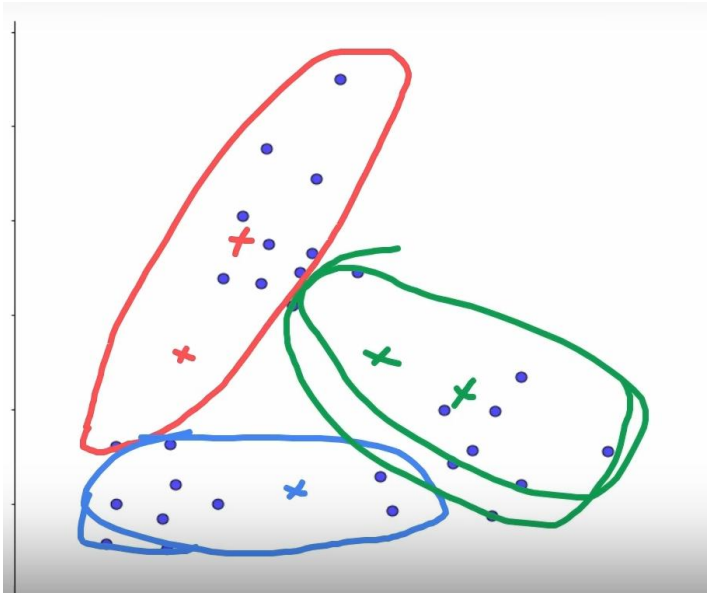


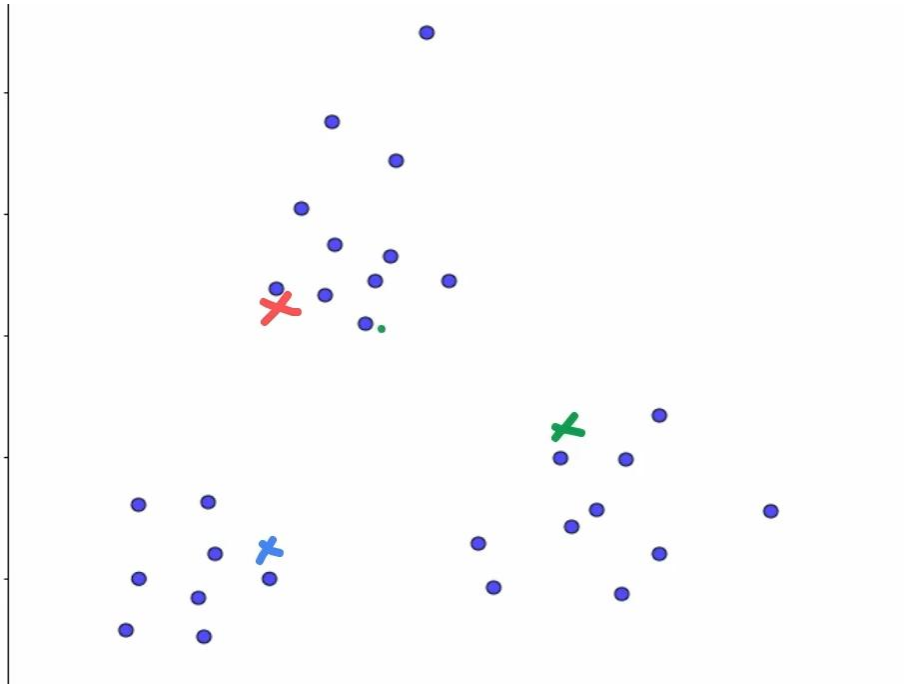
- **Basically to group u need a center!**
- Initially starts OFF WITH SOME RANDOM CENTERS..
- Example: K=3 meaning 3 centers.
- Therefore initially placing 3 random centers.





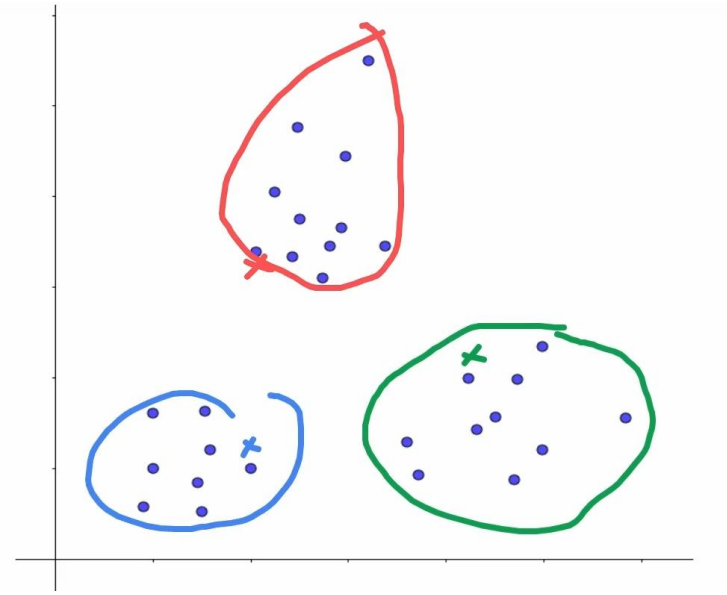
- It forms clusters then what it does next is that:
    - Take average of this clusters , so their lying centers change.



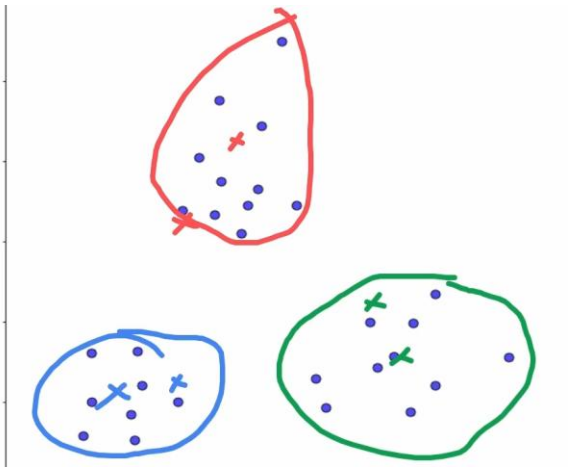


- Likewise it keep on iterating until final centers are formed.

CLUSTERING 2ND ATTEMPT

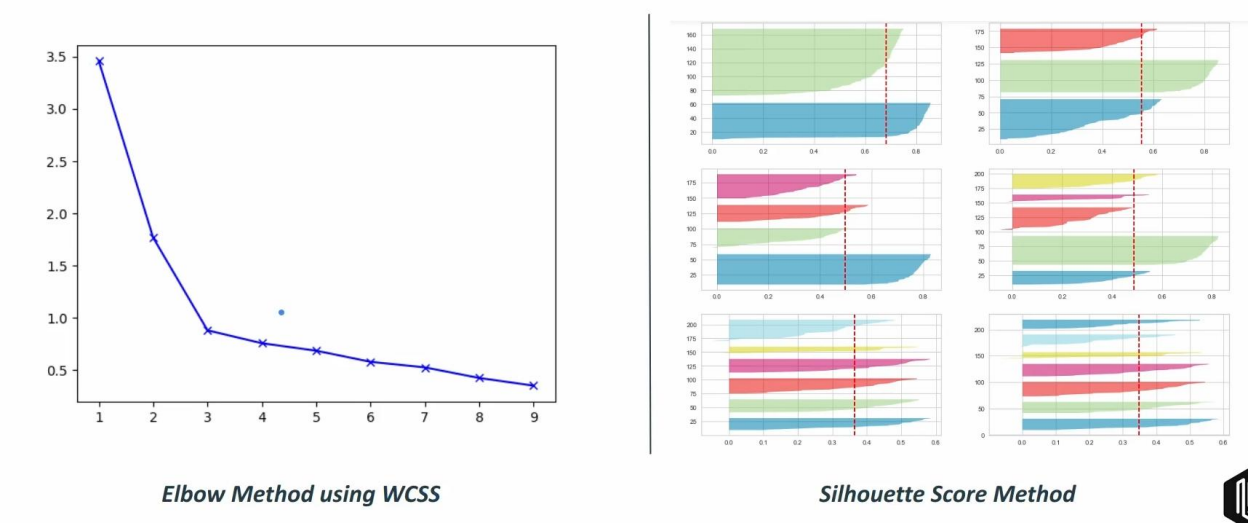


- GETTING AVGS AND MARKING NEW CENTERS.

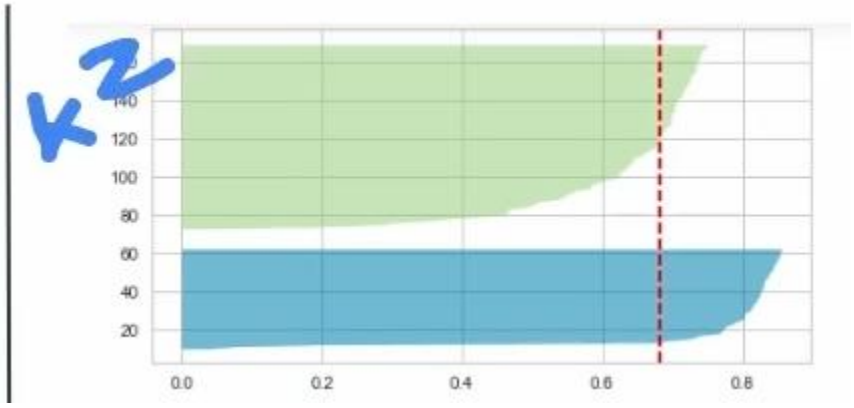


- Likewise it continues until find the best clusters.

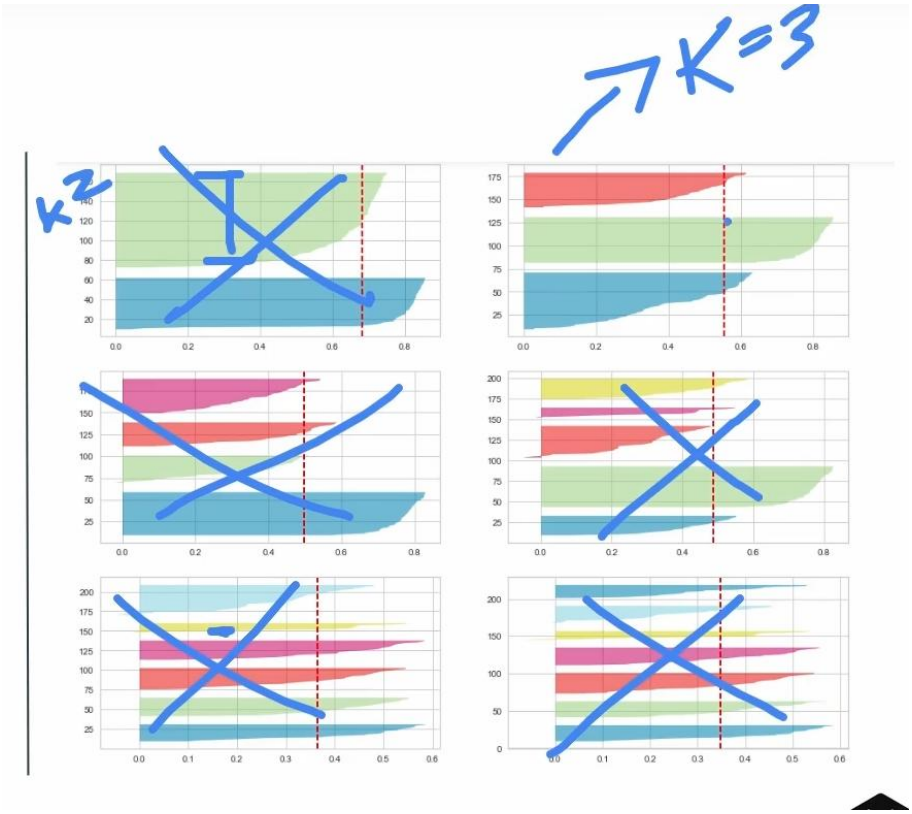
How to find/Select K?



- There are two methods
- (1) Elbow method:
    - The cutting point of the graph is best CLUSTER GROUP VALUE , LOOK AT 3.
- (2) Silhouette Score Method
    - First condition : All should pass the red color boundary
    - Number of patches = K value
    - Ex:



- The second condition: size of each cluster should be similar



Process and Quickstart

1. Select the number K to decide the number of clusters.
2. Select random K points or centroids.
3. Assign each data point to their closest centroid, which will form the predefined K clusters.
4. Calculate the variance and calculate a new centroid for each cluster.
5. Repeat step 3, which means re-assign each datapoint to the new closest centroid of each cluster.
6. If any reassignment occurs (a point is re-assigned to a new centroid), then go to 4 else the model is ready.

```
from sklearn.cluster import KMeans
import numpy as np

dataset = np.array([[1, 2], [1, 4], [1, 0],
                    [10, 2], [10, 4], [10, 0]])

kmeans = KMeans(n_clusters=2)
kmeans.fit(dataset)
```

Descriptive Analysis

Few examples of actions:

- Analyse summary stats for individual clusters.
- If any periodic feature available(date, time, etc), analyse for periodic relations in the clusters.
- Analyse what KPIs contribute towards a clear difference across the clusters.
- Identify features which are common across the clusters.

- Can note down the mean , median , mode of the columns for every cluster.

Descriptive Analysis - SUPER SIMPLE

After K-Means groups your data, you have clusters but **you don't know what they mean yet!**

Descriptive analysis = **Understanding your clusters**

Think of it like this:

You sorted your LEGO pieces into 3 piles (clusters):

```
Pile 1: 🟥🟥🟥🟥
Pile 2: 🟦🟦🟦🟦🟦
Pile 3: 🟩🟩
```

Now what? You need to **describe each pile** to understand them:

The 4 Ways to Describe Your Clusters:

---

1. Summary Stats = "What's in each pile?"

Look at each cluster and describe it:

Pile 1:

- Color: Red
- Average size: Big blocks
- Count: 4 pieces

Pile 2:

- Color: Blue
- Average size: Small blocks
- Count: 5 pieces

Pile 3:

- Color: Green
- Average size: Medium blocks
- Count: 2 pieces

Simple answer: "What does each group look like?"

2. Periodic Relations = "Do they change over time?"

Check if clusters behave differently at different times:

Example with customers:

Cluster 1: Shops a lot in December (holidays!) 🎄
Cluster 2: Shops the same all year round
Cluster 3: Shops more in summer 🌞

Simple answer: "When is each group most active?"

KPI = Key Performance Indicator (important metrics)

Purpose: Find what makes each cluster unique

3. KPI Differences = "What makes them different?"

Find the **biggest differences** between clusters:

Example:

Cluster	Spending	Age
1	$500 💰	35
2	$50	22
3	$200	45

Biggest difference: SPENDING! Cluster 1 spends 10x more than Cluster 2!

Simple answer: "What's the MAIN thing that separates these groups?"

ONE Complete Example:

Let's say you clustered dogs into 3 groups:

After K-Means:

```
Cluster 1: 🐕🐕🐕 (3 dogs)
Cluster 2: 🐕🐕🐕🐕🐕 (5 dogs)
Cluster 3: 🐕🐕 (2 dogs)
```

You have 3 groups but **don't know what they mean yet!**

Now do Descriptive Analysis:

1. Summary Stats:

Cluster 1:

- Average size: Large
- Average weight: 60 lbs
- Common breed: Labrador

Cluster 2:

- Average size: Small
- Average weight: 10 lbs
- Common breed: Chihuahua

Cluster 3:

- Average size: Medium
- Average weight: 35 lbs
- Common breed: Beagle

Now you know: "Cluster 1 = Big dogs, Cluster 2 = Small dogs, Cluster 3 = Medium dogs"

2. Periodic Patterns:

Cluster 1 (Big dogs):

- Play more in morning 🌅
- Sleep in afternoon 😴

Cluster 2 (Small dogs):

- Active all day long! ⚡

Cluster 3 (Medium dogs):

- Play more in evening 🌙

Now you know: "When each type of dog is most active"

3. KPI Differences:

Cluster	Food eaten	Exercise time	Barking
1 (Big)	4 cups 🥣🥣🥣🥣	2 hours	Low
2 (Small)	1 cup 🥣	30 mins	High
3 (Medium)	2 cups 🥣🥣	1 hour	Medium

Now you know: "Food intake is the biggest difference! Big dogs eat 4x more than small dogs"

Now you know: "Food intake is the biggest difference! Big dogs eat 4x more than small dogs"

4. Common Features:

All 3 clusters:

- ✅ All love treats
- ✅ All need daily walks
- ✅ All like belly rubs
- ✅ All respond to training

Now you know: "What to focus on for ALL dogs, regardless of size"

Why Do This?

Before descriptive analysis:

- "I have 3 clusters... so what? 🤷"

After descriptive analysis:

- "Oh! Cluster 1 is big dogs that eat a lot and play in mornings!"
- "Cluster 2 is small dogs that bark a lot and are always active!"
- "Cluster 3 is medium dogs that play in evenings!"

Now you can:

- Name your clusters (Big Dogs, Small Dogs, Medium Dogs)
- Understand each group
- Make decisions (buy more food for Cluster 1!)

Super Simple Summary:

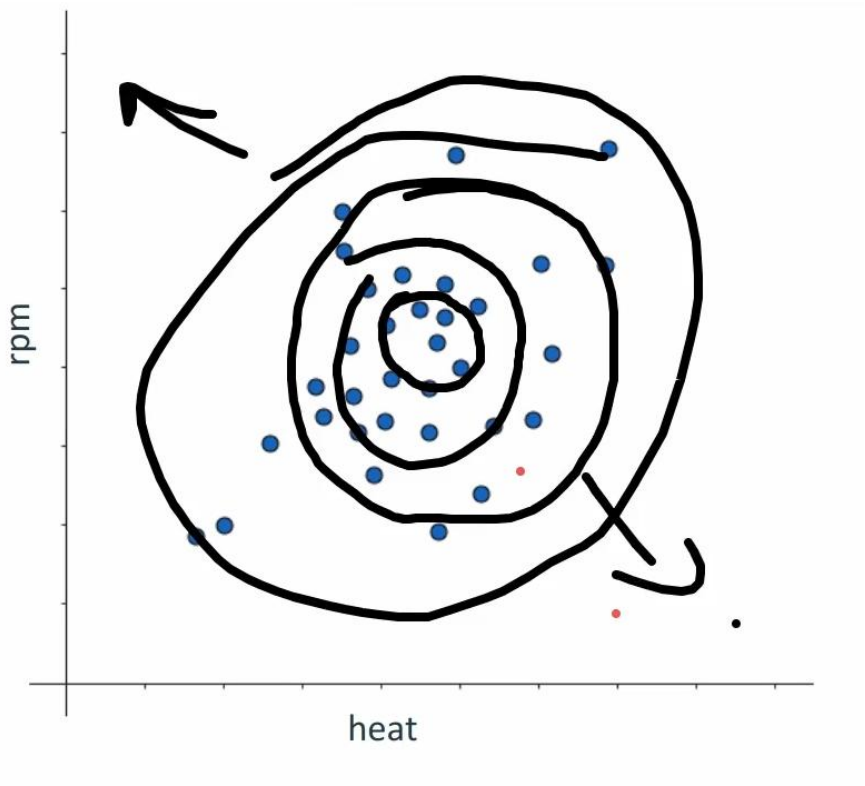
After K-Means groups your data:

Descriptive Analysis answers these questions:

1. **Summary:** What does each group look like? 🔍
2. **Periodic:** When do they behave differently? ⏰
3. **KPIs:** What's the BIGGEST difference? 📊
4. **Common:** What do they ALL share? 🤝

That's it! You're just **describing and understanding** the clusters you created!

Anomaly Detection



- The confidence of the data point validity decreases with distance from the center
- If new point is on center -> valid
- If bit away -> yes
- If kinda far but within -> kind of
- If away f rom all of the circles - > anomaly.

The Split:

Training Set: 6,000 Good engines ONLY

Important: Training uses **ONLY good engines!**

Why? Because in anomaly detection, you want the model to **learn what "normal" looks like.**

Think of it like teaching someone what a healthy apple looks like - you only show them good apples! 🍎

Validation Set: 2,000 Good + 15 Anomalous

This is used to:

- Tune your model
- Check if it can detect anomalies
- Adjust thresholds

Mix of both good and bad so you can see: "Is my model correctly identifying the bad ones?"

Test Set: 2,000 Good + 15 Anomalous

Final evaluation:

- How well does the model perform on completely unseen data?
- Can it catch the bad engines?
- What's the false alarm rate?

Why Split Anomalies 50-50?

30 total bad engines:

- 15 → Validation
- 15 → Test
- 0 → Training ❌

Since there are so few anomalies (only 30!), you split them equally between validation and test to have enough examples in each set to evaluate properly.

Real-World Analogy:

Imagine you're a security guard learning to spot suspicious behavior:

Training:

- Watch 6,000 normal people walking by
- Learn: "This is how normal people behave"

Validation:

- Watch 2,000 normal + 15 suspicious people
- Practice: "Can I spot the suspicious ones?"
- Adjust your detection skills

Test:

- Watch 2,000 normal + 15 NEW suspicious people
- Final check: "How good am I at catching bad actors?"

Key Takeaway:

Anomaly detection is different from normal classification:

Normal classification:

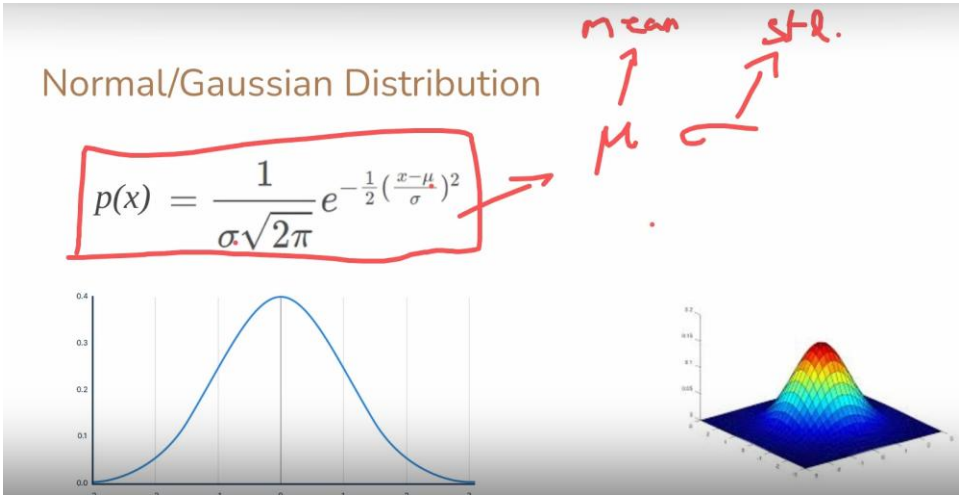
- Train on BOTH classes (dogs AND cats)

Anomaly detection:

- Train on ONLY normal class (good engines)
- Validate/Test on BOTH (to check detection ability)

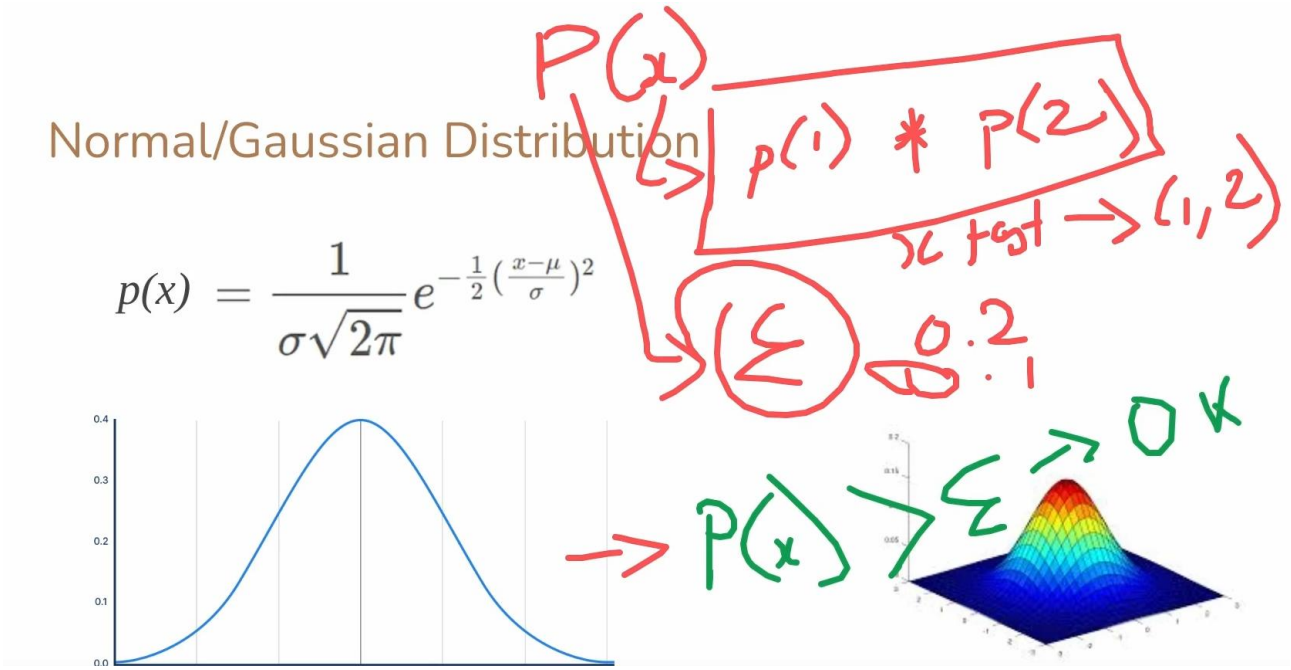
- We use anomalies for validation because we want to makesure the model does know how to find the anomalies because we only train using the good engines or good data.

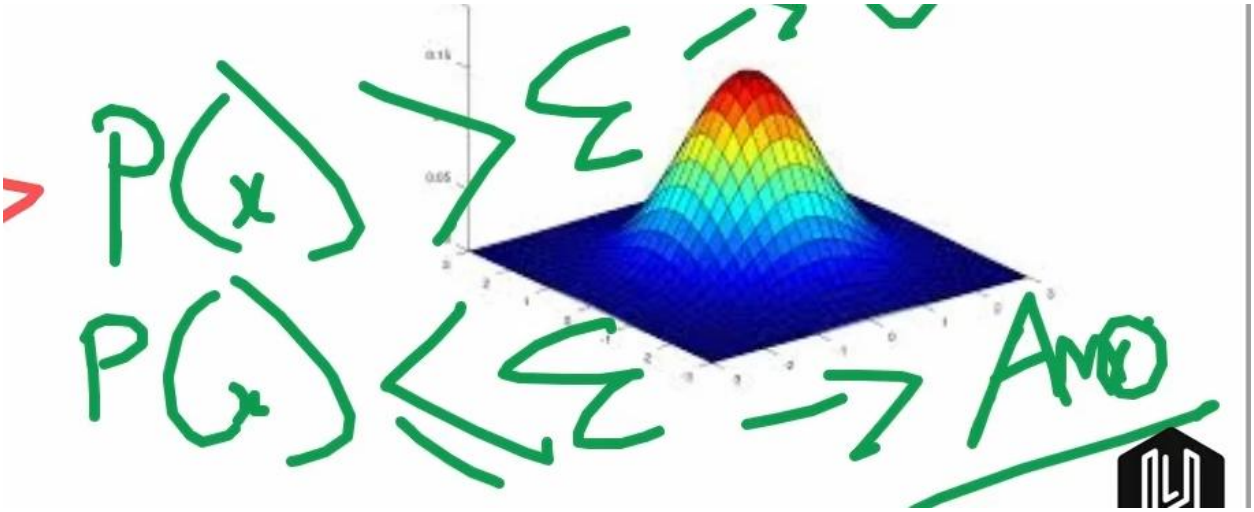
Normal/Gaussian Distribution

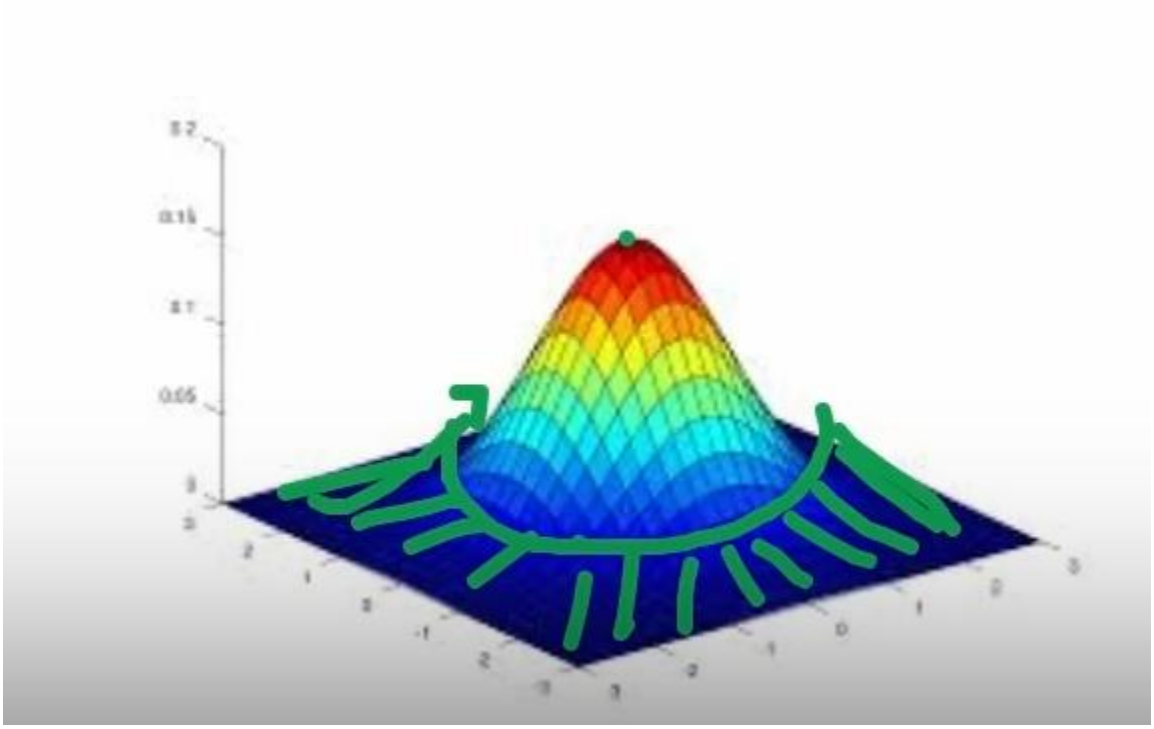


The above is probability , the below is different one

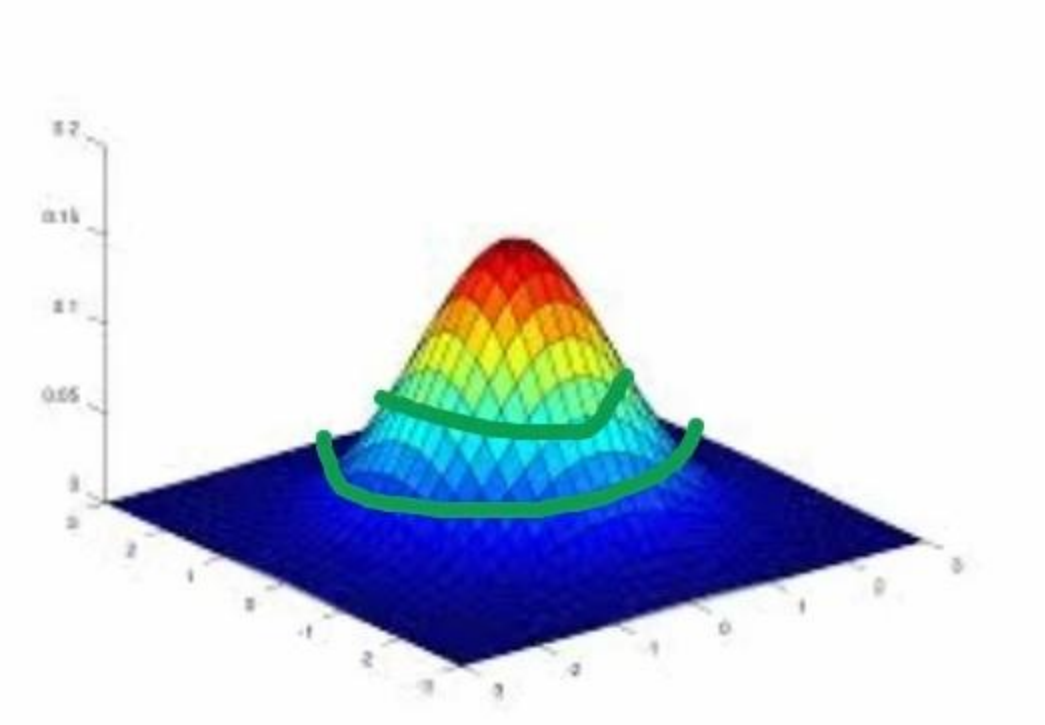
- So basically take an example where we have a model P(X) = p(1) * p(2)
- And we have X_TEST = (1,2)
- Therefore we calculate the model value
- However , we cant interpret , so to interpret the model we use a threshold which is ε







- As the curve climbs or moves up, the chances of anomalies reduces and top is when anomaly is 0.



- The boundary is ε **, as you can see it might varies , if tigher value then bar might raise. Likewise. However only 1 line.**

Anomaly Detection	Supervised Learning
• Very few positive/target examples while a large number of negative examples.	• Large number of positive/target examples while a relatively smaller number of negative examples.
• Anomalies could be of various "Type", making it significantly hard for a model to learn from positive examples.	• Enough positive examples are available for the model to learn what is it looking for.
• There is a chance that the properties of anomalies may change in the future.	• The target properties will mostly likely remain the same in the future as well.
• Ex: Fraud Detection, Manufacturing, Monitoring, etc.	• Ex: Email spam classification, weather prediction, etc.