DEEP LEARNING

Let’s start our discussion with basics of deep learning
Following are basic which we will consider to understand each neuron

Disclaimer: Content of this post and many to come are from different courses & books, I will mention references in last
DATA
Data can be of different type numeric, text, structured, unstructured so it various according to the problem with which we are dealing with.
It is very rare to find data readily available in real world, data is always present in unstructured and different way which we need to collect according to our requirement and structure them.

TASK
First of all we need to see what kind of TASK we wanted to do, there various task which can be performed using machine learning & deep learning.
It is just a method where we are come up with a set of inputs and outputs based on data
Few examples of defining or choosing a task are:

- Auto face tagging in Facebook photos.
- Product suggestions in e-commerce websites while placing orders.
- Using face detection for attendance.

MODEL
Using Neural networks come up with some simple or complex which actually maps our input to its corresponding output
A Model is our approximation of relationship between X & Y
There are some tools we can use for model predictions such as:

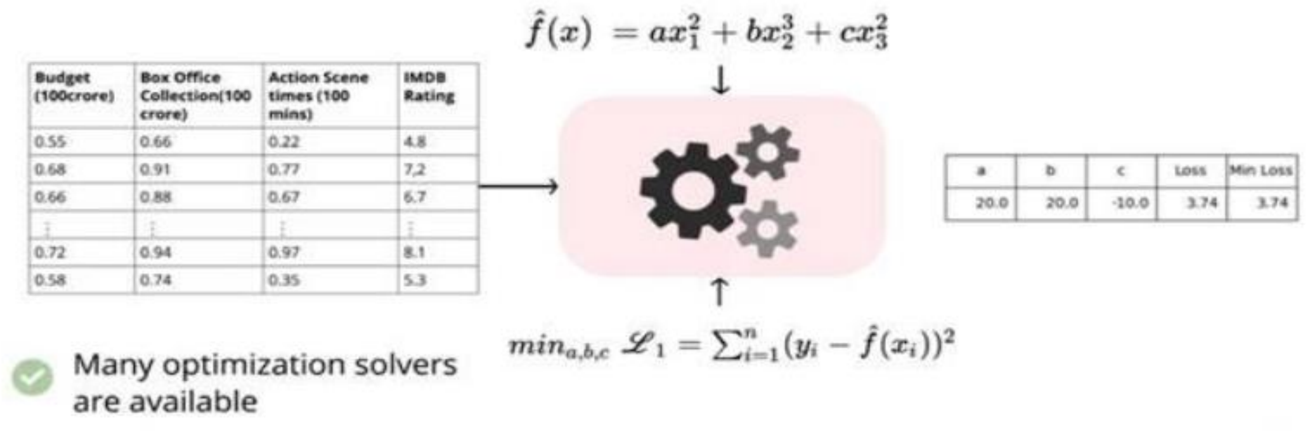
- **Bias-Variance Tradeoff.**
- **Overfitting.**
- **Regularization.**

LOSS
How do we know the model predicted out is close to actual output or not, we need to use Loss function to choose best approximate function from a set of models or functions
Loss function helps machine to estimate the correct set of parameters that ultimately gives out lowest/least error.

LEARNING

Learning answers our question → How do we identify parameters of the models?

the learning algorithm and loss function go hand in hand. This is more like an optimization problem, where we try to predict values of parameters in our model via optimizing our loss function to minimum



$$\hat{f}(x) = ax_1^2 + bx_2^3 + cx_3^2$$

Budget (100crore)	Box Office Collection(100 crore)	Action Scene times (100 mins)	IMDB Rating
0.55	0.66	0.22	4.8
0.68	0.91	0.77	7.2
0.66	0.88	0.67	6.7
⋮	⋮	⋮	⋮
0.72	0.94	0.97	8.1
0.58	0.74	0.35	5.3

a	b	c	Loss	Min Loss
20.0	20.0	-10.0	3.74	3.74

$$min_{a,b,c}\ \mathscr{L}_1 = \sum_{i=1}^{n}(y_i - \hat{f}(x_i))^2$$

Many optimization solvers are available

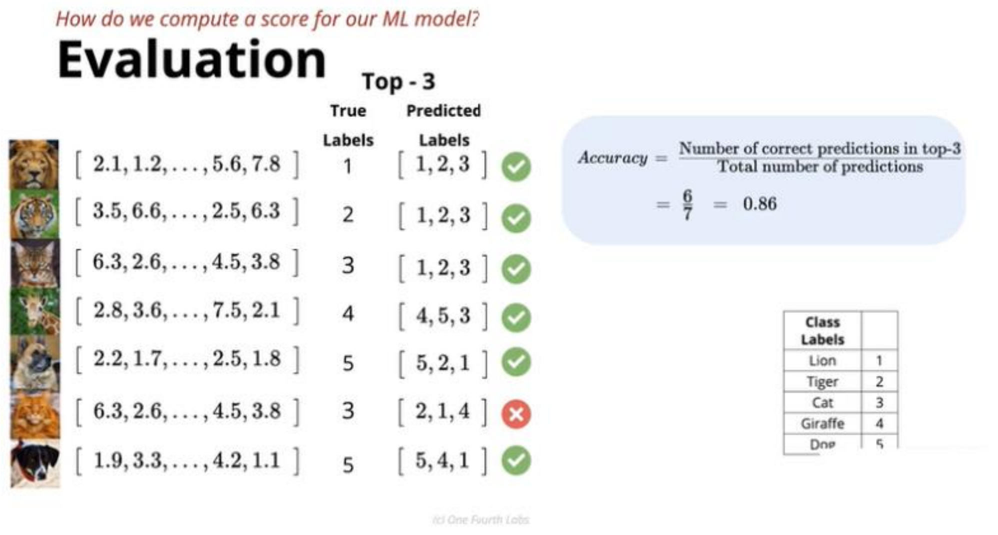
Types of Learning Algorithm / Optimization solvers:

- **Gradient Descent**
- **RMSprop**
- **Backpropagation**
- **Adagrad**
- **and many others…**

EVALUATION

How do we compute a score for our ML models?

Here to understand below example consider we are doing google search, so at least top-3 or top-5 search must be relevant, so if we check above example we can see predicted label and if top-3 output have actual labels then it is considered as accurate predicted output.



How do we compute a score for our ML model?

Evaluation

Top - 3

	True Labels	Predicted Labels	
$[2.1, 1.2, \ldots, 5.6, 7.8]$	1	$[1, 2, 3]$	✓
$[3.5, 6.6, \ldots, 2.5, 6.3]$	2	$[1, 2, 3]$	✓
$[6.3, 2.6, \ldots, 4.5, 3.8]$	3	$[1, 2, 3]$	✓
$[2.8, 3.6, \ldots, 7.5, 2.1]$	4	$[4, 5, 3]$	✓
$[2.2, 1.7, \ldots, 2.5, 1.8]$	5	$[5, 2, 1]$	✓
$[6.3, 2.6, \ldots, 4.5, 3.8]$	3	$[2, 1, 4]$	✗
$[1.9, 3.3, \ldots, 4.2, 1.1]$	5	$[5, 4, 1]$	✓

$$Accuracy = \frac{\text{Number of correct predictions in top-3}}{\text{Total number of predictions}} = \frac{6}{7} = 0.86$$

Class Labels	
Lion	1
Tiger	2
Cat	3
Giraffe	4
Dog	5

(c) One Fourth Labs

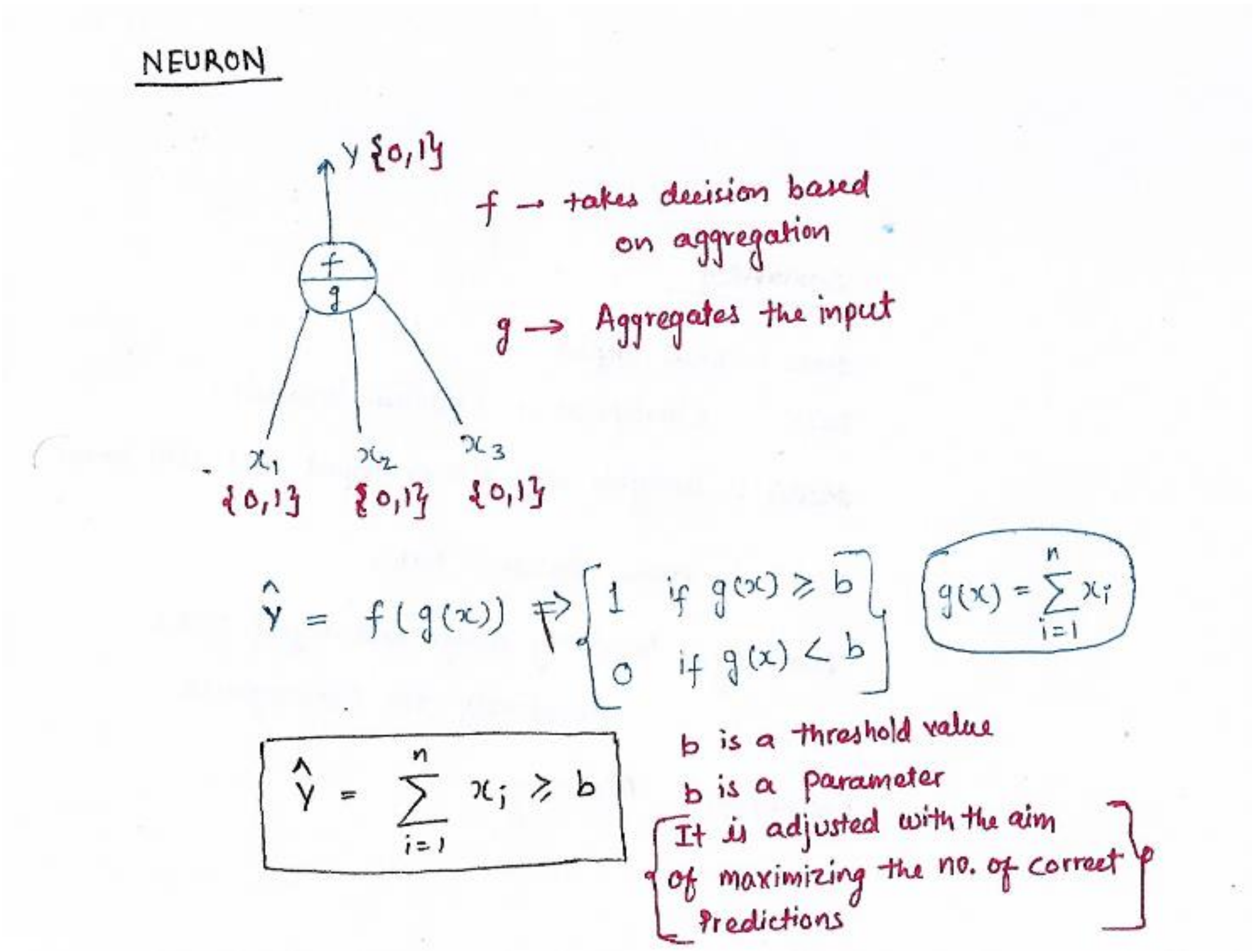
There are different evaluation metrices

- → ACCURACY
- → PRECISION
- → RECALL

BASIC BUILDING BLOCKS

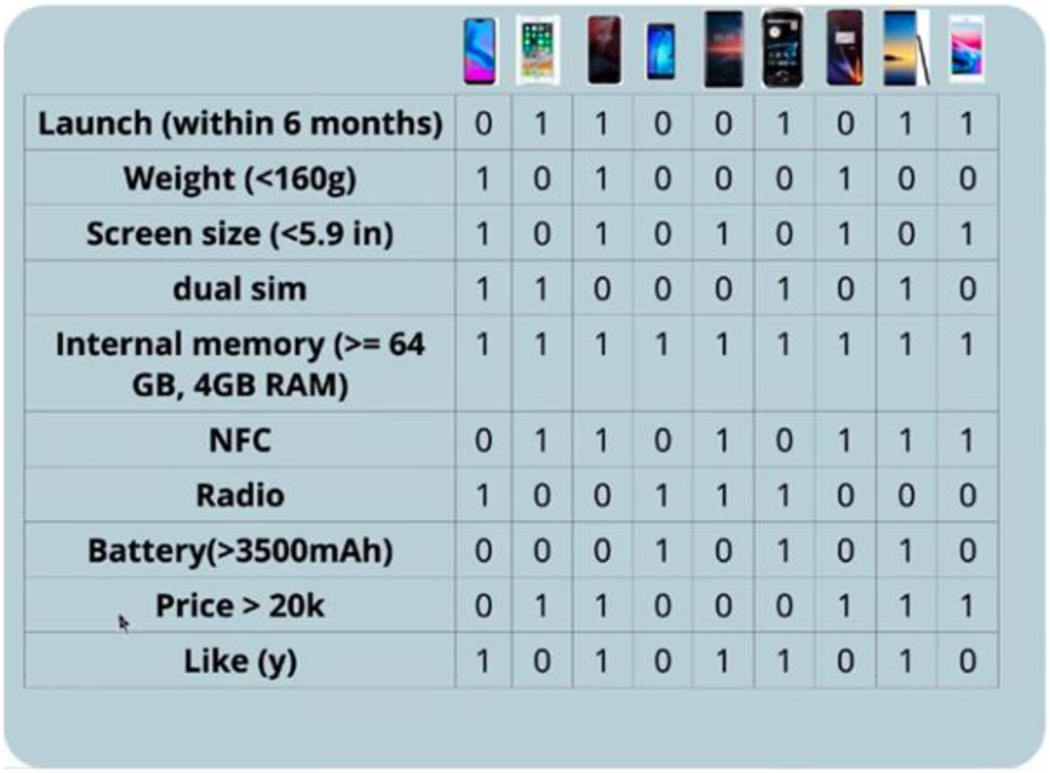
1) NEURON

The fundamental block of deep learning is artificial neuron i.e. it takes a weighted aggregate of inputs, applies a function and gives an output. The very first step towards the artificial neuron was taken by Warren McCulloch and Walter Pitts in 1943 inspired by neurobiology, created a model known as McCulloch-Pitts Neuron.
Disclaimer: The some of the content and the structure of this article is based on the deep learning lectures from One-Fourth Labs — Padhai.


NEURON
y {0,1}
f
g
f → takes decision based on aggregation
g → Aggregates the input
x1 {0,1}
x2 {0,1}
x3 {0,1}
ŷ = f(g(x)) ⇒ 1 if g(x) ≥ b; 0 if g(x) < b
g(x) = Σ_{i=1}^{n} x_i
ŷ = Σ_{i=1}^{n} x_i ≥ b
b is a threshold value
b is a parameter
It is adjusted with the aim of maximizing the no. of correct predictions

The function is actually split into two parts:
g — The aggregates the inputs to a single numeric value
f -- The function **f** produces the output of this neuron by taking the output of the **g** as the input i,e.. a single value as its argument. The function **f** will output the value 1 if the aggregation performed by the function **g** is greater than some threshold else it will return 0.
The inputs x1, x2, ….. xn for the MP Neuron can only take boolean values and the inputs can be inhibitory or excitatory. Inhibitory inputs can have maximum effect on the decision-making process of the model. In some cases, inhibitory inputs can influence the final outcome of the model.

Now to understand further consider this example



Launch (within 6 months)	0	1	1	0	0	1	0	1	1
Weight (<160g)	1	0	1	0	0	0	1	0	0
Screen size (<5.9 in)	1	0	1	0	1	0	1	0	1
dual sim	1	1	0	0	0	1	0	1	0
Internal memory (>= 64 GB, 4GB RAM)	1	1	1	1	1	1	1	1	1
NFC	0	1	1	0	1	0	1	1	1
Radio	1	0	0	1	1	1	0	0	0
Battery(>3500mAh)	0	0	0	1	0	1	0	1	0
Price > 20k	0	1	1	0	0	0	1	1	1
Like (y)	1	0	1	0	1	1	0	1	0

Let's take an example of buying a phone based on some features of the features in the binary format. { y — 0: Not buying a phone and y — 1: buying a phone}

For each particular phone (observation) with a certain threshold value **b**, using the MP Neuron Model, we can predict the outcome using a condition that the summation of the inputs is greater than b then the predicted value will be 1 or else it will be 0. The loss for the particular observation will be squared difference between the Yactual and Ypredicted.

Similarly, for all the observations, calculate the summation of the squared difference between the Yactual and Ypredicted to get the total loss of the model for a particular threshold value **b**.

$$\hat{y} = \sum_{i=1}^{n} x_i \geq b$$

Now once we have the predicted values we can compute LOSS FUNCTION for neuron

MP NEURON LOSS

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

LEARNING in neuron

Now we have the output calculating function to predict the output, loss function and we need the b value that's the bias how will we calculate that.
According to a method we take b weight from 1 to 10 and taking each value at a time.

The purpose of the learning algorithm is to find out the best value for the parameter **b** so that the loss of the model will be minimum. In the ideal scenario, the loss of the model for the best value of **b** would be zero.

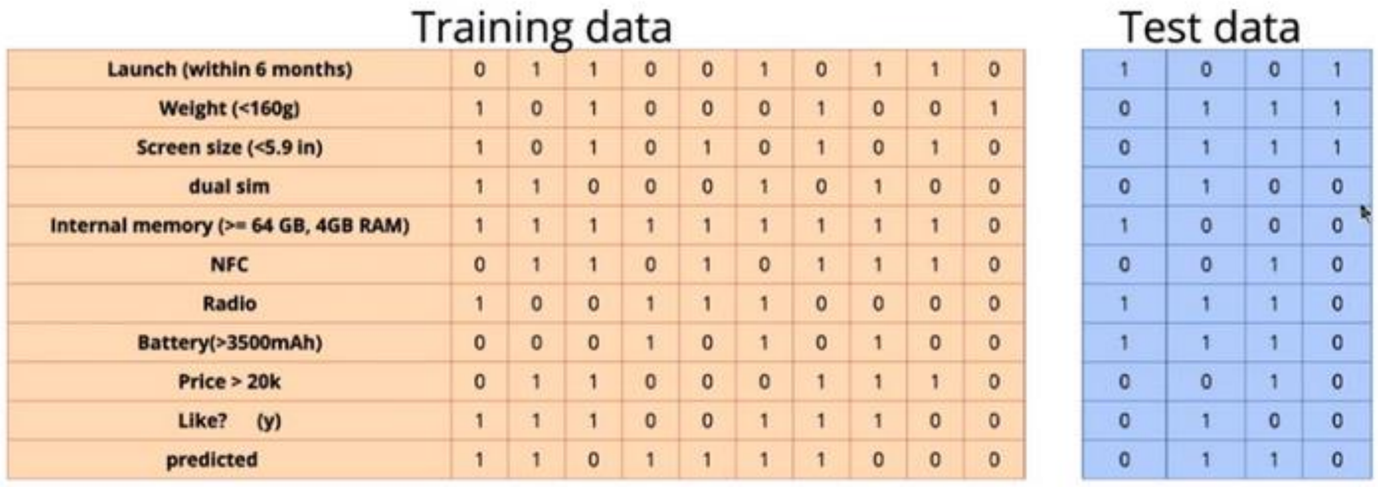
In this case, we have only one parameter, so we can afford to use brute force search.
Here, consider we have n features
b can only range from 0 to n, else it would be a pointless parameter
b has discrete values only, as the inputs are also discrete values
As we have only one parameter with a range of values 0 to n, we can use the brute force approach to find the best value of b.

- **Initialize the b with a random integer [0,n]**
- **For each observation**
- **Find the predicted outcome as we explained before**

Calculate the summation of inputs and check whether its greater than or equal to b. If its greater than or equal to b, then the predicted outcome will be 1 or else it will be 0.

- After finding the predicting outcome compute the loss for each observation.
- Finally, compute the total loss of the model by summing up all the individual losses.
- Similarly, we can iterate over all the possible values of b and find the total loss of the model. Then we can choose the value of b, such that the loss is minimum.

MODEL EVALUATION
After finding the best threshold value **b** from the learning algorithm, we can evaluate the model on the test data by comparing the predicted outcome and the actual outcome.



Training data

Launch (within 6 months)	0	1	1	0	0	1	0	1	1	0
Weight (<160g)	1	0	1	0	0	0	1	0	0	1
Screen size (<5.9 in)	1	0	1	0	1	0	1	0	1	0
dual sim	1	1	0	0	0	1	0	1	0	0
Internal memory (>= 64 GB, 4GB RAM)	1	1	1	1	1	1	1	1	1	0
NFC	0	1	1	0	1	0	1	1	1	0
Radio	1	0	0	1	1	1	0	0	0	0
Battery(>3500mAh)	0	0	0	1	0	1	0	1	0	0
Price > 20k	0	1	1	0	0	0	1	1	1	0
Like? (y)	1	1	1	0	0	1	1	1	0	0
predicted	1	1	0	1	1	1	1	0	0	0

Test data

1	0	0	1
0	1	1	1
0	1	1	1
0	1	0	0
1	0	0	0
0	0	1	0
1	1	1	0
1	1	1	0
0	0	1	0
0	1	0	0
0	1	1	0

MP NEURON EVALUATION

$$\boxed{\text{Accuracy} = \frac{\text{No. of Correct Predictions}}{\text{Total No. of predictions}}}$$

SUMMARY

DATA: All Boolean inputs
TASK: Binary classification (Boolean output)
MODEL: Linear Decision boundary

- ➔ All +ve points lie above the line
- ➔ All -ve points lie below the line

LOSS: Mean Squared Error
LEARNING: Brute Force approach to learn best parameter b
EVALUATION: Accuracy

DISADVANTAGES

- ➔ Linear, we cannot find any degree of polynomial which is more efficient
- ➔ Fixed slope
- ➔ Few possible intercept (b’s)
- ➔ Only for Boolean input
- ➔ Boolean Output

2) PERCEPTRON

An upgrade to McCulloch-Pitts Neuron
Perceptron is a fundamental unit of the neural network which **takes weighted inputs**, process it and capable of performing binary classifications.
Comparing with neuron

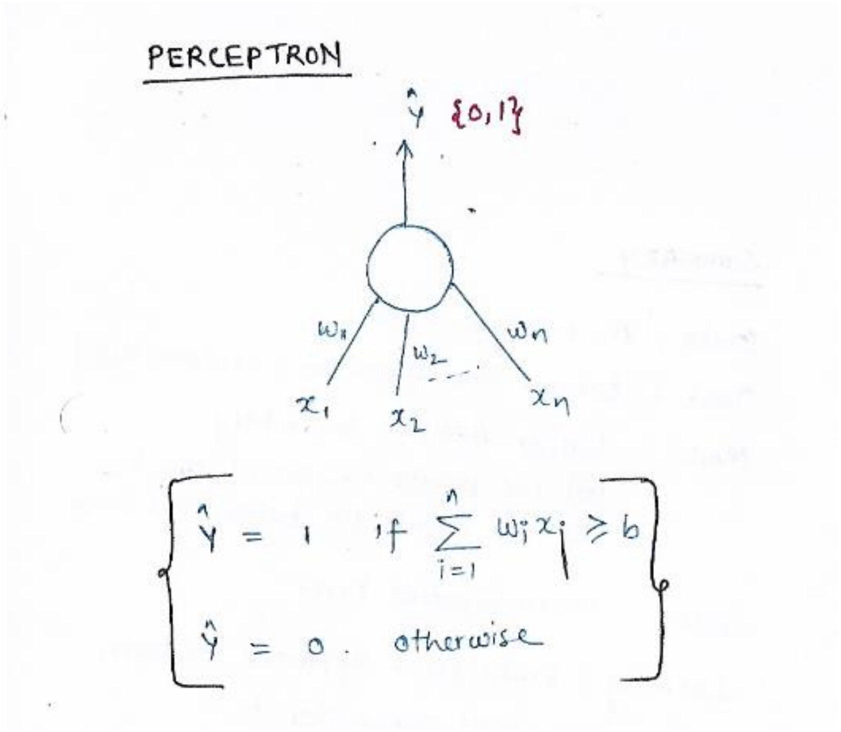
MP Neuron	Perceptron
$\hat{y}$ = 1 if $\Sigma^{n}_{i=1}x_i$ >= b $\hat{y}$ = 0 otherwise	$\hat{y}$ = 1 if $\Sigma^{n}_{i=1}w_ix_i$ >= b $\hat{y}$ = 0 otherwise
Boolean inputs ☹	Real inputs 😃
Linear ☹	Linear ☹
Inputs are not weighted ☹	Weights for each input 😃
Adjustable threshold 😃	Adjustable threshold 😃

- **In the MP Neuron Model, all the inputs have the same weight (same importance)** while calculating the outcome and the parameter **b** can only take fewer values i.e., the parameter space for finding the best parameter is limited.
- In perceptron model inputs can be real numbers unlike the boolean inputs in MP Neuron Model. The output from the model still is boolean outputs {0,1}.
- Mathematical equation

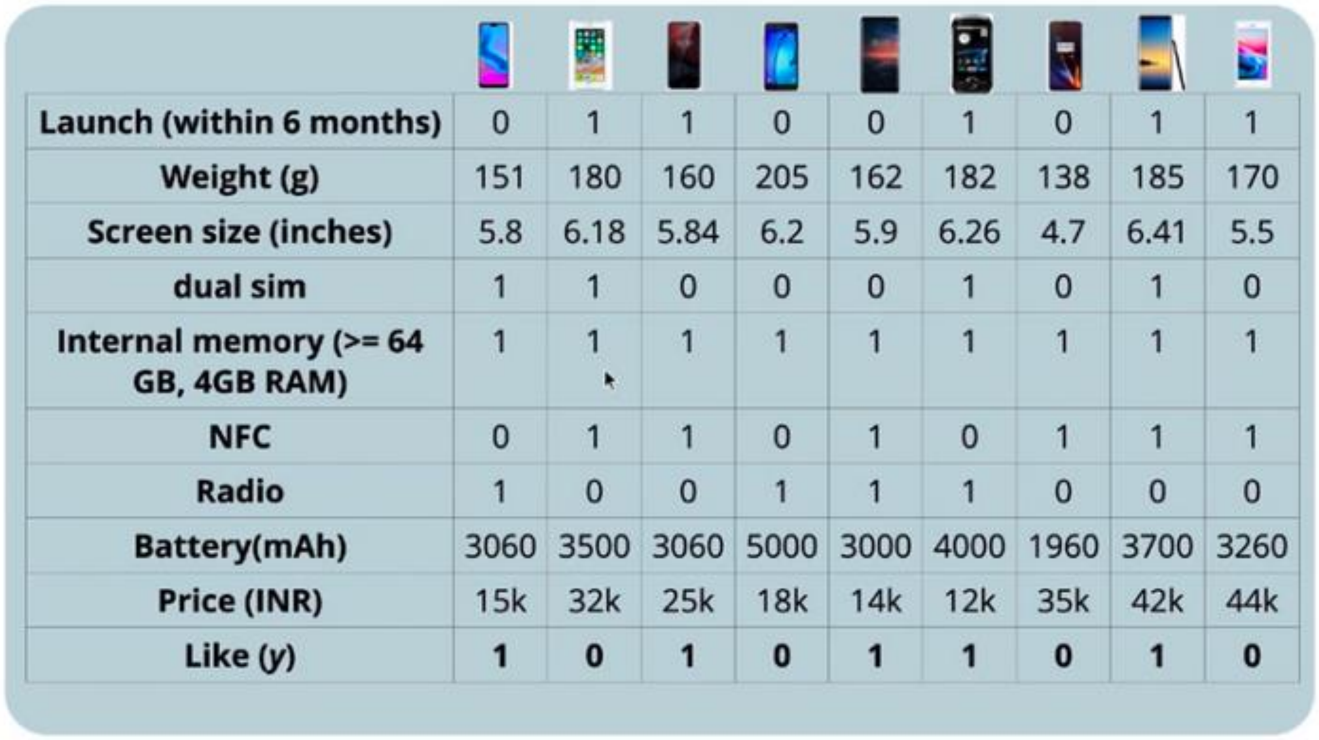
$$\hat{y} = 1 \text{ if } \textstyle\sum_{i=1}^{n} w_i x_i \geq b$$

$$\hat{y} = 0 \text{ otherwise}$$

- The function here has two parameters weights **w**(w1,w2,....,wn) and threshold **b**. The mathematical representation of Perceptron looks an equation of a line (2D) or a plane(3D).


PERCEPTRON
$\hat{y}$ {0,1}
w_1
w_2
w_n
x_1
x_2
x_n
$\hat{y} = 1$ if $\sum_{i=1}^{n} w_i x_i \geq b$
$\hat{y} = 0$. otherwise

Now to understand further consider this example

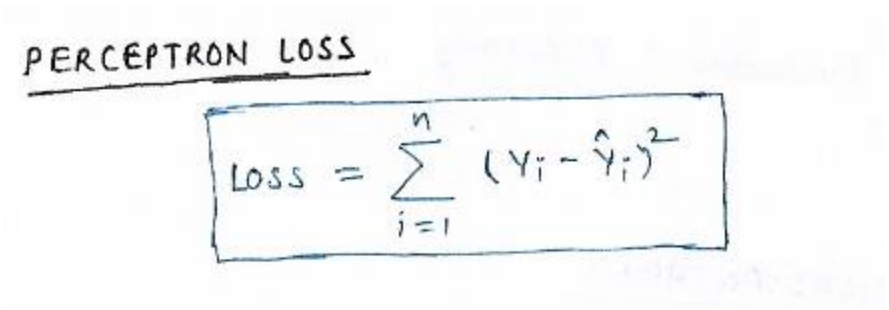


Launch (within 6 months)	0	1	1	0	0	1	0	1	1
Weight (g)	151	180	160	205	162	182	138	185	170
Screen size (inches)	5.8	6.18	5.84	6.2	5.9	6.26	4.7	6.41	5.5
dual sim	1	1	0	0	0	1	0	1	0
Internal memory (>= 64 GB, 4GB RAM)	1	1	1	1	1	1	1	1	1
NFC	0	1	1	0	1	0	1	1	1
Radio	1	0	0	1	1	1	0	0	0
Battery(mAh)	3060	3500	3060	5000	3000	4000	1960	3700	3260
Price (INR)	15k	32k	25k	18k	14k	12k	35k	42k	44k
Like (*y*)	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**0**

DATA: → The inputs to the Perceptron model can be real numbers, and for real numbers data can be having different units for example in above example weight is in grams, screen size is in inches so to maintain the same scale we will perform min-max standardization to bring all values between 0-1

Now we will predict the output using equation which we have explained above

LOSS: → Once we have the predicted output available, we can calculate loss using below loss function



PERCEPTRON LOSS

$$\boxed{\text{LOSS} = \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2}$$

More often, it is represented using an indicator variable
L = 1(y ≠ ŷ)
Or L = 0(y = ŷ)
The Perceptron loss is almost identical to the square error loss function.

PERCEPTRON LEARNING ALGORITHM

Let’s discuss first our equation for perceptron

$$\left\{\begin{array}{ll} \hat{y} = 1 & \text{if } \sum_{i=1}^{n} w_i x_i \geq b \\ \hat{y} = 0 . & \text{otherwise} \end{array}\right\}$$

Now for 2 features it can be written as w1x1 + w2x2 - b >= 0
Let w0 = -b and x0 = 1
Further rewritten as w1x1 + w2x2 + w0x0 >= 0

Generalizing the above equation for n features as shown below

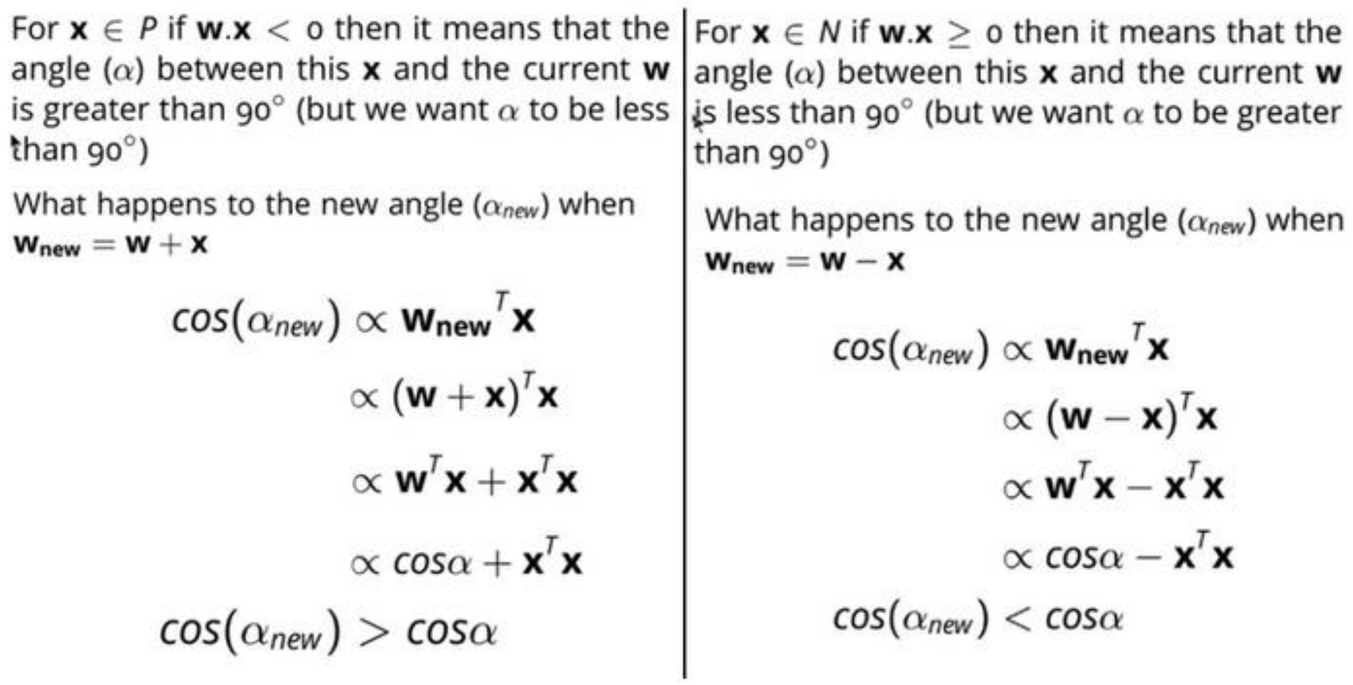
$$\hat{y} = 1 \text{ if } \textstyle\sum_{i=1}^{n} w_i x_i \geq b$$

$$\hat{y} = 0 \text{ otherwise}$$

Here we are finding vector **w** capable of absolutely separating both sets of data. Training data contains two sets of inputs, P (Positive y = 1) and N (Negative y = 0). Perceptron learning algorithm goes like this,

```
Algorithm: Perceptron Learning Algorithm
$P \leftarrow$ inputs with label 1;
$N \leftarrow$ inputs with label 0;
Initialize w randomly;
while !convergence do
    Pick random $\mathbf{x} \in P \cup N$;
    if $\mathbf{x} \in P$ and $\sum_{i=0}^{n} w_i * x_i < 0$ then
        $\mathbf{w} = \mathbf{w} + \mathbf{x}$;
    end
    if $\mathbf{x} \in N$ and $\sum_{i=0}^{n} w_i * x_i \geq 0$ then
        $\mathbf{w} = \mathbf{w} - \mathbf{x}$;
    end
end
//the algorithm converges when all the inputs are
  classified correctly
```

What is the intuition behind the Perceptron learning algorithm?
We need to see why updating weights classifies the positive and negative classes
Consider two vectors W & X
W = [w1, w2, … wn]
X = [x1, x2, … xn]
Cos θ = W.X/||W||X||, here the numerator can be replaced with Σ wixi
The denominator is always positive
Therefore Cos $\theta \propto \Sigma$ wixi
As θ ranges from 0 to 180 degree, cos θ ranges from 1 to -1
If cosθ > 0, it is an acute angle
If cosθ < 0, it is an obtuse angle



For $\mathbf{x} \in P$ if $\mathbf{w}.\mathbf{x} < 0$ then it means that the angle (α) between this $\mathbf{x}$ and the current $\mathbf{w}$ is greater than 90° (but we want α to be less than 90°)

What happens to the new angle (α_{new}) when $\mathbf{w_{new}} = \mathbf{w} + \mathbf{x}$

$$\begin{aligned} cos(\alpha_{new}) &\propto \mathbf{w_{new}}^T\mathbf{x} \\ &\propto (\mathbf{w}+\mathbf{x})^T\mathbf{x} \\ &\propto \mathbf{w}^T\mathbf{x} + \mathbf{x}^T\mathbf{x} \\ &\propto cos\alpha + \mathbf{x}^T\mathbf{x} \end{aligned}$$
$$cos(\alpha_{new}) > cos\alpha$$

For $\mathbf{x} \in N$ if $\mathbf{w}.\mathbf{x} \geq 0$ then it means that the angle (α) between this $\mathbf{x}$ and the current $\mathbf{w}$ is less than 90° (but we want α to be greater than 90°)

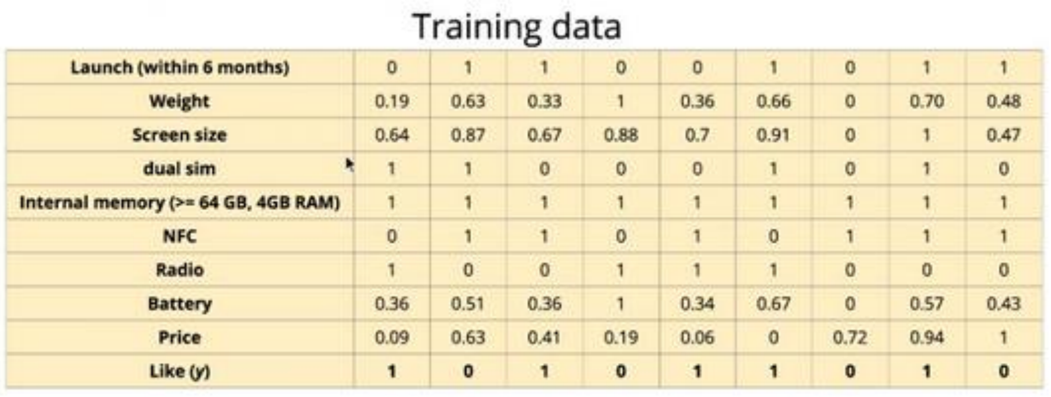
What happens to the new angle (α_{new}) when $\mathbf{w_{new}} = \mathbf{w} - \mathbf{x}$

$$\begin{aligned} cos(\alpha_{new}) &\propto \mathbf{w_{new}}^T\mathbf{x} \\ &\propto (\mathbf{w}-\mathbf{x})^T\mathbf{x} \\ &\propto \mathbf{w}^T\mathbf{x} - \mathbf{x}^T\mathbf{x} \\ &\propto cos\alpha - \mathbf{x}^T\mathbf{x} \end{aligned}$$
$$cos(\alpha_{new}) < cos\alpha$$

Perceptron Learning - Will it always work
Will this algorithm always work?
It will only work if the data is linearly separable
If it is not linearly separable, the algorithm will never converge (i.e, predict all training examples correctly)
Linearly Separable: Two sets P and N of points in an n-dimensional space are called absolutely linearly separable if n+1 real numbers w0, w1, ...wn exist such that
Every point (x0, x1, ... xn) ∈ P satisfies Σ wixi >= w0
Every point (x0, x1, ... xn) ∈ N satisfies Σ wixi < w0
If the sets P and N are finite and linearly separable, the Perceptron learning algorithm will converge in a finite number of steps

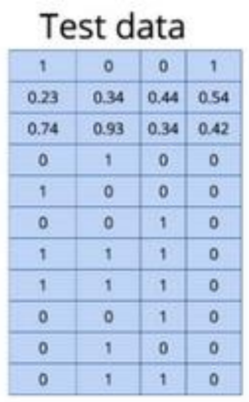
PERCEPTRON EVALUATION

Once we got the best weights **w** from the learning algorithm, we can evaluate the model on the test data by comparing the actual class of observation and the predicted class of the observation.



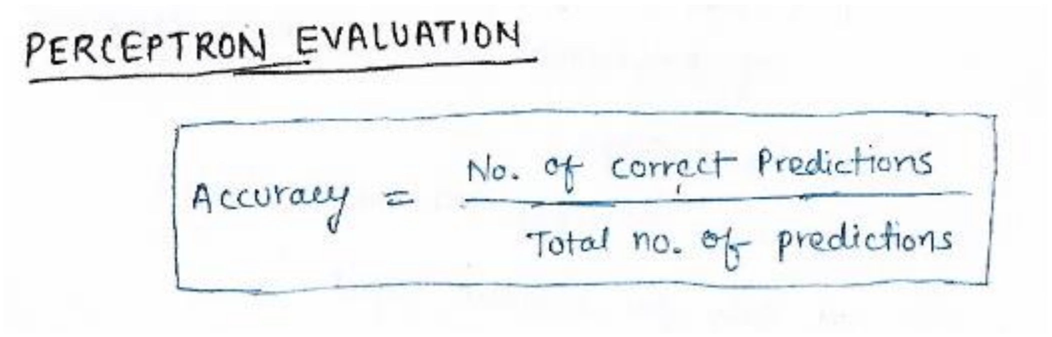
Training data

Launch (within 6 months)	0	1	1	0	0	1	0	1	1
Weight	0.19	0.63	0.33	1	0.36	0.66	0	0.70	0.48
Screen size	0.64	0.87	0.67	0.88	0.7	0.91	0	1	0.47
dual sim	1	1	0	0	0	1	0	1	0
Internal memory (>= 64 GB, 4GB RAM)	1	1	1	1	1	1	1	1	1
NFC	0	1	1	0	1	0	1	1	1
Radio	1	0	0	1	1	1	0	0	0
Battery	0.36	0.51	0.36	1	0.34	0.67	0	0.57	0.43
Price	0.09	0.63	0.41	0.19	0.06	0	0.72	0.94	1
Like (y)	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**0**



Test data

1	0	0	1
0.23	0.34	0.44	0.54
0.74	0.93	0.34	0.42
0	1	0	0
1	0	0	0
0	0	1	0
1	1	1	0
1	1	1	0
0	0	1	0
0	1	0	0
0	1	1	0



PERCEPTRON EVALUATION

$$\boxed{\text{Accuracy} = \frac{\text{No. of correct Predictions}}{\text{Total no. of predictions}}}$$

SUMMARY

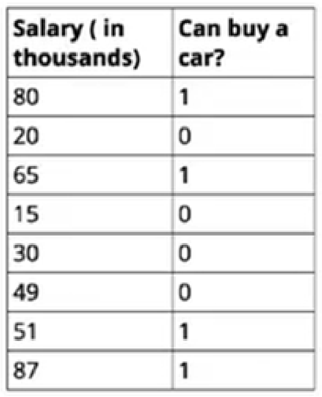
Data: Real inputs
Task: Classification (Boolean outputs)
Model: Weight for every input but still linear
Loss: Mean squared Error
Learning: Randomly assign and adjust w & b iteratively till convergence
Evaluation: Accuracy

3) SIGMOID

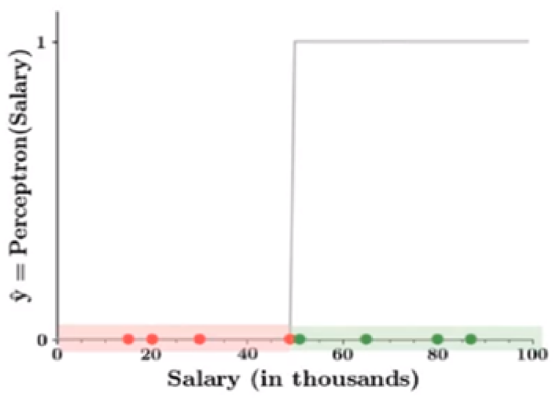
Why SIGMOID NEURON?

Perceptron model takes several real-valued inputs and gives a single binary output. In the perceptron model, every input xi has weight wi associated with it. The weights indicate the importance of the input in the decision-making process. The model output is decided by a threshold $\mathbf{W_o}$ if the weighted sum of the inputs is greater than threshold $\mathbf{W_o}$ output will be 1 else output will be 0. In other words, the model will fire if the weighted sum is greater than the threshold.

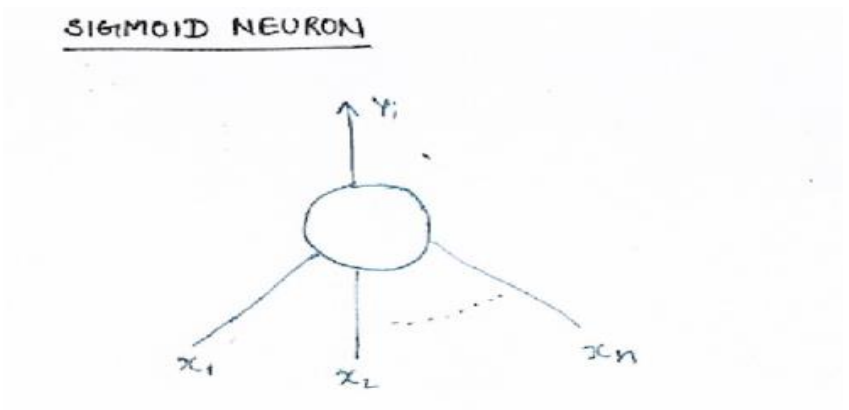
From the mathematical representation, we might say that the thresholding logic used by the perceptron is very harsh. Let's see the harsh thresholding logic with an example. Consider the decision making process of a person, whether he/she would like to purchase a car or not based on only one input X1 — Salary and by setting the threshold $\mathbf{b}(\mathbf{W_o}) = -10$ and the weight $\mathbf{W_1} = 0.2$. The output from the perceptron model will look like in the figure shown below.



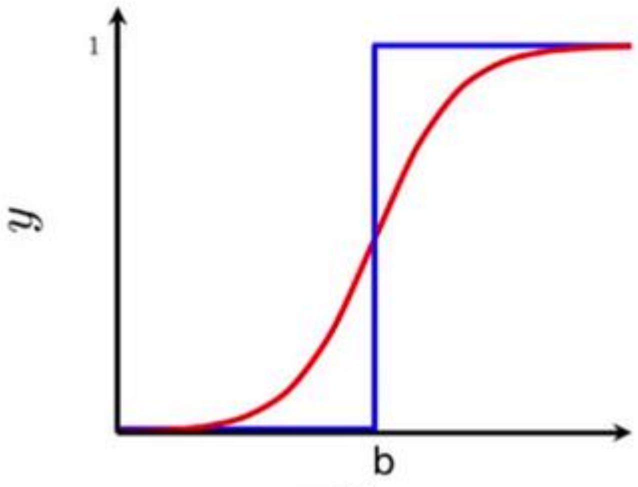
Salary (in thousands)	Can buy a car?
80	1
20	0
65	1
15	0
30	0
49	0
51	1
87	1



Red points indicates that person would not buy a car and green points indicates that person would like to buy a car. Isn't it a bit odd that a person with 50.1K will buy a car but someone with a 49.9K will not buy a car? The small change in the input to a perceptron can sometimes cause the output to completely flip, say from 0 to 1. This behavior is not a characteristic of the specific problem we choose or the specific weight and the threshold we choose. It is a characteristic of the perceptron neuron itself which behaves like a step function. We can overcome this problem by introducing a new type of artificial neuron called a *sigmoid* neuron.



$$\hat{Y} = \frac{1}{1 + e^{-\left(\sum_{i=1}^{n} w_i x_i + b\right)}}$$



$$y = \frac{1}{1+e^{-(w^T x+b)}}$$

$$\cdot \quad \sum_{i=1}^{n} w_i x_i$$

DATA
Regression and Classification
The inputs to the sigmoid neuron can be real numbers unlike the boolean inputs in MP Neuron and the output will also be a real number between 0–1.
In the sigmoid neuron, we are trying to regress the relationship between **X** and **Y** in terms of probability. Even though the output is between 0–1, we can still use the sigmoid function for binary classification tasks by choosing some threshold.

LOSS FUNCTION

Going forward

From probability theory
To understand cross entropy, we need to understand probability
Information content depends on probability of event
A situation with several possible outcomes can be modeled by a random variable X where for any possible outcome xi that has probability pi of occurring:

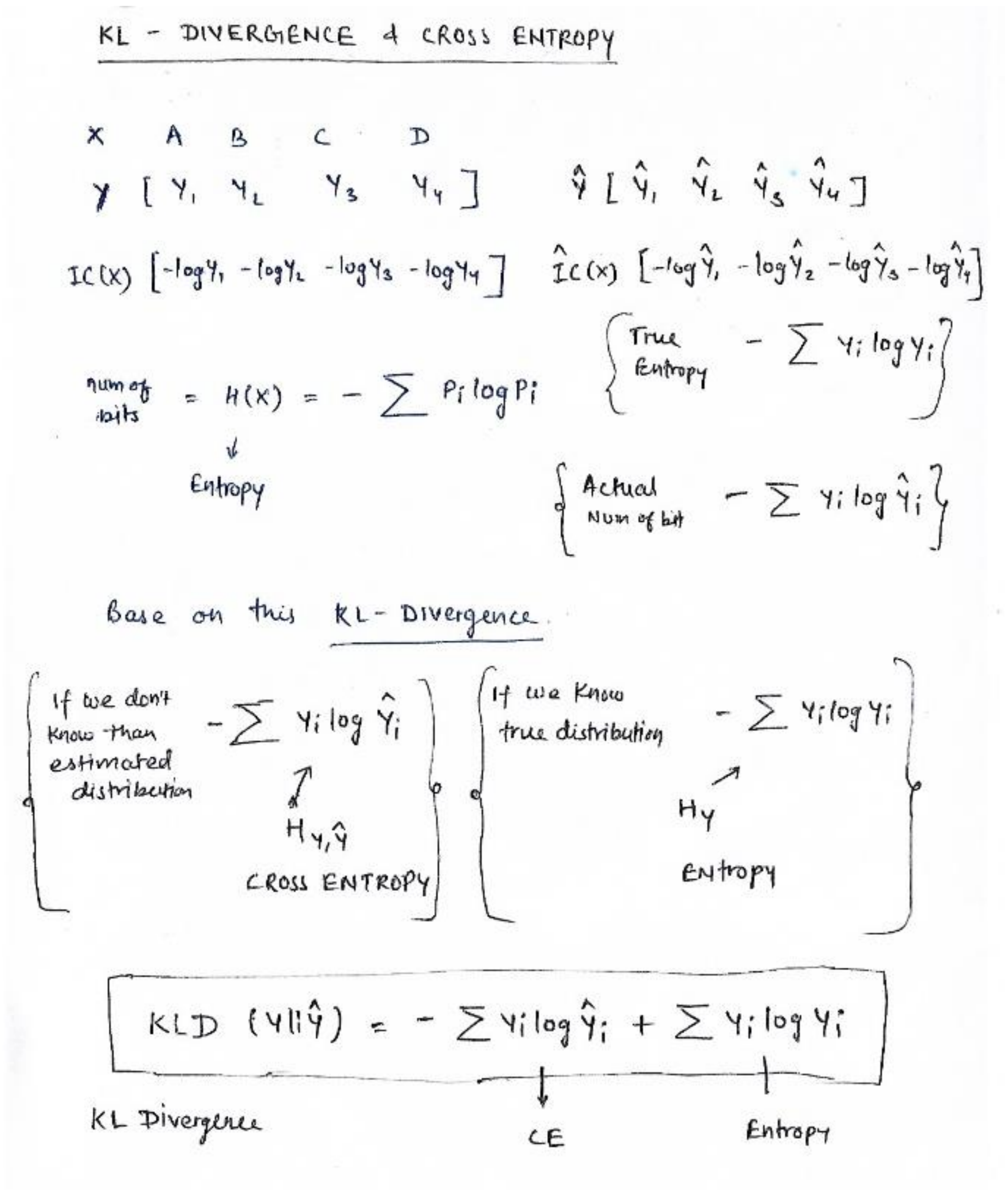
$$\Pr(X = x_i) = p_i.$$

$$I(x_i) = \log_2 \left(\frac{1}{p_i}\right).$$

Entropy

$$S = -\sum_i P_i \log P_i$$

KL-DIVERGENCE & CROSS ENTROPY



KL - DIVERGENCE & CROSS ENTROPY

$$\begin{array}{ccccc} X & A & B & C & D \\ y & [\,y_1 & y_2 & y_3 & y_4\,] \end{array} \qquad \hat{y}\,[\,\hat{y}_1 \;\; \hat{y}_2 \;\; \hat{y}_3 \;\; \hat{y}_4\,]$$

$$IC(X)\;[-\log y_1 \;\; -\log y_2 \;\; -\log y_3 \;\; -\log y_4] \qquad \hat{IC}(x)\;[-\log \hat{y}_1 \;\; -\log \hat{y}_2 \;\; -\log \hat{y}_3 \;\; -\log \hat{y}_4]$$

$$\text{num of bits} = \underbrace{H(X)}_{\text{Entropy}} = -\sum P_i \log P_i$$

$$\left\{\text{True Entropy} \quad -\sum y_i \log y_i\right\}$$

$$\left\{\text{Actual Num of bit} \quad -\sum y_i \log \hat{y}_i\right\}$$

Base on this KL- Divergence

$$\left\{\text{If we don't know than estimated distribution} \quad -\sum y_i \log \hat{y}_i \;\; \uparrow H_{y,\hat{y}} \;\; \text{CROSS ENTROPY}\right\} \quad \left\{\text{If we know true distribution} \quad -\sum y_i \log y_i \;\; \uparrow H_y \;\; \text{ENTROPY}\right\}$$

$$\underbrace{\boxed{KLD(y\|\hat{y}) = \underbrace{-\sum y_i \log \hat{y}_i}_{CE} + \underbrace{\sum y_i \log y_i}_{\text{Entropy}}}}_{\text{KL Divergence}}$$

So Cross entropy and Sigmoid neuron following will be loss function

Consider the Example:

A signboard with the text Mumbai

A random variable X which maps the signboard to: Text, No-Text

The distributions are as follows

X	y (We don't know initially)	ŷ (Predicted using sigmoid)
T	1	0.7
NT	0	0.3

Previously, we were using Squared-error Loss = $\Sigma_i(y_i - \hat{y}_i)^2$
Now, we have a better metric, one that is grounded in probability theory (KL- Divergence)

$$\boxed{KLD\ (y\|\hat{y}) = \underbrace{-\sum y_i \log \hat{y}_i}_{CE} + \underbrace{\sum y_i \log y_i}_{Entropy}}$$

KL Divergence

We aim to minimize loss by KLD with respect to the parameters w, b
From KLD equation, we can see that yi doesn't depend on w, b. So therefore, we are really only trying to minimize the first term, i.e. the cross-entropy
So in practice, we can treat the second term as a constant, and the equation would really be
$min(-\Sigma_i y_i log\hat{y}_i)$ $here\ i\ \in T, NT$

$$Cross\ Entropy\ Loss = -1 * log(0.7) - 0 * log(0.3)$$

The second terms cancels out and we are left with $-log(0.7)$which is the same as $-log\hat{y}$ $(for\ the\ true\ case)$
It can be called $-log\ \hat{y}_c$ where c can take the value 0 or 1 which correspond to NT and T

So loss function will be

KL Divergence / cross Entropy loss

$$\boxed{L = -\left[(1-y)\log(1-\hat{y}) + y\log(\hat{y})\right]}$$

As w changes → slope changes
As b changes → entire fxn shifted

LEARNING ALGORITHM

From given data we wanted to find out the w & b parameters such that loss is minimum so predicted Y is close to actual Y, we need to learn w & b by learning algorithm

Now let's first discuss learning by guessing
Can we try to learn estimate w, b using guess work?

- ➔ We initialize w & b randomly
- ➔ We will see Xi and update value of w & b
    - o Initialize w, b
    - o Iterate over data
        - ▪ w = w + delta w
        - ▪ b = b + delta b
    - o till satisfied
- ➔ This guess work will not work when dimensions are high
- ➔ So, we want appropriate solution so that from anywhere we start we wanted to reach minimum point ASAP, this is what we wanted to achieve

MATHEMATICAL SETUP of LEARNING ALGORITHM

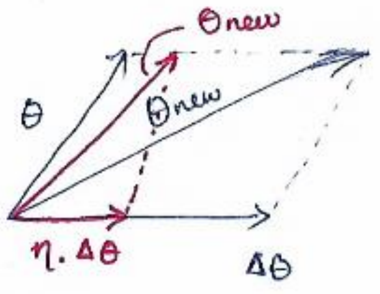
Instead of guessing delta w & delta b we need a principle way of changing w & b based on loss function
So, we will try to formulate more mathematically

can we formulate this more mathematically?

$\theta = [w, b]$ θ is vector => w, b are its components

$\Delta\theta = [\Delta w, \Delta b]$ in every iteration we are going to change it.

$\theta_{i+1} = \theta_i + \Delta\theta$



θ to θ_{new} is large so we take smallest value or step in that direction.

$$\theta_{new} = \theta + \underset{\text{learning rate}}{\eta} \cdot \Delta\theta$$

<u>smaller the η</u> → {smaller are the step it will take}

How to get delta theta?

$\mathcal{L}(w, b)$ $\swarrow$ $\Delta\theta$ such that

$\mathcal{L}(\theta_{new}) < \mathcal{L}(\theta_{old})$

Now how does our algorithm look like?

```
Initialise $w, b$
Iterate over data:
    compute $\hat{y}$
    compute $\mathscr{L}(w, b)$
    $w_{t+1} = w_t - \eta \Delta w_t$
    $b_{t+1} = b_t - \eta \Delta b_t$
till satisfied
```

Now following question remains
What we are doing to update weights?
Why we are updating the weights like this?

Conditions when we will stop

- ➔ When loss = 0
- ➔ When loss < epsilon (very small value)
- ➔ Specific amount of iteration

$$w_{t+1} = w_t - \eta \Delta w_t$$

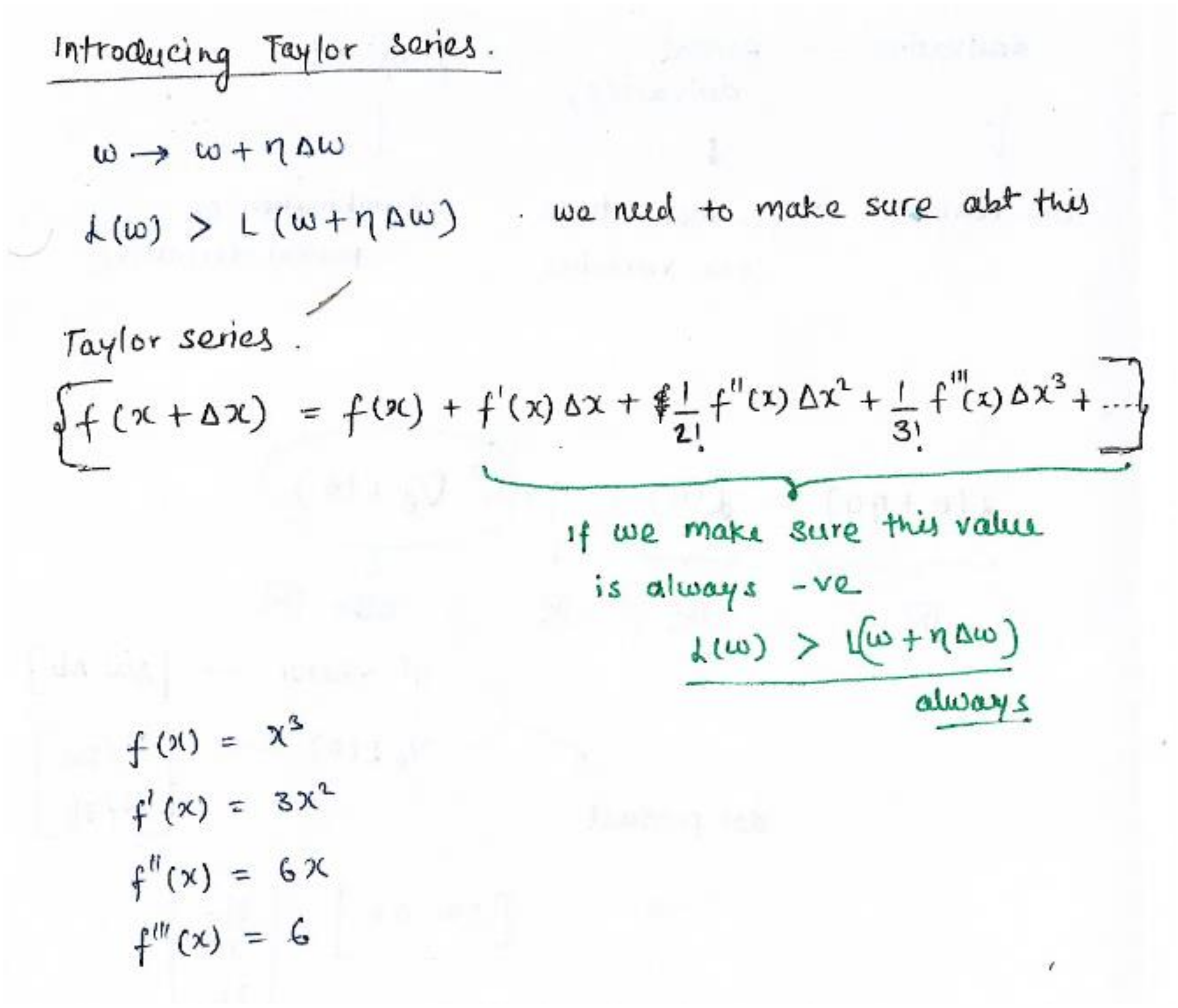
$$b_{t+1} = b_t - \eta \Delta b_t$$

$$\boxed{\Delta w_t = \frac{\partial \mathcal{L}}{\partial w}}$$

grad_w

Now we will discuss abt this

But how do we decide the value of **Δ**θ and what is the right **Δ**θ to use?. How do we get the correct **Δ**θ in a principal manner, so that the loss (which is a function of **w** and **b**) at the new theta should be less than the loss at the old theta. The answer to this question comes from Taylor series.
What Taylor series tells us is that, if we have a function **f** and we know the value of the function at a particular point **x** then the value of a function **f** at a new point that is very close to **x** is given by below formula,



Introducing Taylor series.

$$w \rightarrow w + \eta \Delta w$$

$$L(w) > L(w + \eta \Delta w)$$. we need to make sure abt this

Taylor series.

$$f(x+\Delta x) = f(x) + \underbrace{f'(x)\Delta x + \frac{1}{2!} f''(x)\Delta x^2 + \frac{1}{3!} f'''(x)\Delta x^3 + \dots}_{\text{If we make sure this value is always -ve}}$$

$$\underline{L(w) > L(w+\eta\Delta w)}$$
always

$$f(x) = x^3$$
$$f'(x) = 3x^2$$
$$f''(x) = 6x$$
$$f'''(x) = 6$$

As mentioned above quantity highlighted by green is dependent on delta X
If I am able to find delta **x** such that, the quantity represented in the green would be negative, then I would know that the new loss at new **x** is less than the loss at old **x**.

Now we will discuss Taylor series in terms of Sigmoid neuron

Taylor Series on loss fxn

$$\mathcal{L}(w+\Delta w) = \mathcal{L}(w) + \underbrace{\mathcal{L}'(w)\Delta w + \frac{1}{2!}\mathcal{L}''(w)\Delta w^2 + \frac{1}{3!}\mathcal{L}'''(w)\Delta w^3 + \dots}_{-\text{ve so that}}$$

$\mathcal{L}(w) > L(w+\eta\Delta w)$

not only for w
also for b.

$\mathcal{L}(b) > \mathcal{L}(b+\Delta b)$

$$\underbrace{\mathcal{L}(w,b) > \mathcal{L}(w+\eta\Delta w, b+\eta\Delta b)}_{\text{loss fxn is fxn for both } w+b}$$

If we change w or b $\longrightarrow$ $\hat{y} = \frac{1}{1+e^{-(wx+b)}}$ change.

$\downarrow$

$\frac{1}{n}\sum(\hat{y}-y)^2$ chages

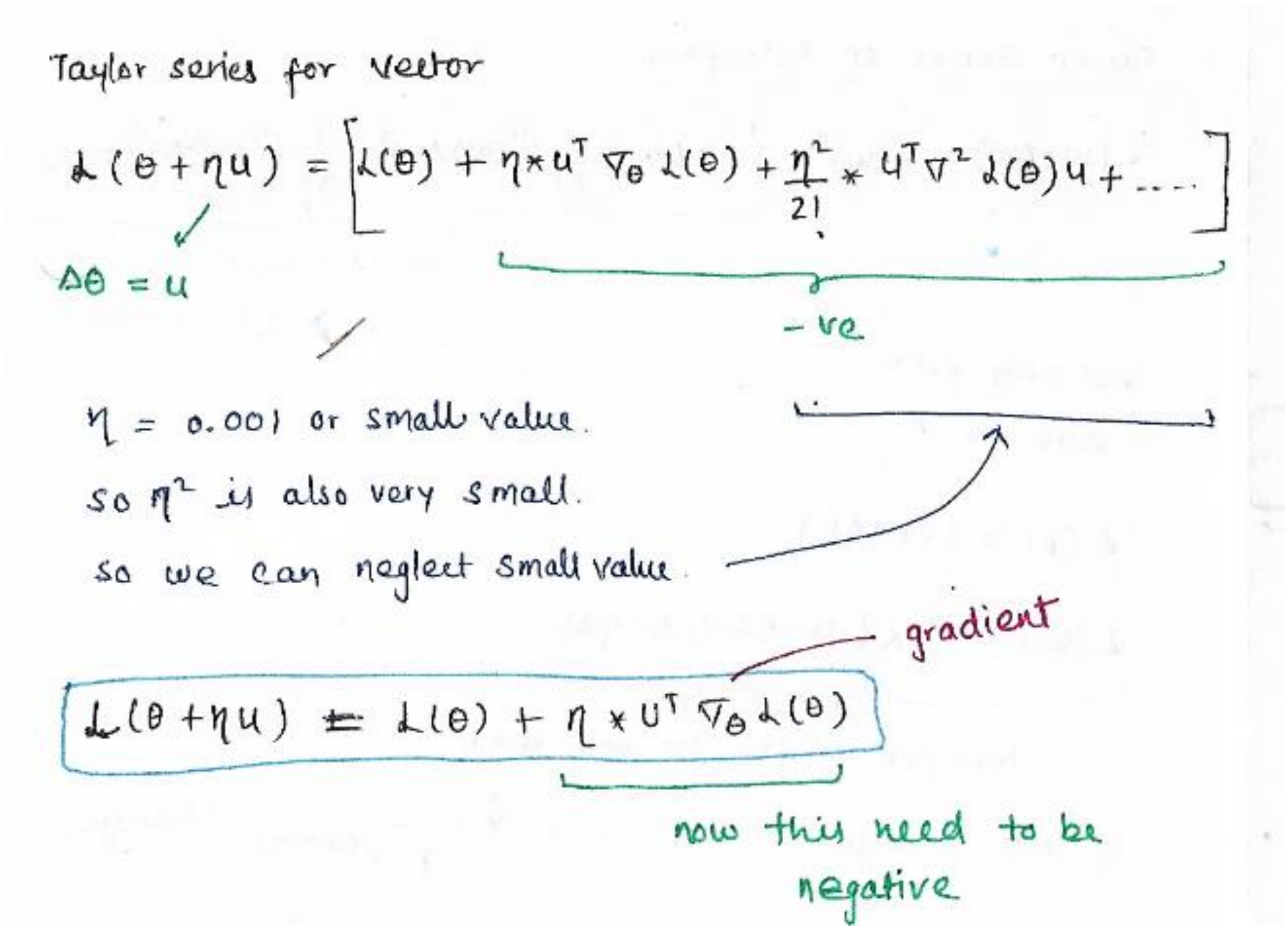
$\mathcal{L}(\theta) > \mathcal{L}(\theta+\eta\theta)$ $\because$ $\theta = [w,b]$ vector.

$$\theta_{new} = \theta_{old} + \eta\Delta\theta$$

$$\begin{bmatrix} w_{old} \\ b_{old} \end{bmatrix} \begin{matrix} +\eta \\ +\eta \end{matrix} \begin{bmatrix} \Delta w \\ \Delta b \end{bmatrix}$$

As θ is vector now so do we have a taylor series of vector.

Going forward



Taylor series for vector

$$\mathcal{L}(\theta+\eta u) = \Big[\mathcal{L}(\theta) + \underbrace{\eta * u^T \nabla_\theta \mathcal{L}(\theta) + \frac{\eta^2}{2!} * u^T \nabla^2 \mathcal{L}(\theta) u + \dots}_{-ve}\Big]$$

$\Delta\theta = u$ ✓

$\eta = 0.001$ or small value.

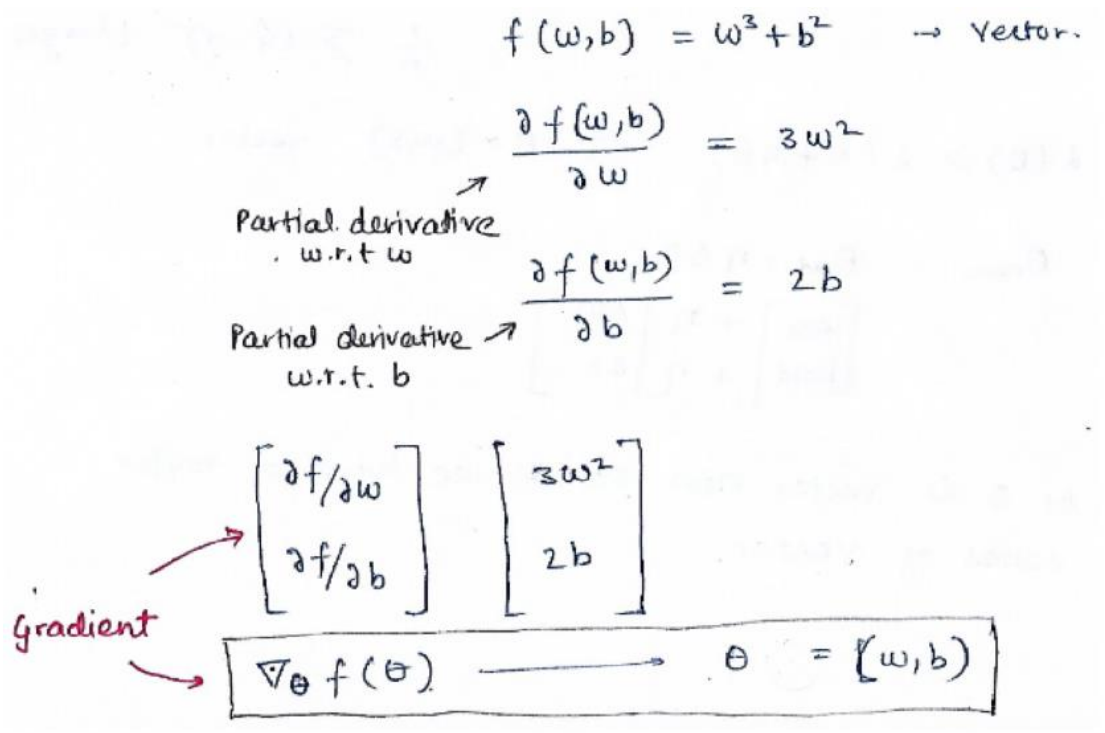
so η^2 is also very small.

so we can neglect small value

gradient

$$\boxed{\mathcal{L}(\theta+\eta u) \doteq \mathcal{L}(\theta) + \underbrace{\eta * u^T \nabla_\theta \mathcal{L}(\theta)}_{\text{now this need to be negative}}}$$

Now let's consider following example



$$f(w,b) = w^3 + b^2 \rightarrow \text{Vector.}$$

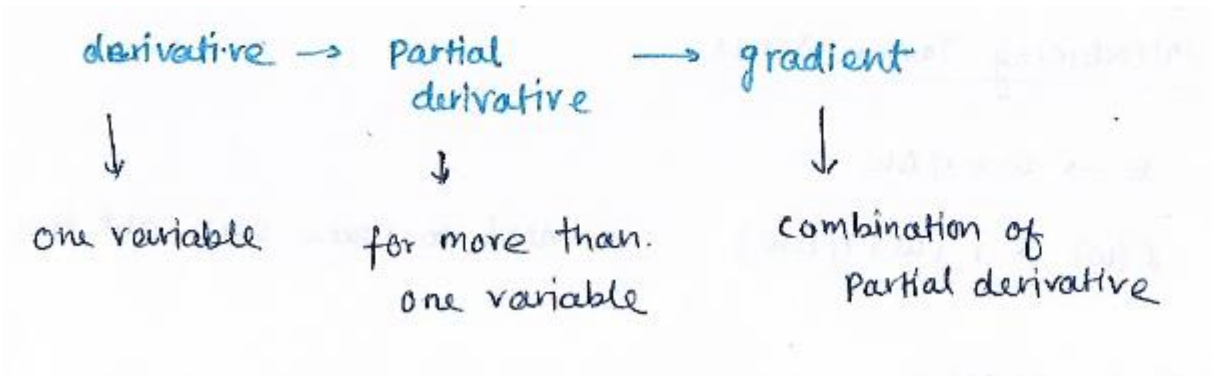
Partial derivative w.r.t w ↗ $$\frac{\partial f(w,b)}{\partial w} = 3w^2$$

Partial derivative w.r.t. b ↗ $$\frac{\partial f(w,b)}{\partial b} = 2b$$

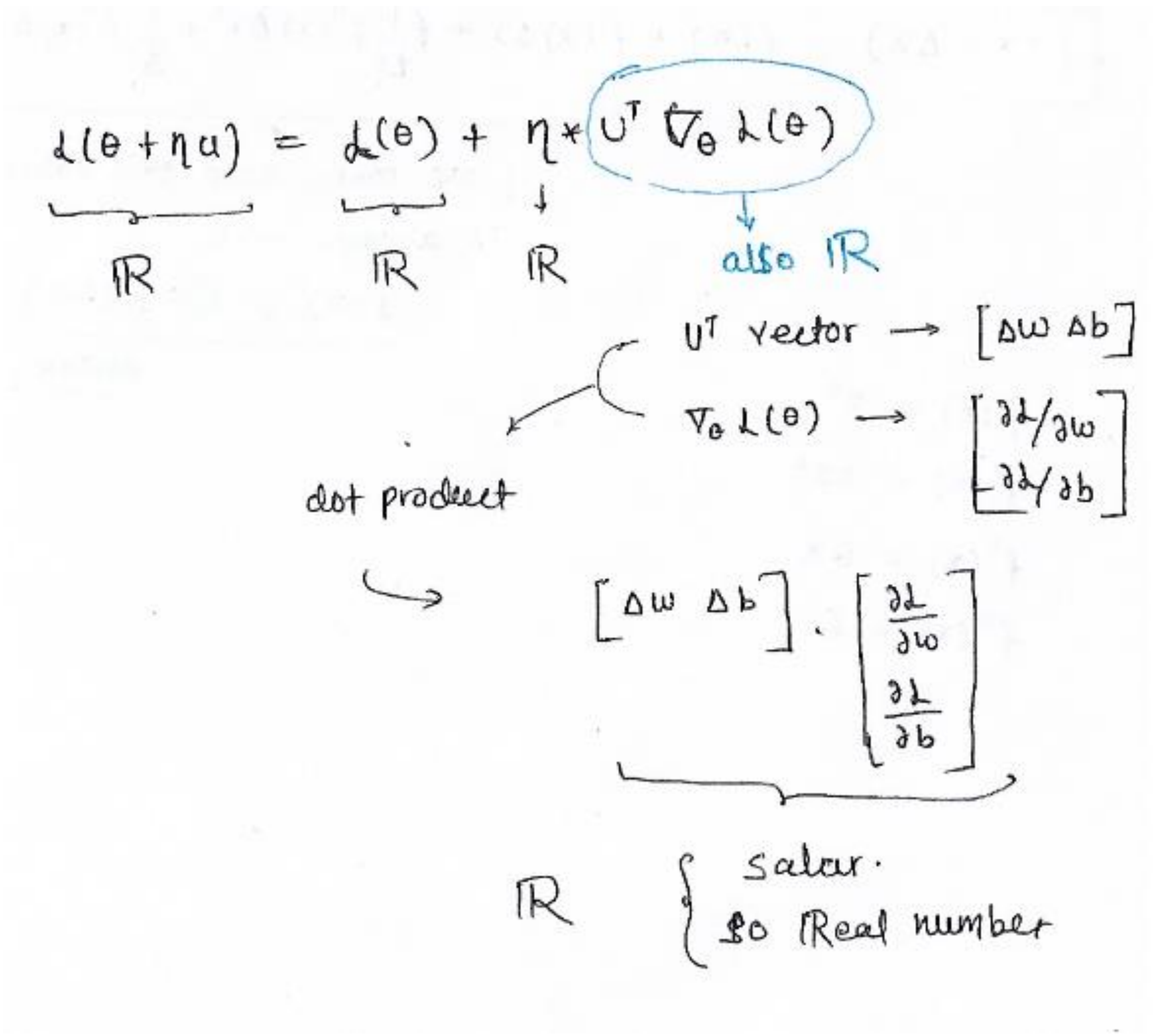
Gradient

$$\begin{bmatrix} \partial f/\partial w \\ \partial f/\partial b \end{bmatrix} \quad \begin{bmatrix} 3w^2 \\ 2b \end{bmatrix}$$

$$\boxed{\nabla_\theta f(\theta) \longrightarrow \theta = [w,b)}$$



$$\begin{tikzcd}
\text{derivative} \arrow[r] \arrow[d] & \text{Partial derivative} \arrow[r] \arrow[d] & \text{gradient} \arrow[d] \\
\text{one variable} & \text{for more than. one variable} & \text{combination of Partial derivative}
\end{tikzcd}$$



$$\underbrace{L(\theta+\eta u)}_{\mathbb{R}} = \underbrace{L(\theta)}_{\mathbb{R}} + \underbrace{\eta}_{\mathbb{R}} * \underbrace{\boxed{U^T \nabla_\theta L(\theta)}}_{\text{also } \mathbb{R}}$$

U^T vector $\rightarrow \begin{bmatrix} \Delta w & \Delta b \end{bmatrix}$

$\nabla_\theta L(\theta) \rightarrow \begin{bmatrix} \partial L/\partial w \\ \partial L/\partial b \end{bmatrix}$

dot prodcct

$$\hookrightarrow \underbrace{\begin{bmatrix} \Delta w & \Delta b \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial L}{\partial w} \\ \frac{\partial L}{\partial b} \end{bmatrix}}$$

$\mathbb{R}$ $\left\{ \begin{array}{l} \text{salar.} \\ \text{so Real number} \end{array} \right.$

Deriving Gradient Descent Update rule

After getting rid of some terms we get following equation as explained above

$$\mathscr{L}(\theta+\eta u) \approx \mathscr{L}(\theta) + \eta * u^T \nabla_\theta \mathscr{L}(\theta)$$

The change vector u^T is optimal only if the loss at the new θ is less than old θ,

$$\mathscr{L}(\theta + \eta u) - \mathscr{L}(\theta) = \eta * u^T \nabla_\theta \mathscr{L}(\theta)$$

Note that the move ηu would be favorable only if,

$$\mathscr{L}(\theta + \eta u) - \mathscr{L}(\theta) < 0 \quad \text{[i.e. if the new loss is less than the previous loss]}$$

By comparing the above two equations (change vector u^T optimization equation and truncated Taylor series equation) we can imply that,

$$u^T \nabla_\theta \mathscr{L}(\theta) < 0$$

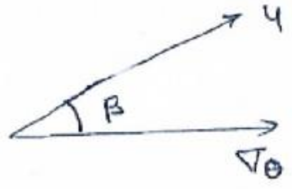
Explaining above equation a bit

This implies.,

$$\underbrace{u^T}_{\mathbb{R}^2} \overbrace{\nabla_\theta L(\theta)}^{\mathbb{R}^2} < 0$$

$$\begin{bmatrix} \Delta w \\ \Delta b \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial L}{\partial w} \\ \frac{\partial L}{\partial b} \end{bmatrix}$$

let β be angle b/w u & $\nabla_\theta L(\theta)$



$$\cos\beta = \frac{u^T \nabla_\theta L(\theta)}{\|u\| \; \|\nabla_\theta L(\theta)\|}$$

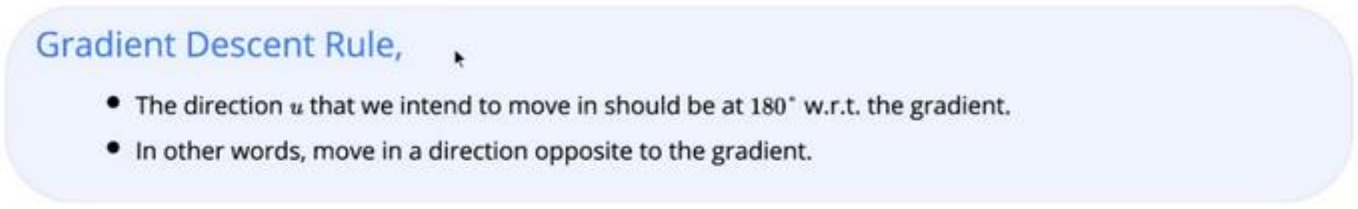
$$\boxed{-1 \le \left[\cos(\beta) = \frac{u^T \nabla_\theta L(\theta)}{\|u\| * \|\nabla_\theta L(\theta)\|}\right] \le 1}$$

multiply thoroughout by $K = \|u\| * \|\nabla_\theta L(\theta)\|$

$$\boxed{-k \le u^T \nabla_\theta L(\theta) \le k}$$

$$\hookrightarrow L(\theta + \eta u) - L(\theta)$$

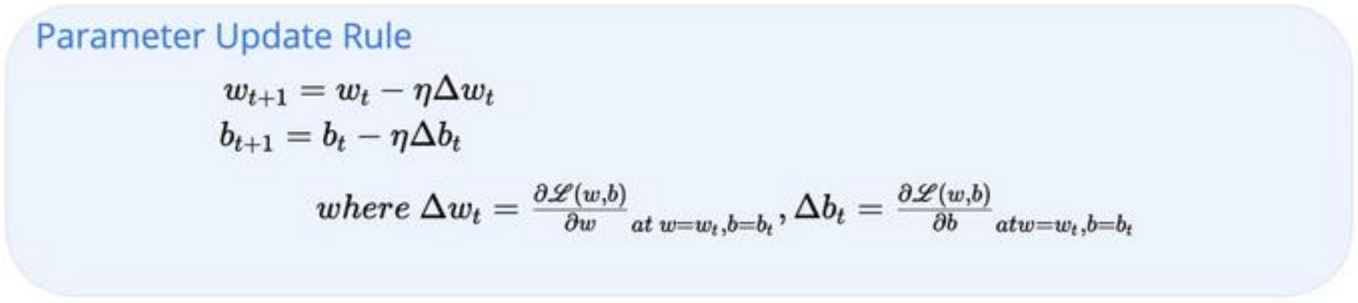
Since the denominator of cosine angle formula is a magnitude and it will always be positive, we can multiply the cosine formula throughout by **k** (denominator) without affecting the sign of the inequality. We want the quantity present in-between (marked in the blue box) the inequality to be negative, that is only possible if the cosine angle between the u^T and ∇ is equal to -1. If the cosine angle is equal to -1 then we know the angle between the vectors is equal to 180^o.
If the angle is equal to 180^o that means the direction of change vector u^T we choose should be opposite to the gradient vector. We can summarise the gradient descent rule as follows,



Gradient Descent Rule,

- The direction u that we intend to move in should be at 180° w.r.t. the gradient.
- In other words, move in a direction opposite to the gradient.

Similarly, we can write the parameter update rule as follows,



Parameter Update Rule

$$w_{t+1} = w_t - \eta \Delta w_t$$
$$b_{t+1} = b_t - \eta \Delta b_t$$
$$where\ \Delta w_t = \frac{\partial \mathscr{L}(w,b)}{\partial w}_{at\ w=w_t,b=b_t}, \Delta b_t = \frac{\partial \mathscr{L}(w,b)}{\partial b}_{at w=w_t,b=b_t}$$

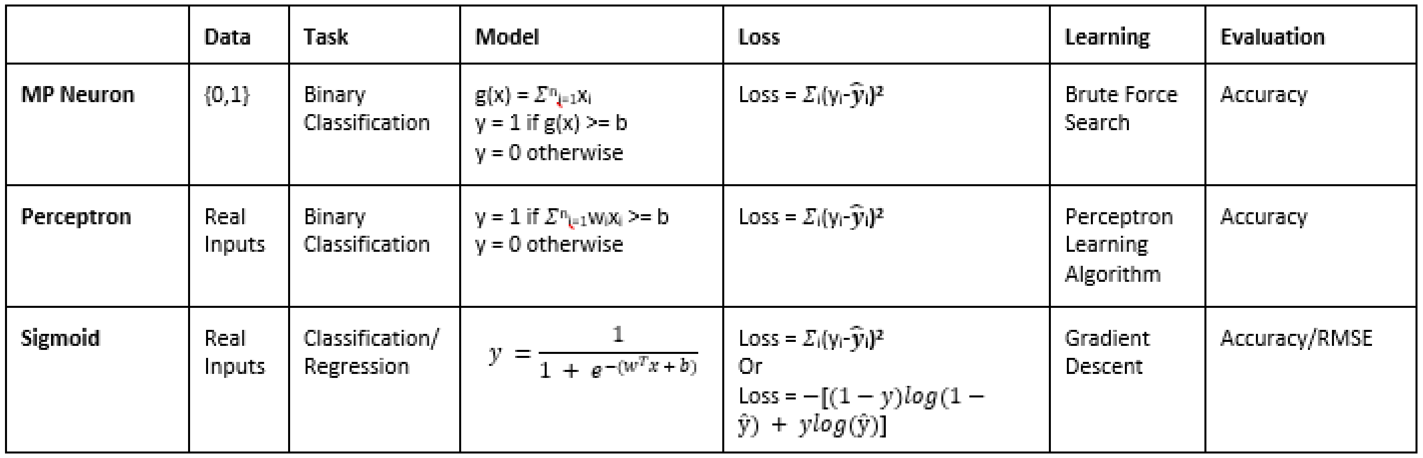
So now we should know how to calculate partial derivatives

LEARNING ALGORITHM for CROSS ENTROPY FUNCTION

* Initialize $w + b$ randomly
* Iterate over data.
  - ⇒ Compute $\hat{y}$
  - ⇒ Compute Loss $L(w,b)$ (Cross-entropy loss)
  - ⇒ Derivative $\nabla_w L$, $\nabla_b L$ $\quad \nabla_w L = \begin{bmatrix} \partial L/\partial w_1 \\ \partial L/\partial w_2 \\ \vdots \\ \partial L/\partial w_n \end{bmatrix}$
  - ⇒ $w_{new} = w_{old} - \eta \left[\nabla_w L\right]$

    $b_{new} = b_{old} - \eta \left[\nabla_b L\right]$
* Till satisfied.
  - ⇒ Number of epochs is reached
  - ⇒ Continue till loss $< \varepsilon$ (some defined value)
  - ⇒ continue till $Loss(w,b)_{t+1} \approx Loss(w,b)_t$.

Each one has its limitations and characteristics



	Data	Task	Model	Loss	Learning	Evaluation
MP Neuron	{0,1}	Binary Classification	$g(x) = \Sigma^n_{i=1} x_i$ $y = 1$ if $g(x) >= b$ $y = 0$ otherwise	Loss = $\Sigma_i (y_i - \hat{y}_i)^2$	Brute Force Search	Accuracy
Perceptron	Real Inputs	Binary Classification	$y = 1$ if $\Sigma^n_{i=1} w_i x_i >= b$ $y = 0$ otherwise	Loss = $\Sigma_i (y_i - \hat{y}_i)^2$	Perceptron Learning Algorithm	Accuracy
Sigmoid	Real Inputs	Classification/ Regression	$y = \frac{1}{1 + e^{-(w^T x + b)}}$	Loss = $\Sigma_i (y_i - \hat{y}_i)^2$ Or Loss = $-[(1 - y) log(1 - \hat{y}) + y log(\hat{y})]$	Gradient Descent	Accuracy/RMSE

- ➔ None of above 3 can handle non-linear separable data
- ➔ We care about continuous function because our learning algorithm (Gradient Descent) requires that the input functions be differential (i.e. continuous)
- ➔ In real world the functions will be complex and problem will be non-linear in nature
- ➔ Although sigmoid neuron cannot handle completely non-linear data but is step towards that and for non-linear we will need deep neural networks.
- ➔ Now to deal with complex real world function we are using lots of neurons which is known as Neural Network
- ➔ A deep neural network with a certain number of hidden layers would be able to approximate any function between input and output this is called **Universal Approximation Theorem**

Reference
https://github.com/mbadry1/DeepLearning.ai-Summary/tree/master/1-%20Neural%20Networks%20and%20Deep%20Learning
https://towardsdatascience.com/sigmoid-neuron-deep-neural-networks-a4cd35b629d7

Applied AI course
Depplearning.ai
Oneforthlab