Deep Learning:-

→ Deep learning attempts to mimic the human brain.

→ Deep learning is a subset of machine learning, which is essentially a neural network with 3 or more layers. These neural networks attempt to simulate the behavior of the human brain.

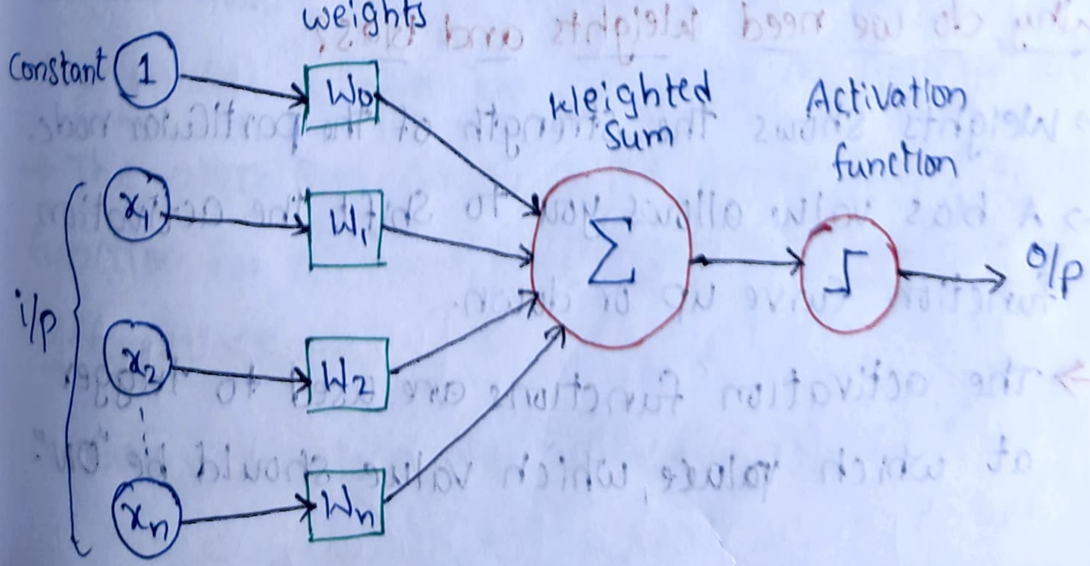
Neural Network:

Neural Network in deep learning uses interconnected nodes or neurons in a layered structure that resembles the human brain.

Perceptron:-

Perceptron is a single layer neural network. And a multi-layer perceptron is called Neural Networks.

→ Perceptron is a linear classifier (binary). It is used in supervised learning and helps to classify the given input data.



The perceptron consists of 4 parts.

1. Input values or one input layer.
2. Weights and Bias.
3. Net Sum
4. Activation function.

How does it work?

a. All the inputs x are multiplied with their weights w.

Ex:- $x_1 \cdot w_1, x_2 \cdot w_2, \dots x_n \cdot w_n$

b. Add all the multiplied values and call them weighted Sum.

$\Rightarrow x_1 \cdot w_1 + x_2 \cdot w_2 + \dots x_n \cdot w_n$

c. Apply that weighted Sum to the Correct Activation Function to obtain desired output.

This activation function is also known as the Step function and is represented by 'f'.

Why do we need weights and Bias?

→ Weights shows the strength of the particular node.

→ A bias value allows you to shift the activation function curve up or down.

→ The activation functions are used to trigger at which place, which value should be "ON".

→ Perceptron is used to classify the data into two parts. Therefore, it is also known as Linear Binary classifier.

Forward stage: Activation functions start from the input layer in the forward stage and terminate on the output layer.

Backward stage:- In the backward stage, weight and bias values are modified as per the model's requirement. In this stage, the error between actual output and demanded originated backward on the output layer and ended on the input layer.

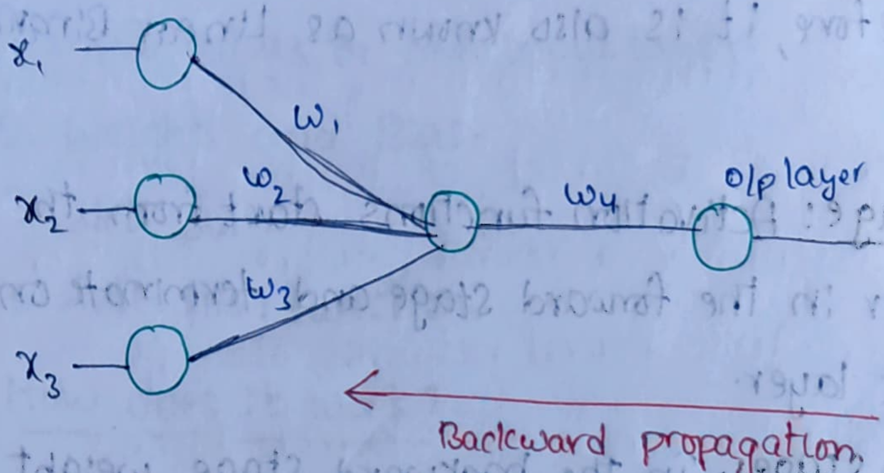
Back Propagation :-

Backpropagation is the essence of neural network training. It is the method of fine-tuning the weights of a ~~netual~~ network based on the error rate obtained in previous iteration. Proper tuning of weights allows you to reduce error rates and make the model reliable by increasing its generalization.

→ The algorithm computes the gradient of the loss function for a single weight by the Chain rule of Differentiation.

Error = Actual output − Desired output

$$= y - \hat{y}$$



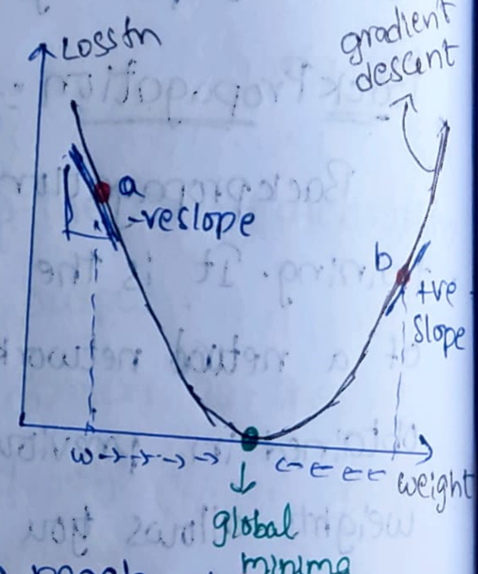
Weight updation formula:

$$\boxed{W_{new} = W_{old} - \eta \frac{\partial L}{\partial W_{old}}}$$

η – Learning rate

L – Loss

$\frac{\partial L}{\partial W_{old}} \Rightarrow$ slope



→ Let's consider our value is at point 'a'.

→ Now, Calculate slope by drawing a tangent line.

→ So, our slope is -ve slope, so, to reach global minima we have to increase the weight.

$\Rightarrow W_{new} = W_{old} - \eta(-ve)$ ∵ -ve slope

$W_{new} = W_{old} + \eta(+ve) \Rightarrow \boxed{W_{new} >> W_{old}}$

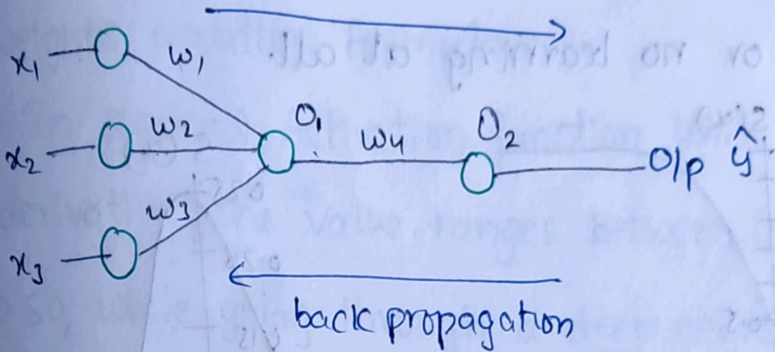
→ Let's consider our value as 'b'. So, it will be a +ve slope. To reach global minima, we have to decrease the weight.

⇒ $w_{new} = w_{old} - \eta(+ve)$

⇒ $\boxed{w_{new} \ll w_{old}}$

→ To Calculate slope type
→ if tangent right side is pointing downwards ⇒ −ve slope
→ if pointing upwards ⇒ +ve slope

Chain Rule of differentiation



$$w_{new} = w_{old} - \eta \frac{\delta L}{\delta w_{old}}$$

→ So, we are trying to update weight w_4

$$w_{4new} = w_{4old} - \eta \frac{\delta L}{\delta w_{4old}}$$

$$\boxed{\frac{\delta L}{\delta w_{4old}} = \frac{\delta L}{\delta O_2} \times \frac{\delta O_2}{\delta w_{4old}}}$$ ⇒ chain rule of diff

→ $$w_{1new} = w_{1old} - \eta \frac{\delta L}{\delta w_{1old}}$$

$$\frac{\delta L}{\delta w_{1old}} = \frac{\delta L}{\delta O_2} \times \frac{\delta O_2}{\delta O_1} \times \frac{\delta O_1}{\delta w_{1old}}$$

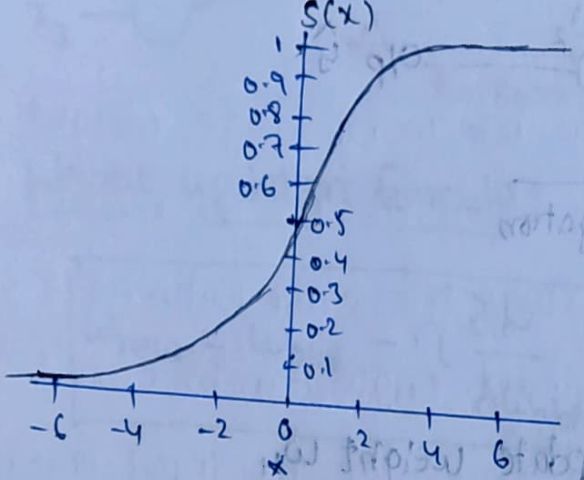
Vanishing Gradient Problem:-

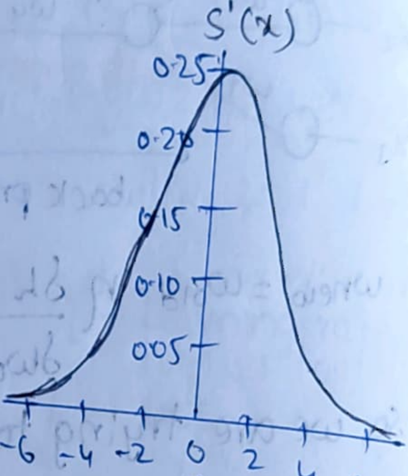
→ Sigmoid Activation functions are used frequently in neural networks to activate neurons.

$$\sigma(x) = \frac{1}{1+e^{-x}}$$ ⇒ sigmoid fn.

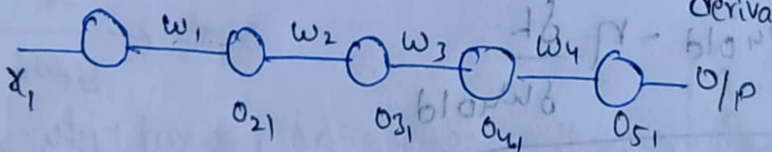
→ The use of Sigmoid fn restricted the training of deep neural networks because it caused the vanishing gradient problem.

→ This caused the neural network to learn at a slower pace or no learning at all.





derivative of Sigmoid Activation fn.



$$W_{new} = W_{1old} - \eta \frac{\delta L}{\delta W_{1old}}$$

$$\left[\frac{\delta L}{\delta W_{1old}}\right] = \frac{\delta L}{\delta O_{51}} \times \frac{\delta O_{51}}{\delta O_{41}} \times \frac{\delta O_{41}}{\delta O_{31}} \times \frac{\delta O_{31}}{\delta O_{21}} \times \frac{\delta O_{21}}{\delta W_1}$$

$$= \underbrace{0.25 \times 0.15 \times 0.10 \times 0.05 \times 0.02}_{\text{very small value}}$$

$$\Rightarrow W_{1new} = W_{1old} - \eta \left[\text{small value}\right]$$

$$\Rightarrow \boxed{W_{1new} \approx W_{1old}} \Rightarrow \text{Vanishing gradient problem.}$$

→ While doing Back propagation in a deep neural network, we have to update the weights.

→ To update the weights to "w_{new}", we have to calculate derivative of old weights based on the weights updation formula.

→ In sigmoid Activation function, while calculating derivative, the value ranges between [0 to 0.25]

→ So, while going through a deep neural network, the multiplication of weights which are small values [0 - 0.25], which leads to derivative with a very small value.
$$\left[\frac{\partial L}{\partial w_{old}}\right]$$

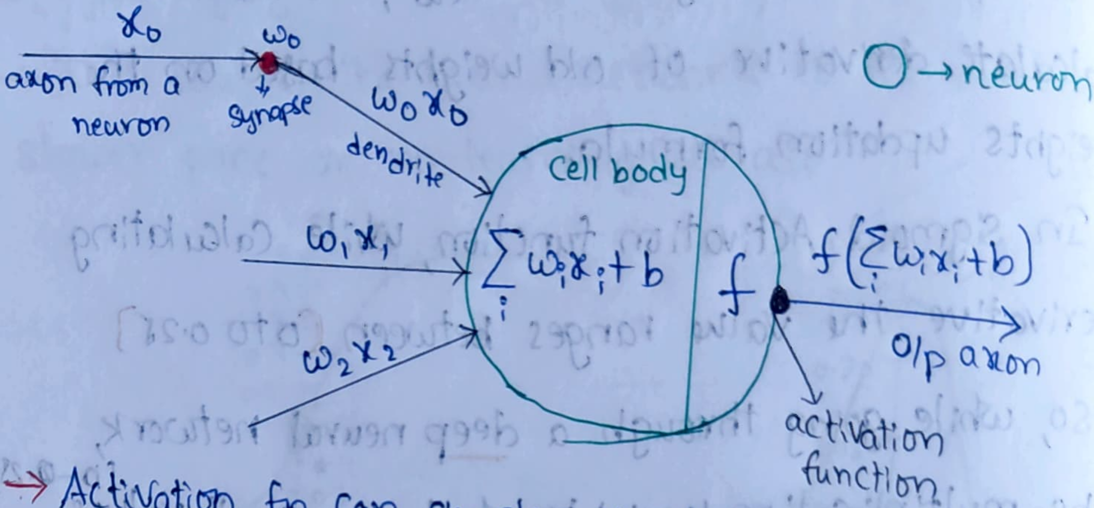
→ Due to the small derivative value, leads to minor updation of weights or New weights and old weights are similar because derivative is neglisble [near to 0].

→ This disadvantage is called vanishing gradient problem. This can be overcome by using of other Activation functions like 'ReLU'.

Activation Function:

Activation functions helps to determine the output of a neural network. These type of functions are attached to each neuron in the network, and determines whether it should be activated or not, based on whether each neuron's input is relevant for the model's prediction.

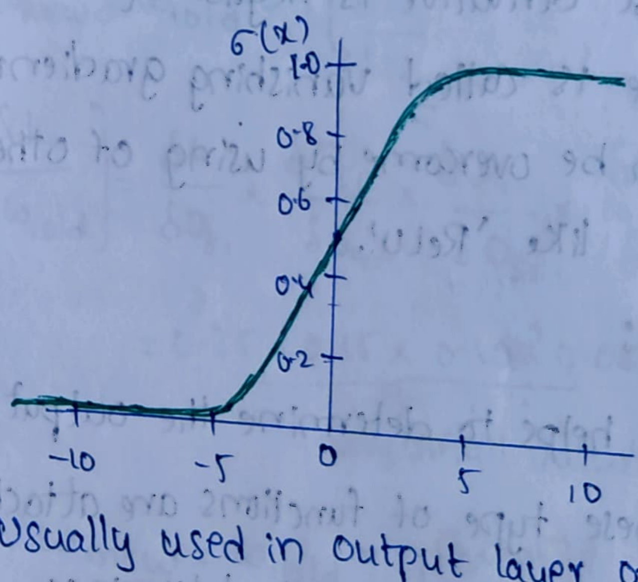
→ Activation function also helps to normalize the o/p of each neuron to a range b/w 1 & 0 or b/w -1 & 1.



→ Activation fn can simple act as a switch that turns the neuron output "on" and "off", depending on a rule or threshold.

Types :-

1. **Sigmoid function:-**



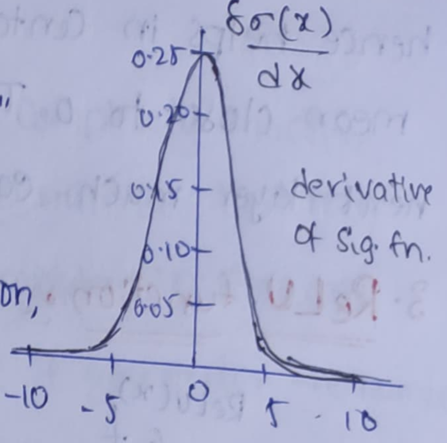
$$\boxed{\sigma(x) = \frac{1}{1+e^{-x}}}$$

Value Range: [0 to 1]

→ Usually used in output layer of a binary classification where result is either 0 or 1, as value for sigmoid function lies between 0 and 1 only so result can be predicted easily to be 1 if value is greater than 0.5 and 0 otherwise.

Advantages:-

1. smooth gradient, preventing "jumps" in output values.

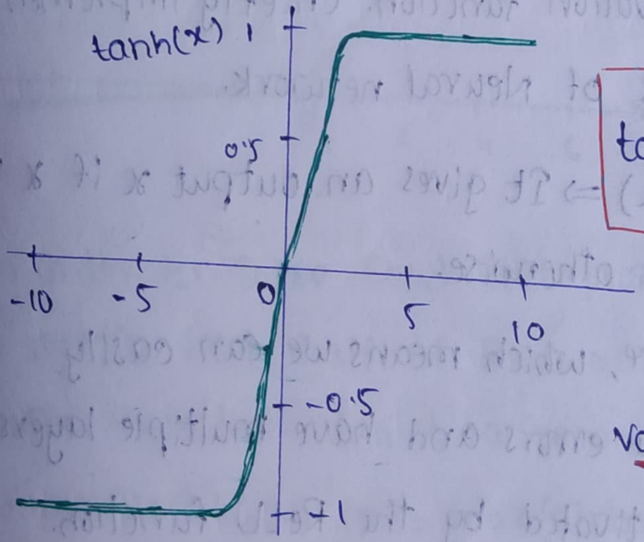


2. Normalizing the o/p of each neuron, clear predictions, i.e very close to 1 or 0.

Disadvantages:

1. Prone to gradient vanishing
2. Function o/p is not zero-centered.
3. Power operations are relatively time consuming.

2. tanh function:-



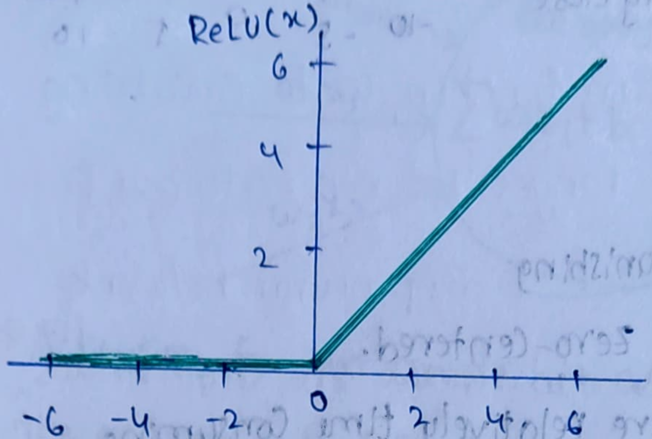
$$\boxed{\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}}$$

Value Range: [-1 to +1]

→ The Tanh function works almost better than sigmoid function.

→ Usually used in hidden layer ~~comes out to be 0 or close to~~ of a neural network as it's values lies between -1 to 1. hence the mean for the hidden layer comes out to be 0 or very close to it, hence helps in centering the data by bringing mean close to 0. This makes learning for the next layer much easier.

3. ReLU function :-



$\boxed{ReLU = max(0, x)}$

Value Range :- [0, inf)

→ ReLU stands for Rectified Linear Unit. It is the most widely used activation function. Chiefly implemented in hidden layers of Neural network.

→ $\boxed{\sigma(x) = max(0, x)}$ ⇒ It gives an output x if x is positive and 0 otherwise.

→ Non-linear nature, which means we can easily backpropagate the errors and have multiple layers of neurons being activated by the ReLU function.

→ When the i/p is +ve, there is no gradient saturation problem.

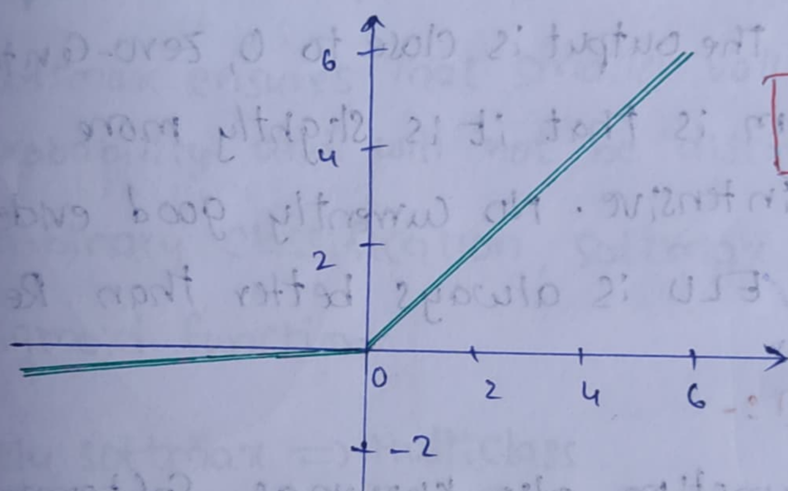
→ ReLU is less computationally expensive than sigmoid and tanh because it involves simpler mathematical operations. At a time only a few neurons are activated making the network sparse making it efficient and easy for computation.

Some disadvantages:-

1. When the i/p is -ve, ReLU is completely inactive, which means that once a (-ve) number entered, ReLU will die. This is not a problem in forward propagation, but in Back prop. if you enter -ve number the gradient will be completely zero.

2. ReLU is not a 0-centric function.

4. Leaky ReLU function:-



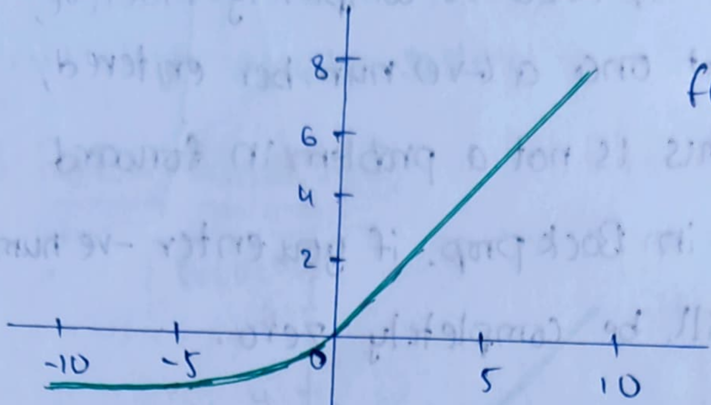
$$\boxed{f(x) = \max(0.01x, x)}$$

→ In order to solve Dead ReLU problem, people proposed to set the first half of ReLU 0.01x instead of 0.

→ Another intuitive idea is a parameter based method, Parameteric ReLU: $f(x) = \max(\text{alpha}\, x, x)$, which alpha can be learned from back propagation.

→ Leaky ReLU has not been fully proved that Leaky ReLU is always better than ReLU.

5. ELU (Exponential Linear Units) function:



$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1), & \text{otherwise} \end{cases}$$

→ ELU is proposed to solve the problems of ReLU.

1. No Dead ReLU issues
2. The mean of the output is close to 0, zero-centered.

→ One small problem is that it is slightly more computationally intensive. No currently good evidence in practice that ELU is always better than ReLU.

Softmax function:-

The Softmax function, also known as Softargmax converts a vector of k real numbers into a probability distribution of k possible outcomes. It is a generalization of the logistic function to multiple dimensions (classes)

→ Prior to apply softmax, some vector components could be -ve, greater than 1 and might not sum to 1. But after applying softmax, each component will be in the interval (0, 1) and the components will add up to 1, so that they can be interpreted as probabilities.

$$\boxed{\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}}$$ for $i = 1, \ldots, K$

$z = (z_1, \ldots z_K) \in \mathbb{R}^K$

→ In simple words, it applies the standard exponential function to each element z_i of the input vector z and normalizes these values by dividing by the sum of all these exponentials; this normalization ensures that the sum of the components of the output vector $\sigma(z)$ is 1.

→ Softmax ensures that smaller values have a smaller probability and will not be discarded directly.

→ In binary classification, softmax is degraded to sigmoid function.

Relu, softmax ⇒ Multiclass

Relu, sigmoid ⇒ Binary

Adam Optimizer :-

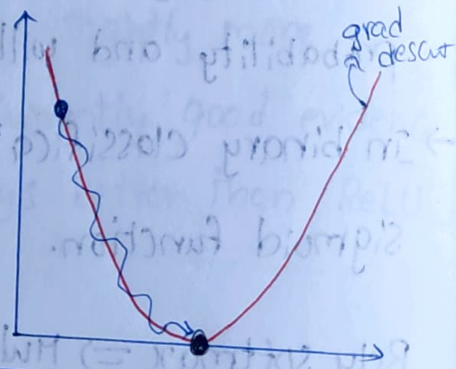
The name adam is derived from adaptive moment estimation. This optimization algorithm is a further extension of stochastic gradient descent to update network weights during training. Unlike maintaining a single learning rate through training in SGD, Adam optimizer updates the learning rate for each network weight individually.

→ The creators of the Adam optimization algorithm know the benefits of AdaGrad and RMSProp algorithms, which are also extensions of the stochastic gradient descent algorithms.

→ Hence the Adam optimizers inherit the features of both Adagrad and RMS prop algorithms.

→ In adam, instead of adapting learning rates based upon the first moment (mean) as in RMS prop, it also uses the second moment of the gradients.

→ We mean the uncentred variance by the second moment of the gradients.



$$w_{new} = w_{old} - \eta \frac{\delta L}{\delta w_{old}}$$

$$\Rightarrow w_t = w_{t-1} - \eta' V_{dw}$$

$$\boxed{V_{dw_t} = \beta \times V_{dw_{t-1}} + (1-\beta)\frac{\delta L}{\delta w_{t-1}}}$$

$$\boxed{\eta' = \frac{\eta}{\sqrt{S_{dw_t} + \epsilon}}}$$

⇒ ① Smoothening
\+
② Learning Rate Adaptive.