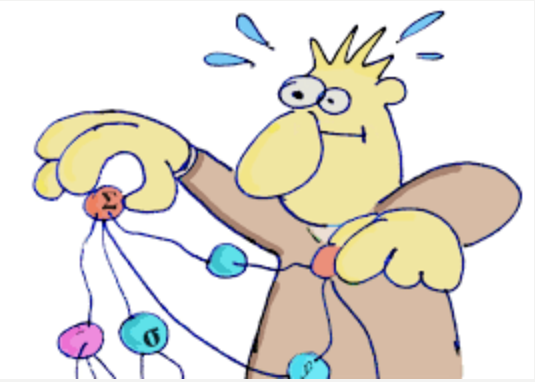
PERCEPTRON & BACKPROPAGATION NETWORKS

DR. KRISHNENDU GUHA
ASSISTANT PROFESSOR (ON CONTRACT)
NATIONAL INSTITUTE OF TECHNOLOGY (NIT), JAMSHEDPUR
EMAIL: KRISHNENDU.CA@NITJSR.AC.IN

PERCEPTRON

- In machine learning, the **perceptron** is an algorithm for supervised learning of binary classifiers.
- A binary classifier is a function which can decide whether or not an input, represented by a vector of numbers, belongs to some specific class.
- It is a type of linear classifier, i.e. a classification algorithm that makes its predictions based on a linear predictor function combining a set of weights with the feature vector.
- A perceptron is a neural network unit (an artificial neuron) that does certain computations to detect features or business intelligence in the input data.

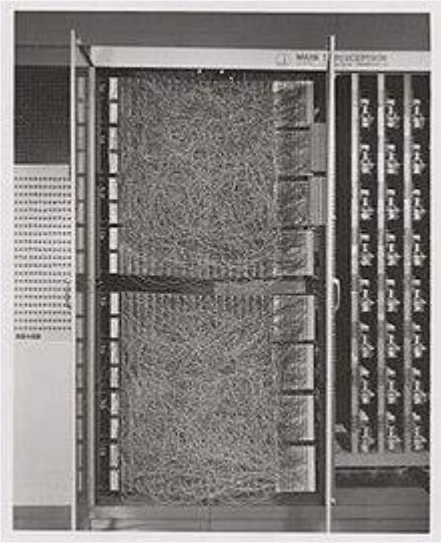


HISTORY

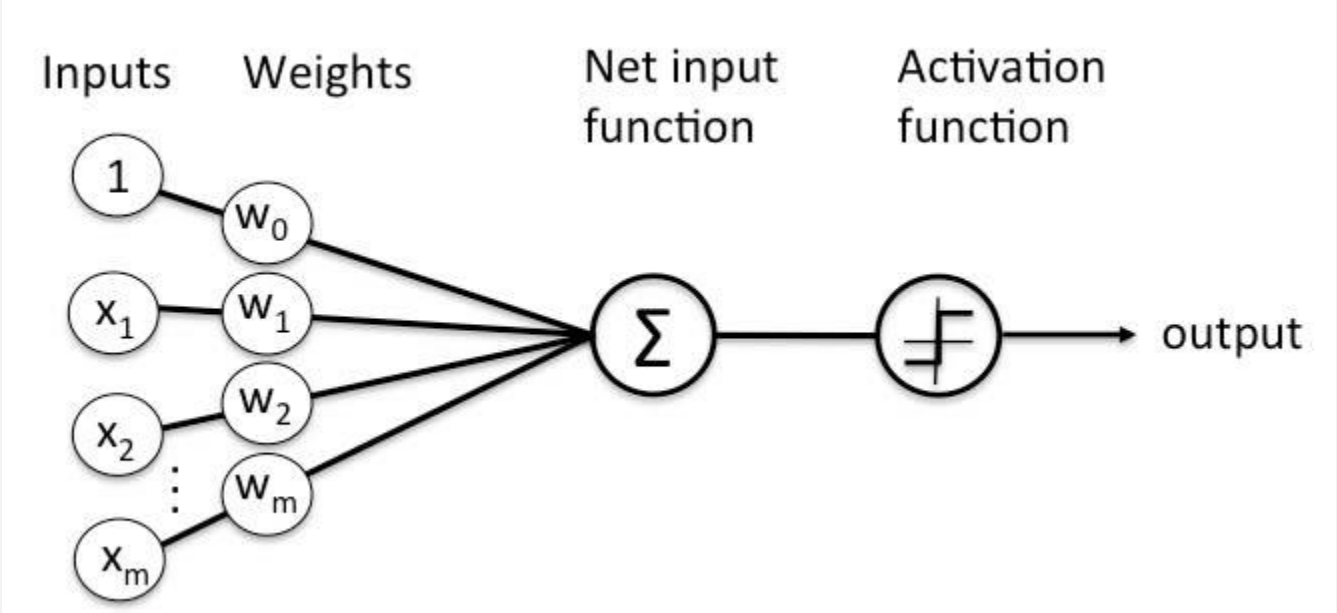
- The perceptron algorithm was invented in 1958 at the Cornell Aeronautical Laboratory by Frank Rosenblatt,[3] funded by the United States Office of Naval Research

Frank Rosenblatt
(1928-1971)

- The perceptron was intended to be a machine, rather than a program, and while its first implementation was in software for the IBM 704, it was subsequently implemented in custom-built hardware as the "Mark 1 perceptron".
- This machine was designed for image recognition

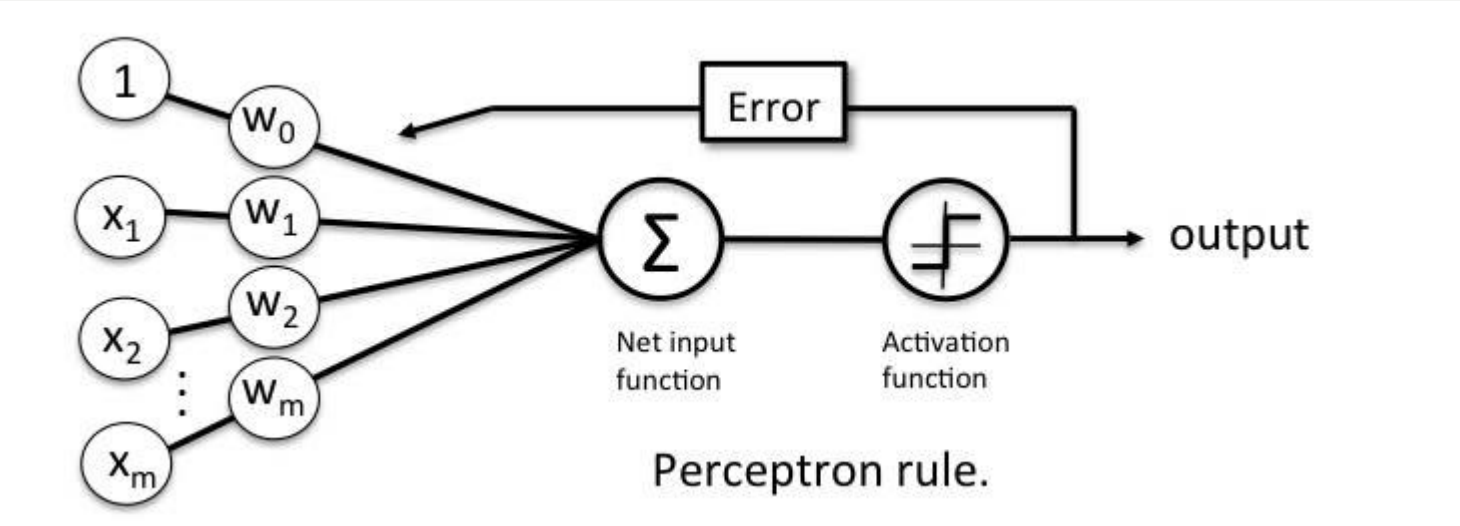


- A Perceptron is an algorithm for supervised learning of binary classifiers.
- This algorithm enables neurons to learn and processes elements in the training set one at a time.
- There are two types of Perceptrons: Single layer and Multilayer.
- Single layer Perceptrons can learn only linearly separable patterns.
- Multilayer Perceptrons or feedforward neural networks with two or more layers have the greater processing power.



PERCEPTRON LEARNING RULE

- Perceptron Learning Rule states that the algorithm would automatically learn the optimal weight coefficients.
- The input features are then multiplied with these weights to determine if a neuron fires or not
- The Perceptron receives multiple input signals, and if the sum of the input signals exceeds a certain threshold, it either outputs a signal or does not return an output.
- In the context of supervised learning and classification, this can then be used to predict the class of a sample.



PERCEPTRON FUNCTION

- Perceptron is a function that maps its input “x,” which is multiplied with the learned weight coefficient; an output value ”f(x)”is generated

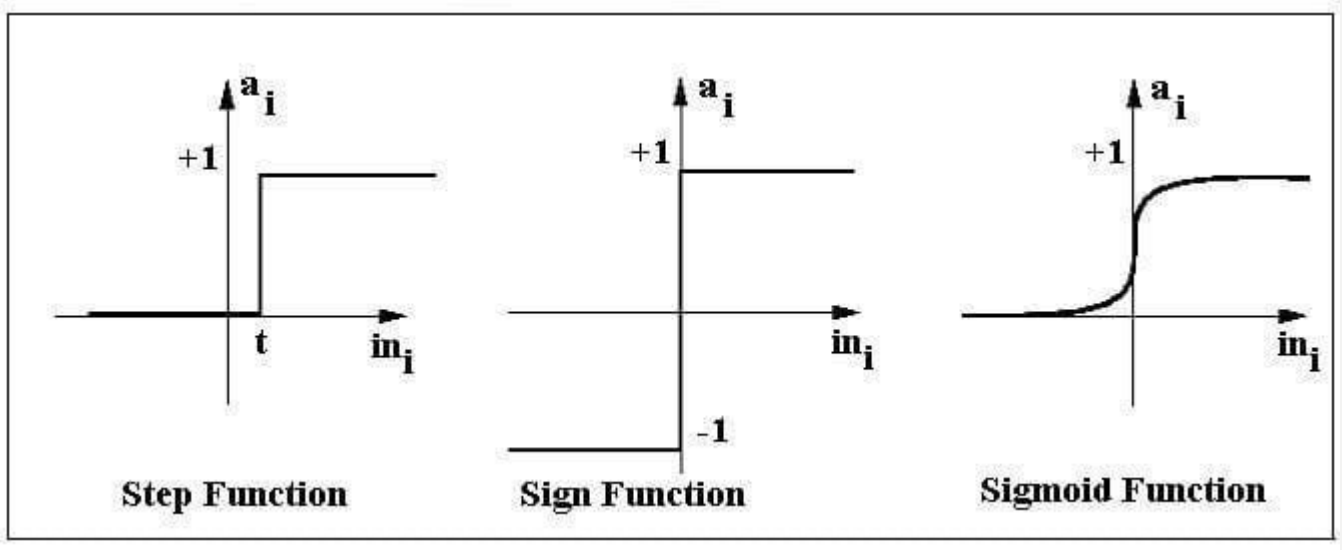
$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\sum_{i=1}^{m} w_i x_i$$

- In the equation given above:
- “w” = vector of real-valued weights
- “b” = bias (an element that adjusts the boundary away from origin without any dependence on the input value)
- “x” = vector of input x values
- “m” = number of inputs to the Perceptron
- The output can be represented as “1” or “0.” It can also be represented as “1” or “-1” depending on which activation function is used.

ACTIVATION FUNCTION

- The activation function applies a step rule (convert the numerical output into +1 or -1) to check if the output of the weighting function is greater than zero or not



For example:
If ∑ wixi> 0 => then final output “o” = 1 (issue bank loan)
Else, final output “o” = -1 (deny bank loan)

IMPLEMENT BASIC LOGIC GATES WITH PERCEPTRON

- AND
- If the two inputs are TRUE (+1), the output of Perceptron is positive, which amounts to TRUE.
- This is the desired behavior of an AND gate

x1= 1 (TRUE), x2= 1 (TRUE)

w0 = -.8, w1 = 0.5, w2 = 0.5

=> o(x1, x2) => -.8 + 0.5*1 + 0.5*1 = 0.2 > 0

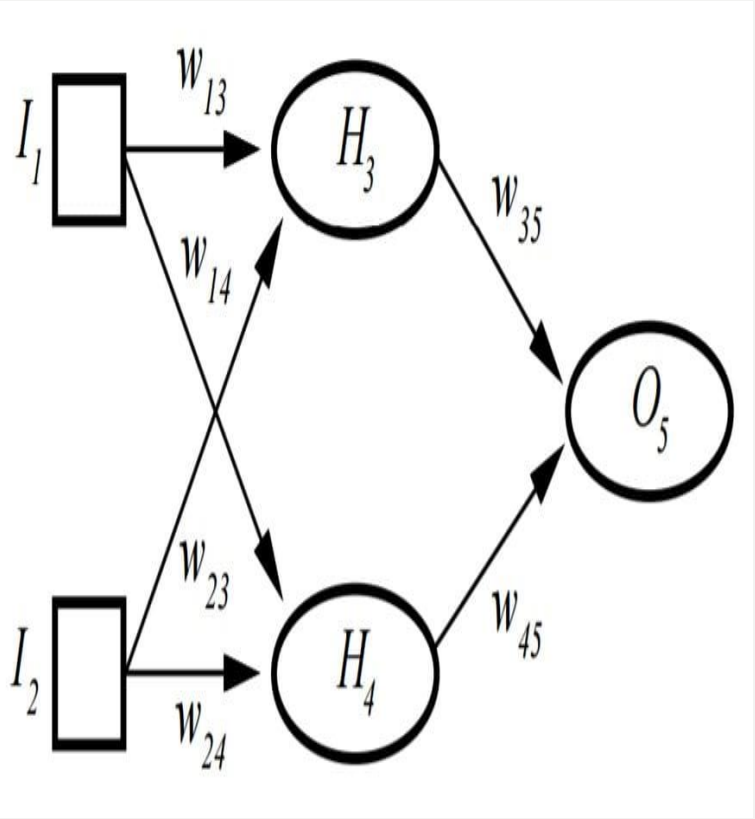
- OR
- If either of the two inputs are TRUE (+1), the output of Perceptron is positive, which amounts to TRUE.
- This is the desired behavior of an OR gate

x1 = 1 (TRUE), x2 = 0 (FALSE)

w0 = -.3, w1 = 0.5, w2 = 0.5

=> o(x1, x2) => -.3 + 0.5*1 + 0.5*0 = 0.2 > 0

- XOR Gate with Neural Networks
- Unlike the AND and OR gate, an XOR gate requires an intermediate hidden layer for preliminary transformation in order to achieve the logic of an XOR gate



- An XOR gate assigns weights so that XOR conditions are met.
- It cannot be implemented with a single layer Perceptron and requires Multi-layer Perceptron or MLP.
- H represents the hidden layer, which allows XOR implementation.
- I1, I2, H3, H4, O5are 0 (FALSE) or 1 (TRUE)
- t3= threshold for H3; t4= threshold for H4; t5= threshold for O5
- H3= sigmoid (I1*w13+ I2*w23−t3); H4= sigmoid (I1*w14+ I2*w24−t4)
- O5= sigmoid (H3*w35+ H4*w45−t5);

LEARNING ALGORITHM

We first define some variables:

- r is the learning rate of the perceptron. Learning rate is between 0 and 1, larger values make the weight changes more volatile.
- $y = f(\mathbf{z})$ denotes the *output* from the perceptron for an input vector $\mathbf{z}$.
- $D = \{(\mathbf{x}_1, d_1), \ldots, (\mathbf{x}_s, d_s)\}$ is the *training set* of s samples, where:
  - $\mathbf{x}_j$ is the n-dimensional input vector.
  - d_j is the desired output value of the perceptron for that input.

We show the values of the features as follows:

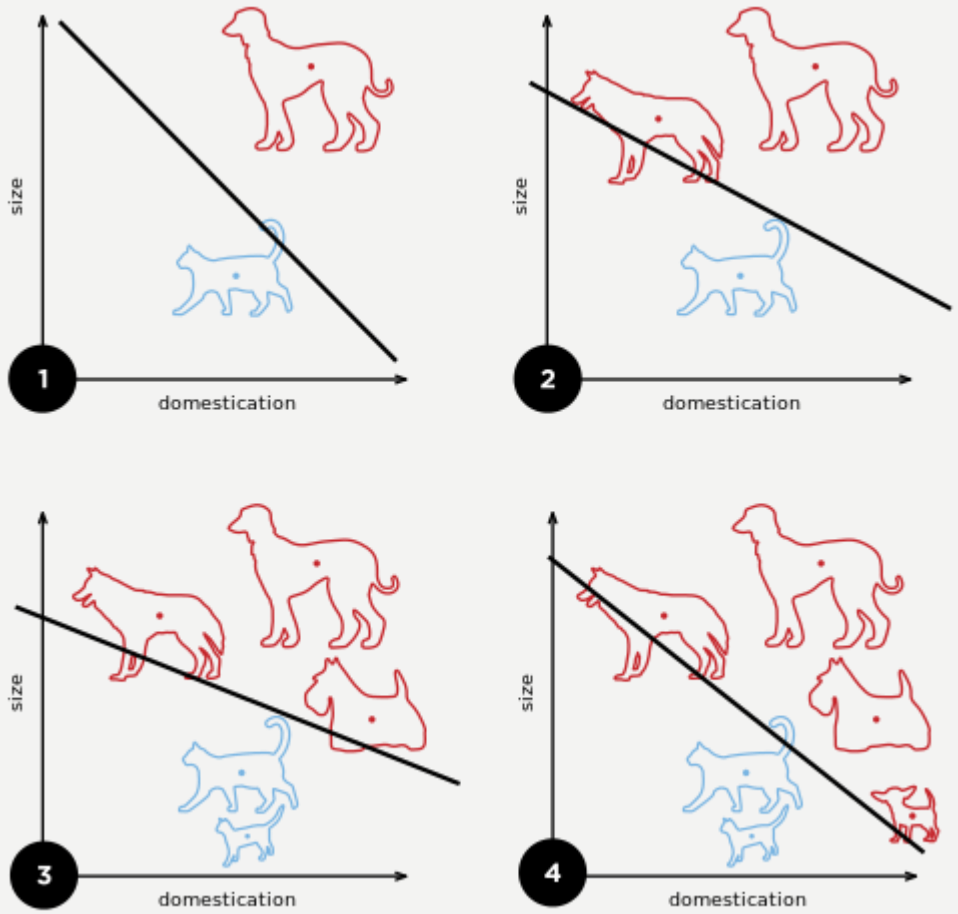
- $x_{j,i}$ is the value of the ith feature of the jth training *input vector*.
- $x_{j,0} = 1$.

To represent the weights:

- w_i is the ith value in the *weight vector*, to be multiplied by the value of the ith input feature.
- Because $x_{j,0} = 1$, the w_0 is effectively a bias that we use instead of the bias constant b.

To show the time-dependence of $\mathbf{w}$, we use:

- $w_i(t)$ is the weight i at time t.



A diagram showing a perceptron updating its linear boundary as more training examples are added

1. Initialize the weights and the threshold. Weights may be initialized to 0 or to a small random value. In the example below, we use 0.
2. For each example j in our training set D, perform the following steps over the input $\mathbf{x}_j$ and desired output d_j:

    a. Calculate the actual output:

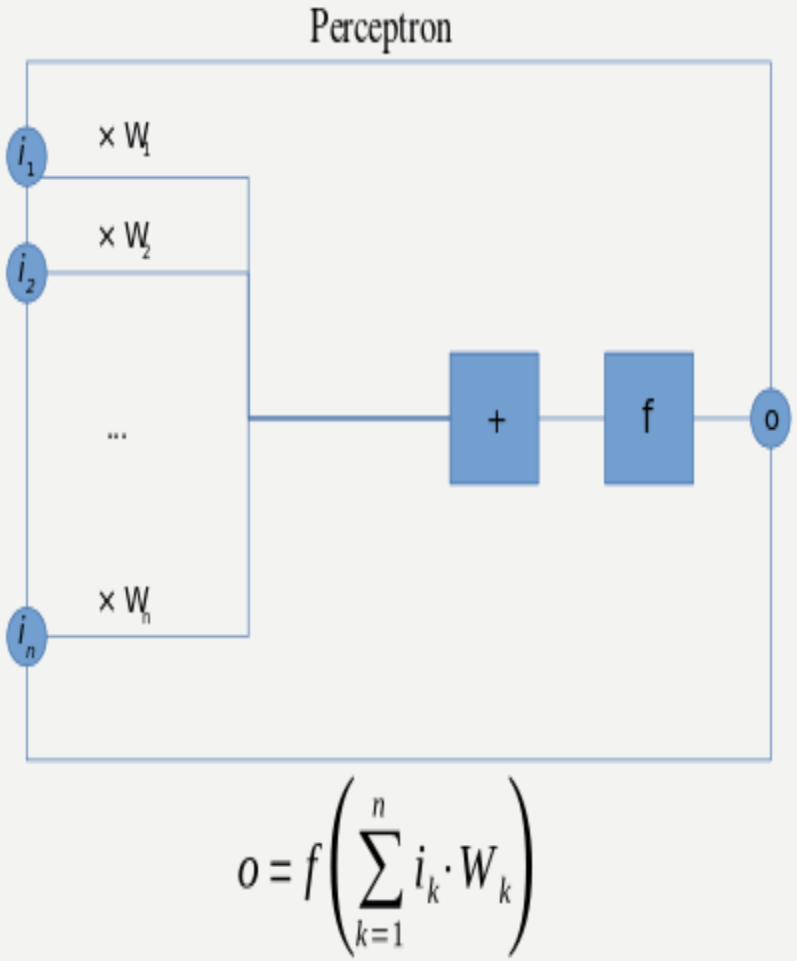
    $$\begin{aligned} y_j(t) &= f[\mathbf{w}(t) \cdot \mathbf{x}_j] \\ &= f[w_0(t)x_{j,0} + w_1(t)x_{j,1} + w_2(t)x_{j,2} + \cdots + w_n(t)x_{j,n}] \end{aligned}$$

    b. Update the weights:

    $w_i(t+1) = w_i(t) + r \cdot (d_j - y_j(t))x_{j,i}$, for all features $0 \leq i \leq n$, r is the learning rate.

3. For offline learning, the second step may be repeated until the iteration error $\frac{1}{s}\sum_{j=1}^{s}|d_j - y_j(t)|$ is less than a user-specified error threshold γ, or a predetermined number of iterations have been completed, where s is again the size of the sample set.

The algorithm updates the weights after steps 2a and 2b. These weights are immediately applied to a pair in the training set, and subsequently updated, rather than waiting until all pairs in the training set have undergone these steps.

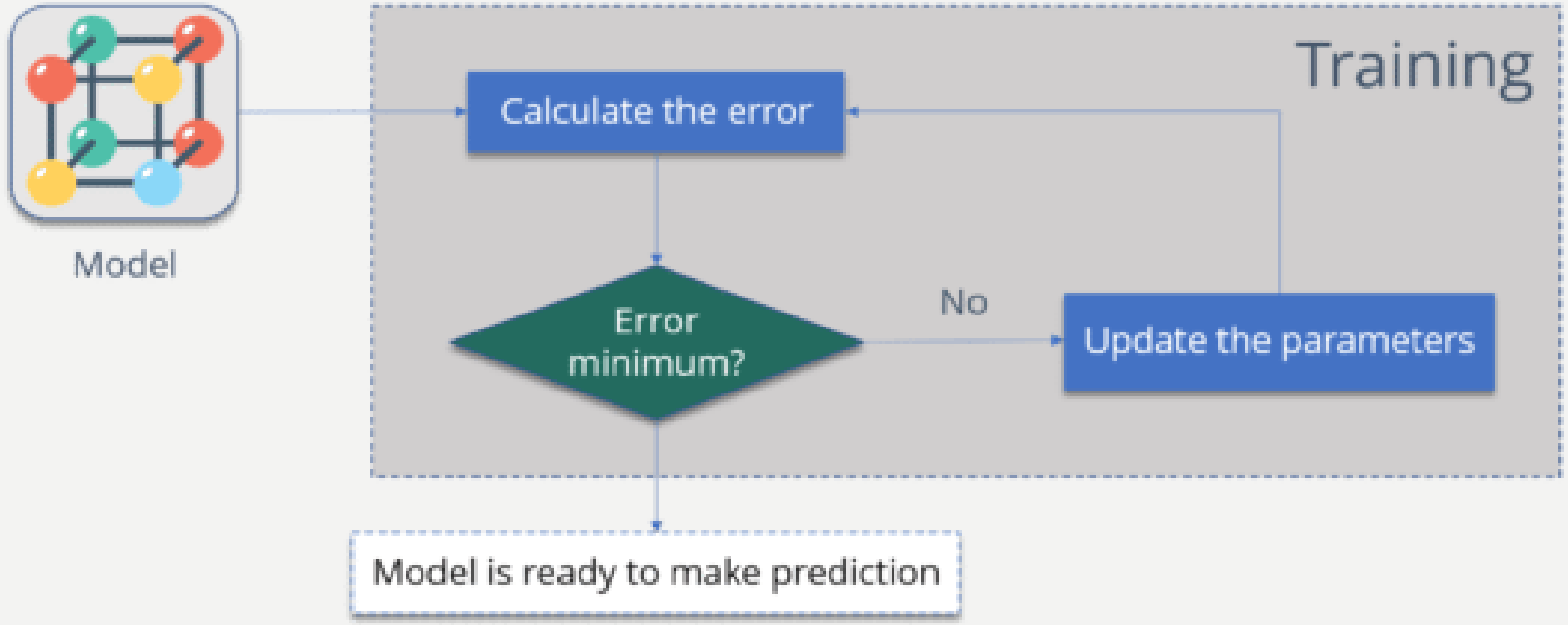


$$o = f\left(\sum_{k=1}^{n} i_k \cdot W_k\right)$$

BACKPROPAGATION NETWORK

- Backpropagation is a supervised learning algorithm, for training Multi-layer Perceptrons (Artificial Neural Networks).
- **Why We Need Backpropagation?**
- While designing a Neural Network, in the beginning, we initialize weights with some random values or any variable for that fact.
- So, it's not necessary that whatever weight values we have selected will be correct, or it fits our model the best.
- our model output is way different than our actual output i.e. the error value is huge.
- Now, how will you reduce the error?
- Basically, what we need to do, we need to somehow explain the model to change the parameters (weights), such that error becomes minimum.
- Let's put it in an another way, we need to train our model.
- One way to train our model is called as Backpropagation.

- **Calculate the error** – How far is your model output from the actual output.
- **Minimum Error** – Check whether the error is minimized or not.
- **Update the parameters** – If the error is huge then, update the parameters (weights and biases). After that again check the error. Repeat the process until the error becomes minimum.
- **Model is ready to make a prediction** – Once the error becomes minimum, you can feed some inputs to your model and it will produce the output.



- The Backpropagation algorithm looks for the minimum value of the error function in weight space using a technique called the delta rule or gradient descent.
- The weights that minimize the error function is then considered to be a solution to the learning problem

Consider the below table:

Input	Desired Output
0	0
1	2
2	4

Now the output of your model when 'W" value is 3:

Input	Desired Output	Model output (W=3)
0	0	0
1	2	3
2	4	6

Notice the difference between the actual output and the desired output:

Input	Desired Output	Model output (W=3)	Absolute Error	Square Error
0	0	0	0	0
1	2	3	1	1
2	4	6	2	4

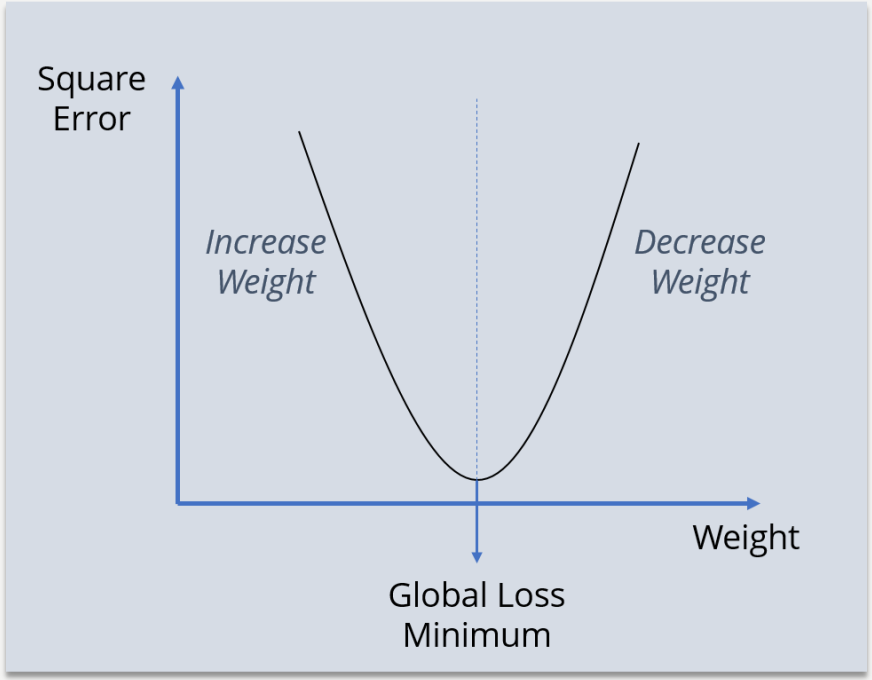
Let's change the value of 'W'. Notice the error when 'W' = '4'

Input	Desired Output	Model output (W=3)	Absolute Error	Square Error	Model output (W=4)	Square Error
0	0	0	0	0	0	0
1	2	3	1	1	4	4
2	4	6	2	4	8	16

Now if you notice, when we increase the value of 'W' the error has increased. So, obviously there is no point in increasing the value of 'W' further. But, what happens if I decrease the value of 'W'?

Input	Desired Output	Model output (W=3)	Absolute Error	Square Error	Model output (W=2)	Square Error
0	0	0	0	0	0	0
1	2	3	2	4	3	0
2	4	6	2	4	4	0

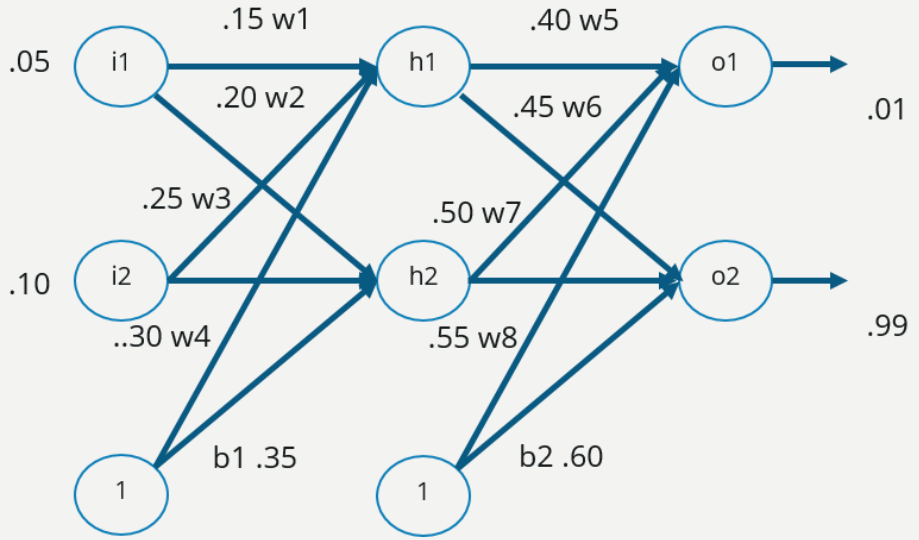
- we are trying to get the value of weight such that the error becomes minimum.
- Basically, we need to figure out whether we need to increase or decrease the weight value.
- Once we know that, we keep on updating the weight value in that direction until error becomes minimum.
- You might reach a point, where if you further update the weight, the error will increase.
- At that time you need to stop, and that is your final weight value.



- We need to reach the ‘Global Loss Minimum’.
- This is nothing but Backpropagation

HOW BACKPROPAGATION WORKS (VIA MATHEMATICAL DESCRIPTION)

- Consider the below Neural Network:



- The above network contains the following:
- two inputs
- two hidden neurons
- two output neurons
- two biases

- Below are the steps involved in Backpropagation:
- Step – 1: Forward Propagation
- Step – 2: Backward Propagation
- Step – 3: Putting all the values together and calculating the updated weight value

Step – 1: Forward Propagation

We will start by propagating forward

Net Input For h1:

net h1 = w1*i1 + w2*i2 + b1*1 → net h1 = 0.15*0.05 + 0.2*0.1 + 0.35*1 = 0.3775

Output Of h1:

out h1 = $1/1+e^{-\text{net h1}}$ → $1/1+e^{.3775}$ = 0.593269992

Output Of h2:

out h2 = 0.596884378

We will repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.

Output For o1:

net o1 = w5*out h1 + w6*out h2 + b2*1 → 0.4*0.593269992 + 0.45*0.596884378 + 0.6*1 = 1.105905967

Out o1 = $1/1+e^{-\text{net o1}}$ → $1/1+e^{-1.105905967}$ = 0.75136507

Output For o2:

Out o2 = 0.772928465

Now, let's see what is the value of the error:

Error For o1:

$E_{o1} = \Sigma 1/2(target - output)^2$ → ½ $(0.01 - 0.75136507)^2 = 0.274811083$

Error For o2:

$E_{o2} = 0.023560026$

Total Error:

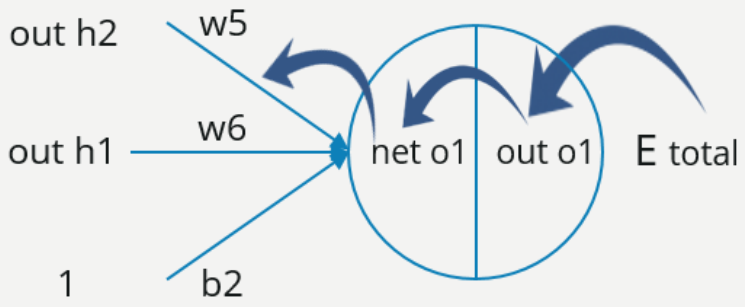
$E_{total} = E_{o1} + E_{o2}$ → $0.274811083 + 0.023560026 = 0.298371109$

Step – 2: Backward Propagation

Now, we will propagate backwards. This way we will try to reduce the error by changing the values of weights and biases.

Consider W5, we will calculate the rate of change of error w.r.t change in weight W5.

$$\frac{\delta Etotal}{\delta w5} = \frac{\delta Etotal}{\delta out\ o1} * \frac{\delta out\ o1}{\delta net\ o1} * \frac{\delta net\ o1}{\delta w5}$$



Since we are propagating backwards, first thing we need to do is, calculate the change in total errors w.r.t the output O1 and O2.

$$E_{total} = 1/2(\text{target o1} - \text{out o1})^2 + 1/2(\text{target o2} - \text{out o2})^2$$

$$\frac{\delta Etotal}{\delta out\ o1} = -(\text{target o1} - \text{out o1}) = -(0.01 - 0.75136507) = 0.74136507$$

Now, we will propagate further backwards and calculate the change in output O1 w.r.t to its total net input.

$$\text{out o1} = 1/1+e^{-neto1}$$

$$\frac{\delta out\ o1}{\delta net\ o1} = \text{out o1 (1 - out o1)} = 0.75136507\ (1 - 0.75136507) = 0.186815602$$

Let's see now how much does the total net input of O1 changes w.r.t W5?

$$\text{net o1} = w5 * \text{out h1} + w6 * \text{out h2} + b2 * 1$$

$$\frac{\delta net\ o1}{\delta w5} = 1 * \text{out h1}\ w5^{(1-1)} + 0 + 0 = 0.593269992$$

Step – 3: Putting all the values together and calculating the updated weight value

Now, let's put all the values together:

$$\frac{\delta Etotal}{\delta w5} = \frac{\delta Etotal}{\delta out\ o1} * \frac{\delta out\ o1}{\delta net\ o1} * \frac{\delta net\ o1}{\delta w5} \rightarrow 0.082167041$$

Let's calculate the updated value of W5:

$$w5^{+} = w5 - n\frac{\delta Etotal}{\delta w5} \rightarrow w5^{+} = 0.4 - 0.5 * 0.082167041$$

Updated w5 → 0.35891648

- Similarly, we can calculate the other weight values as well.
- After that we will again propagate forward and calculate the output. Again, we will calculate the error.
- If the error is minimum we will stop right there, else we will again propagate backwards and update the weight values.
- This process will keep on repeating until error becomes minimum.

BACKPROPAGATION ALGORITHM

Backpropagation Algorithm:

initialize network weights (often small random values)
  do
forEach training example named ex
    prediction = neural-net-output(network, ex) *// forward pass*
    actual = teacher-output(ex)
    compute error (prediction - actual) at the output units
    compute {displaystyle Delta w_{h}} for all weights from hidden layer to output layer *// backward pass*
    compute {displaystyle Delta w_{i}} for all weights from input layer to hidden layer *// backward pass continued*
    update network weights *// input layer not modified by error estimate*
until all examples classified correctly or another stopping criterion satisfied
return the network