

MLP



THIS BOOK

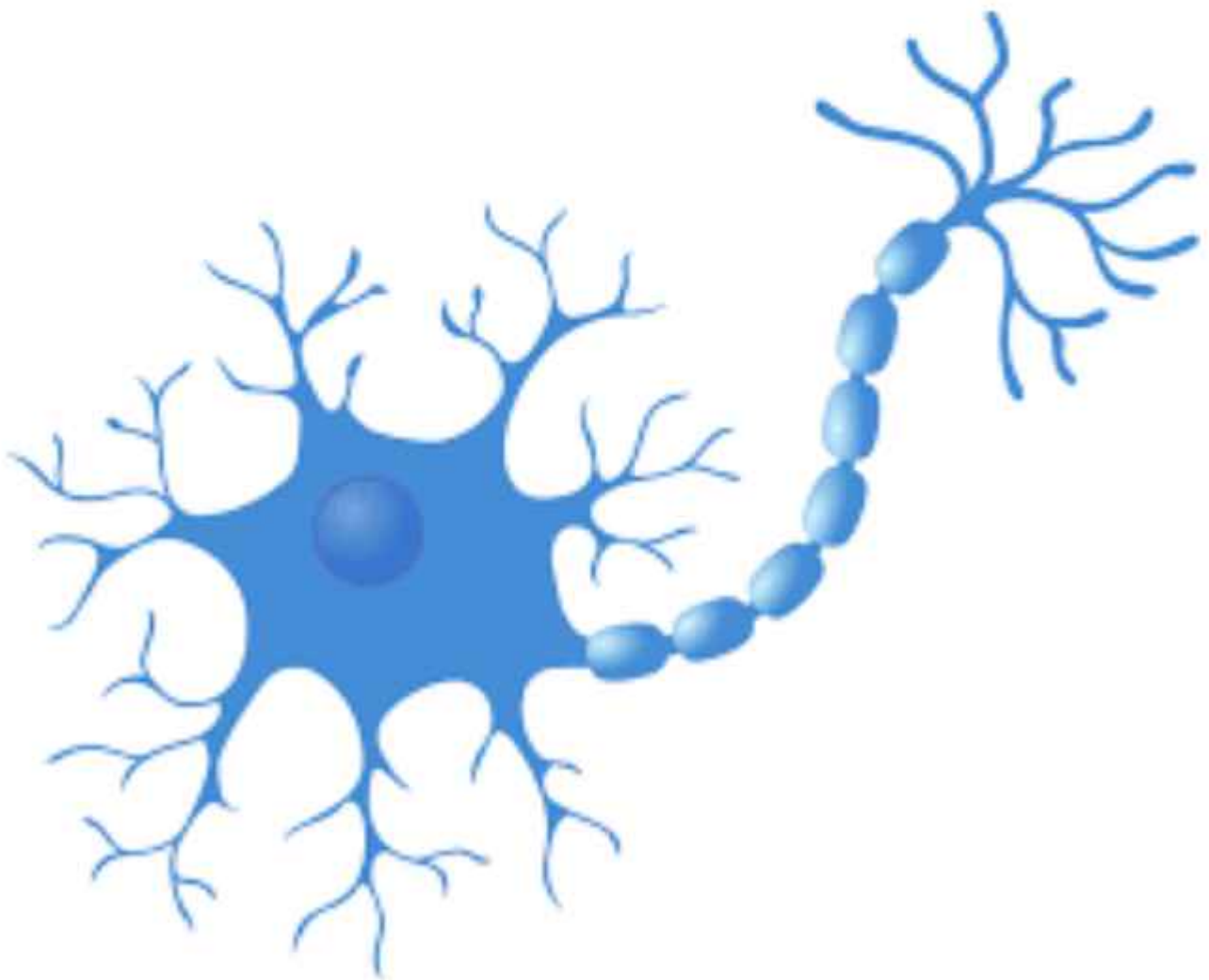
This book is written (typed) by Ari, who hails from the South and has keen interests in Computer Science, Biology, and Tamil Literature. Occasionally, he updates his website, where you can reach out to him.

<https://arihara-sudhan.github.io>



THE HISTORY

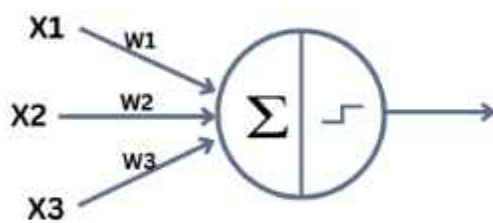
Neuron is the fundamental unit of the nervous system (Another guy I know is Nephron, the fundamental unit of kidney), specialized for receiving, processing, and transmitting information throughout the body.



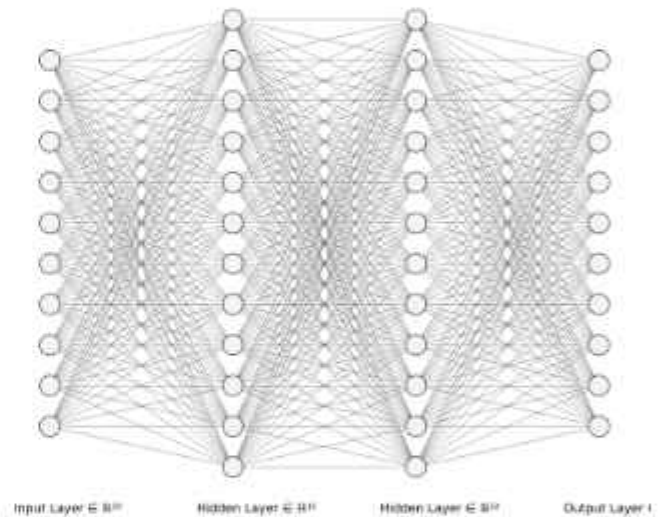
A scientist named Camilo Golgi proposed the Reticular Theory, which suggested that the nervous system is a continuous network, with neurons connected directly to one another. This theory was supported by staining experiments conducted by Santiago Ramon, who further advanced its concepts. However, with the advent of advanced microscopes, researchers discovered the existence of synapses (the gaps between neurons) which made clear that the nervous system is not continuous. A new understanding of neural function emerged, leading to the Neuron Doctrine, proposed by Santiago. This doctrine argued that neurons are individual, discrete units.

PERCEPTRON

The perceptron is a foundational concept in Artificial Neural Networks. A perceptron is the simplest type of neural network model with only one layer and one neuron (or node).



Single-layer perceptron

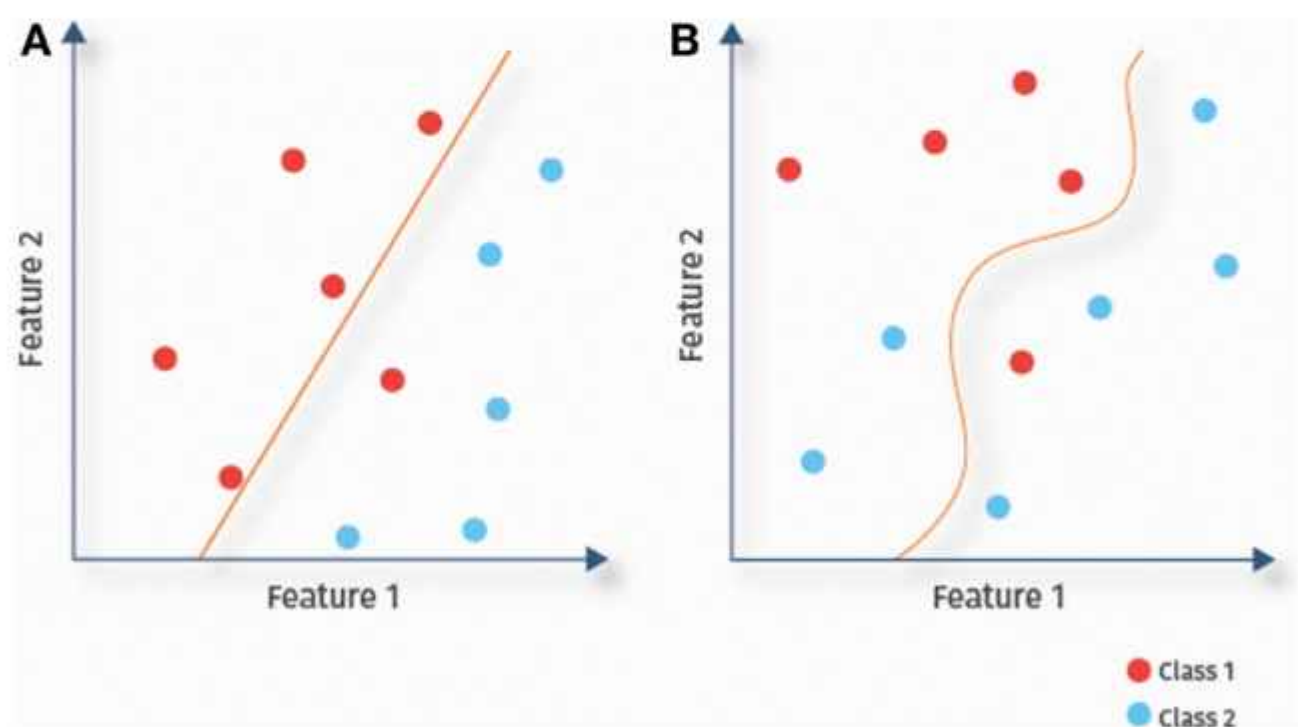


Multi-layer perceptron

It's a binary classifier that makes a decision based on input features and their corresponding weights.

Mathematically, it calculates the weighted sum of the inputs, and if

the result exceeds a certain threshold, it outputs one class (say, 1); otherwise, it outputs the other class (0). Meanwhile, A Multilayer Perceptron (MLP) is a neural network with one or more layers of neurons between the input and output layers, known as hidden layers.



MLPs are capable of learning complex patterns and performing non-linear classification, which single-layer perceptrons cannot achieve. A single-layer perceptron can only handle linearly separable data (data that can be separated by a straight line or a hyperplane in higher dimensions). If the data isn't linearly separable, a single-layer perceptron won't find a solution, no matter how many training iterations we use. Meanwhile, a multilayer perceptron (MLP) with one or more hidden layers can model non-linear decision boundaries. (All because of hidden layers and non-linear activation functions)

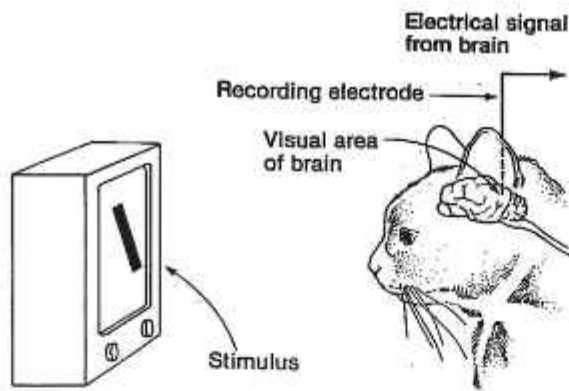
The more complex the network, the better it can capture intricate patterns and relationships in data.

Network	Error	Layers
AlexNet ^[13]	16.0%	8
ZFNet ^[14]	11.2%	8
VGGNet ^[15]	7.3%	19
GoogLeNet ^[16]	6.7%	22
MS ResNet ^[17]	3.6%	152!!

More layers and neurons require more computational power, memory, and storage, especially during training. Complex networks often need powerful GPUs or specialized hardware for efficient processing.

CATS TO CNN

Hubel and Wiesel cat experiment showed how the brain processes visual information by studying how cats respond to different shapes and lines. They found that certain brain cells in the cat's visual cortex react only to specific shapes, like lines or edges in a particular direction (e.g., vertical or horizontal). Some cells were very picky, responding only to certain orientations of lines. Their experiment showed that the brain builds up complex images in layers. First, it detects simple things like edges, then combines those edges into more complex shapes and patterns.

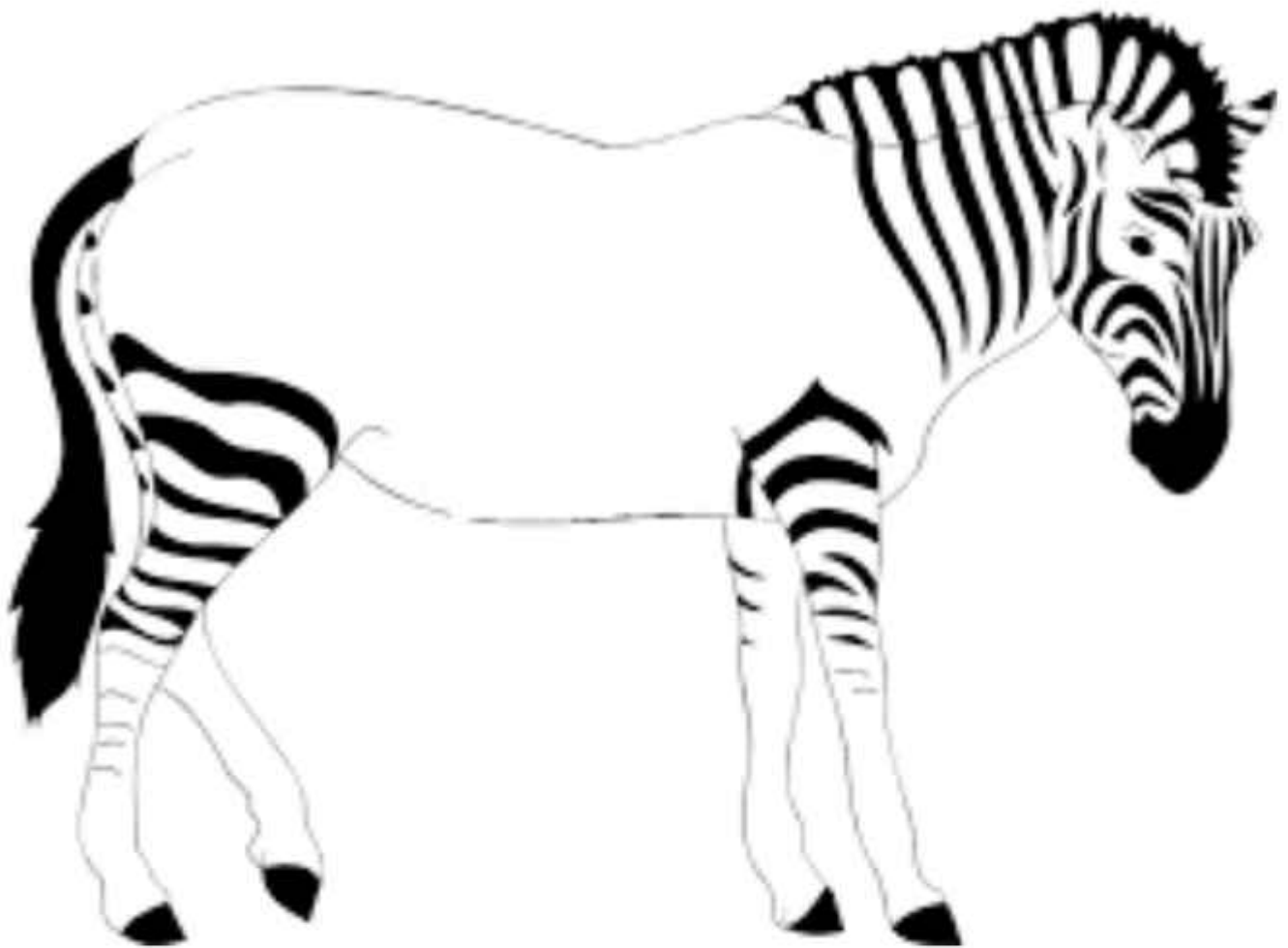


D. Hubel



T. Wiesel

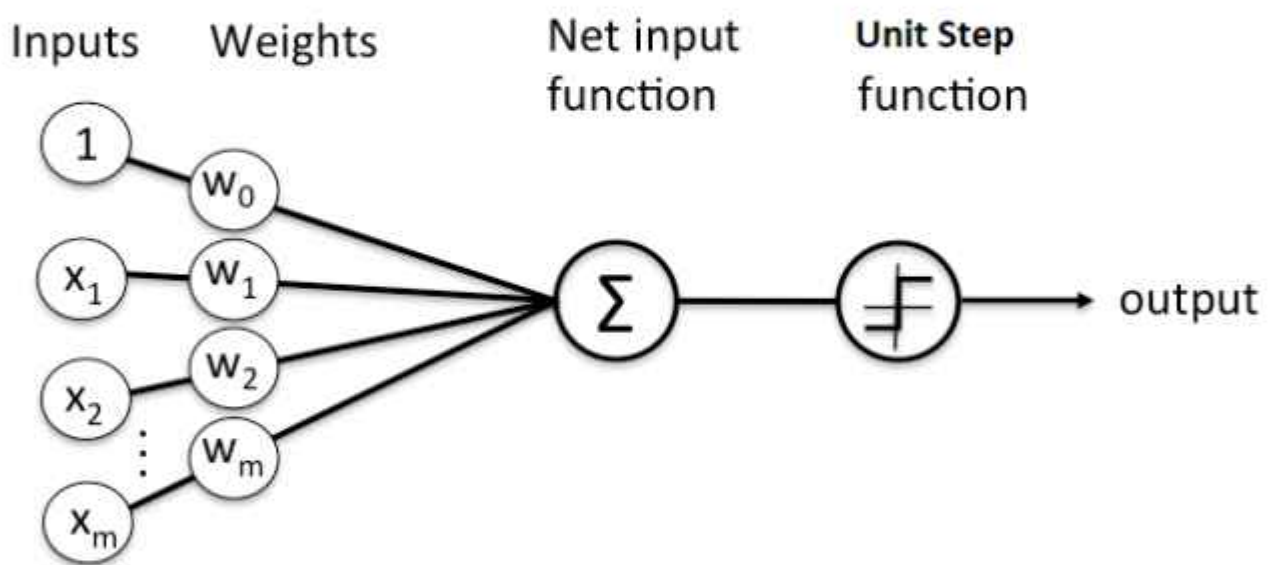
A receptive field is the specific area of the visual field where the presence of a stimulus (like light or a shape) will trigger a response in that particular neuron. Each neuron responds to visual stimuli within a defined area. For example, a neuron might only respond when a vertical line appears in a specific location in the visual field. If we remove those neurons, one cannot recognize vertical lines.



Maybe, if we have a neural network without neurons which fire on vertical lines, we couldn't see the vertical stripes on the body of Zebra (*Equus quagga*).

THE SLP

A Single Layer Perceptron is one of the simplest type of feed forward network. It consists of a single layer of output neurons directly connected to a layer of input nodes, with no hidden layers between them.



The input layer receives data features, each connected to the output neuron with an associated weight that determines its influence.

$$y = wx + b$$

These inputs are multiplied by their respective weights and summed up in a summation function. The result is passed through an activation function, typically a step or sign function, to produce the final output. Adjustments to weights during training help the perceptron learn and improve its predictions over time.

$$z = \sum_{j=1}^m x_j w_j = w^T x$$

We will repeat updating the weights until the model converges (There is adequate learning).

Now, how do we update the weight?

It is done like,

$$\mathbf{W}_{\text{new}} = \mathbf{W}_{\text{old}} + \mathbf{Error} * \mathbf{Input}$$

Here, there will be something called learning rate, which I don't want to include for now. Weights are updated using this **Perceptron Learning Rule**, which adjusts weights to minimize classification errors. This process of updating the weights is repeated over multiple inputs until the perceptron produces the correct output for all training samples or until the error reaches an acceptable level. Look at the following Python simple script

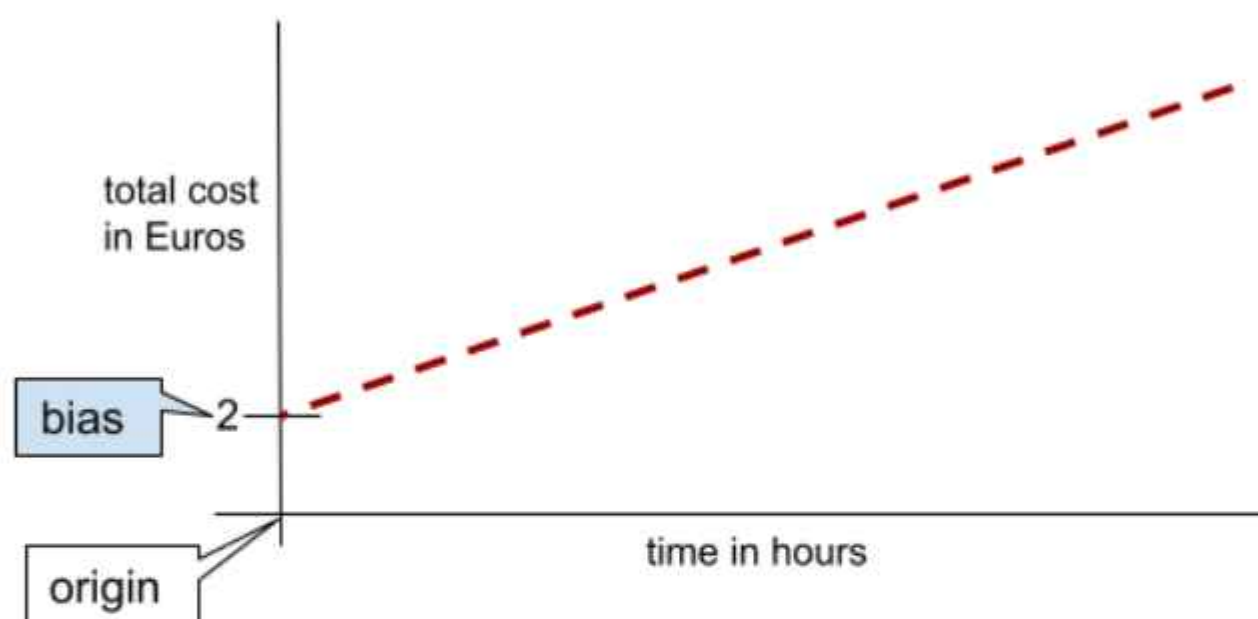
which learns to output 0 when the inputs are [1,0]

```
2  w1, w2 = 0, 0
3  b = 0
4  inputs = [1, 0]
5  target = 0
6
7  def activation(z):
8      return 1 if z >= 0 else 0
9
10 max_iterations = 10
11 for i in range(max_iterations):
12     z = w1 * inputs[0] + w2 * inputs[1] + b
13     output = activation(z)
14     error = target - output
15     print(f"Loss: {error} | w1: {w1} | w2: {w2} | b: {b}")
16     if error == 0:
17         break
18     w1 += error * inputs[0]
19     w2 += error * inputs[1]
20     b += error
21
22 x1 = int(input("Enter Input Feature1: "))
23 x2 = int(input("Enter Input Feature2: "))
24 output = activation(w1*x1 + w2*x2 + b)
25 print(f"OUTPUT: {output}")
```

The bias and weights are adjusted until the network converges.

```
Loss: -1 | w1: 0 | w2: 0 | b: 0
Loss: 0 | w1: -1 | w2: 0 | b: -1
Enter Input Feature1: 1
Enter Input Feature2: 0
OUTPUT: 0
```

Bias acts like an offset, shifting the activation function and giving the model the ability to adjust its decision boundary independently of the inputs.



Bias allows the boundary to be shifted away from the origin, giving the perceptron more freedom to find the optimal boundary.

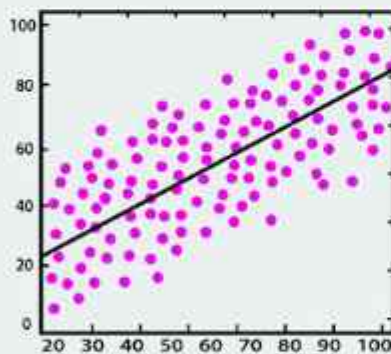
Now, let's code for OR Operation.

```
2  x = [[1,1], [1,0], [0,1], [0,0]]
3  y = [1, 1, 1, 0]
4
5  def activate(wsum):
6      return 0 if wsum < 0 else 1
7
8  def train(EPOCHS = 5):
9      (w1, w2, BIAS) = (0, 0, 0)
10     for epoch in range(EPOCHS):
11         for i,data in enumerate(x):
12             wsum = data[0]*w1 + data[1]*w2 + BIAS
13             y_hat = activate(wsum)
14             ERR = y[i] - y_hat
15             print(f"Error: ${ERR}")
16             if ERR != 0:
17                 w1 = w1 + ERR*data[0]
18                 w2 = w2 + ERR*data[1]
19                 BIAS = BIAS + ERR
20     return w1, w2, BIAS
21
22 w1, w2, BIAS = train()
23 model = lambda x1, x2 : activate(w1*x1 + w2*x2 + BIAS)
24
25 print(model(0,0)) #0
26 print(model(0,1)) #1
27 print(model(1,0)) #1
28 print(model(1,0)) #1
```

The model is a Single Layer Perceptron with two inputs and one output.

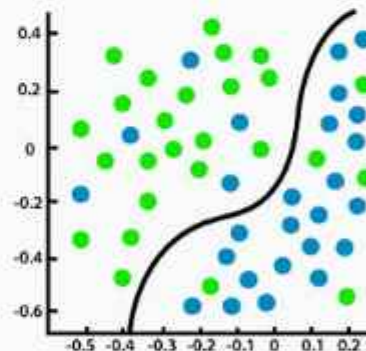
THE MLP

A Multi-Layer Perceptron (MLP) is a type of artificial neural network that consists of multiple layers of nodes (neurons). It is a supervised learning algorithm that can be used for both classification and regression tasks.



Regression

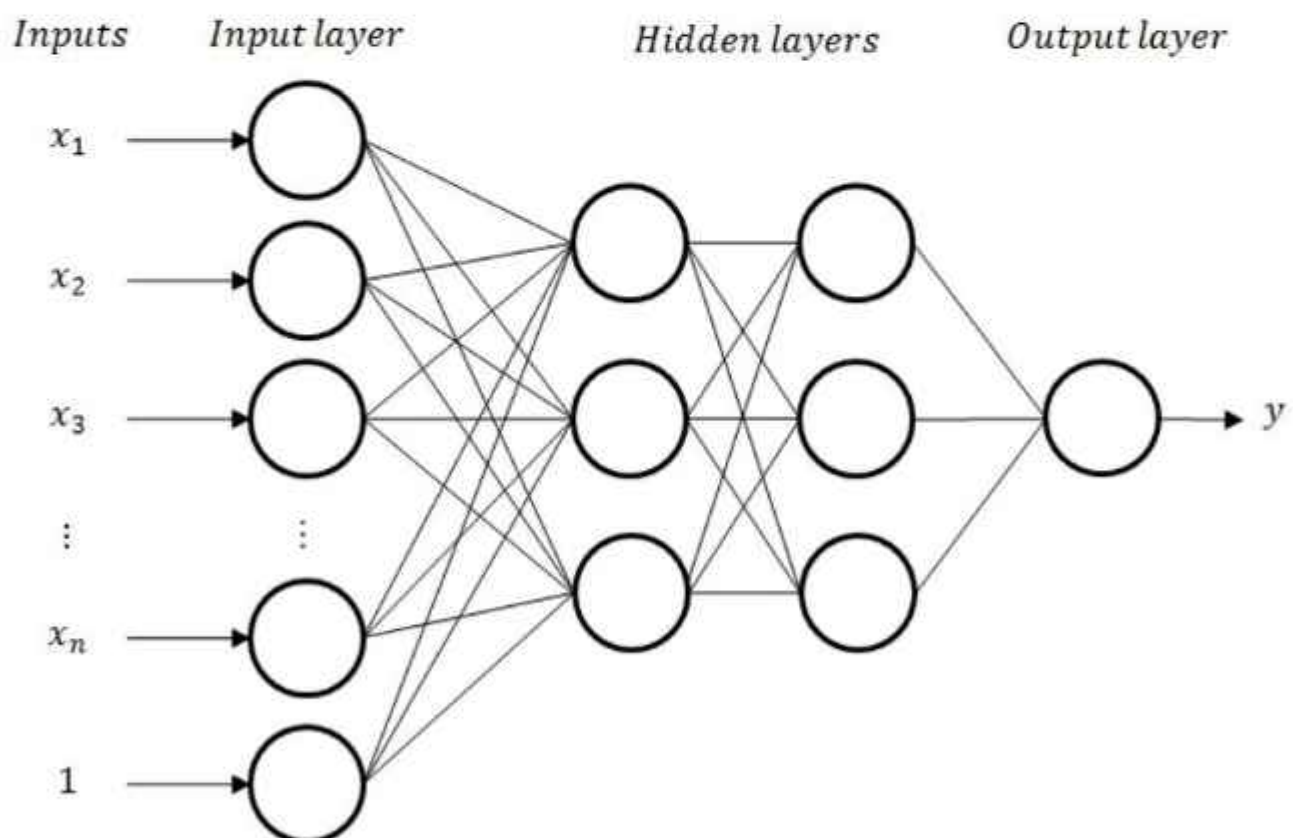
versus



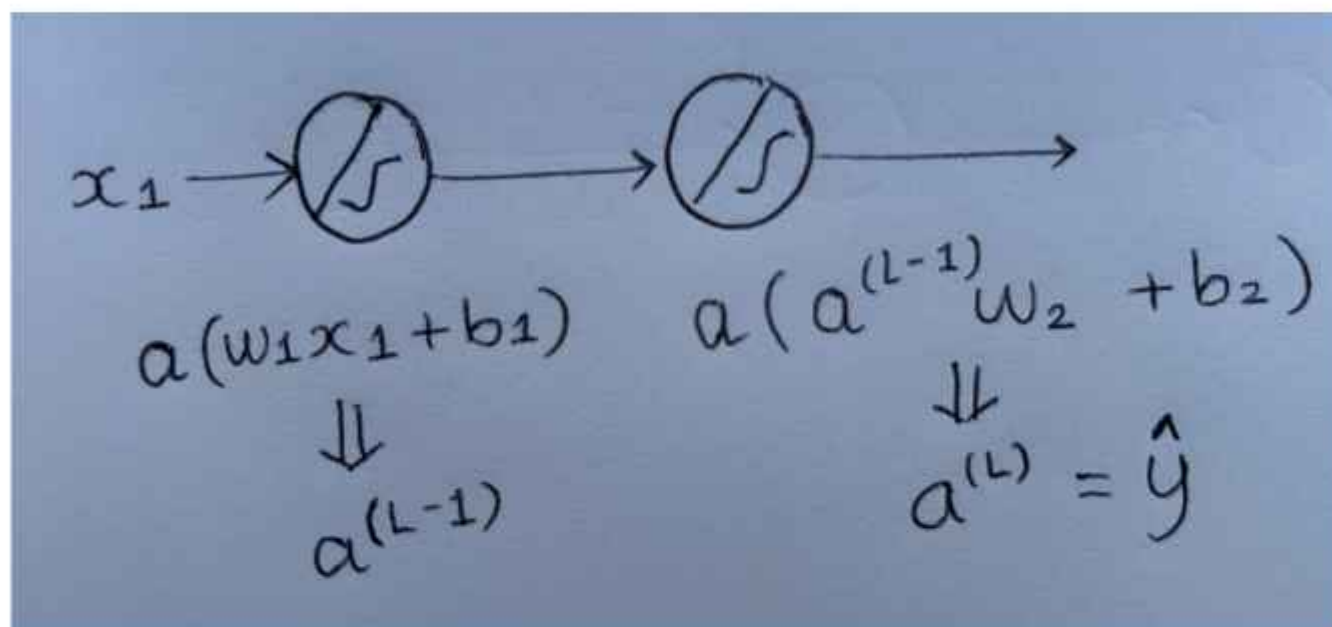
Classification

Each neuron in the MLP processes input data through a weighted sum followed by a

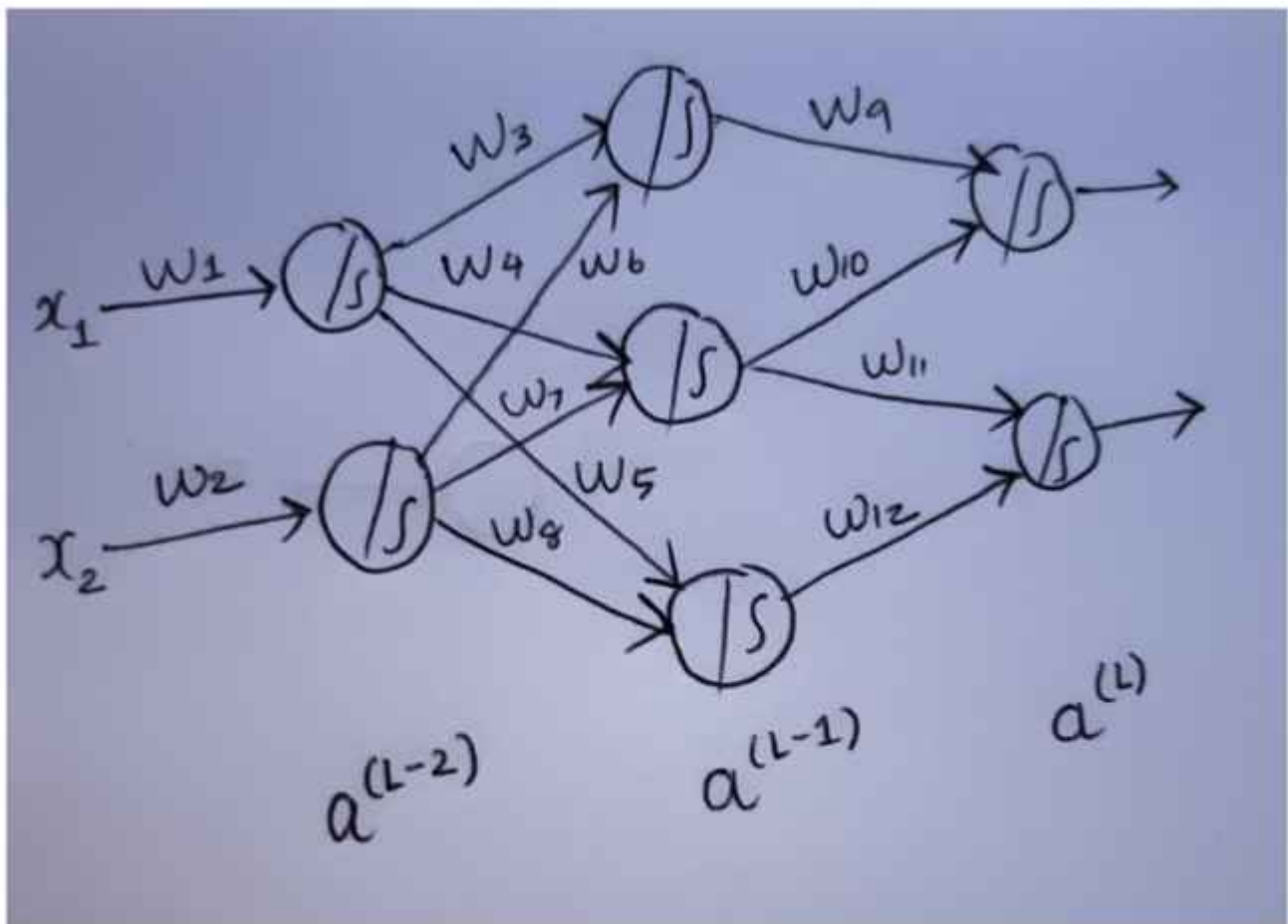
non-linear activation function, enabling the network to learn complex patterns. During training, the model uses forward propagation to compute outputs and back-propagation to adjust weights based on the error measured by a loss function.



Now, how will the things happen here? Let's consider the following sequence of neurons.



Here, the activation of one layer is fed to the another layer as an input. This is how, forward propagation happens. As we can see, the predicted output \hat{y} depends on the weights of each preceding layers. Now, let's draw another neural network as shown below.



Now, we can understand more how it works... This network can have more training parameters (weights and biases) to represent complex patterns. But, how are we going to make it learn? Can we do the weight adjustment we did in SLP? I mean,

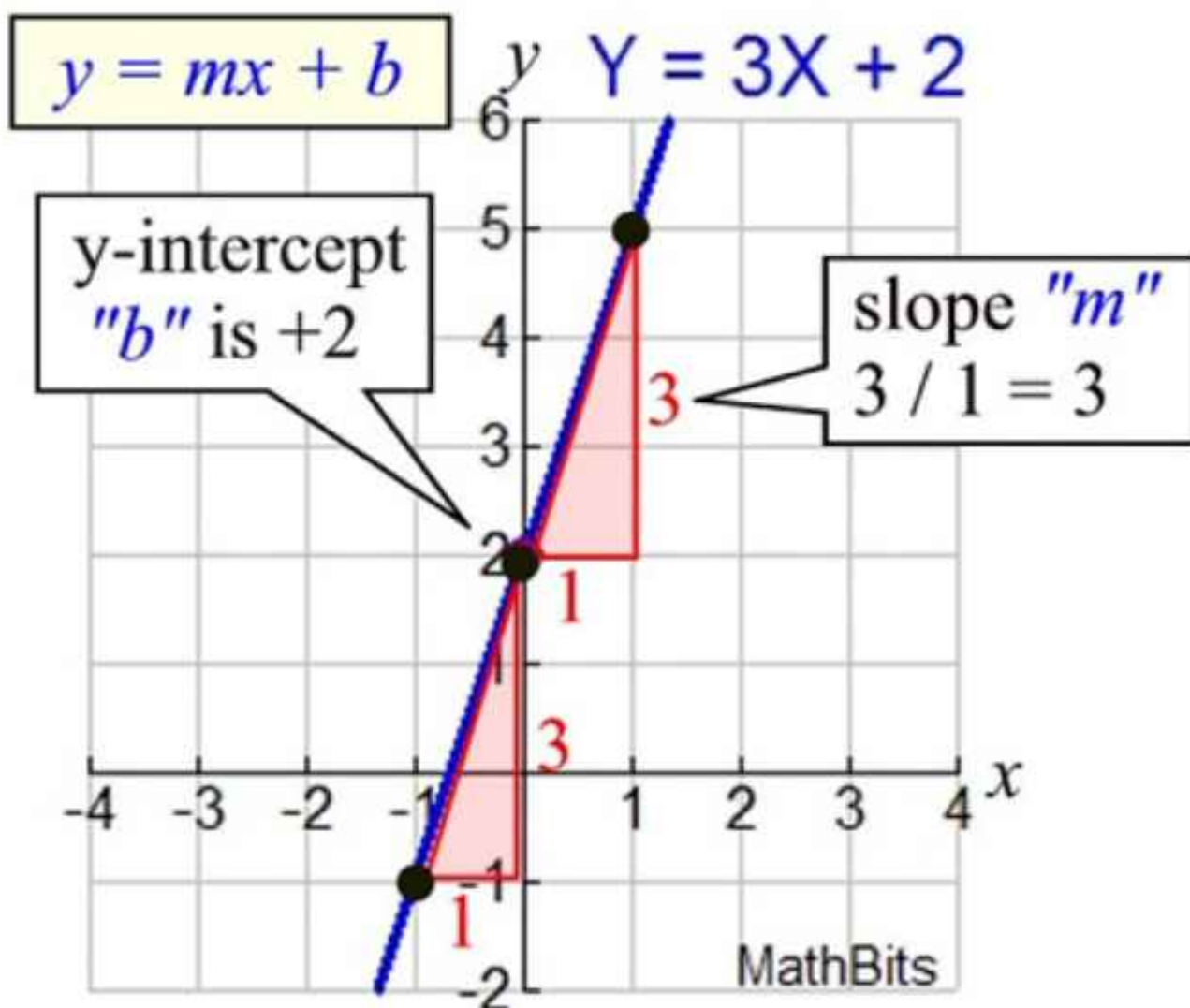
$$W_{\text{new}} = W_{\text{old}} + \text{ERR} * \text{Input}$$

Will it work? NO!

In an SLP, there is only one layer of weights, so adjusting weights based on the direct error (ERR) is straightforward. However, in an MLP with hidden layers, the error is not directly observable for the weights in the hidden layers. Therefore, we need a way to propagate the error backward from the output layer through the hidden layers.

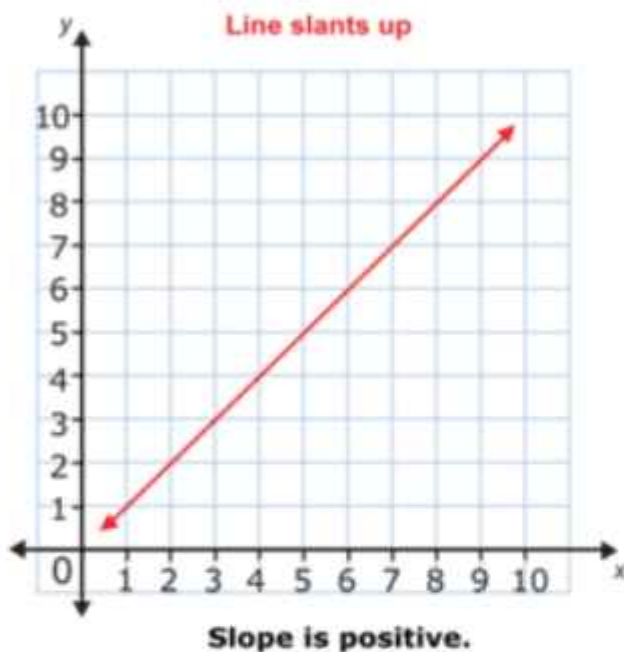
Instead of using a simple update rule, we calculate the gradient of the loss function with respect to each weight. This gradient indicates the direction and magnitude by which each weight should be adjusted to minimize the loss.

Let's learn some basics. What does the slope of a function exactly define?



A positive slope means the function is increasing (upward movement as we move right).

A negative slope means the function is decreasing (downward movement as you move right).

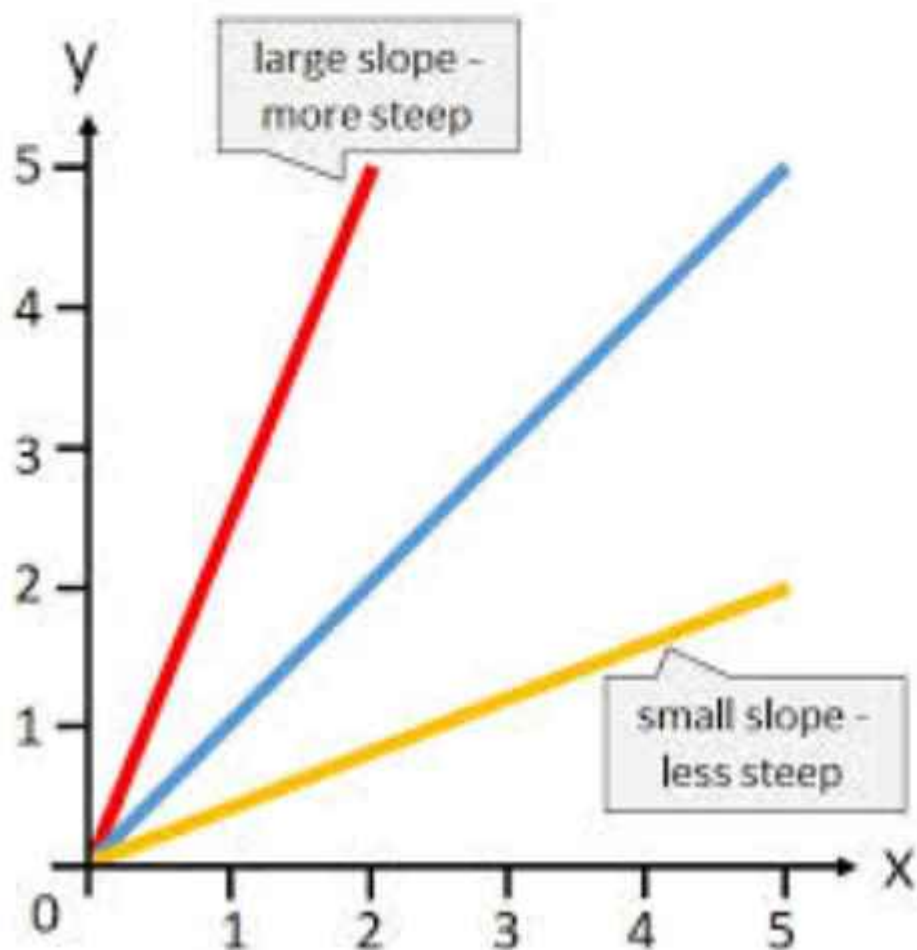


A zero slope indicates a flat spot, where the function is neither increasing nor decreasing, which often corresponds to a local maximum, minimum, or saddle point.



The slope of a horizontal line is always 0.

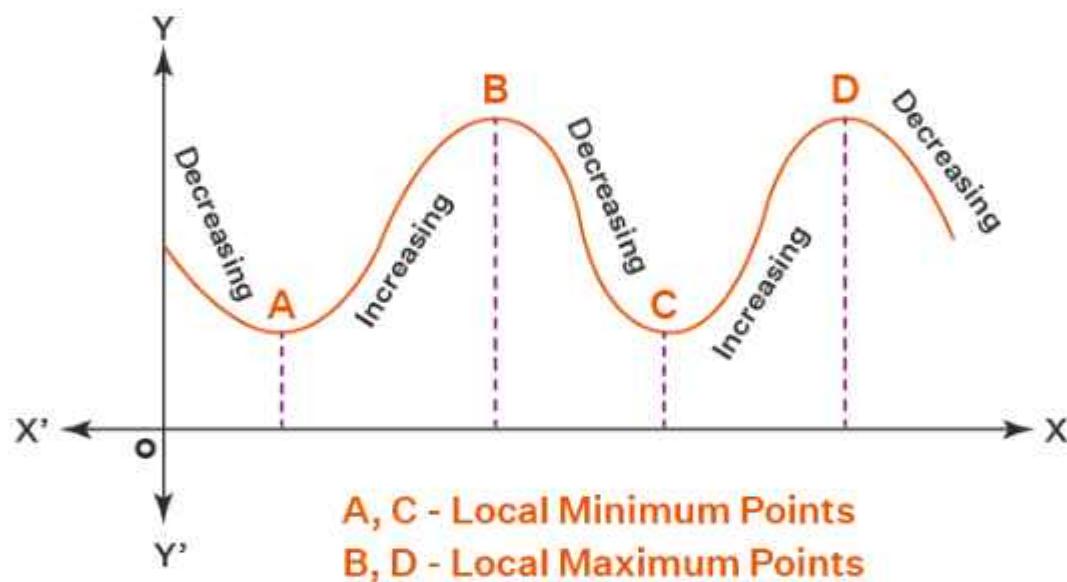
Steepness describes the intensity of the slope at a given point and reflects how quickly or slowly the function rises or falls. A steeper slope means the function value is changing rapidly with a small change in x , resulting in a more vertical rise or fall. A gentler slope means the function value changes more slowly, resulting in a flatter appearance.



The slope can be written as dy/dx in calculus. It is used to represent the slope of any function at any point, capturing how y changes with x , even when the rate of change is not constant. At a local maximum (a peak) or a local minimum (a valley), the slope of the function at that point is zero. This means, $dy/dx=0$. The zero slope indicates that the function's rate of change has temporarily "paused" or flattened out, meaning there's no increase or decrease in y at that exact point. These are also called critical points.

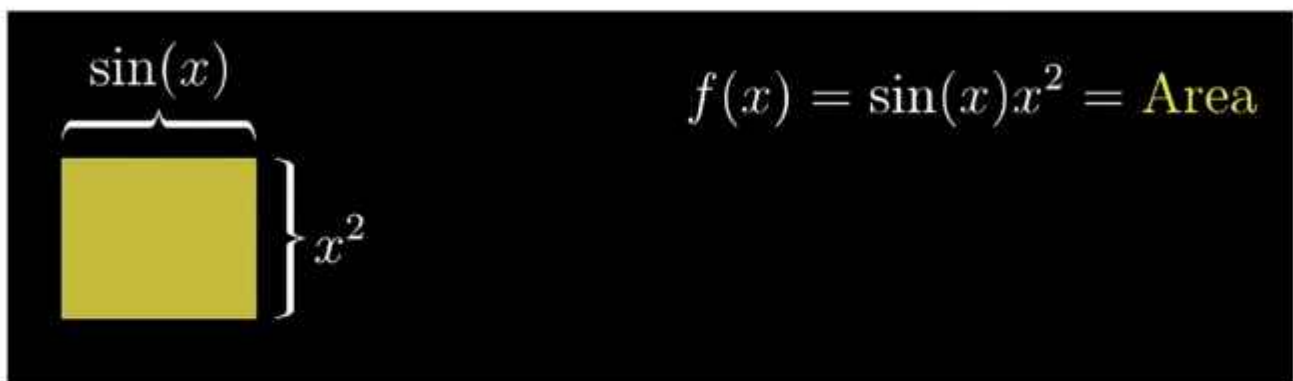
If the second derivative d^2y/dx^2 at this point is positive, then the point is a local minimum. If d^2y/dx^2 is negative, then the point is a local maximum.

Local Maximum and Minimum

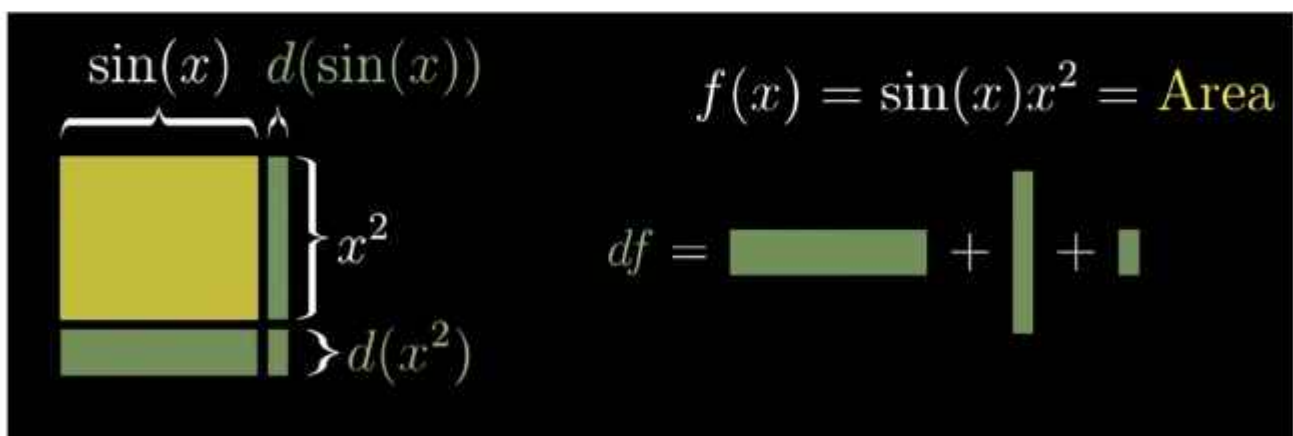


With these all said, let's learn one more thing! The Chain Rule!

To understand the most out of it, we need to know of Product Rule and Chain Rule. Consider the following figure (from 3Blue1Brown).



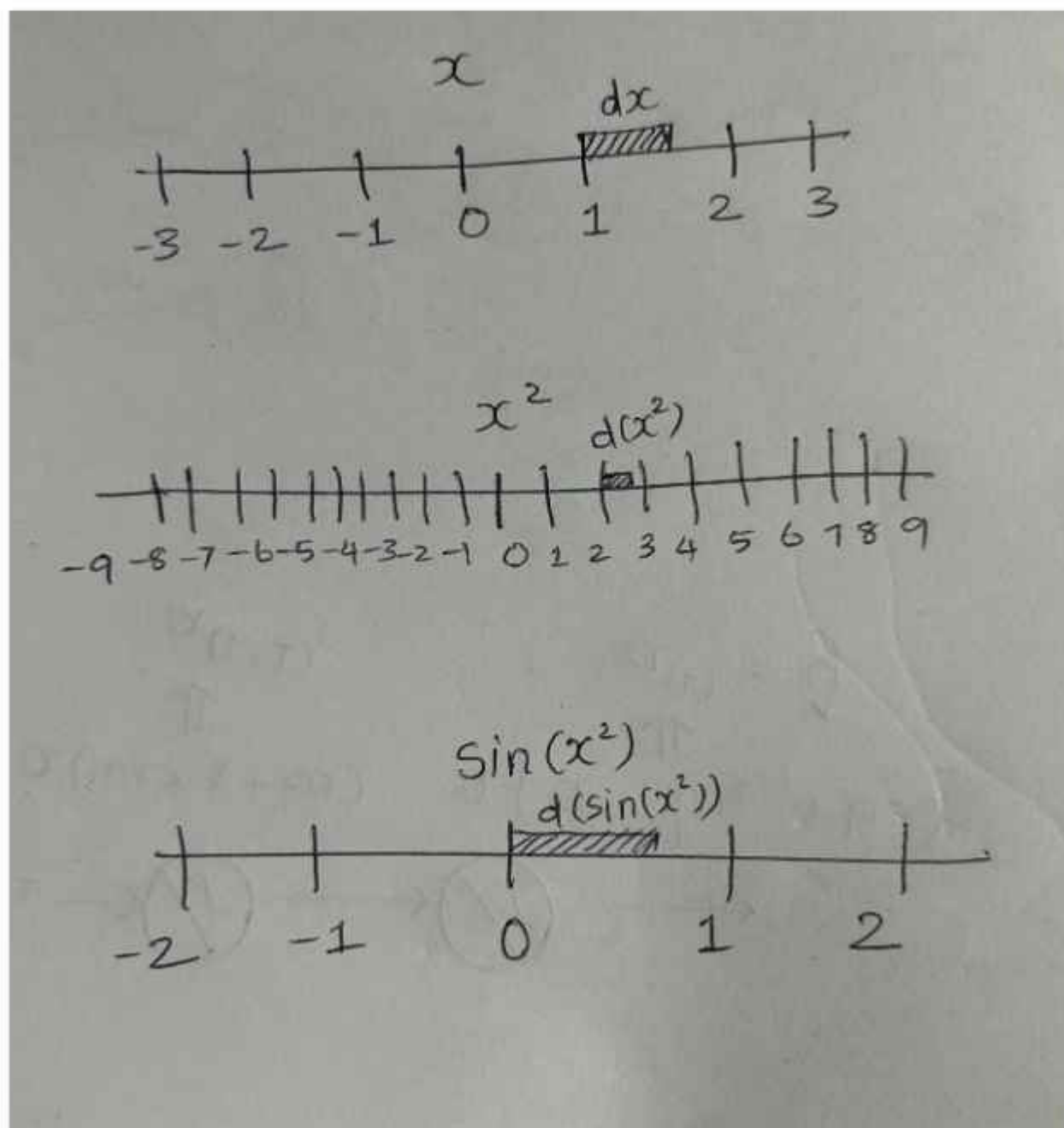
To find the RISE over RUN (Impact of A Tiny nudge), let's change x slightly.



$$df = \sin(x)d(x^2) + x^2d(\sin(x))$$

Ignore $\rightarrow +$

The newly formed area can be computed as shown above. The product of $d(\sin(x))$ and $d(x^2)$ is ignorable. This is the intuition of Product Rule. This being said, look at the following.



$\sin(x^2)$ is apparently, a function of x^2 , which is a function of x . A small change in x stimulates a chain reaction in x^2 and $\sin(x^2)$. So, it should be calculated this way...

$$\begin{aligned} d(\sin(x^2)) &= \cos(x^2) d(x^2) \\ &= \cos(x^2) \cdot 2x \cdot dx \end{aligned}$$

This is,

$$[d(\sin(x^2)) / d(x^2)] \cdot [d(x^2) / d(x)] \cdot [d(x)/d(x)]$$

Let's take another example.

Chain Rule

$$z = y^2 + 1 \quad (\text{Function of } y)$$

$$y = 2x + 3 \quad (\text{Function of } x)$$

$$z = g(y) \quad \& \quad y = f(x)$$

$$z = g(f(x))$$

How does z changes
w.r.to x ? $[x=2, y=7, z=50]$

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

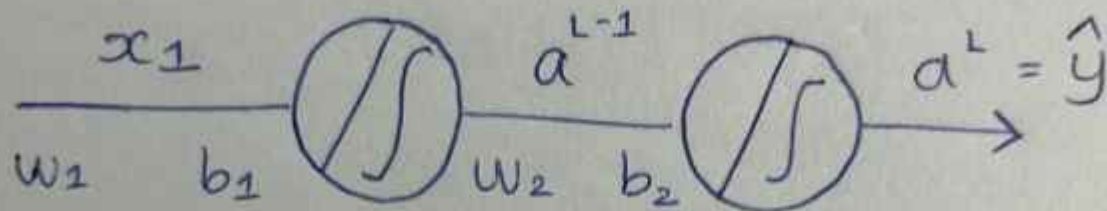
$$= 2y \cdot 2$$

$$\frac{dz}{dx} = 4y = 28$$

Now, we may have grasped the intuition of these rules, I hope.



What if our MLP is a Two Layer Perceptron, each layer with one neuron?



$$z_1 = w_1 x_1 + b_1$$

$$a^{L-1} = f(z_1)$$

$$z_2 = w_2 a^{L-1} + b_2$$

$$a^L = f(z_2) \text{ (or) } g(z_2)$$

$$L = \frac{(\hat{y} - y)^2}{2}$$

$$L = \frac{1}{2} (g(w_2 f(z_1) + b_2) - y)^2$$

$$L = \frac{1}{2} (g(w_2 f(w_1 x_1 + b_1) + b_2) - y)^2$$

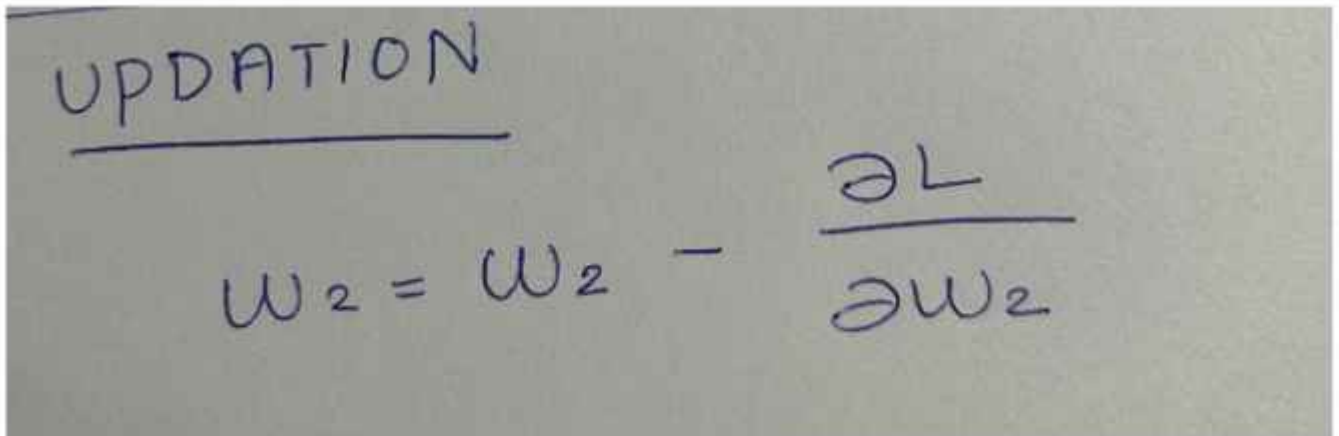
Here, we use a special Loss called Mean Squared Error. Look at the following image.

Handwritten equations on a piece of paper:

$$\hat{y} = g(z_2) \quad [\text{Function of } z_2]$$
$$z_2 = w_2 \cdot a_1 + b_2 \quad [\text{Function of } a_1, w_2, b_2]$$
$$a_1 = f(z_1) \quad [\text{Function of } z_1]$$
$$z_1 = w_1 \cdot x + b_1 \quad [\text{Function of } w_1, b_1, x]$$

Apparently, the loss is a function of lot of parameters. Those parameters in turn, are functions of some other params!

If I have to update w_2 , I should update it like the following.



A photograph of a piece of paper with handwritten text. The word 'UPDATE' is written in blue ink and underlined. Below it, the equation $w_2 = w_2 - \frac{\partial L}{\partial w_2}$ is written in blue ink.

$$\text{UPDATE}$$
$$w_2 = w_2 - \frac{\partial L}{\partial w_2}$$

But, what is the need to update the weights this way? That's the highest intuition we will get when we complete these all! For now, keep it as it is! We'll understand this later... So, we need to compute differentiation of Loss with respect to the respective weight. Now, how can we compute that? CHAIN RULE!

$$L = \frac{1}{2} (\hat{y} - y)^2$$

$$L = \frac{1}{2} (g(z_2) - y)^2$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_2}$$

$$\text{Gradient} = (g(z_2) - y) \cdot g'(z_2) \cdot a_1$$

The word is Gradient!

After calculating something called Gradient, we update the respective weight as shown below.

UPDATION

$$w_2 = w_2 - \frac{\partial L}{\partial w_2}$$

Let's consider the hill climbing race..



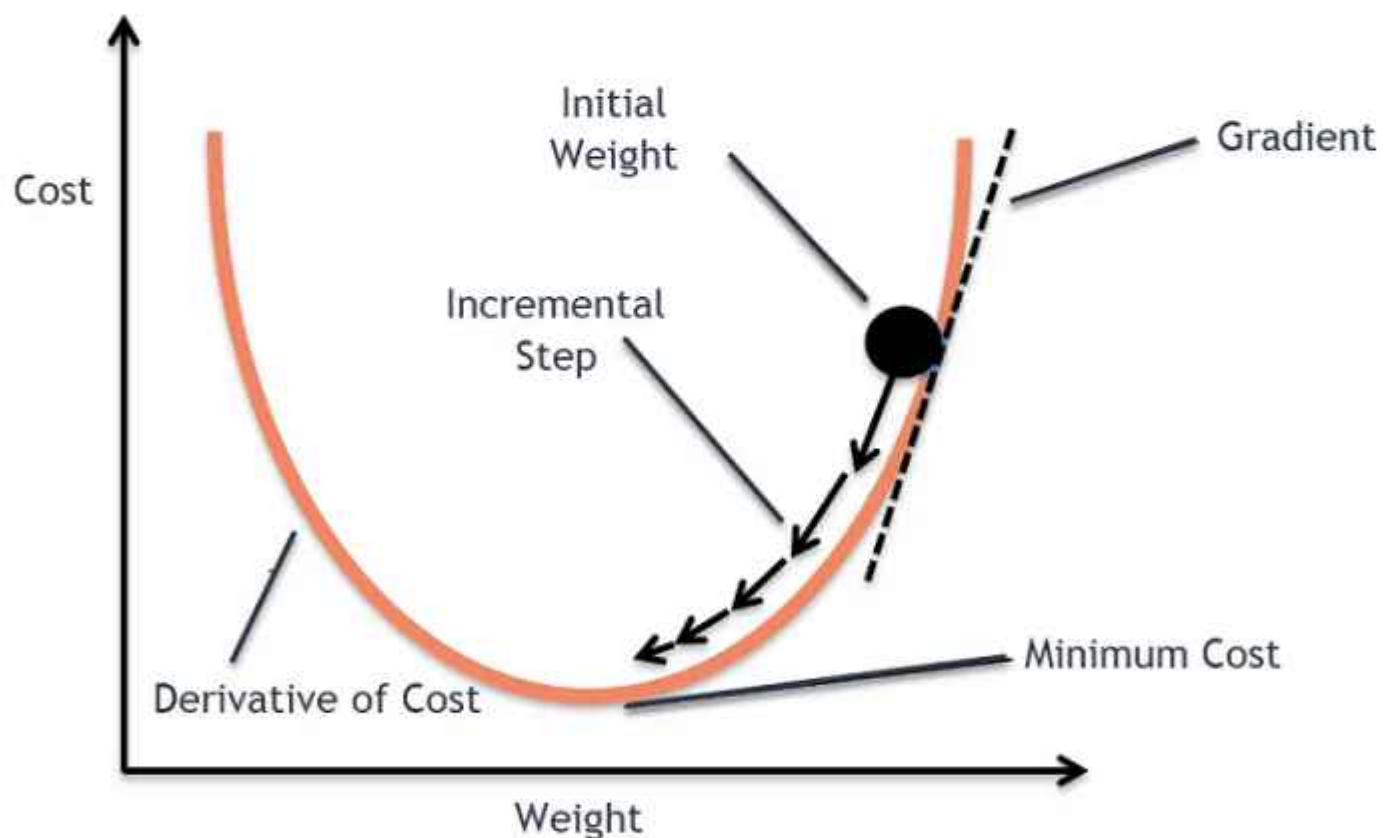
How does this car game to the minimum point? It's because of the acceleration we give and also some simulated physics. Let the hills represent areas of higher loss and valleys representing lower loss. I mean, the hill landscape is the loss function visualized. The lowest valley is the optimal solution the car must reach (Let's consider so). Loss is the function of parameters.

We adjust those parameters in the way such that the loss approaches minimum. How do we adjust? Through Gradient calculation and subtracting it from the weight... That's okay... But, why does it works?

The landscape is continuous, meaning the car can move smoothly across it similar to how a loss function behaves as we adjust weights in a neural network. At any point in time, the car has a specific location on the landscape that corresponds to a set of weights in a neural network.

The car can "feel" the steepness of the ground around it, which corresponds to evaluating the loss function at that position. This steepness can be thought of as the gradient, which tells the car how much it needs to change its position to get lower. The gradient at a given point represents the direction and rate of steepest ascent. Steepest ascent refers to the direction in which a function increases the most rapidly from a given point. The gradient is essentially a vector that points in the direction of the steepest increase in the loss function.

Conversely, moving in the opposite direction of the gradient leads to the steepest decrease, which is what we want when minimizing loss.



Let's consider the following Neural Net. Although it is not an MLP, it will help us to understand how the weight adjustment happens.

$$w = w - \frac{\partial \text{Loss}}{\partial w}$$

$$\frac{\partial \text{Loss}}{\partial w} = \frac{\partial \text{Loss}}{\partial z} \cdot \frac{\partial z}{\partial w}$$

$$b = b - \frac{\partial \text{Loss}}{\partial b}$$

$$\frac{\partial \text{Loss}}{\partial b} = \frac{\partial \text{Loss}}{\partial z} \cdot \frac{\partial z}{\partial b}$$

$$z = w \cdot x + b$$

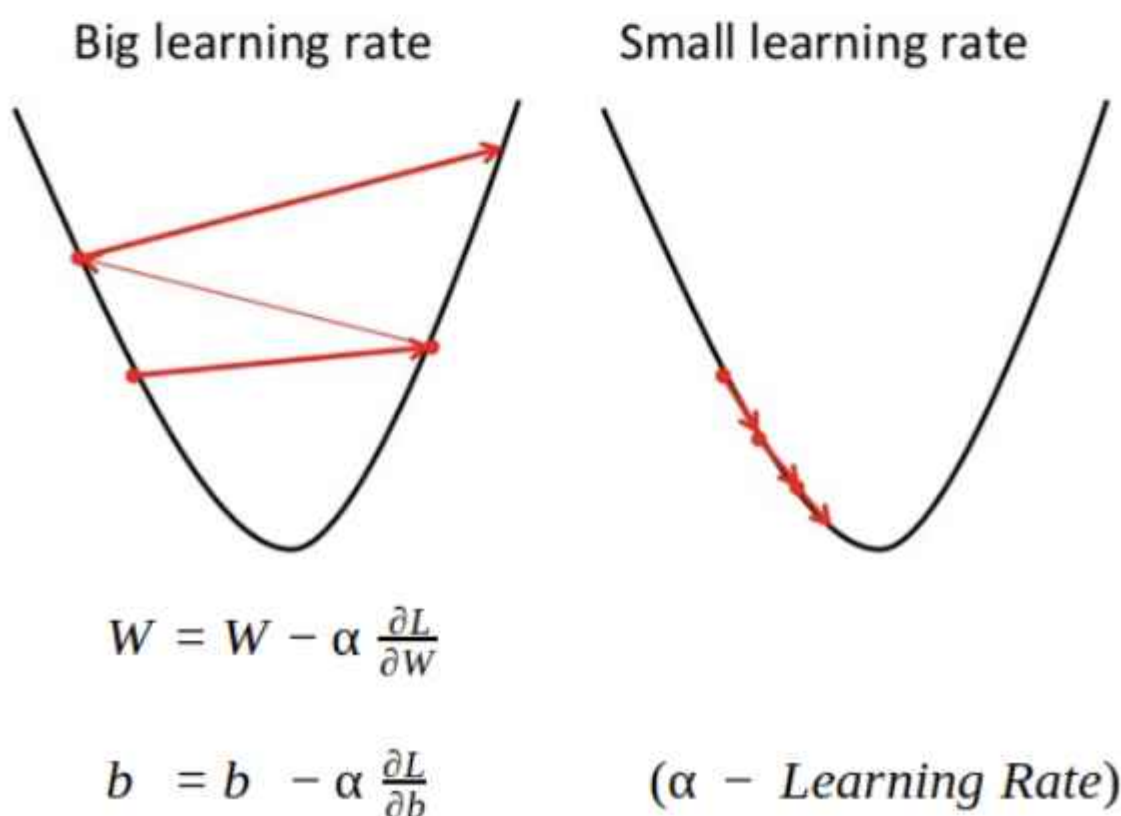
$$\text{ReLU}(z) = \max(0, z)$$

$$\text{Loss} = \frac{1}{2} (y - \hat{y})^2$$

So, following is the table. Through the experiment, we have come to the conclusion that this way of weight updation actually works.

Epoch	x	w	b	$z = w \cdot x + b$	$\text{ReLU}(z)$ (Predicted \hat{y})	Loss $\frac{1}{2} (y - \hat{y})^2$	$\frac{\partial \text{Loss}}{\partial w}$	$\frac{\partial \text{Loss}}{\partial b}$
1	1	0.5	0.5	1	1	0.5	1	1
2	1	-0.5	-0.5	-1	0	0	0	0

The learning rate determines the step size that gradient descent takes towards the minimum of the loss function. A small learning rate means small steps, leading to slow convergence but potentially more stable updates. Conversely, a large learning rate leads to larger steps, which can speed up convergence but may cause the model to overshoot the optimal point or even diverge.



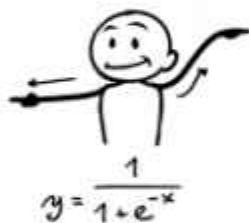
During training, alpha controls the rate at which the model's weights are updated. For each iteration, the weight update is calculated as the following.

$$\mathbf{W}_{\text{new}} = \mathbf{W}_{\text{old}} - \alpha(\text{gradient})$$

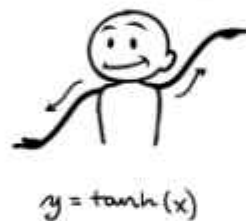
The gradient represents the direction of the steepest ascent (or descent in our case) in the loss landscape. Multiplying the gradient by alpha ensures the updates are proportional to the learning rate, impacting how quickly the model moves towards minimizing the loss.

Without activation functions, a neural network (even a deep one) behaves like a linear model, no matter how many layers it has. Activation functions enable the network to capture complex patterns by adding non-linear transformations to the data.

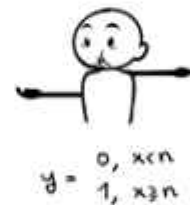
Sigmoid



Tanh



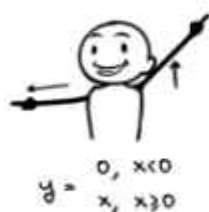
Step Function



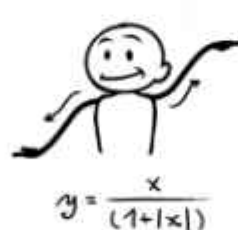
Softplus



ReLU



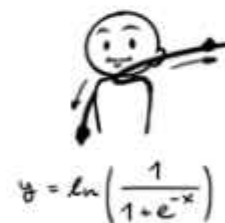
Softsign



ELU



Log of Sigmoid



Swish



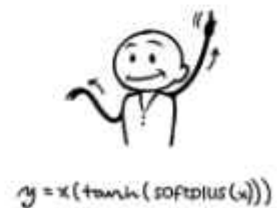
Sinc



Leaky ReLU



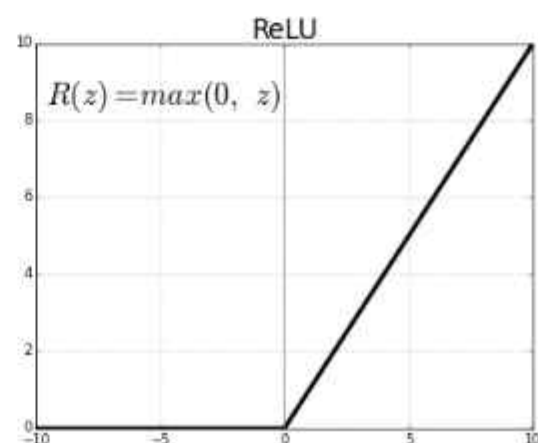
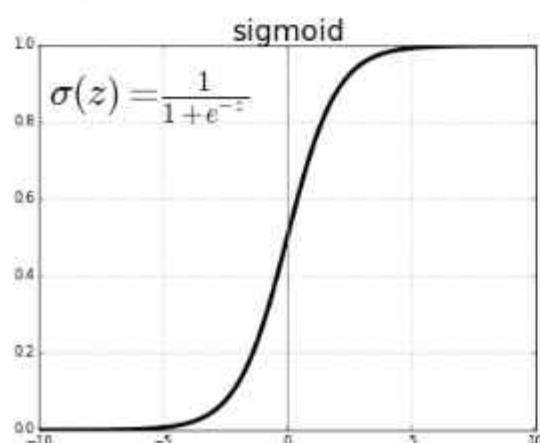
Mish



Let's learn two of them for now.

Sigmoid : Squashes the output to be between 0 and 1 | Commonly used in binary classification tasks | Useful for probabilities | Can lead to vanishing gradients, slowing down learning in deep networks

RELU : Sets all negative values to zero and leaves positive values unchanged | Simple and computationally efficient; helps with sparse representations | Can lead to “dead neurons” (neurons that always output zero) if gradients repeatedly push weights in the negative direction



Let's code for XOR using MLP in Torch.

```
1
2 import torch
3 import torch.nn as nn
4 import torch.optim as optim
5
6 class XORMLP(nn.Module):
7     def __init__(self):
8         super(XORMLP, self).__init__()
9         self.hidden_one = nn.Linear(2, 8)
10        self.hidden_two = nn.Linear(8, 4)
11        self.output = nn.Linear(4, 1)
12        self.relu = nn.ReLU()
13
14        def forward(self, x):
15            x = self.relu(self.hidden_one(x))
16            x = self.relu(self.hidden_two(x))
17            x = torch.sigmoid(self.output(x))
18            return x
19
20 model = XORMLP()
21 criterion = nn.BCELoss()
22 optimizer = optim.SGD(model.parameters(), lr=0.1)
23
24 X = torch.tensor([[0.0, 0.0],
25                   [0.0, 1.0],
26                   [1.0, 0.0],
27                   [1.0, 1.0]])
28 Y = torch.tensor([[0.0], [1.0], [1.0], [0.0]])
```

Here, we have defined an MLP. The loss used is Binary Cross Entropy. The optimizing algorithm is Stochastic Gradient Descent.

```

30 epochs = 2000
31 for epoch in range(epochs):
32     output = model(X)
33     loss = criterion(output, Y)
34     optimizer.zero_grad()
35     loss.backward()
36     optimizer.step()
37
38     if (epoch + 1) % 10 == 0:
39         print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')
40
41 with torch.no_grad():
42     for i, sample in enumerate(X):
43         output = model(sample)
44         print(f'Input: {sample.tolist()}, Output: {output.item():.4f}, Expected: {Y[i].item()}')
45

```

Now, if we train the network, we could see the progress.

Epoch [10/2000], Loss: 0.6864

```

Epoch [20/2000], Loss: 0.6829
Epoch [30/2000], Loss: 0.6791
Epoch [40/2000], Loss: 0.6744
Epoch [50/2000], Loss: 0.6694
Epoch [60/2000], Loss: 0.6633
Epoch [70/2000], Loss: 0.6553
Epoch [80/2000], Loss: 0.6448
Epoch [90/2000], Loss: 0.6317
Epoch [100/2000], Loss: 0.6163
Epoch [110/2000], Loss: 0.5970
Epoch [120/2000], Loss: 0.5733
Epoch [130/2000], Loss: 0.5427
Epoch [140/2000], Loss: 0.5040
Epoch [150/2000], Loss: 0.4569
Epoch [160/2000], Loss: 0.4117
Epoch [170/2000], Loss: 0.3650
Epoch [180/2000], Loss: 0.3149
Epoch [190/2000], Loss: 0.2665
Epoch [200/2000], Loss: 0.2256
Epoch [210/2000], Loss: 0.1874
Epoch [220/2000], Loss: 0.1551
Epoch [230/2000], Loss: 0.1287
Epoch [240/2000], Loss: 0.1065
Epoch [250/2000], Loss: 0.0893
Epoch [260/2000], Loss: 0.0763
Epoch [270/2000], Loss: 0.0648
Epoch [280/2000], Loss: 0.0560
Epoch [290/2000], Loss: 0.0494
Epoch [300/2000], Loss: 0.0432
Epoch [310/2000], Loss: 0.0384
Epoch [320/2000], Loss: 0.0345
Epoch [330/2000], Loss: 0.0311
Epoch [340/2000], Loss: 0.0282
Epoch [350/2000], Loss: 0.0257
Epoch [360/2000], Loss: 0.0235
Epoch [370/2000], Loss: 0.0217
Epoch [380/2000], Loss: 0.0200
Epoch [390/2000], Loss: 0.0186
Epoch [400/2000], Loss: 0.0173

```

Epoch	[410/2000]	Loss: 0.0162
Epoch	[420/2000]	Loss: 0.0151
Epoch	[430/2000]	Loss: 0.0142
Epoch	[440/2000]	Loss: 0.0134
Epoch	[450/2000]	Loss: 0.0126
Epoch	[460/2000]	Loss: 0.0119
Epoch	[470/2000]	Loss: 0.0113
Epoch	[480/2000]	Loss: 0.0108
Epoch	[490/2000]	Loss: 0.0102
Epoch	[500/2000]	Loss: 0.0097
Epoch	[510/2000]	Loss: 0.0093
Epoch	[520/2000]	Loss: 0.0089
Epoch	[530/2000]	Loss: 0.0085
Epoch	[540/2000]	Loss: 0.0081
Epoch	[550/2000]	Loss: 0.0078
Epoch	[560/2000]	Loss: 0.0075
Epoch	[570/2000]	Loss: 0.0072
Epoch	[580/2000]	Loss: 0.0069
Epoch	[590/2000]	Loss: 0.0067
Epoch	[600/2000]	Loss: 0.0065
Epoch	[610/2000]	Loss: 0.0062
Epoch	[620/2000]	Loss: 0.0060
Epoch	[630/2000]	Loss: 0.0058
Epoch	[640/2000]	Loss: 0.0056
Epoch	[650/2000]	Loss: 0.0055
Epoch	[660/2000]	Loss: 0.0053
Epoch	[670/2000]	Loss: 0.0051
Epoch	[680/2000]	Loss: 0.0050
Epoch	[690/2000]	Loss: 0.0048
Epoch	[700/2000]	Loss: 0.0047
Epoch	[710/2000]	Loss: 0.0046
Epoch	[720/2000]	Loss: 0.0044
Epoch	[730/2000]	Loss: 0.0043
Epoch	[740/2000]	Loss: 0.0042
Epoch	[750/2000]	Loss: 0.0041
Epoch	[760/2000]	Loss: 0.0040
Epoch	[770/2000]	Loss: 0.0039
Epoch	[780/2000]	Loss: 0.0038
Epoch	[790/2000]	Loss: 0.0037
Epoch	[800/2000]	Loss: 0.0036
Epoch	[810/2000]	Loss: 0.0035
Epoch	[820/2000]	Loss: 0.0035
Epoch	[830/2000]	Loss: 0.0034
Epoch	[840/2000]	Loss: 0.0033
Epoch	[850/2000]	Loss: 0.0032
Epoch	[860/2000]	Loss: 0.0032
Epoch	[870/2000]	Loss: 0.0031
Epoch	[880/2000]	Loss: 0.0030
Epoch	[890/2000]	Loss: 0.0030
Epoch	[900/2000]	Loss: 0.0029
Epoch	[910/2000]	Loss: 0.0029
Epoch	[920/2000]	Loss: 0.0028
Epoch	[930/2000]	Loss: 0.0027
Epoch	[940/2000]	Loss: 0.0027
Epoch	[950/2000]	Loss: 0.0026
Epoch	[960/2000]	Loss: 0.0026
Epoch	[970/2000]	Loss: 0.0025
Epoch	[980/2000]	Loss: 0.0025
Epoch	[990/2000]	Loss: 0.0025
Epoch	[1000/2000]	Loss: 0.0024
Epoch	[1010/2000]	Loss: 0.0024
Epoch	[1020/2000]	Loss: 0.0023
Epoch	[1030/2000]	Loss: 0.0023
Epoch	[1040/2000]	Loss: 0.0023

Epoch	[1050/2000]	Loss: 0.0022
Epoch	[1060/2000]	Loss: 0.0022
Epoch	[1070/2000]	Loss: 0.0022
Epoch	[1080/2000]	Loss: 0.0021
Epoch	[1090/2000]	Loss: 0.0021
Epoch	[1100/2000]	Loss: 0.0021
Epoch	[1110/2000]	Loss: 0.0020
Epoch	[1120/2000]	Loss: 0.0020
Epoch	[1130/2000]	Loss: 0.0020
Epoch	[1140/2000]	Loss: 0.0019
Epoch	[1150/2000]	Loss: 0.0019
Epoch	[1160/2000]	Loss: 0.0019
Epoch	[1170/2000]	Loss: 0.0018
Epoch	[1180/2000]	Loss: 0.0018
Epoch	[1190/2000]	Loss: 0.0018
Epoch	[1200/2000]	Loss: 0.0018
Epoch	[1210/2000]	Loss: 0.0017
Epoch	[1220/2000]	Loss: 0.0017
Epoch	[1230/2000]	Loss: 0.0017
Epoch	[1240/2000]	Loss: 0.0017
Epoch	[1250/2000]	Loss: 0.0017
Epoch	[1260/2000]	Loss: 0.0016
Epoch	[1270/2000]	Loss: 0.0016
Epoch	[1280/2000]	Loss: 0.0016
Epoch	[1290/2000]	Loss: 0.0016
Epoch	[1300/2000]	Loss: 0.0016
Epoch	[1310/2000]	Loss: 0.0015
Epoch	[1320/2000]	Loss: 0.0015
Epoch	[1330/2000]	Loss: 0.0015
Epoch	[1340/2000]	Loss: 0.0015
Epoch	[1350/2000]	Loss: 0.0015
Epoch	[1360/2000]	Loss: 0.0014
Epoch	[1370/2000]	Loss: 0.0014
Epoch	[1380/2000]	Loss: 0.0014
Epoch	[1390/2000]	Loss: 0.0014
Epoch	[1400/2000]	Loss: 0.0014
Epoch	[1410/2000]	Loss: 0.0014
Epoch	[1420/2000]	Loss: 0.0013
Epoch	[1430/2000]	Loss: 0.0013
Epoch	[1440/2000]	Loss: 0.0013
Epoch	[1450/2000]	Loss: 0.0013
Epoch	[1460/2000]	Loss: 0.0013
Epoch	[1470/2000]	Loss: 0.0013
Epoch	[1480/2000]	Loss: 0.0013
Epoch	[1490/2000]	Loss: 0.0013
Epoch	[1500/2000]	Loss: 0.0012
Epoch	[1510/2000]	Loss: 0.0012
Epoch	[1520/2000]	Loss: 0.0012
Epoch	[1530/2000]	Loss: 0.0012
Epoch	[1540/2000]	Loss: 0.0012
Epoch	[1550/2000]	Loss: 0.0012
Epoch	[1560/2000]	Loss: 0.0012
Epoch	[1570/2000]	Loss: 0.0012
Epoch	[1580/2000]	Loss: 0.0011
Epoch	[1590/2000]	Loss: 0.0011
Epoch	[1600/2000]	Loss: 0.0011
Epoch	[1610/2000]	Loss: 0.0011
Epoch	[1620/2000]	Loss: 0.0011
Epoch	[1630/2000]	Loss: 0.0011
Epoch	[1640/2000]	Loss: 0.0011
Epoch	[1650/2000]	Loss: 0.0011
Epoch	[1660/2000]	Loss: 0.0011
Epoch	[1670/2000]	Loss: 0.0010
Epoch	[1680/2000]	Loss: 0.0010


```
Epoch [1690/2000], Loss: 0.0010
Epoch [1700/2000], Loss: 0.0010
Epoch [1710/2000], Loss: 0.0010
Epoch [1720/2000], Loss: 0.0010
Epoch [1730/2000], Loss: 0.0010
Epoch [1740/2000], Loss: 0.0010
Epoch [1750/2000], Loss: 0.0010
Epoch [1760/2000], Loss: 0.0010
Epoch [1770/2000], Loss: 0.0010
Epoch [1780/2000], Loss: 0.0010
Epoch [1790/2000], Loss: 0.0009
Epoch [1800/2000], Loss: 0.0009
Epoch [1810/2000], Loss: 0.0009
Epoch [1820/2000], Loss: 0.0009
Epoch [1830/2000], Loss: 0.0009
Epoch [1840/2000], Loss: 0.0009
Epoch [1850/2000], Loss: 0.0009
Epoch [1860/2000], Loss: 0.0009
Epoch [1870/2000], Loss: 0.0009
Epoch [1880/2000], Loss: 0.0009
Epoch [1890/2000], Loss: 0.0009
Epoch [1900/2000], Loss: 0.0009
Epoch [1910/2000], Loss: 0.0009
Epoch [1920/2000], Loss: 0.0008
Epoch [1930/2000], Loss: 0.0008
Epoch [1940/2000], Loss: 0.0008
Epoch [1950/2000], Loss: 0.0008
Epoch [1960/2000], Loss: 0.0008
Epoch [1970/2000], Loss: 0.0008
Epoch [1980/2000], Loss: 0.0008
Epoch [1990/2000], Loss: 0.0008
Epoch [2000/2000], Loss: 0.0008
```

```
Input: [0.0, 0.0], Output: 0.0017, Expected: 0.0
Input: [0.0, 1.0], Output: 0.9995, Expected: 1.0
Input: [1.0, 0.0], Output: 0.9995, Expected: 1.0
Input: [1.0, 1.0], Output: 0.0005, Expected: 0.0
```

Alright, let's learn Gradient Descent and its variant algorithms next!

Check at <https://arihara-sudhan.github.io>

Thank You!