

# Mod-1

## Need of activation function

### 1. Introducing Non-Linearity:

- Without activation functions, a neural network would simply be a linear combination of inputs and weights, limited to modeling only linear relationships.

### 2. Learning Complex Representations:

- Activation functions enable neurons to learn more intricate representations of data.

### 3. Decision-Making Capabilities:

- Activation functions introduce decision-making power to neurons.

- Based on the activation function's output, a neuron can decide whether to "fire" or not, influencing the flow of information through the network and ultimately shaping the model's output.


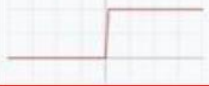





### 4. Controlling Output Range:

- Activation functions can regulate the output range of neurons.

- Some activation functions, like sigmoid and tanh, constrain outputs between 0 and 1 or -1 and 1, making them suitable for probability-related tasks.

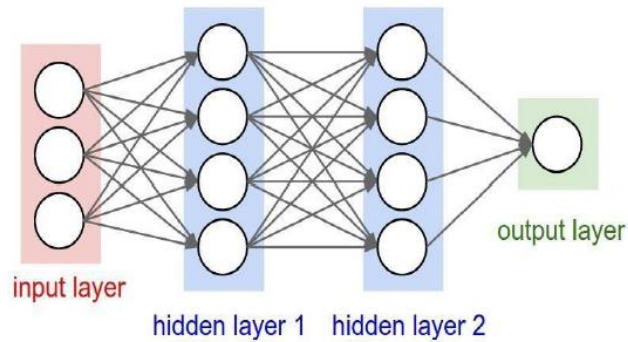
- Others, like ReLU, allow for unbounded positive outputs, useful for representing real-valued quantities.

# Activation Function

Identity		$f(x) = x$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Logistic (a.k.a. Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$
ArcTan		$f(x) = \tan^{-1}(x)$
Softsign <sup>[7][8]</sup>		$f(x) = \frac{x}{1 +  x }$
Rectified linear unit (ReLU) <sup>[9]</sup>		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$

## Feedforward Networks

A feedforward neural network, also called a multi-layer perceptron, is a collection of neurons, organized in *layers*.



It is used to approximate some function  $f^*$ . For instance,  $f^*$  could be a classifier that maps an input vector  $x$  to a category  $y$ .

The neurons are arranged in the form of a directed acyclic graph i.e., the information only flows in one direction - input  $x$  to output  $y$ . Hence the term **feedforward**.

## Gradient Descent

- Gradient descent is an optimization algorithm used to adjust the weights and biases of the network, ultimately leading it to learn and improve its performance on a specific task.
- How it works:
  - 1. Objective Function and Error:
    - We start with an objective function, which quantifies the network's performance on the training data. Examples include mean squared error for regression or cross-entropy for classification.
    - During training, the network predicts outputs for each data point, and these predictions are compared to the actual targets. The difference between the prediction and the target is called the error.

## Why Negative Gradient?

- The gradient of the loss function points in the direction of the steepest increase in loss.
- We want to minimize the loss, so we take a step in the opposite direction, which is achieved by multiplying the gradient by -1.

Imagine a landscape:

- The loss function forms a landscape with valleys (minima) and hills (maxima).
- The negative gradient points us down the steepest slope of the hill towards the valley.
- By iteratively taking steps in this direction, we eventually reach the bottom of the valley (minimum loss), improving the network's accuracy.

## Why Negative Gradient?

Benefits of using the negative gradient:

- Helps find the optimal parameters that minimize the loss function.
- Enables efficient learning by guiding the network towards better predictions.
- Forms the basis for various optimization algorithms used in neural network training.
- the negative gradient in neural networks plays a critical role in guiding the learning process and adjusting parameters to minimize the loss function, ultimately leading to improved network performance and accurate predictions.

# Backpropagation

- Backpropagation is an algorithm that trains neural networks by effectively adjusting their weights and biases to minimize the error between their predictions and the desired outputs.
- It works by propagating the error back through the network, layer by layer, using the chain rule of calculus to calculate how much each weight and bias contributed to the error.

## Steps of Backpropagation

### 1. Forward Pass:

1. Input data is fed into the input layer of the network.
2. Each neuron in the network performs calculations based on its inputs and activation function, passing its output to the neurons in the next layer.
3. This process continues until the output layer produces a prediction.

### 2. Error Calculation:

1. The predicted output is compared to the actual target value, and the difference (error) is calculated using a loss function.

### 3. Backward Pass:

1. The error signal is propagated backward through the network, layer by layer, using the chain rule of differentiation.
2. At each layer, the algorithm calculates the partial derivative of the error with respect to each weight and bias.

# Steps of Backpropagation

## 4. Parameter Update:

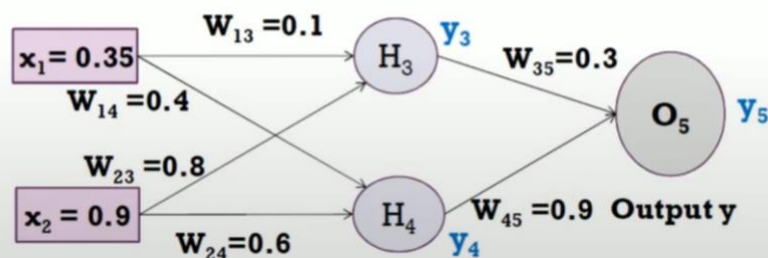
1. The calculated partial derivatives are used to update the weights and biases of the network, moving them in a direction that reduces the error.
2. The amount of change is determined by the learning rate, which controls how quickly the network learns.

## 5. Iteration:

1. Steps 1-4 are repeated for a large number of training examples, adjusting the weights and biases after each example.
2. Over time, the network learns to make better predictions by minimizing the error on the training data.

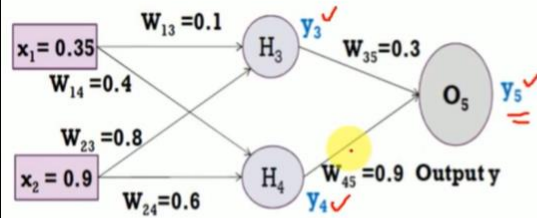
## Back Propagation Solved Example - 1

- Assume that the neurons have a sigmoid activation function, perform a forward pass and a backward pass on the network. Assume that the actual output of  $y$  is 0.5 and learning rate is 1. Perform another forward pass.





## Back Propagation Solved Example - 1



$$\text{Error} = y_{\text{target}} - y_5 = -0.19$$

$$0.5 - 0.69$$

- Forward Pass: Compute output for  $y_3$ ,  $y_4$  and  $y_5$ .

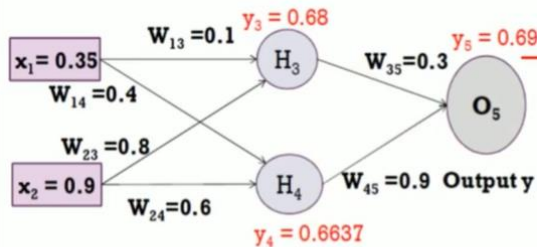
$$a_j = \sum_i (w_{ij} * x_i) \quad y_j = F(a_j) = \frac{1}{1 + e^{-a_j}}$$

$$\begin{aligned} a_1 &= (w_{13} * x_1) + (w_{23} * x_2) \\ &= (0.1 * 0.35) + (0.8 * 0.9) = 0.755 \\ y_3 &= f(a_1) = 1 / (1 + e^{-0.755}) = 0.68 \end{aligned}$$

$$\begin{aligned} a_2 &= (w_{14} * x_1) + (w_{24} * x_2) \\ &= (0.4 * 0.35) + (0.6 * 0.9) = 0.68 \\ y_4 &= f(a_2) = 1 / (1 + e^{-0.68}) = 0.6637 \end{aligned}$$

$$\begin{aligned} a_3 &= (w_{35} * y_3) + (w_{45} * y_4) \\ &= (0.3 * 0.68) + (0.9 * 0.6637) = 0.801 \\ y_5 &= f(a_3) = 1 / (1 + e^{-0.801}) = 0.69 \text{ (Network Output)} \end{aligned}$$

## Back Propagation Solved Example - 1



- Backward Pass: Compute  $\delta_3$ ,  $\delta_4$  and  $\delta_5$ .

For output unit:

$$\begin{aligned} \delta_5 &= y(1-y) (y_{\text{target}} - y) \\ &= 0.69 * (1 - 0.69) * (0.5 - 0.69) = -0.0406 \end{aligned}$$

For hidden unit:

$$\begin{aligned} \delta_3 &= y_3(1-y_3) w_{35} * \delta_5 \\ &= 0.68 * (1 - 0.68) * (0.3 * -0.0406) = -0.00265 \end{aligned}$$

$$\Delta w_{ji} = \eta \delta_j o_i$$

$$\delta_j = o_j(1-o_j)(t_j - o_j)$$

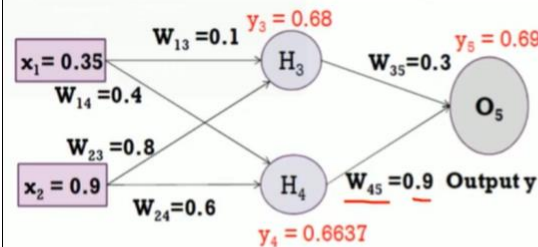
$$\delta_j = o_j(1-o_j) \sum_k \delta_k w_{kj}$$

if  $j$  is an output unit

if  $j$  is a hidden unit

$$\begin{aligned} \delta_4 &= y_4(1-y_4) w_{45} * \delta_5 \\ &= 0.6637 * (1 - 0.6637) * (0.9 * -0.0406) = -0.0082 \end{aligned}$$

## Back Propagation Solved Example - 1



• Backward Pass: Compute  $\delta_3$ ,  $\delta_4$  and  $\delta_5$ .

For output unit:

$$\delta_5 = y(1-y)(y_{\text{target}} - y) = 0.69 * (1 - 0.69) * (0.5 - 0.69) = -0.0406$$

For hidden unit:

$$\delta_3 = y_3(1-y_3)w_{35} * \delta_5 = 0.68 * (1 - 0.68) * (0.3 * -0.0406) = -0.00265$$

Compute new weights

$$\Delta w_{ji} = \eta \delta_j o_i$$

$$\Delta w_{45} = \eta \delta_5 y_4 = 1 * -0.0406 * 0.6637 = -0.0269$$

$$w_{45}(\text{new}) = \Delta w_{45} + w_{45}(\text{old}) = -0.0269 + (0.9) = 0.8731$$

$$\delta_4 = y_4(1-y_4)w_{45} * \delta_5 = 0.6637 * (1 - 0.6637) * (0.9 * -0.0406) = -0.0082$$

$$\Delta w_{14} = \eta \delta_4 x_1 = 1 * -0.0082 * 0.35 = -0.00287$$

$$w_{14}(\text{new}) = \Delta w_{14} + w_{14}(\text{old}) = -0.00287 + 0.4 = 0.3971$$

## Samples Versus Population

- Population - All of the units we'd like to study or model in our experiment.
- Sample- Subset of the population of data that hopefully represents the accurate distribution of the data without introducing sampling bias



# Resampling Methods

Method	Definition	Limitations
<b>Holdout</b>	The original data is divided into two sets, training and test set. The model is induced using the training set and then its performance is evaluated using the test set. The accuracy of the induced model on the test set is used to estimate the accuracy of the classifier	<ol style="list-style-type: none"> <li>1. Because the data was divided into two sets fewer label examples are available for training. As a result, the induced model may not be as good</li> <li>2. It may be highly dependent on the composition of the training and test sets.</li> <li>3. The two sets are no longer independent because they are subsets of a larger set.</li> </ol>
<b>Random Subsampling</b>	Repeats the holdout method several times to improve estimation of a classifier's performance. Overall accuracy, $acc_{sub}$ is based on the accuracy for each run, $acc_i$ . $acc_{sub} = \sum_{i=1}^k acc_i / k$	<ol style="list-style-type: none"> <li>1. Still encounters some of the holdout problems.</li> <li>2. Some records might be used for training more often than others.</li> </ol>
<b>Cross-Validation</b>	Partitions the data into k disjoint subset. It then uses k-1 subsets for the training model and one set for the test set. It does this k number of times switching the test set each time thereby each record is used the same number of times for training and exactly one time for testing. The total error is found by summing the errors of all k runs. Leave-one-out $k=n$ : basically all the data except for one is used for the training set and the one is used for the test set. This is good because it uses as much data as possible for training	<ol style="list-style-type: none"> <li>1. Training algorithm has to be rerun from scratch k times.</li> <li>2. When the data set size is small splitting it compromises it integrity.</li> </ol> <ol style="list-style-type: none"> <li>1. Computationally expensive</li> <li>2. The variance of the estimated performance metric is high</li> </ol>
<b>Bootstrap</b>	The training records are sampled with replacement. "A common variation used is the .632 bootstrap which computes the overall accuracy by combining the accuracies of each bootstrap sample with the accuracy computed from a training set that contains all the labeled examples in the original data" <sup>10</sup>	<ol style="list-style-type: none"> <li>1. Bootstrap can yield poor results in certain situations</li> <li>2. Though the bootstrap easily accommodates some violations of traditional statistical assumptions (e.g., non-normality), it is susceptible to others (e.g., non-independence)<sup>2</sup>.</li> </ol>

## Likelihood

- ❓ Likelihood that an event will occur yet do not specifically reference its numeric probability.
- ❓ About an event that has a reasonable probability of happening but still might not.
- ❓ Informally, likelihood is also used as a synonym for probability.

## Confusion

ma

		Positive prediction Label was positive	Negative prediction Label was positive
		P' (Predicted)	N' (Predicted)
P (Actual)	True Positive	False Negative	
N (Actual)	False Positive	True Negative	Negative prediction Label was negative

## Evaluating Models

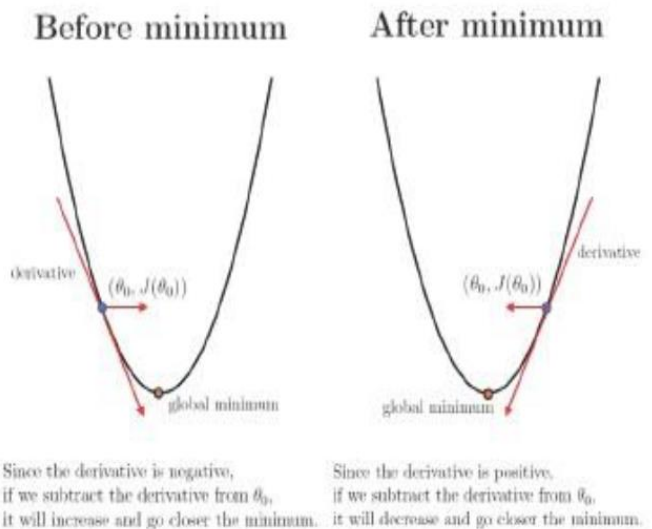
- False positive - “type I error”
- False negative - “type II error”

## Review: Gradient Descent (GD)

- Optimization algorithm used to find minima of a given differentiable function
- At each step, parameters ( $\theta$ ) are pushed in negative direction of gradient of a cost function ( $J(\theta; x, y)$ , in figure alongside) w.r.t parameters

$$\theta_{new} = \theta_{old} - \alpha \Delta \theta_{old}$$

where  $\alpha$  is learning rate



## Gradient Descent Algorithm

---

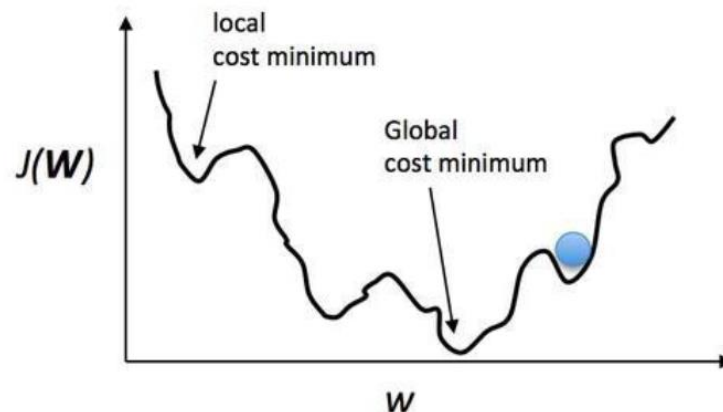
**Require:** Learning rate  $\alpha$ , initial parameters  $\theta_t$ , training dataset  $\mathcal{D}_{tr}$

---

- 1: **while** stopping criterion not met **do**
  - 2:     Initialize parameter updates  $\Delta \theta_t = 0$
  - 3:     **for each**  $(x^{(i)}, y^{(i)})$  in  $\mathcal{D}_{tr}$  **do**
  - 4:         Compute gradient using backpropagation  $\nabla_{\theta_t} \mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$
  - 5:         Aggregate gradient  $\Delta \theta_t = \Delta \theta_t + \nabla_{\theta_t} \mathcal{L}$
  - 6:     **end for**
  - 7:     Apply update  $\theta_{t+1} = \theta_t - \alpha \frac{1}{|\mathcal{D}_{tr}|} \Delta \theta_t$
  - 8: **end while**
-

## Local Minima

- Unlike convex objective functions that have a global minimum, non-convex functions as in deep neural networks have multiple local minima<sup>1</sup>



## GD: Pros and Cons

- For every parameter update, G D parses the entire dataset, hence called Batch G D
- Advantages of Batch G D
  - Conditions of convergence well-understood
  - Many acceleration techniques (e.g. conjugate gradient) operate in batch G D setting
- Disadvantages of Batch G D
  - Computationally slow
  - E.g. ImageNet (<http://www.image-net.org>), a commonly used dataset in vision, has ~14:2million samples; an iteration over it is going to be very slow

# Stochastic Gradient Descent

**Stochastic GD (SGD):** Randomly shuffle the training set, and update parameters after gradients are computed for each training example

---

**Require:** Learning rate  $\alpha$ , initial parameters  $\theta_t$ , training dataset  $\mathcal{D}_{tr}$

---

```
1: while stopping criterion not met do
2:   for each  $(x^{(i)}, y^{(i)})$  in  $\mathcal{D}_{tr}$  do
3:     Compute gradient using backpropagation  $\nabla_{\theta_t} \mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$ 
4:     Gradient  $\Delta\theta_t = \nabla_{\theta_t} \mathcal{L}$ 
5:     Apply update  $\theta_{t+1} = \theta_t - \alpha \Delta\theta_t$ 
6:   end for
7: end while
```

---

## SGD: Pros and Cons

### Advantages of SGD

- Usually much faster than batch learning; because there is lot of redundancy in batch learning
- Often results in better solutions; SGD's noise can help in escaping local minima (provided neighborhood provides enough gradient information) and saddle points<sup>1</sup>.
- Can be used for tracking changes

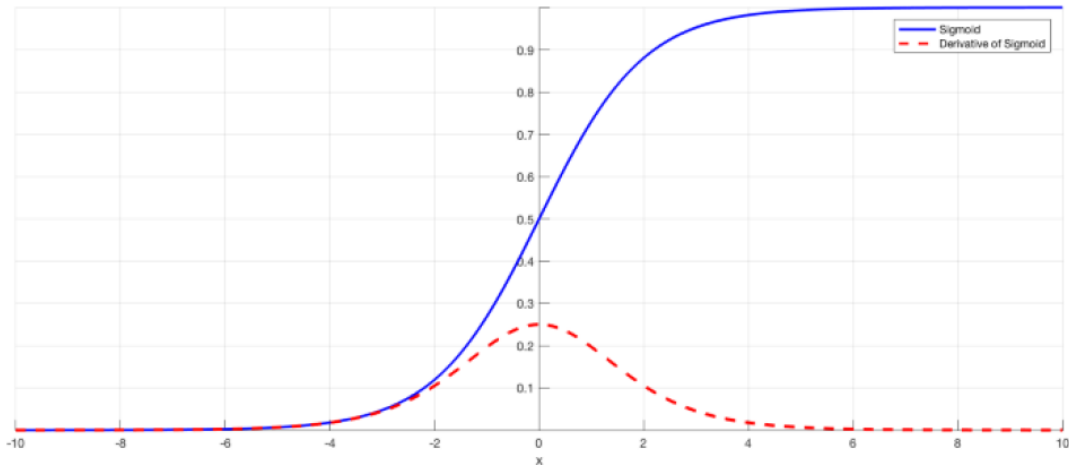
### Disadvantages of SGD

- Noise in SGD weight updates -can lead to no convergence!
- Can be controlled using learning rate, but identifying proper learning rate is a problem of its own

<sup>1</sup><http://mitadnes.github.io/ift6085-2019/ift-6085-bonus-lecture-saddle-points-notes.pdf>



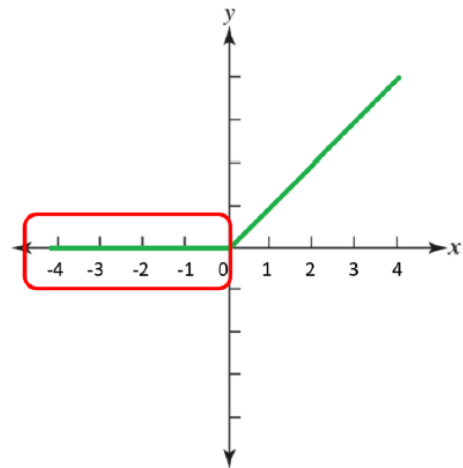
## Vanishing Gradient



## Dying ReLU

### What's the Dying ReLU problem?

The dying ReLU problem refers to the scenario when many ReLU neurons only output values of 0. The red outline below shows that this happens when the inputs are in the **negative** range.



# Dying ReLU- Causes

❓ High learning rate

$$W_{ij}^* = W_{ij} - a \left( \frac{\partial E}{\partial W_{ij}} \right)$$

The diagram illustrates the weight update equation  $W_{ij}^* = W_{ij} - a \left( \frac{\partial E}{\partial W_{ij}} \right)$ . Below the equation, four labels are positioned with arrows pointing to specific terms: 'New Weight' points to  $W_{ij}^*$ , 'Old Weight' points to  $W_{ij}$ , 'Learning Rate' points to  $a$ , and 'Derivative of Error with respect to Weight' points to  $\left( \frac{\partial E}{\partial W_{ij}} \right)$ .

New Weight

Old Weight

Learning Rate

Derivative of Error with respect to Weight