Unit -1

Machine Learning Basics

AI, ML and Deep Learning

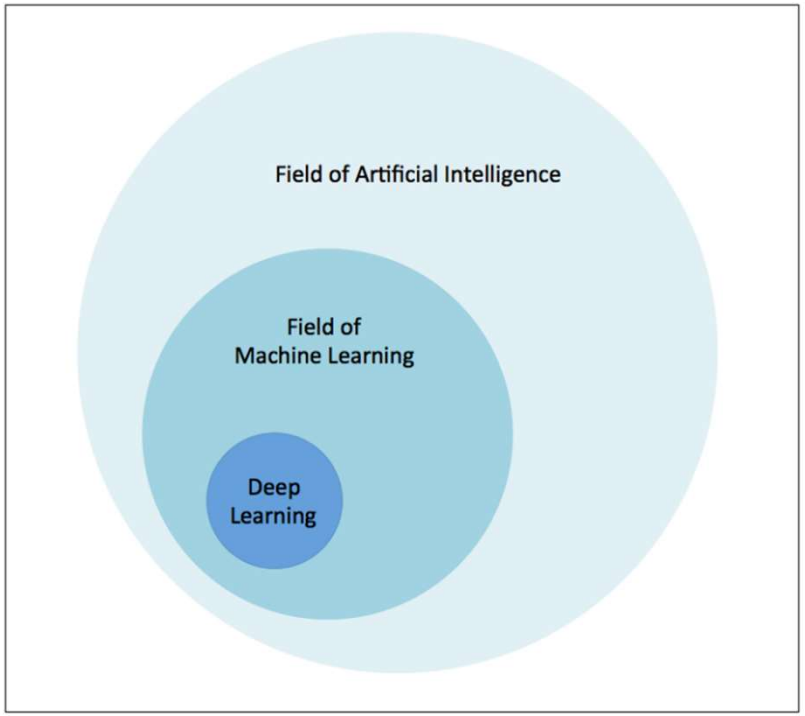
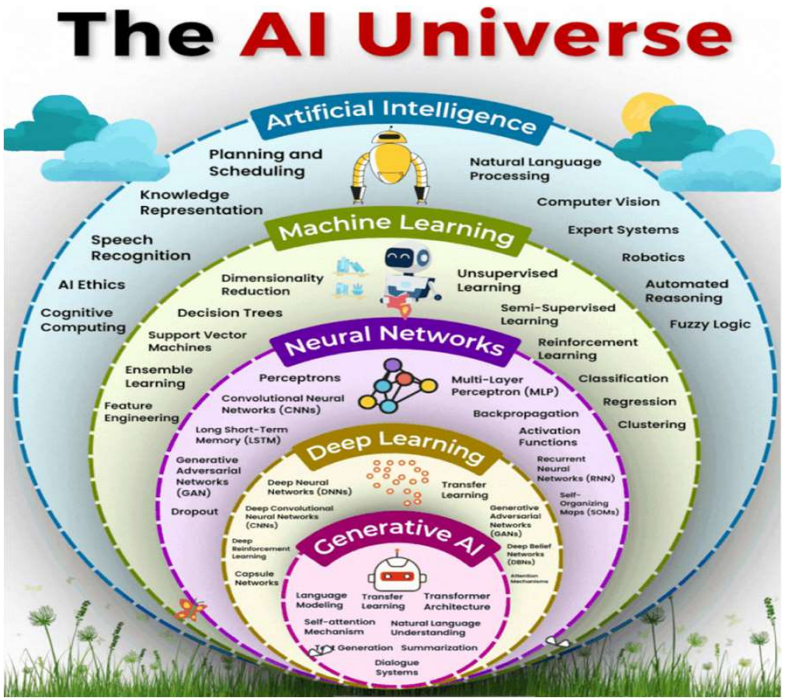


Figure 1-1. The relationship between AI and deep learning



Learning Algorithms

- “A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.”
- **The Task, T**
- Machine learning allows us to tackle tasks that are too difficult to solve with fixed programs written and designed by human beings.
- the process of learning itself is not the task. Learning is our means of attaining the ability to perform the task. For example, if we want a robot to be able to walk, then walking is the task. We could program the robot to learn to walk, or we could attempt to directly write a program that specifies how to walk manually.
- Many kinds of tasks can be solved with machine learning: classification, classification with missing inputs, Regression, transcription, anomaly detection etc

- **The Performance Measure, P**
- In order to evaluate the abilities of a machine learning algorithm, we must design a quantitative measure of its performance.
- Usually this performance measure P is specific to the task T being carried out by the system.
- For tasks such as classification, classification with missing inputs, and transcription, we often measure the accuracy of the model. Accuracy is just the proportion of examples for which the model produces the correct output
- We can also obtain equivalent information by measuring the error rate
- **Accuracy**, **Confusion Matrix, Precision, Recall, F-Score**, **AUC(Area Under the Curve)-ROC**

- **The Experience, E**
- Machine learning algorithms can be broadly categorized as unsupervised or supervised by what kind of experience they are allowed to have during the learning process.
- Unsupervised learning algorithms experience a dataset containing many features, then learn useful properties of the structure of this dataset.
- In the context of deep learning, we usually want to learn the entire probability distribution that generated a dataset, whether explicitly as in density estimation or implicitly for tasks like synthesis or denoising.
- Some other unsupervised learning algorithms perform other roles, like clustering, which consists of dividing the dataset into clusters of similar examples.

- Supervised learning algorithms experience a dataset containing features, but each example is also associated with a label or target.
- For example, the Iris dataset is annotated with the species of each iris plant.
- A supervised learning algorithm can study the Iris dataset and learn to classify iris plants into three different species based on their measurements.
- **Linear Regression**

Maximum Likelihood Estimation

- Maximum Likelihood Estimation (MLE) is a statistical method used to estimate the parameters of a probability distribution that best describe a given dataset.
- The fundamental idea behind MLE is to find the values of the parameters that maximize the likelihood of the observed data, assuming that the data are generated by the specified distribution.
- to analyze the data that we have been presented with. Naturally, the first thing would be to identify the distribution from which we have obtained our data.
- Next, we need to use our data to find the parameters of our distribution.
- A parameter is a numerical characteristic of a distribution.
- Normal distributions, as we know, have mean (μ) & variance (σ^2) as parameters. Binomial distributions have the number of trials (n) & probability of success (p) as parameters. Gamma distributions have shape (k) and scale (θ) as parameters.

- Exponential distributions have the inverse mean (λ) as the parameter. The list goes on.
- if a population is known to follow a normal distribution but the mean and variance are unknown, MLE can be used to estimate them using a limited sample of the population, by finding particular values of the mean and variance so that the observation is the most likely result to have occurred.
- These parameters or numerical characteristics are vital for understanding the size, shape, spread, and other properties of a distribution.
- Since the data that we have is mostly randomly generated, we often don't know the true values of the parameters characterizing our distribution.
- That's when estimators step in. An estimator is like a function of your data that gives you approximate values of the parameters that you're interested in.

- For instance, the sample-mean estimator, which is perhaps the most frequently used estimator.
- It's calculated by taking the mean of our observations and comes in very handy when trying to estimate parameters that represent the mean of their distribution (for example the parameter μ for a normal distribution).
- use the sample mean estimator whenever the parameter is the mean of your distribution.

:

- In general, a statistical model for a random experiment is the pair:

$$(E, \quad \{\mathbb{P}_\theta\}_{\theta \in \Theta})$$

- 1) E represents the sample space of an experiment. By experiment we mean the data that we've collected- the observable data. So, E's the range of values that our data can take (based on the distribution that we've assigned to it).
- 2) $\mathbb{P}_\theta$ represents the family of probability-measures on E. In other words, it indicates the probability distribution that we've assigned to our data (based on our observations).

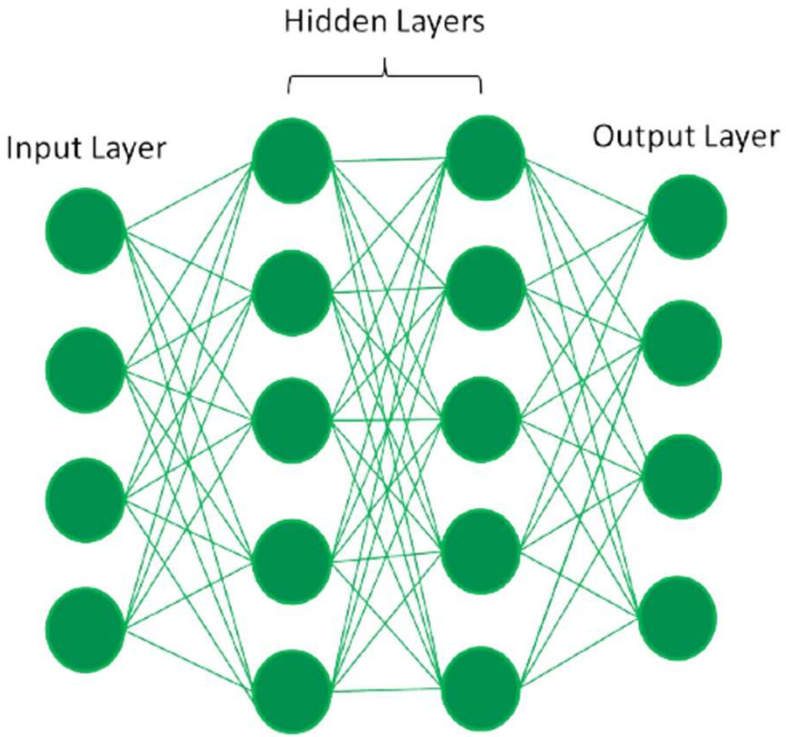
- 3) θ represents the set of unknown parameters that characterize the distribution $\mathbb{P}_\theta$. All those numerical features we wish to estimate are represented by θ. For now, it's enough to think of θ as a single parameter that we're trying to estimate. We'll later see how to deal with multi-dimensional parameters.
- 4) Θ represents the parameter space i.e., the range or the set of all possible values that the parameter θ could take.

Deep Learning

- The definition of Deep learning is that it is the branch of machine learning that is based on artificial neural network architecture.
- An artificial neural network or ANN uses layers of interconnected nodes called neurons that work together to process and learn from the input data.
- In a fully connected Deep neural network, there is an input layer and one or more hidden layers connected one after the other.
- Each neuron receives input from the previous layer neurons or the input layer. The output of one neuron becomes the input to other neurons in the next layer of the network, and this process continues until the final layer produces the output of the network.
- The layers of the neural network transform the input data through a series of nonlinear transformations, allowing the network to learn complex representations of the input data.

Artificial Neural Network

- Artificial neural networks are built on the principles of the structure and operation of human neurons. It is also known as neural networks or neural nets.
- An artificial neural network's input layer, which is the first layer, receives input from external sources and passes it on to the hidden layer, which is the second layer.
- Each neuron in the hidden layer gets information from the neurons in the previous layer, computes the weighted total, and then transfers it to the neurons in the next layer.
- These connections are weighted, which means that the impacts of the inputs from the preceding layer are more or less optimized by giving each input a distinct weight. These weights are then adjusted during the training process to enhance the performance of the model.

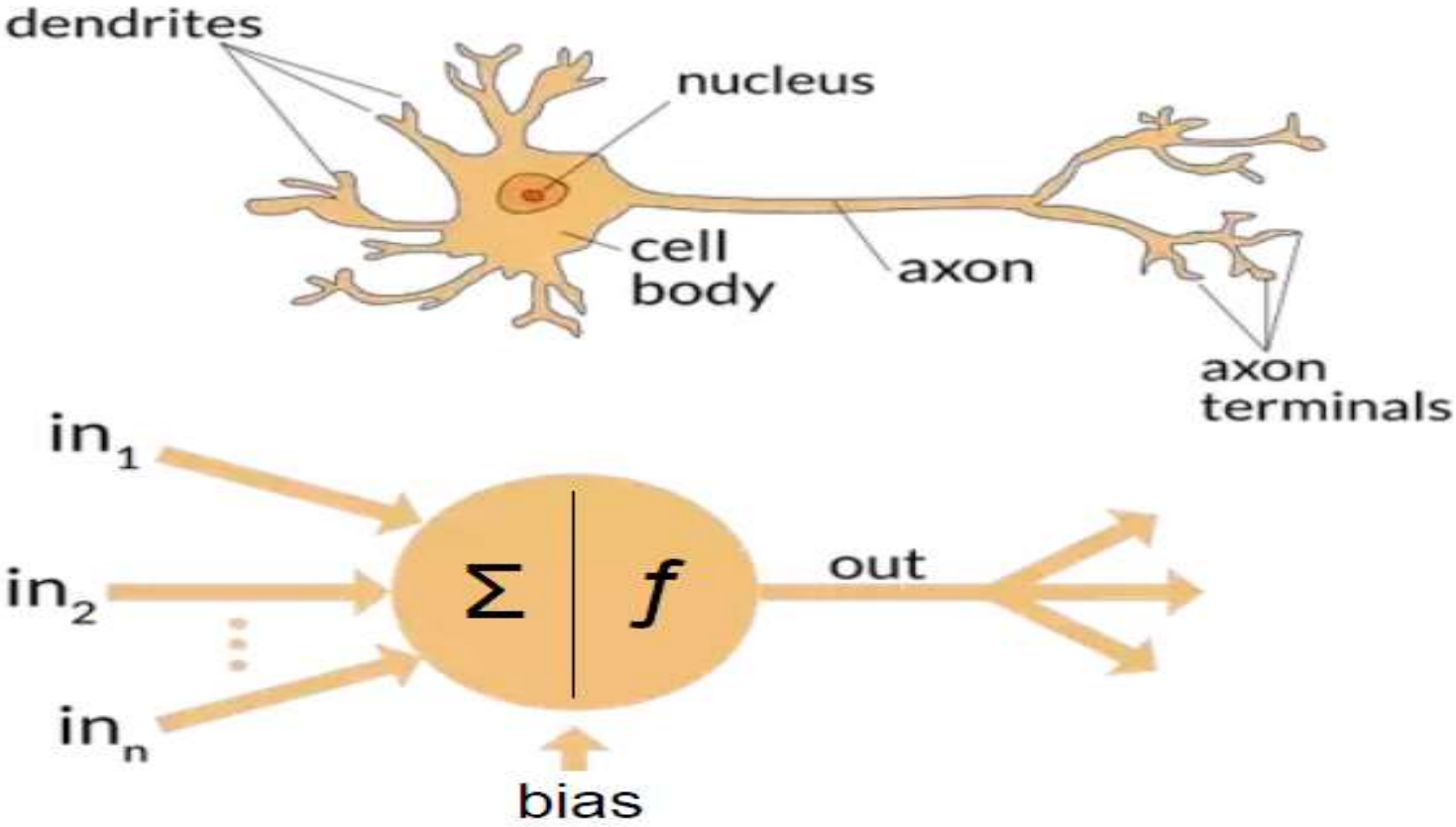

Hidden Layers
Input Layer
Output Layer

Neural Networks Multilayer Perceptron

- Biological neural networks (brains) are composed of roughly 86 billion neurons connected to many other neurons.
- From an information processing point of view a biological neuron is an excitable unit that can process and transmit information via electrical and chemical signals.
- A neuron in the biological brain is considered a main component of the brain, spinal cord of the central nervous system, and the ganglia of the peripheral nervous system.
- Biological neural networks are considerably more complex (several orders of magnitude) than the artificial neural network versions

The biological neuron

- It is a nerve cell that provides the fundamental functional unit for the nervous systems of all animals.
- Neurons exist to communicate with one another, and pass electro-chemical impulses across synapses, from one cell to the next, as long as the impulse is strong enough to activate the release of chemicals across a synaptic cleft.
- The strength of the impulse must surpass a minimum threshold or chemicals will not be released.
- the major parts of the nerve cell are:
  - **Soma**
  - **Dendrites**
  - **Axons**
  - **Synapses**

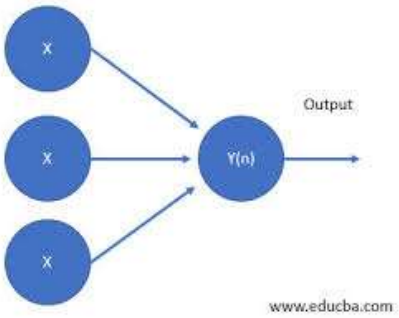

dendrites
nucleus
cell body
axon
axon terminals
in1
in2
inn
Σ
f
out
bias

Biological Neuron	Artificial Neuron
Cell Nucleus (Soma)	Node
Dendrites	Input
Synapse	Weights or interconnections
Axon	Output

- The behavior of neural networks is shaped by its network architecture.
- A network's architecture can be defined (in part) by the following:
  - **Number of neurons**
  - **Number of layers**
  - **Types of connections between layers**

- There are two main properties of artificial neural networks that follow the general idea of how the brain works.
- First is that the most basic unit of the neural network is the artificial neuron (or node in shorthand).
- Artificial neurons are modeled on the biological neurons of the brain, and like biological neurons, they are stimulated by inputs.
- These artificial neurons pass on some—but not all—information they receive to other artificial neurons, often with transformations.
- Second, much as the neurons in the brain can be trained to pass forward only signals that are useful in achieving the larger goals of the brain, we can train the neurons of a neural network to pass along only useful signals

- The most well-known and simplest-to-understand neural network is the feedforward multilayer neural network.
- It has an input layer, one or many hidden layers, and a single output layer.
- Each layer can have a different number of neurons and each layer is fully connected to the adjacent layer. The connections between the neurons in the layers form an acyclic graph



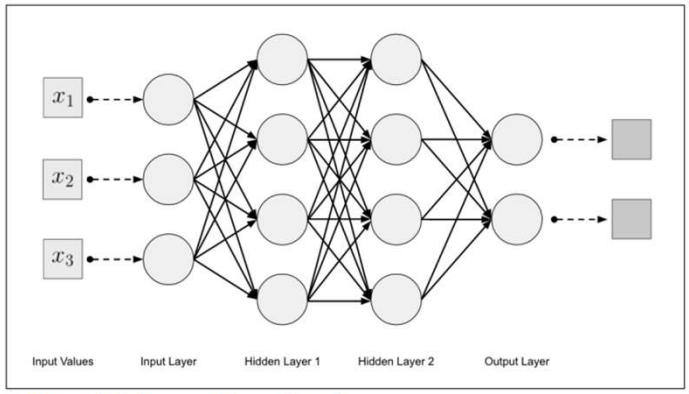


Figure 2-1. Multilayer neural network topology

- Neural networks are trained using techniques called **feedforward propagation** and **backpropagation**.
- During feedforward propagation, input data is passed through the network layer by layer, with each layer performing a computation based on the inputs it receives and passing the result to the next layer.
- Backpropagation is an algorithm used to train neural networks by iteratively adjusting the network's weights and biases in order to minimize the loss function.

- A **feed-forward multilayer neural network** can represent any function, given enough artificial neuron units.
- It is generally trained by a learning algorithm called backpropagation learning.
- Backpropagation uses gradient descent on the weights of the connections in a neural network to minimize the error on the output of the network
- Training a neural network involves feeding it a set of input-output pairs (training data) and adjusting the weights to minimize the difference between the network's predictions and the actual outputs.
- This process, often referred to as supervised learning, utilizes techniques like gradient descent to iteratively update the weights based on the error between the predicted and actual outputs.

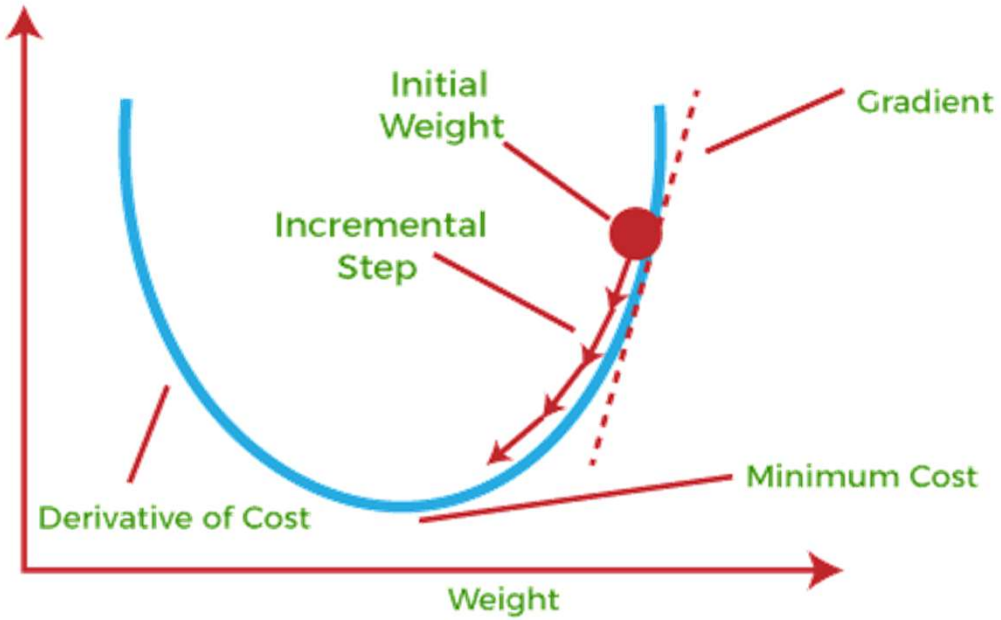
Gradient Descent

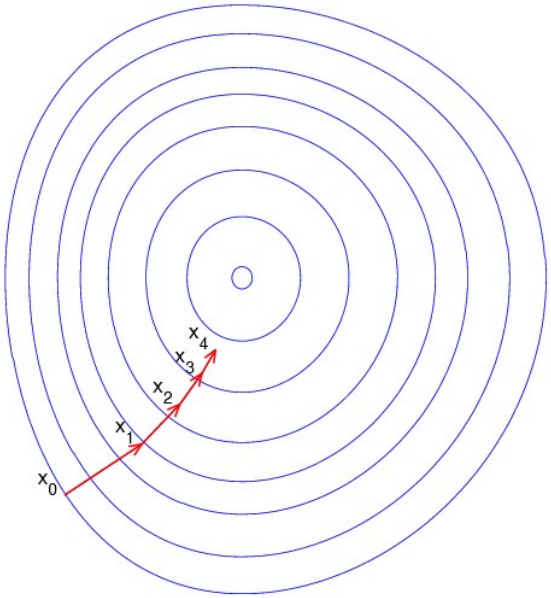
- Picture yourself hiking down a mountain in thick fog. You can't see the bottom, but you want to reach it (find the minimum of a function). You take small steps in the direction that seems to go downhill the most (gradient).
- Sometimes the path might be steep (large gradients) or flat (small gradients), and you adjust your steps accordingly.
- Gradient descent helps in finding the optimal parameters for a model by iteratively moving in the direction that reduces the error.
- Harnessing the potential of gradient descent is like mastering the art of navigating through uncertainty—each step guided by data, precision, and perseverance

Gradient Descent

- Gradient Descent is known as one of the most commonly used optimization algorithms to train machine learning models by means of **minimizing errors between actual and expected results**. Further, gradient descent is also used to train Neural Networks.
- gradient descent is the go-to algorithm for navigating the complex landscape of machine learning and deep learning.
- It helps models find the optimal set of parameters by iteratively adjusting them in the opposite direction of the gradient.
- Gradient descent is an optimization algorithm used in machine learning to minimize the cost function by iteratively adjusting parameters in the direction of the negative gradient, aiming to find the optimal set of parameters.

- ***Gradient Descent is defined as one of the most commonly used iterative optimization algorithms of machine learning to train the machine learning and deep learning models. It helps in finding the local minimum of a function.***
- The best way to define the local minimum or local maximum of a function using gradient descent is as follows:
- If we move towards a negative gradient or away from the gradient of the function at the current point, it will give the **local minimum** of that function.
- Whenever we move towards a positive gradient or towards the gradient of the function at the current point, we will get the **local maximum** of that function.


Initial
Weight
Gradient
Incremental
Step
Minimum Cost
Derivative of Cost
Weight


x0
x1
x2
x3
x4

- A gradient simply measures the change in all weights with regard to the change in error.
- You can also think of a gradient as the slope of a function. The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops learning.
- "A gradient measures how much the output of a function changes if you change the inputs a little bit." — Lex Fridman (MIT)
- The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops learning. In mathematical terms, a gradient is a partial derivative with respect to its inputs.
- In mathematical terms, a gradient is a partial derivative with respect to its inputs.
- Example: guessing age of a person
- **New value = old value – step size**
- **Step size = learning rate * slope**

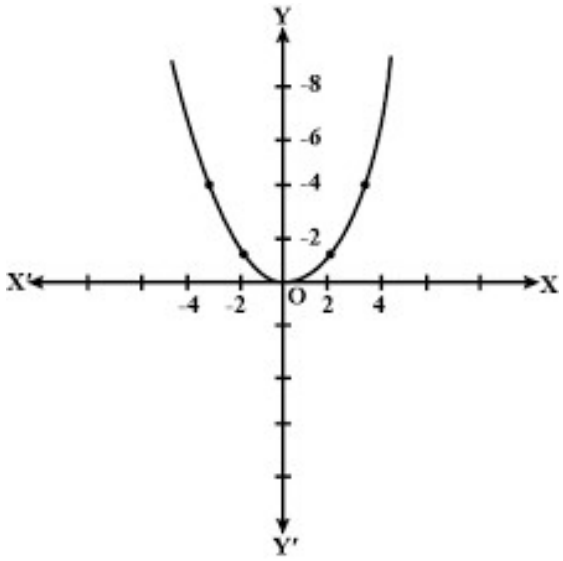
- ***The main objective of using a gradient descent algorithm is to minimize the cost function using iteration.*** To achieve this goal, it performs two steps iteratively:
- Calculates the first-order derivative of the function to compute the gradient or slope of that function.
- Move away from the direction of the gradient, which means slope increased from the current point by alpha times, where Alpha is defined as Learning Rate. It is a tuning parameter in the optimization process which helps to decide the length of the steps.

- ***The cost function is defined as the measurement of difference or error between actual values and expected values at the current position and present in the form of a single real number.***
- It helps to increase and improve machine learning efficiency by providing feedback to this model so that it can minimize error and find the local or global minimum.
- Further, it continuously iterates along the direction of the negative gradient until the cost function approaches zero. At this steepest descent point, the model will stop learning further.
- Although cost function and loss function are considered synonymous, also there is a minor difference between them. The slight difference between the loss function and the cost function is about the error within the training of machine learning models, as loss function refers to the error of one training example, while a cost function calculates the average error across an entire training set.
- The cost function is calculated after making a hypothesis with initial parameters and modifying these parameters using gradient descent algorithms over known data to reduce the cost function.

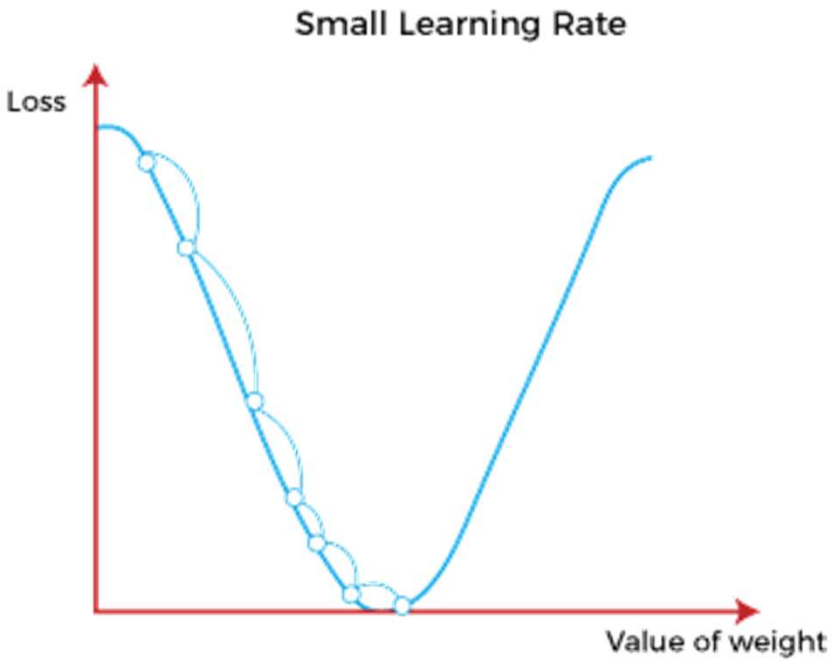
Example

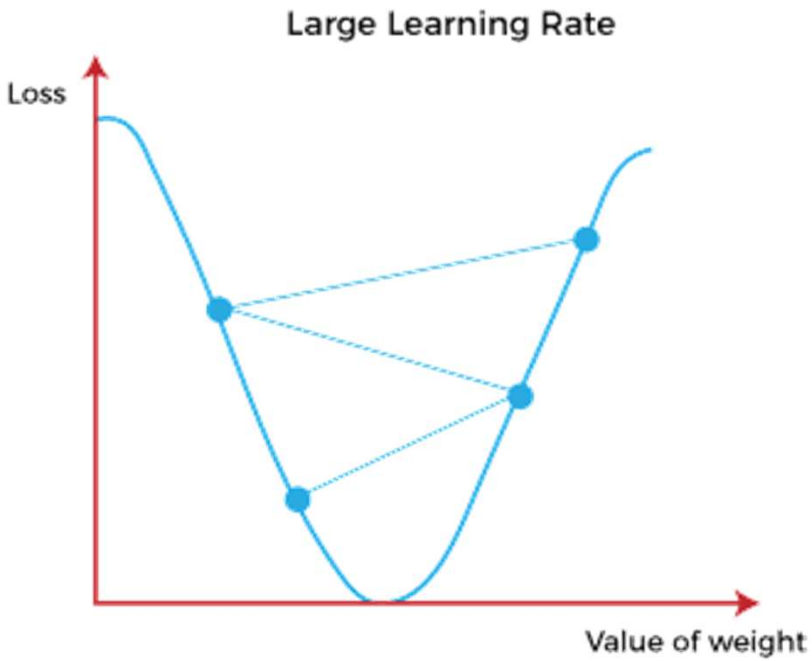
- Lets take a function $f(x) = x^2$
- Function needs to be minimized, we have to decide in which direction to move (up or down) to reach 0 for that we have to compute
- Derivative (slope)
- Slope helps us decide whether to move upward

or downward



- **Learning Rate:**
- It is defined as the step size taken to reach the minimum or lowest point.
- This is typically a small value that is evaluated and updated based on the behavior of the cost function.
- If the learning rate is high, it results in larger steps but also leads to risks of overshooting the minimum.
- At the same time, a low learning rate shows the small step sizes, which compromises overall efficiency but gives the advantage of more precision.


Small Learning Rate
Loss
Value of weight


Large Learning Rate
Loss
Value of weight

- **Example: Minimizing a Quadratic Function**
- Let's consider a simple quadratic function: **$f(x)=x^2+4x+4$**
- **Step-by-Step Explanation:**

1. **Objective**: Minimize the function f(x).
2. **Gradient Calculation**: The gradient (or derivative) of the function gives the slope of the function at any point **$f'(x)=d/dx(x^2+4x+4)=2x+4$**
3. **Initial Guess**: Start with an initial guess, say x0.
4. **Learning Rate**: Choose a learning rate (step size), α which determines how large a step we take in each iteration. Let's set α=0.1

5. **Iteration:** Update the value of x iteratively using the formula:$x_{new}=x_{old}-\alpha \cdot f'(x_{old})$

Let's go through a few iterations to see how x changes

Iteration 1:Initial value: $x_0=3$

Gradient at x_0 : $f'(3)=2(3)+4=10$

Update: $x_1= 3 – 0.1 * 10 = 3-1 =2$

Iteration 2: New value: $x_1=2$

Gradient at x_1 $f'(2)=2(2)+4=8$

Update: $x_2=2-0.1* 8=2-0.8 = 1.2$

Iteration 3: New value: x_2=1.2

Gradient at x_2 f'(1.2)=2(1.2)+4=6.4

Update: x_3=1.2−0.1* 6.4=2−0.8 = 0.56

Iteration 3: New value: x_3=0.56

Gradient at x_3 f'(0.56)=2(0.56)+4=5.12

Update: x_4=0.56−0.1* 5.12= 0.048

- As you can see, the value of x is getting closer to the minimum point of the function with each iteration.

Types of Gradient Descent

- Based on the error in various training models, the Gradient Descent learning algorithm can be divided into **Batch gradient descent, stochastic gradient descent, and mini-batch gradient descent.**
- **1. Batch Gradient Descent:**
- Batch gradient descent (BGD) is used to find the error for each point in the training set and update the model after evaluating all training examples. This procedure is known as the training epoch. In simple words, it is a greedy approach where we have to sum over all examples for each update.
- Batch gradient descent is a variant of the gradient descent optimization algorithm where the entire dataset is used to compute the gradient of the cost function before updating the model parameters.
- **Advantages of Batch gradient descent:**
- It produces less noise in comparison to other gradient descent.
- It produces stable gradient descent convergence.
- It is Computationally efficient as all resources are used for all training samples.
- **(When all data points are used, individual outliers or noisy data points have less impact on the computed gradient. The averaging effect smooths out the noise, leading to a more stable and consistent update direction.)**

- **2. Stochastic gradient descent**
- Stochastic gradient descent (SGD) is a type of gradient descent that runs one training example per iteration. Or in other words, it processes a training epoch for each example within a dataset and updates each training example's parameters one at a time.
- In stochastic gradient descent, the gradient is computed using a single randomly chosen data point at each iteration.
- As it requires only one training example at a time, hence it is easier to store in allocated memory. However, it shows some computational efficiency losses in comparison to batch gradient systems as it shows frequent updates that require more detail and speed. Further, due to frequent updates, it is also treated as a noisy gradient.
- **Advantages of Stochastic gradient descent:**
- In Stochastic gradient descent (SGD), learning happens on every example, and it consists of a few advantages over other gradient descent.
- It is easier to allocate in desired memory.
- It is relatively fast to compute than batch gradient descent.
- It is more efficient for large datasets.

- **3. MiniBatch Gradient Descent:**
- Mini Batch gradient descent is the combination of both batch gradient descent and stochastic gradient descent.
- It divides the training datasets into small batch sizes then performs the updates on those batches separately.
- Splitting training datasets into smaller batches make a balance to maintain the computational efficiency of batch gradient descent and speed of stochastic gradient descent. Hence, we can achieve a special type of gradient descent with higher computational efficiency and less noisy gradient descent.
- **Advantages of Mini Batch gradient descent:**
- It is easier to fit in allocated memory.
- It is computationally efficient.
- It produces stable gradient descent convergence.

Working of SGD

1. **Initialization:** SGD starts with an initial set of model parameters (weights and biases) randomly or using some predefined method.
2. **Iterative Optimization:** The aim of this step is to find the minimum of a loss function, by iteratively moving in the direction of the steepest decrease in the function's value.

- For each iteration (or epoch) of training:
- Shuffle the training data to ensure that the model doesn't learn from the same patterns in the same order every time.
- Split the training data into mini-batches (small subsets of data).

- For each mini-batch:
- Compute the gradient of the loss function with respect to the model parameters using only the data points in the mini-batch. This gradient estimation is a stochastic approximation of the true gradient.
- Update the model parameters by taking a step in the opposite direction of the gradient, scaled by a learning rate:
- $\Theta t+1 = \theta t - n * \nabla J(\theta t)$
- Where:
- **θt** represents the model parameters at iteration t. This parameter can be the weight
- **$\nabla J(\theta t)$** is the gradient of the loss function J with respect to the parameters θt
- **n** is the learning rate, which controls the size of the steps taken during optimization

- **3. Direction of Descent:** The gradient of the loss function indicates the direction of the steepest ascent. To minimize the loss function, gradient descent moves in the opposite direction, towards the steepest descent.
- **4. Learning Rate:** The step size taken in each iteration of gradient descent is determined by a parameter called the learning rate, denoted above as n. This parameter controls the size of the steps taken towards the minimum. If the learning rate is too small, convergence may be slow; if it is too large, the algorithm may oscillate or diverge.
- **5. Convergence:** Repeat the process for a fixed number of iterations or until a convergence criterion is met (e.g., the change in loss function is below a certain threshold).

- Stochastic gradient descent updates the model parameters more frequently using smaller subsets of data, making it computationally efficient, especially for large datasets.
- The randomness introduced by SGD can have a regularization effect, preventing the model from overfitting to the training data.
- It is also well-suited for online learning scenarios where new data becomes available incrementally, as it can update the model quickly with each new data point or mini-batch.

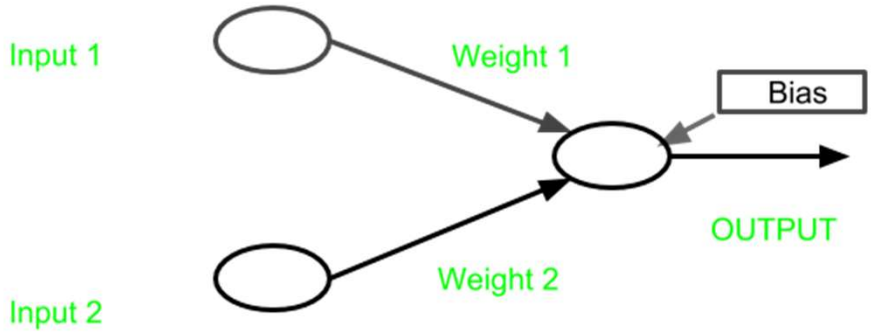
- **Practical 1:-** Implement gradient descent algorithm for y = mx+c for finding optimal values of m and c.

The Perceptron

- A Perceptron is an artificial neuron, and thus a neural network unit. It performs computations to detect features or patterns in the input data. It is an algorithm for supervised learning of binary classifiers. It is this algorithm that allows artificial neurons to learn and process features in a data set.
- The perceptron is a linear-model binary classifier with a simple input–output relationship as depicted, which shows we're summing n number of inputs times their associated weights and then sending this "net input" to a step function with a defined threshold.
- Typically with perceptron, this is a Heaviside step function with a threshold value of 0.5. This function will output a real-valued single binary value (0 or a 1), depending on the input.

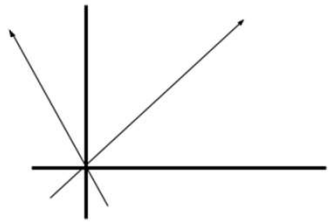
- In Neural network, some inputs are provided to an artificial neuron, and with each input a weight is associated.
- Weight increases the steepness of activation function. This means weight decide how fast the activation function will trigger
- whereas bias is used to delay the triggering of the activation function.
- The weight shows the effectiveness of a particular input. More the weight of input, more it will have impact on network.
- On the other hand Bias is like the intercept added in a linear equation. It is an additional parameter in the Neural Network which is used to adjust the output along with the weighted sum of the inputs to the neuron. Therefore Bias is a constant which helps the model in a way that it can fit best for the given data.

- The processing done by a neuron is thus denoted as :
- **output = sum (weights * inputs) + bias**

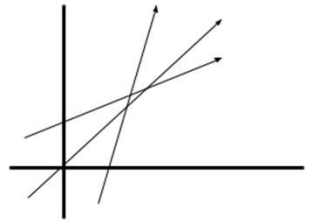


Output = weight1 * input1 + weight2 * input2 +bias

- **Need of bias**
- y = mx+c
- where
- m = weight and c = bias
- Now suppose if c (bias) is absent



Y = mx



Y = mx + c

- Due to absence of bias, model will train over point passing through origin only, which is not in accordance with real-world scenario. Also with the introduction of bias, the model will become more flexible.
- Overall, the bias term in neural networks plays a crucial role in enhancing the model's flexibility, improving its ability to capture complex relationships in the data, and ultimately, boosting its performance and robustness in various machine learning tasks.

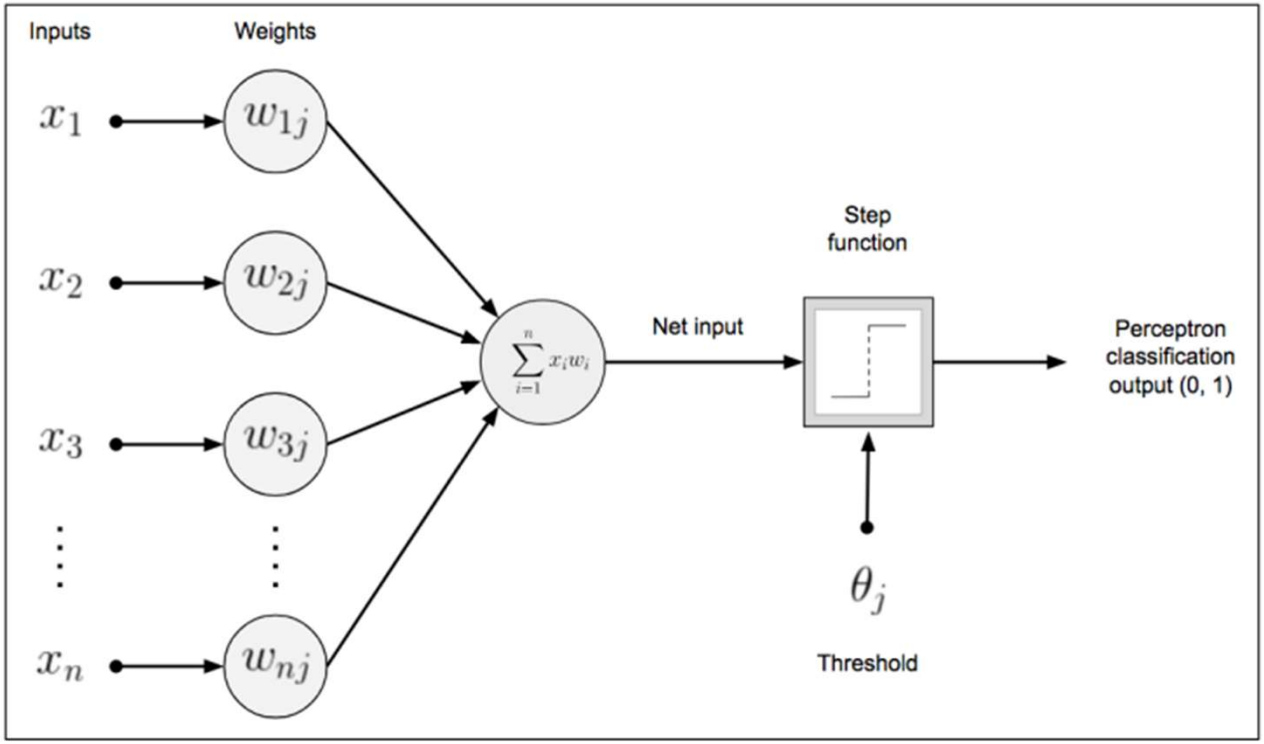


Figure 2-3. Single-layer perceptron

- We can model the decision boundary and the classification output in the Heaviside step function equation, as follows:

$$f(x) = \begin{cases} 0 & x < 0 \\ 1 & x >= 0 \end{cases}$$

- To produce the net input to **the activation function** (here, the Heaviside step function) we take the dot product of the input and the connection weights.

Table 2-1. The summation function parameters

Function parameter	Description
w	Vector of real-valued weights on the connections
w · **x**	Dot product ($\sum_{i=1}^{n} w_i x_i$)
n	Number of inputs to the perceptron
b	The bias term (input value does not affect its value; shifts decision boundary away from origin)

Characteristics of perceptron

- The Perceptron model in machine learning is characterized by the following key points:
- **Binary Linear Classifier**: The Perceptron is a type of binary classifier that assigns input data points to one of two possible categories.
- **Input Processing:** It takes multiple input signals and processes them, each multiplied by a corresponding weight. The inputs are aggregated, and the model produces a single output.
- **Activation Function:** The Perceptron uses an activation function, typically a step function, to determine the output based on the aggregated inputs and weights. If the result exceeds a certain threshold, the output is one category; otherwise, it's the other.

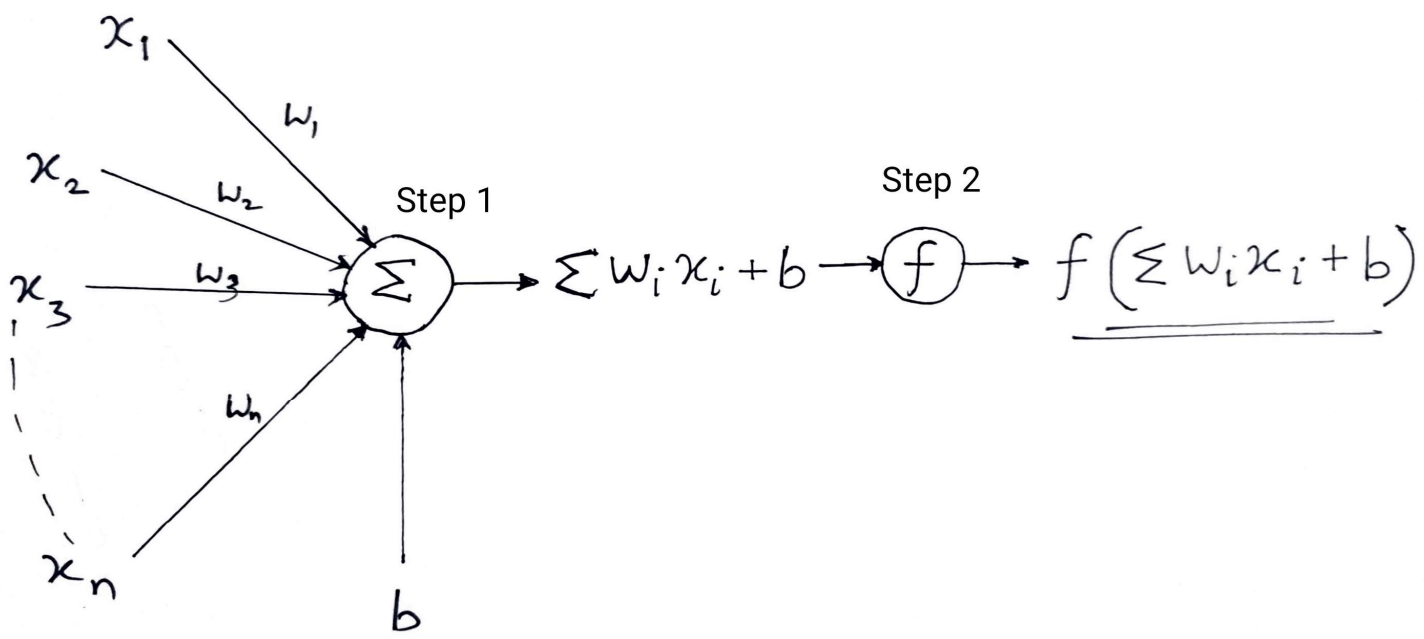
- **Training Process:** During training, the model adjusts its weights based on the error in its predictions compared to the actual outcomes. This adjustment helps improve the model's accuracy over time.
- **Single-Layer Model:** The Perceptron is a single-layer neural network since it has only one layer of output after processing the inputs.
- **Limitations:** While effective for linearly separable data, the Perceptron has limitations in handling more complex patterns, leading to the development of more sophisticated neural network architectures.

Main Components of Perceptron

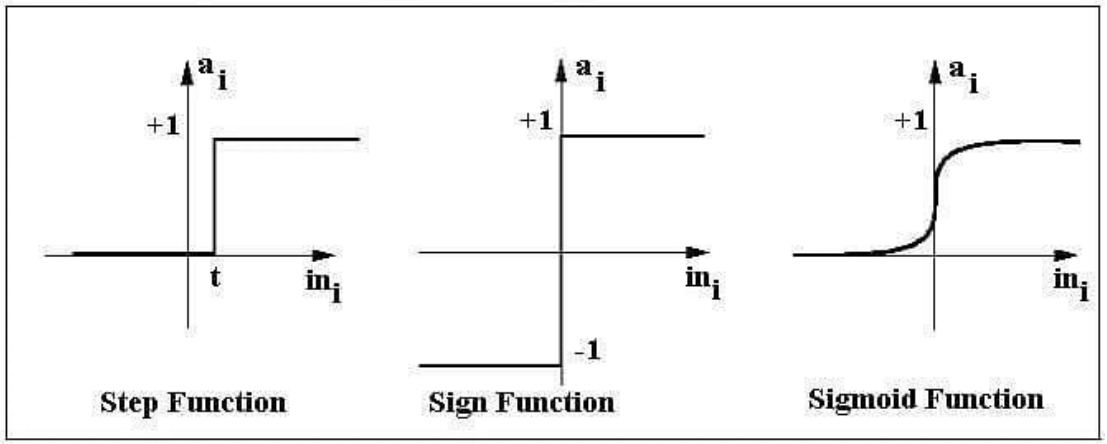
- The perceptron consists of 3 main parts:
- **Input nodes or input layer:** The input layer takes the initial data into the system for further processing. Each input node is associated with a numerical value. It can take any real value.
- **Weights and bias:** Weight parameters represent the strength of the connection between units. Higher is the weight, stronger is the influence of the associated input neuron to decide the output. Bias plays the same as the intercept in a linear equation.
- **Activation function:** The activation function determines whether the neuron will fire or not. At its simplest, the activation function is a step function, but based on the scenario, different activation functions can be used.

Working of a Perceptron

- In the first step, all the input values are multiplied with their respective weights and added together.
- The result obtained is called weighted sum $\sum wi*xi$, or stated differently, $x1*w1 + x2*w2 + \ldots wn*xn$.
- This sum gives an appropriate representation of the inputs based on their importance. Additionally, a bias term b is added to this sum $\sum wi*xi + b$. Bias serves as another model parameter (in addition to weights) that can be tuned to improve the model's performance.
- In the second step, an activation function f is applied over the above sum $\sum wi*xi + b$ to obtain output $Y = f(\sum wi*xi + b)$. Depending upon the scenario and the activation function used, the Output is either binary {1, 0} or a continuous value.


x_1
w_1
x_2
w_2
Step 1
Step 2
x_3
w_3
Σ
$\sum w_i x_i + b$
f
$f\left(\sum w_i x_i + b\right)$
w_n
x_n
b

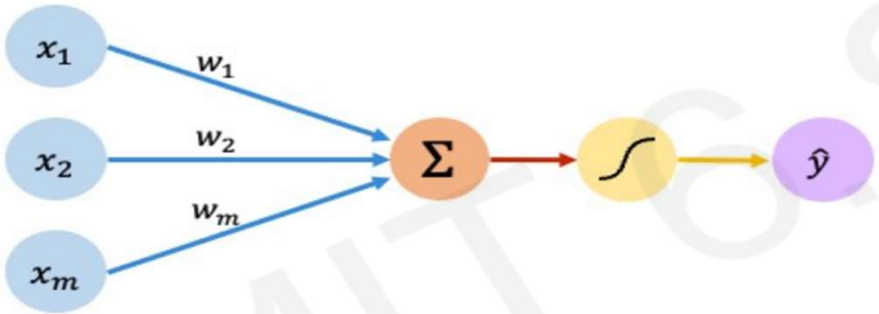
```
If  ∑wi*xi + b > 0:
    output = 1
else:
    output = 0
```



Types of Perceptron models

- **Single Layer Perceptron model:** One of the easiest ANN(Artificial Neural Networks) types consists of a feed-forward network and includes a threshold transfer inside the model. The main objective of the single-layer perceptron model is to analyze the linearly separable objects with binary outcomes. A Single-layer perceptron can learn only linearly separable patterns.
- **Multi-Layered Perceptron model:** It is mainly similar to a single-layer perceptron model but has more hidden layers.

The Perceptron: Forward Propagation



Inputs Weights Sum Non-Linearity Output

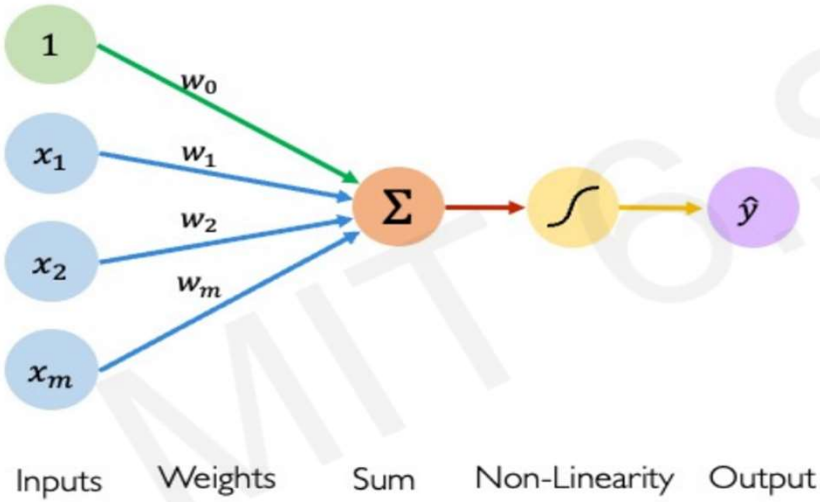
Output

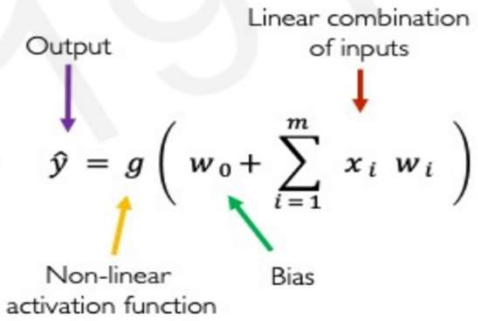
Linear combination of inputs

$$\hat{y} = g\left(\sum_{i=1}^{m} x_i \, w_i\right)$$

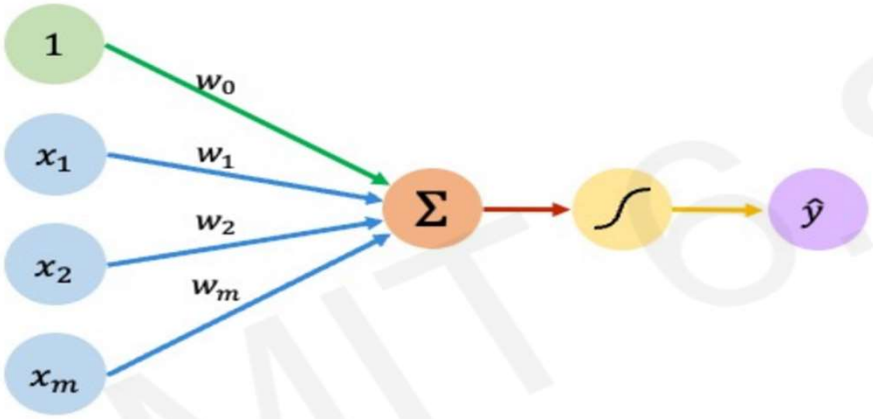
Non-linear activation function

The Perceptron: Forward Propagation





The Perceptron: Forward Propagation

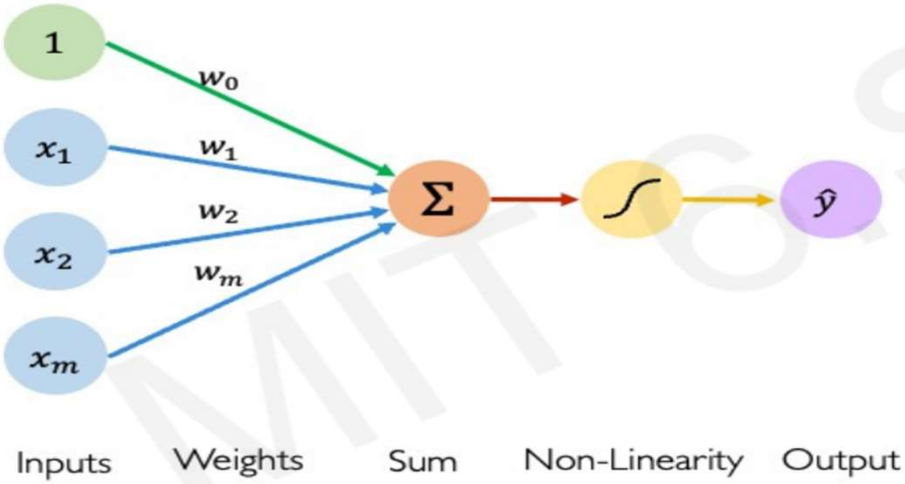


$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i\, w_i\right)$$

$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W}\right)$$

where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

The Perceptron: Forward Propagation

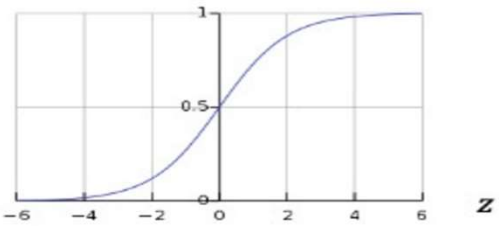


Activation Functions

$$\hat{y} = g(w_0 + \boldsymbol{X}^T \boldsymbol{W})$$

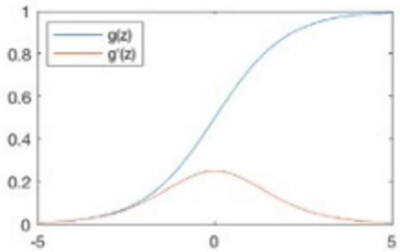
- Example: sigmoid function

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



Common Activation Functions

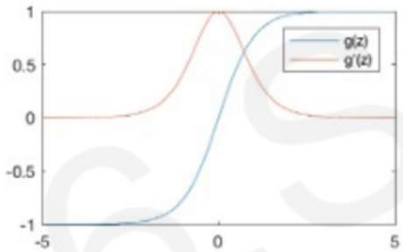
Sigmoid Function



$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

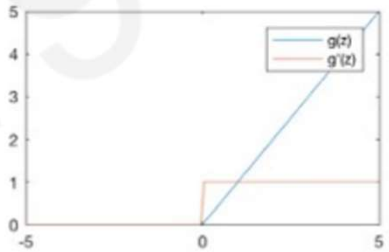
Hyperbolic Tangent



$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)

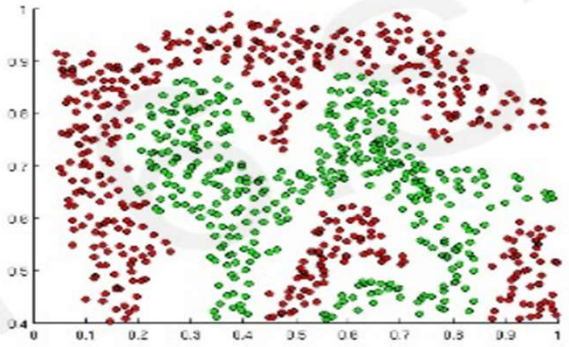


$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

Importance of Activation Functions

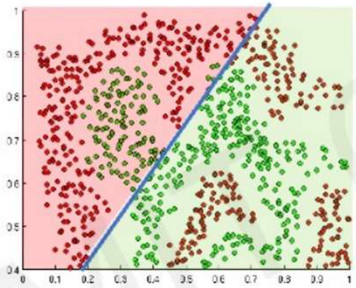
*The purpose of activation functions is to **introduce non-linearities** into the network*



What if we wanted to build a neural network to distinguish green vs red points?

Importance of Activation Functions

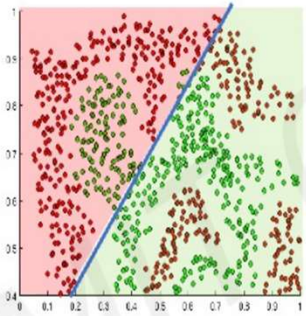
*The purpose of activation functions is to **introduce non-linearities** into the network*



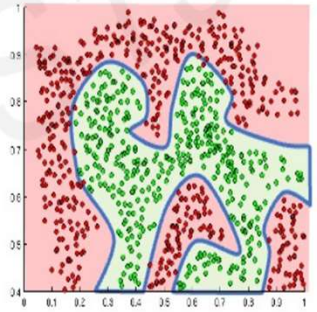
Linear activation functions produce linear decisions no matter the network size

Importance of Activation Functions

*The purpose of activation functions is to **introduce non-linearities** into the network*



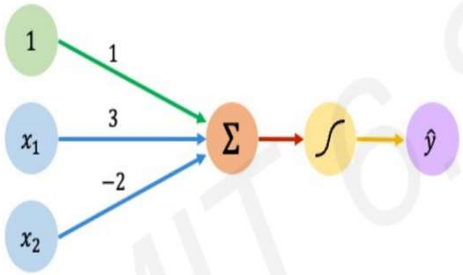
Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

MIT 6.S191

The Perceptron: Example



We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

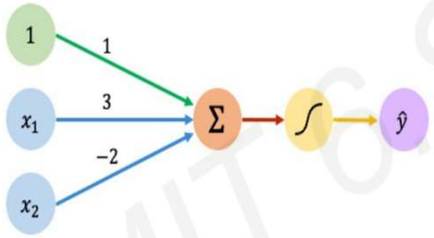
$$\hat{y} = g(w_0 + \boldsymbol{X}^T \boldsymbol{W})$$

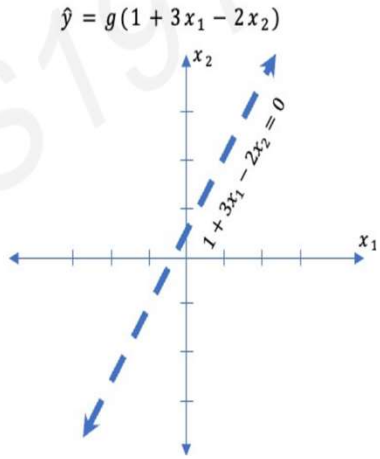
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$

$$\hat{y} = g(1 + 3x_1 - 2x_2)$$

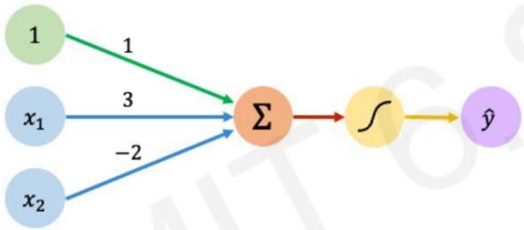
This is just a line in 2D!

The Perceptron: Example



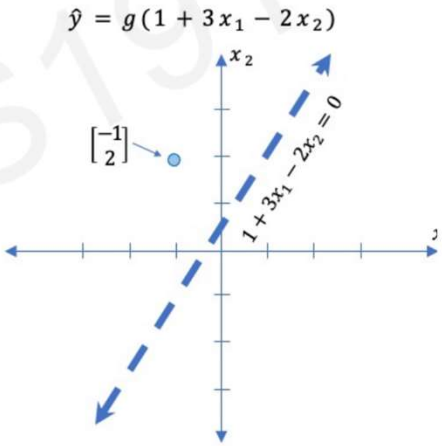


The Perceptron: Example

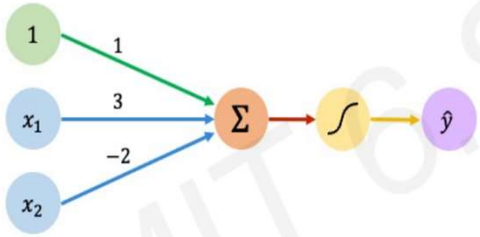


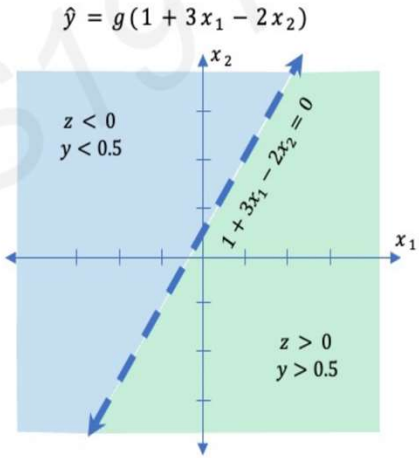
Assume we have input: $\boldsymbol{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

$\hat{y} = g(1 + (3 * -1) - (2 * 2))$
$= g(-6) \approx 0.002$



The Perceptron: Example





- **Practical 2:- Implement all activation functions in python**
- **Practical 3:- Implement a perceptron in python from scratch**

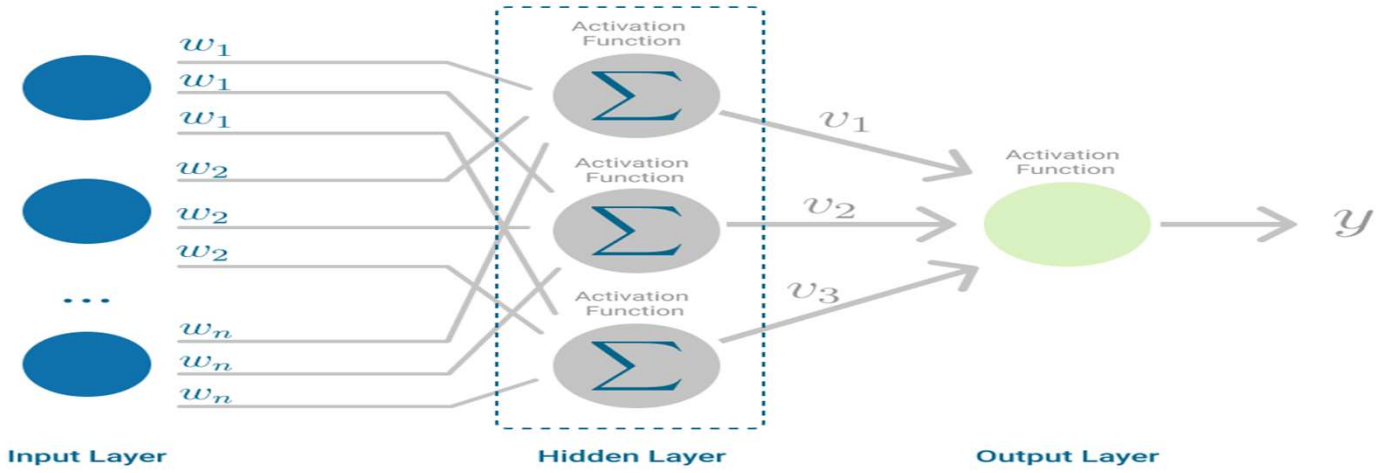
Limitations of the Single Layer Perceptron Algorithm

- The single layer perceptron is a very simple algorithm that can only learn linear decision boundaries.
- This means that it is not well suited for more complex problems where the data is not linearly separable.
- Additionally, the single layer perceptron is a batch learning algorithm, which means that it cannot learn from new data points incrementally.
- single layer perceptron is vulnerable to overfitting. This means that if the training data is not representative of the real data, the model will perform poorly on unseen data.
- single layer perceptron is also sensitive to noise. This means that if there is any noise in the training data, it will adversely affect the performance of the model.

Multilayer Perceptron

- The **Multilayer Perceptron** was developed to tackle limitations of single layer perceptron. It is a neural network where the mapping between inputs and output is non-linear
- A multi-layer perceptron (MLP) is a type of artificial neural network consisting of multiple layers of neurons.
- The neurons in the MLP typically use nonlinear activation functions, allowing the network to learn complex patterns in data.
- MLPs are significant in machine learning because they can learn nonlinear relationships in data, making them powerful models for tasks such as classification, regression, and pattern recognition.

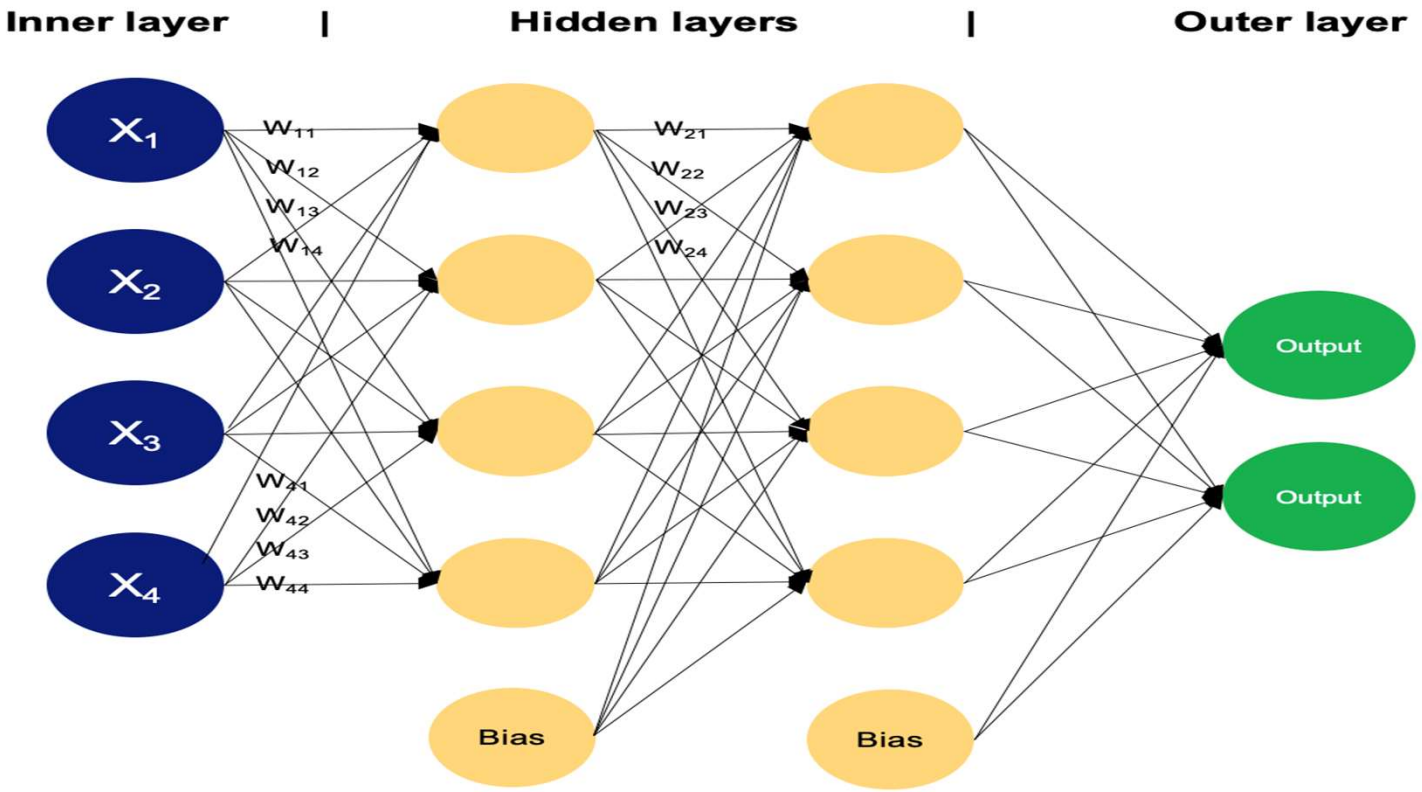
- A Multilayer Perceptron has input and output layers, and one or more **hidden layers** with many neurons stacked together. And while in the Perceptron the neuron must have an activation function that imposes a threshold, like ReLU or sigmoid, neurons in a Multilayer Perceptron can use any arbitrary activation function.



- Multilayer Perceptron falls under the category of **feedforward algorithms**, because inputs are combined with the initial weights in a weighted sum and subjected to the activation function, just like in the Perceptron. But the difference is that each linear combination is propagated to the next layer.
- Each layer is *feeding* the next one with the result of their computation, their internal representation of the data. This goes all the way through the hidden layers to the output layer.
- If the algorithm only computed the weighted sums in each neuron, propagated results to the output layer, and stopped there, it wouldn't be able to *learn* the weights that minimize the cost function. If the algorithm only computed one iteration, there would be no actual learning.
- This is where **Backpropagation** comes into play.

Working of MLP

- In a multilayer perceptron, neurons process information in a step-by-step manner, performing computations that involve weighted sums and nonlinear transformations. Let's walk layer by layer to see the magic that goes within.
- **Input layer**
- The input layer of an MLP receives input data, which could be features extracted from the input samples in a dataset. Each neuron in the input layer represents one feature.
- Neurons in the input layer do not perform any computations; they simply pass the input values to the neurons in the first hidden layer.


Inner layer
|
Hidden layers
|
Outer layer
X1
X2
X3
X4
W11
W12
W13
W14
W21
W22
W23
W24
W41
W42
W43
W44
Bias
Bias
Output
Output

- **Hidden layers**
- The hidden layers of an MLP consist of interconnected neurons that perform computations on the input data.
- Each neuron in a hidden layer receives input from all neurons in the previous layer. The inputs are multiplied by corresponding weights, denoted as w. The weights determine how much influence the input from one neuron has on the output of another.
- In addition to weights, each neuron in the hidden layer has an associated bias, denoted as b. The bias provides an additional input to the neuron, allowing it to adjust its output threshold. Like weights, biases are learned during training.
- For each neuron in a hidden layer or the output layer, the weighted sum of its inputs is computed. This involves multiplying each input by its corresponding weight, summing up these products, and adding the bias:

$$\text{Weighted Sum} = \sum_{i=1}^{n} (w_i * x_i) + b$$

- Where n is the total number of input connections, wi is the weight for the i-th input, and xi is the i-th input value.
- The weighted sum is then passed through an activation function, denoted as f. The activation function introduces nonlinearity into the network, allowing it to learn and represent complex relationships in the data. The activation function determines the output range of the neuron and its behavior in response to different input values. The choice of activation function depends on the nature of the task and the desired properties of the network.

- **Output layer**
- The output layer of an MLP produces the final predictions or outputs of the network. The number of neurons in the output layer depends on the task being performed (e.g., binary classification, multi-class classification, regression).
- Each neuron in the output layer receives input from the neurons in the last hidden layer and applies an activation function. This activation function is usually different from those used in the hidden layers and produces the final output value or prediction.

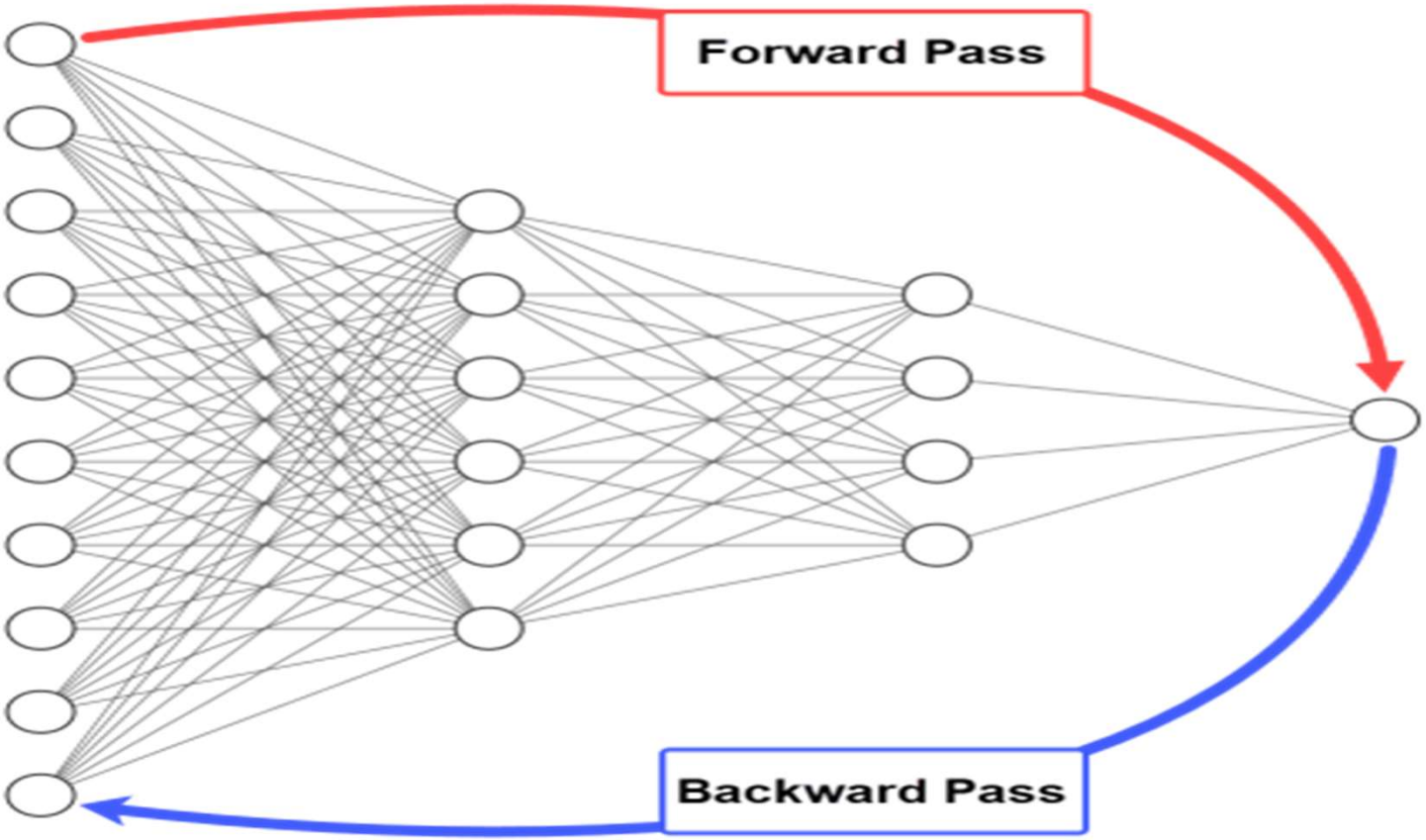
- During the training process, the network learns to adjust the weights associated with each neuron's inputs to minimize the discrepancy between the predicted outputs and the true target values in the training data.
- By adjusting the weights and learning the appropriate activation functions, the network learns to approximate complex patterns and relationships in the data, enabling it to make accurate predictions on new, unseen samples.
- This adjustment is guided by an optimization algorithm, such as stochastic gradient descent (SGD), which computes the gradients of a loss function with respect to the weights and updates the weights iteratively.

Backpropagation

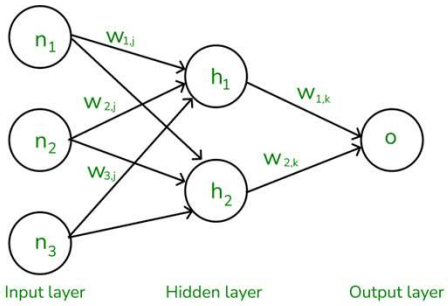
- Backpropagation is short for “backward propagation of errors.” In the context of backpropagation, SGD involves updating the network's parameters iteratively based on the gradients computed during each batch of training data.
- Instead of computing the gradients using the entire training dataset (which can be computationally expensive for large datasets), SGD computes the gradients using small random subsets of the data called mini-batches. Here’s an overview of how backpropagation algorithm works:
- **Backpropagation** is one of the important concepts of a neural network. Our task is to classify our data best. For this, we have to update the weights of parameter and bias, but how can we do that in a deep neural network?
- In the linear regression model, we use gradient descent to optimize the parameter. Similarly here we also use gradient descent algorithm using Backpropagation.

- The main features of Backpropagation are the iterative, recursive and efficient method through which it calculates the updated weight to improve the network until it is not able to perform the task for which it is being trained.
- Derivatives of the activation function to be known at network design time is required to Backpropagation.
- Now, how error function is used in Backpropagation and how Backpropagation works? Let start with an example and do it mathematically to understand how exactly updates the weight using Backpropagation.

- The Backpropagation algorithm works by two different passes, they are:
- **Forward pass**
- **Backward pass**
- **How does Forward pass work?**
- In forward pass, initially the input is fed into the input layer. Since the inputs are raw data, they can be used for training our neural network.
- The inputs and their corresponding weights are passed to the hidden layer. The hidden layer performs the computation on the data it receives. If there are two hidden layers in the neural network, for instance, h_1 and h_2 are the two hidden layers, and the output of h_1 can be used as an input of h_2. Before applying it to the activation function, the bias is added.


Forward Pass
Backward Pass

- To the weighted sum of inputs, the activation function is applied in the hidden layer to each of its neurons.
- One such activation function that is commonly used is ReLU can also be used, which is responsible for returning the input if it is positive otherwise it returns zero.
- By doing this so, it introduces the non-linearity to our model, which enables the network to learn the complex relationships in the data. And finally, the weighted outputs from the last hidden layer are fed into the output to compute the final prediction, this layer can also use the activation function called the softmax function which is responsible for converting the weighted outputs into probabilities for each class.



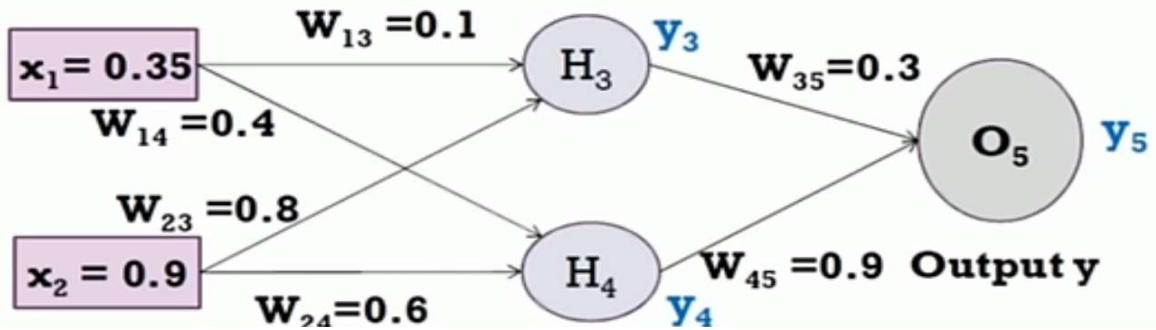
- **How does backward pass work?**
- In the backward pass process shows, the error is transmitted back to the network which helps the network, to improve its performance by learning and adjusting the internal weights.
- To find the error generated through the process of forward pass, we can use one of the most commonly used methods called mean squared error which calculates the difference between the predicted output and desired output. The formula for mean squared error is:$Meansquarederror=(predictedoutput-actualoutput)^2$

- Once we have done the calculation at the output layer, we then propagate the error backward through the network, layer by layer.
- The key calculation during the backward pass is determining the gradients for each weight and bias in the network. This gradient is responsible for telling us how much each weight/bias should be adjusted to minimize the error in the next forward pass. The chain rule is used iteratively to calculate this gradient efficiently.
- In addition to gradient calculation, the activation function also plays a crucial role in backpropagation, it works by calculating the gradients with the help of the derivative of the activation function.

- **Backpropagation Algorithm Steps:**

1. Initialize the weights of the network randomly.
2. Forward propagate an input through the network to get the predicted output.
3. Compute the error between the predicted output and the actual output.
4. Backward propagate the error through the network to compute the gradient of the loss function with respect to each weight.
5. Update the weights in the opposite direction of the gradient using an optimization algorithm such as Stochastic Gradient Descent (SGD).
6. Repeat steps 2–5 for multiple iterations until the weights converge.

- 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.



Curse of dimensionality

- Refer PCA for dimensionality reduction