

---

## Chapter-11

# Neurons, NNs, Linear Discriminants

Instructor Name: B N V Narasimha Raju

---

### 11.1 Neural Networks

Neural networks are used to mimic the basic functioning of the human brain and are inspired by how the human brain interprets information. It is used to solve various real-time tasks because of its ability to perform computations quickly and its fast responses.

An Artificial Neural Network model contains various components that are inspired by the biological nervous system. An artificial neural network has a huge number of interconnected processing elements, also known as nodes. Each node is designed to behave similarly to a neuron in the brain. These nodes are connected with other nodes using a connection link. The connection link contains weights, these weights contain information about the input signal. Each iteration and input in turn leads to an update of these weights.

After inputting all the data instances from the training data set, the final weights of the neural network, along with its architecture, are known as the trained neural network. This process is called Training of Neural Networks. This trained neural network is used to solve specific problems, as defined in the problem statement. Types of tasks that can be solved using an artificial neural network include Classification problems, Pattern Matching, Data Clustering, etc.

We use artificial neural networks because they learn very efficiently and adaptively. They have the capability to learn “how” to solve a specific problem from the training data it receive. After learning, it can be used to solve that specific problem very quickly and efficiently with high accuracy.

An artificial neuron can be thought of as a simple or multiple linear regression model with an activation function at the end. A neuron from layer  $i$  will take the output of all the neurons from layer  $i-1$  as inputs, calculate the weighted sum, and add bias to it. After this, it is sent to an activation function, as shown in figure 11.1. The activation function calculates the output value for the neuron.

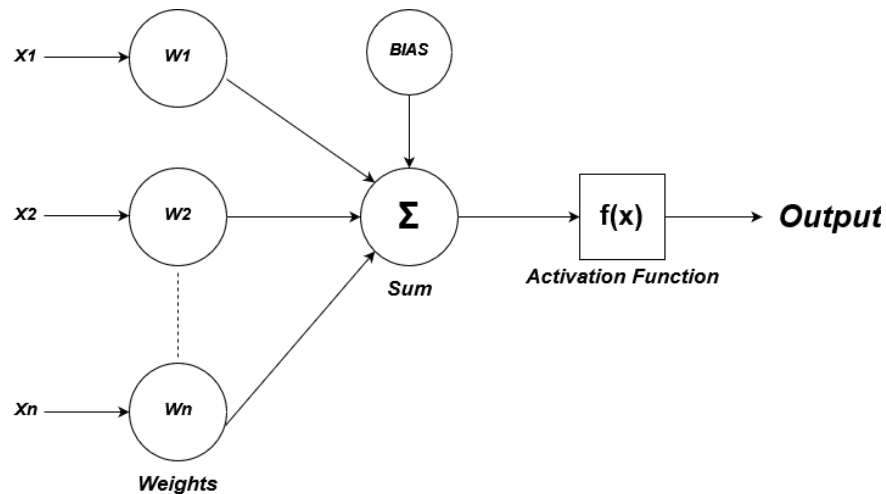


Figure 11.1: Working of Neural Network

The first neuron from the first hidden layer is connected to all the inputs from the previous layer. Similarly, the second neuron from the first hidden layer will also be connected to all the inputs from the previous layer and so on for all the neurons in the first hidden layer, as shown in figure 11.2.

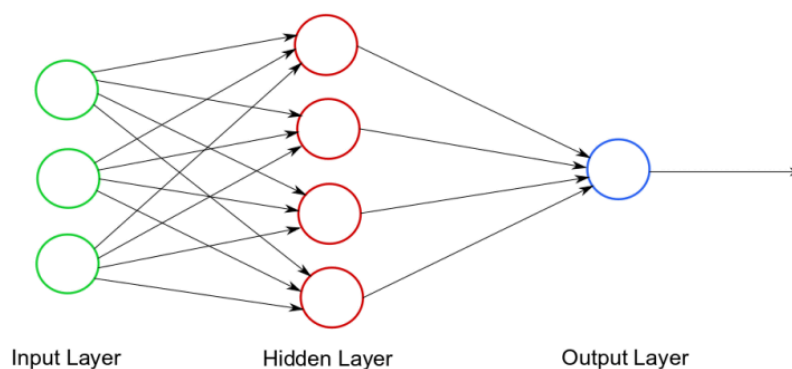


Figure 11.2: Layers of Neural Network

For neurons in the second hidden layer, the outputs of the previously hidden layer are considered as inputs and each of these neurons are connected to previous neurons, likewise. This whole process is called Forward propagation.

Once we have predicted the output, it is then compared to the actual output. We then calculate the loss and try to minimize it. To minimize the loss, use back propagation. In back propagation, first the loss is calculated, then weights and biases are adjusted in such a way that they try to minimize the loss. Weights and biases are updated with the help of an algorithm called gradient descent.

## 11.2 The perceptron

A perceptron is the smallest element of a neural network. Perceptron is a single-layer neural network or machine learning algorithm used for the supervised learning of various binary classifiers. It works as an artificial neuron to perform computations by learning elements and processing them to detect the business intelligence and capabilities of the input data. A perceptron network is a group of simple logical statements that come together to create an array of complex logical statements, known as the neural network.

The human brain is a complex network of billions of interconnected cells known as neurons. These cells process and transmit signals. Biological neurons respond to both chemical and electrical signals to create the Biological Neural Network (BNN). The input and output signals can either increase or decrease the potential of the neuron to fire.

### 11.2.1 Artificial Neuron

An artificial neuron is based on a model of biological neurons, but it is a mathematical function. The neuron takes inputs in the form of binary values, i.e., 1 or 0, meaning that they can either be ON or OFF. The output of an artificial neuron is usually calculated by applying a threshold function to the sum of its input values.

The threshold function can be either linear or nonlinear. A linear threshold function produces an output of 1 if the sum of the input values is greater than or equal to a certain threshold, and an output of 0 if the sum of the input values is less than that threshold. A nonlinear threshold function, on the other hand, can produce any output value between 0 and 1, depending on the inputs.

An Artificial Neural Network (ANN) is built on artificial neurons and based on a Feed-Forward strategy. It is known as the simplest type of neural network as it continues learning irrespective of whether the data is linear or nonlinear. The information flow through the nodes is continuous and stops only after reaching the output node.

### 11.2.2 Biological Neural Network Vs Artificial Neural Network

The structure of artificial neurons is derived from that of biological neurons and the network is also formed on a similar principle but there are some differences between a biological neural network and an artificial neural network.

| Biological Neural Network                   | Artificial Neural Network           |
|---|-------------------------------------|
| The speed of processing information is slow | Faster compared to BNN              |
| Storage allocation to new process can be    | Storage allocation to new processes |

|   |   |
|---|---|
| done easily by adjusting the interconnection strengths                            | is not possible as each location is dedicated to a specified process  |
| Massive parallel applications can be conducted simultaneously                     | Sequential operations only  |
| Information can be retrieved from the sub-nodes even if it gets corrupted         | Once corrupted the information cannot be retrieved                    |
| The information being transmitted and processed into the network is not monitored | A control unit monitors all information and activities on the network |

### 11.2.3 Perceptron Vs Neuron

The perceptron is a mathematical model of the biological neuron. It produces binary outputs from input values while taking into consideration weights and threshold values. Though created to imitate the working of biological neurons, the perceptron model has since been replaced by more advanced models like backpropagation networks for training artificial neural networks. Perceptrons use a brittle activation function to give a positive or negative output based on a specific value.

A neuron, also known as a node in a backpropagation artificial neural network produces graded values between 0 and 1. It is a generalization of the idea of the perceptron as the neuron also adds weighted inputs. However, it does not produce a binary output but a graded value based on the proximity of the input to the desired value of 1. The results are biased towards the extreme values of 0 or 1 as the node uses a sigmoidal output function.

### 11.2.4 Components of a Perceptron

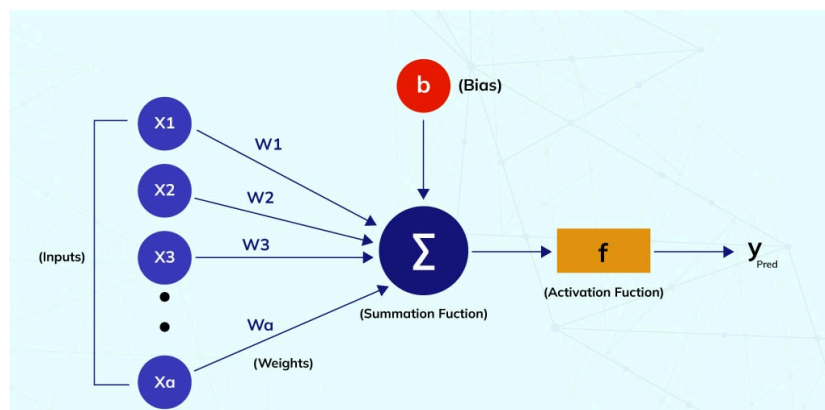


Figure 11.3: Perceptron

As shown in figure 11.3 each perceptron comprises different parts

**Input Values:** A set of values or a dataset for predicting the output value. They are also described as a dataset's features and dataset.

**Weights:** The real value of each feature is known as weight. It tells the importance of that feature in predicting the final value.

**Bias:** The activation function is shifted towards the left or right using bias.

**Summation Function:** The summation function binds the weights and inputs together. It is a function to find their sum.

**Activation Function:** It introduces non-linearity in the perceptron model.

### 11.2.5 Need of Weight and Bias

Weight and bias are two important aspects of the perceptron model. These are learnable parameters and as the network gets trained it adjusts both parameters to achieve the desired values and the correct output.

Weights are used to measure the importance of each feature in predicting output value. Features with values close to zero are said to have lesser weight or significance. These have less importance in the prediction process compared to the features with values further from zero known as weights with a larger value. Besides high-weighted features having greater predictive power than low-weighting ones, the weight can also be positive or negative. If the weight of a feature is positive then it has a direct relation with the target value, and if it is negative then it has an inverse relationship with the target value.

In contrast to weight in a neural network that increases the speed of triggering an activation function, bias delays the trigger of the activation function as shown in figure 11.4. Bias is a constant used to adjust the output and help the model to provide the best fit output for the given data.

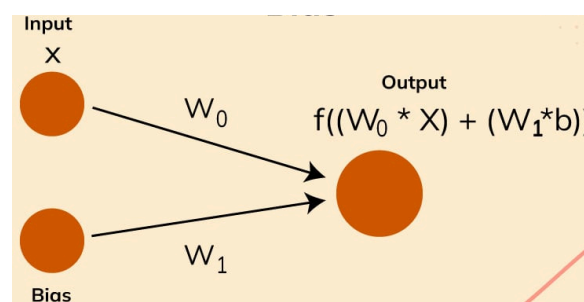


Figure 11.4: Bias

### 11.2.6 Perceptron Learning Rule

A new type of neural network called perceptrons works like neurons. These networks are trained using perceptron learning rules. According to this rule, perceptrons can automatically learn to produce the desired results by adjusting their weight values, as shown in Figure 11.5.

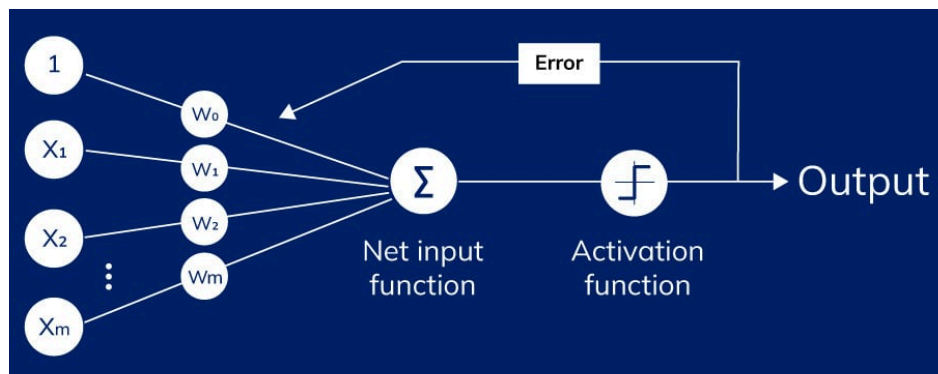


Figure 11.5: Perceptron Rule

### 11.2.7 Perceptron in Machine Learning

A perceptron in machine learning is used for supervised learning, specifically in binary classification tasks. It is one of the simplest models of artificial neural networks. The perceptron learning algorithm operates as a supervised machine learning system, utilizing binary classifiers for decision-making.

In machine learning, a binary classifier determines whether input data, represented as vectors of numbers, belongs to a specific category. These classifiers are linear, considering both feature values and weights. This helps the algorithm determine the classification outcome or the probability distribution around the prediction point.

### 11.2.8 The Perceptron in Neural Network

Neural networks are computer models designed to understand data and process information. They are inspired by the structure of the human brain, where perceptrons act like neurons. The perceptron model is an early and useful type of supervised machine learning. It is a basic algorithm for binary classification, meaning it can categorize inputs into one of two groups. These models use visual inputs to organize data, allowing machine learning algorithms to classify, identify, and analyze patterns. Perceptrons help separate different classes and patterns based on numerical or visual data.

Originally, the perceptron model was used for image recognition and is considered one of the first artificial neural networks, marking a significant advancement in AI. However, it had some limitations. As a single-layer model, it could only work with linearly separable classes. This limitation was addressed with the development of multi-layer perceptron algorithms.

### 11.2.9 Single Layer Perceptron Model

A single-layer perceptron model is the simplest type of artificial neural network. It includes a feed-forward network that can analyze only linearly separable objects while being dependent on a threshold transfer function. The model returns only binary outcomes (target) i.e. 1, and 0.

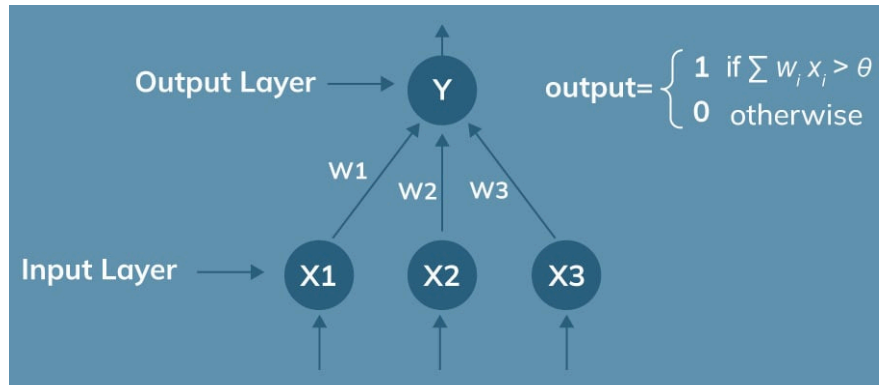


Figure 11.5: Single Layer Perceptron

The algorithm in a single-layered perceptron model does not have any previous information initially. The weights are allocated inconsistently, so the algorithm simply adds up all the weighted inputs. If the value of the sum is more than the threshold or a predetermined value then the output is delivered as 1 and the single-layer perceptron is considered to be activated.

$$w_1x_1 + w_2x_2 + \dots + w_nx_n > \theta \rightarrow 1 \text{ (output)}$$

$$w_1x_1 + w_2x_2 + \dots + w_nx_n \leq \theta \rightarrow 0 \text{ (output)}$$

When the values of input are similar to those desired for its predicted output, then we can say that the perceptron has performed satisfactorily. If there is any difference between what was expected and obtained, then the weights will need adjusting to limit how much these errors affect future predictions based on unchanged parameters.

$$\Delta w = \eta \times d \times X$$

Where

- d is predicted output - desired output
- $\eta$  is learning rate, usually less than 1
- X is the input data

However, since the single-layer perceptron is a linear classifier and it does not classify cases if they are not linearly separable. So, due to the inability of the perceptron to solve problems with linearly non-separable cases, the learning process will never reach the point with all cases properly classified.

### 11.2.10 Multilayer Perceptron Model

A multi-layer perceptron model uses the backpropagation algorithm. Though it has the same structure as that of a single-layer perceptron, it has one or more hidden layers.

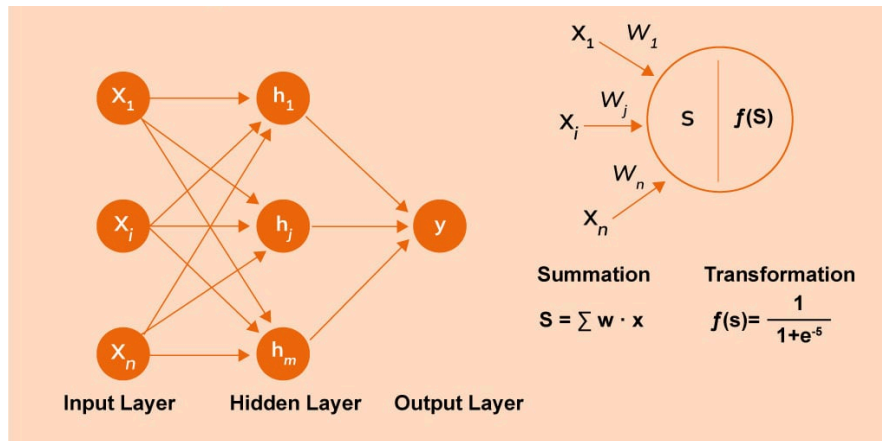


Figure 11.6: Multi Layer Perceptron

The backpropagation algorithm is executed in two phases:

**Forward phase-** Activation functions propagate from the input layer to the output layer. All weighted inputs are added to compute outputs using the sigmoid threshold.

**Backward phase-** The errors between the observed actual value and the demanded nominal value in the output layer are propagated backward. The weights and bias values are modified to achieve the requested value. The modification is done by apportioning the weights and bias to each unit according to its impact on the error.

- Error in any output neuron is  $d_o = y \times (1 - y) \times (t - y)$
- Error in any hidden neuron is  $d_i = y_i \times (1 - y_i) \times (w_i - d_o)$
- Change the weights  $\Delta w = \eta \times d \times X$

### 11.2.11 Perceptron Function

Perceptron function  $f(x)$  is generated by multiplying the input 'x' with the learned weight coefficient 'w'. The same can be expressed through the following mathematical equation:

$$f(x) = 1; \text{ if } w \cdot x + b > 0$$

$$\text{Otherwise, } f(x) = 0$$

where  $w$  is the real-valued weight,  $b$  is the bias and  $x$  is a vector of input  $x$  values

### 11.2.12 Limitations of the Perceptron Model

A perceptron model has the following limitations



- The input vectors must be presented to the network one at a time or in batches so that the corrections can be made to the network based on the results of each presentation.
- The perceptron generates only a binary number (0 or 1) as an output due to the hard limit transfer function.
- It can classify linearly separable sets of inputs easily whereas non-linear input vectors cannot be classified properly.

## 11.3 Backpropagation

Backpropagation is a learning algorithm used in neural networks. In simple terms, a neural network consists of connected input and output units, with each connection having a specific weight. During the learning phase, the network adjusts these weights to accurately predict the correct class label for the input data.

### 11.3.1 A Multilayer Feed-Forward Neural Network

The backpropagation algorithm performs learning on a multilayer feed-forward neural network. It iteratively learns a set of weights for prediction of the class label of tuples. A multilayer feed-forward neural network consists of an input layer, one or more hidden layers, and an output layer. An example of a multilayer feed-forward network is shown in Figure 11.7.

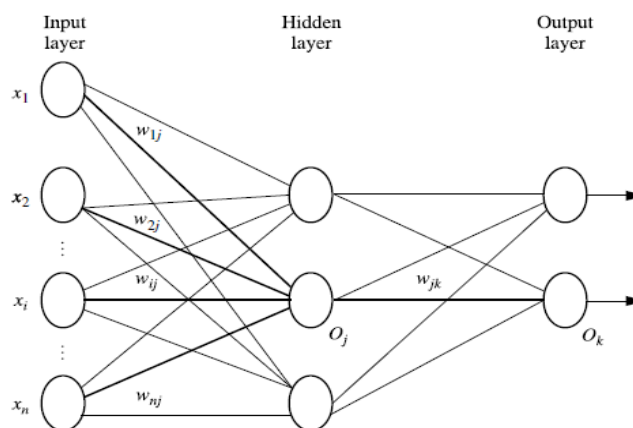



Figure 11.7: Multilayer feed-forward neural network

Each layer is made up of units. The inputs to the network correspond to the attributes measured for each training tuple. The inputs are fed simultaneously into the units making up the input layer. These inputs pass through the input layer and are then weighted and fed simultaneously to a second layer of “neuronlike” units, known as a hidden layer. The outputs of the hidden layer units can be input to another hidden layer, and so on. The number of hidden layers is arbitrary, although in practice, usually only one is used. The weighted outputs



of the last hidden layer are input to units making up the output layer, which emits the network's prediction for given tuples.

The units in the input layer are called input units. The units in the hidden layers and output layer are sometimes referred to as neurodes, due to their symbolic biological basis, or as output units. The multilayer neural network shown in Figure 11.7 has two layers of output units. Therefore, we say that it is a two-layer neural network. (The input layer is not counted because it serves only to pass the input values to the next layer.) Similarly, a network containing two hidden layers is called a three-layer neural network, and so on. It is a feed-forward network since none of the weights cycle back to an input unit or to a previous layer's output unit. It is fully connected in that each unit provides input to each unit in the next forward layer.

Each output unit takes, as input, a weighted sum of the outputs from units in the previous layer. It applies a nonlinear (activation) function to the weighted input. Multilayer feed-forward neural networks are able to model the class prediction as a nonlinear combination of the inputs. From a statistical point of view, they perform nonlinear regression. Multilayer feed-forward networks, given enough hidden units and enough training samples, can closely approximate any function.

Backpropagation is learned by iteratively processing a data set of training tuples, comparing the network's prediction for each tuple with the actual known target value. The target value may be the known class label of the training tuple (for classification problems) or a continuous value (for numeric prediction). For each training tuple, the weights are modified so as to minimize the mean-squared error between the network's prediction and the actual target value. These modifications are made in the "backwards" direction (i.e., from the output layer) through each hidden layer down to the first hidden layer (hence the name backpropagation). Although it is not guaranteed, in general the weights will eventually converge, and the learning process stops.

**Initialize the weights:** The weights in the network are initialized to small random numbers (e.g., ranging from -1.0 to 1.0, or -0.5 to 0.5). Each unit has a bias associated with it, as explained later. The biases are similarly initialized to small random numbers.

Each training tuple,  $X$ , is processed by the following steps.

**Propagate the inputs forward:** First, the training tuple is fed to the network's input layer. The inputs pass through the input units, unchanged. That is, for an input unit,  $j$ , its output,  $O_j$ , is equal to its input value,  $I_j$ . Next, the net input and output of each unit in the hidden and output layers are computed. The net input to a unit in the hidden or output layers is computed as a linear combination of its inputs. Each such unit has a number of inputs to it that are, in fact, the outputs of the units connected to it in the previous layer. Each connection has a weight. To compute the net input to the unit, each input connected to the unit is multiplied by its

corresponding weight, and this is summed. Given a unit,  $j$  in a hidden or output layer, the net input,  $I_j$ , to unit  $j$  is

$$I_j = \sum_i w_{ij} O_i + \theta_j,$$

where  $w_{ij}$  is the weight of the connection from unit  $i$  in the previous layer to unit  $j$ ;  $O_i$  is the output of unit  $i$  from the previous layer; and  $\theta_j$  is the bias of the unit. The bias acts as a threshold in that it serves to vary the activity of the unit.

Each unit in the hidden and output layers takes its net input and then applies an activation function to it. The function symbolizes the activation of the neuron represented by the unit. The logistic, or sigmoid, function is used. Given the net input  $I_j$  to unit  $j$ , then  $O_j$ , the output of unit  $j$ , is computed as

$$O_j = \frac{1}{1 + e^{-I_j}}.$$

This function is also referred to as a squashing function, because it maps a large input domain onto the smaller range of 0 to 1. The logistic function is nonlinear and differentiable, allowing the backpropagation algorithm to model classification problems that are linearly inseparable.

We compute the output values,  $O_j$ , for each hidden layer, up to and including the output layer, which gives the network's prediction. In practice, it is a good idea to cache (i.e., save) the intermediate output values at each unit as they are required again later when backpropagating the error. This trick can substantially reduce the amount of computation required.

**Backpropagate the error:** The error is propagated backward by updating the weights and biases to reflect the error of the network's prediction. For a unit  $j$  in the output layer, the error  $Err_j$  is computed by

$$Err_j = O_j(1 - O_j)(T_j - O_j)$$

where  $O_j$  is the actual output of unit  $j$ , and  $T_j$  is the known target value of the given training tuple. Note that  $O_j(1 - O_j)$  is the derivative of the logistic function. To compute the error of a hidden layer unit  $j$ , the weighted sum of the errors of the units connected to unit  $j$  in the next layer are considered. The error of a hidden layer unit  $j$  is

$$Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}$$

where  $w_{jk}$  is the weight of the connection from unit  $j$  to a unit  $k$  in the next higher layer, and  $Err_k$  is the error of unit  $k$ . The weights and biases are updated to reflect the propagated errors. Weights are updated by the following equations, where  $\Delta w_{ij}$  is the change in weight  $w_{ij}$ :

$$\Delta w_{ij} = (l)Err_j O_i.$$

$$w_{ij} = w_{ij} + \Delta w_{ij}.$$

The variable  $l$  is the learning rate, a constant typically having a value between 0.0 and 1.0. Backpropagation learns using a gradient descent method to search for a set of weights that fits the training data so as to minimize the mean-squared distance between the network's class prediction and the known target value of the tuples. The learning rate helps avoid getting stuck at a local minimum in decision space (i.e., where the weights appear to converge, but are not the optimum solution) and encourages finding the global minimum. If the learning rate is too small, then learning will occur at a very slow pace. If the learning rate is too large, then oscillation between inadequate solutions may occur. A rule of thumb is to set the learning rate to  $1/t$ , where  $t$  is the number of iterations through the training set so far.

Biases are updated by the following equations, where  $\Delta \theta_j$  is the change in bias  $\theta_j$ :

$$\Delta \theta_j = (l)Err_j.$$

$$\theta_j = \theta_j + \Delta \theta_j.$$

Note that here we are updating the weights and biases after the presentation of each tuple. This is referred to as case updating. Alternatively, the weight and bias increments could be accumulated in variables, so that the weights and biases are updated after all the tuples in the training set have been presented. This latter strategy is called epoch updating, where one iteration through the training set is an epoch. In theory, the mathematical derivation of backpropagation employs epoch updating, yet in practice, case updating is more common because it tends to yield more accurate results.

**Terminating condition:** Training stops when

- All  $\Delta w_{ij}$  in the previous epoch are so small as to be below some specified threshold, or
- The percentage of tuples misclassified in the previous epoch is below some threshold, or
- A prespecified number of epochs has expired.

Sample calculations for learning by the backpropagation algorithm. Figure 11.8 shows a multilayer feed-forward neural network. Let the learning rate be 0.9. The initial weight and bias values of the network are given in Table 11.1, along with the first training tuple,  $X = (1, 0, 1)$ , with a class label of 1.

This example shows the calculations for backpropagation, given the first training tuple,  $X$ . The tuple is fed into the network, and the net input and output of each unit are computed. These values are shown in Table 11.2. The error of each unit is computed and propagated backward. The error values are shown in Table 11.3. The weight and bias updates are shown in Table 11.4.

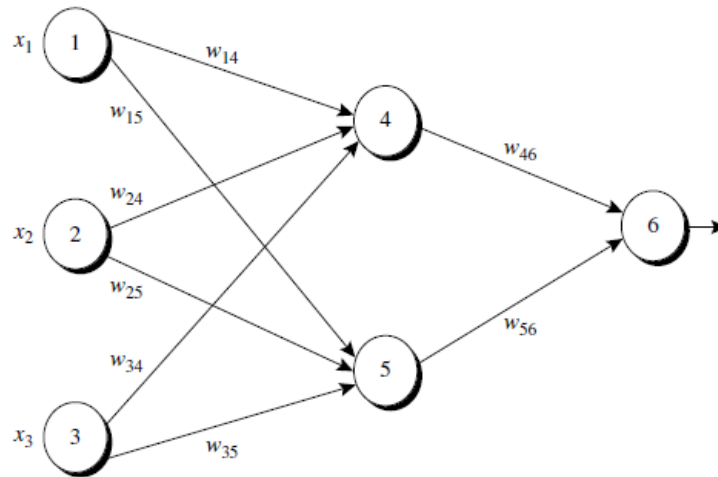


Figure 11.8: Example of a multilayer feed-forward neural network

| $x_1$ | $x_2$ | $x_3$ | $w_{14}$ | $w_{15}$ | $w_{24}$ | $w_{25}$ | $w_{34}$ | $w_{35}$ | $w_{46}$ | $w_{56}$ | $\theta_4$ | $\theta_5$ | $\theta_6$ |
|-------|-------|-------|----------|----------|----------|----------|----------|----------|----------|----------|------------|------------|------------|
| 1     | 0     | 1     | 0.2      | -0.3     | 0.4      | 0.1      | -0.5     | 0.2      | -0.3     | -0.2     | -0.4       | 0.2        | 0.1        |

Table 11.1: Initial Input,Weight, and Bias Values

| Unit, $j$ | Net Input, $I_j$                              | Output, $O_j$               |
|-----------|---|-----------------------------|
| 4         | $0.2 + 0 - 0.5 - 0.4 = -0.7$                  | $1/(1 + e^{0.7}) = 0.332$   |
| 5         | $-0.3 + 0 + 0.2 + 0.2 = 0.1$                  | $1/(1 + e^{-0.1}) = 0.525$  |
| 6         | $(-0.3)(0.332) - (0.2)(0.525) + 0.1 = -0.105$ | $1/(1 + e^{0.105}) = 0.474$ |

Table 11.2: Net Input and Output Calculations

| Unit, $j$ | Err <sub><math>j</math></sub>                |
|-----------|--|
| 6         | $(0.474)(1 - 0.474)(1 - 0.474) = 0.1311$     |
| 5         | $(0.525)(1 - 0.525)(0.1311)(-0.2) = -0.0065$ |
| 4         | $(0.332)(1 - 0.332)(0.1311)(-0.3) = -0.0087$ |

Table 11.3: Calculation of the Error at Each Node

| <i>Weight<br/>or Bias</i> | <i>New Value</i>                       |
|---------------------------|--|
| $w_{46}$                  | $-0.3 + (0.9)(0.1311)(0.332) = -0.261$ |
| $w_{56}$                  | $-0.2 + (0.9)(0.1311)(0.525) = -0.138$ |
| $w_{14}$                  | $0.2 + (0.9)(-0.0087)(1) = 0.192$      |
| $w_{15}$                  | $-0.3 + (0.9)(-0.0065)(1) = -0.306$    |
| $w_{24}$                  | $0.4 + (0.9)(-0.0087)(0) = 0.4$        |
| $w_{25}$                  | $0.1 + (0.9)(-0.0065)(0) = 0.1$        |
| $w_{34}$                  | $-0.5 + (0.9)(-0.0087)(1) = -0.508$    |
| $w_{35}$                  | $0.2 + (0.9)(-0.0065)(1) = 0.194$      |
| $\theta_6$                | $0.1 + (0.9)(0.1311) = 0.218$          |
| $\theta_5$                | $0.2 + (0.9)(-0.0065) = 0.194$         |
| $\theta_4$                | $-0.4 + (0.9)(-0.0087) = -0.408$       |

Table 11.4: Calculations for Weight and Bias Updating

To classify an unknown tuple,  $X$ , the tuple is input to the trained network, and the net input and output of each unit are computed. (There is no need for computation and/or backpropagation of the error.) If there is one output node per class, then the output node with the highest value determines the predicted class label for  $X$ . If there is only one output node, then output values greater than or equal to 0.5 may be considered as belonging to the positive class, while values less than 0.5 may be considered negative.

## Chapter-12

# Reinforcement Learning

Instructor Name: B N V Narasimha Raju

The best way to train your dog is by using a reward system. You give the dog a treat when it behaves well, and you punish it when it does something wrong. This same policy can be applied to machine learning models also. This type of machine learning method, where we use a reward system to train our model, is called Reinforcement Learning.

## 12.1 Need for Reinforcement Learning

A major drawback of machine learning is that a tremendous amount of data is needed to train models. The more complex a model, the more data it may require. But this data may not be available to us. It may not exist or we simply may not have access to it. Further, the data collected might not be reliable. It may have false or missing values or it might be outdated.

Also, learning from a small subset of actions will not help expand the vast realm of solutions that may work for a particular problem. This is going to slow the growth that technology is capable of. Machines need to learn to perform actions by themselves and not just learn from humans.

Reinforcement learning addresses all of these challenges by introducing a model to a controlled environment that simulates the problem to be solved. Instead of relying on actual data, the model learns by interacting with this simulated environment.

## 12.2 Reinforcement Learning

Reinforcement learning is a sub-branch of Machine Learning that trains a model to return an optimum solution for a problem by taking a sequence of decisions by itself.

Reinforcement learning is about taking suitable action to maximize reward in a particular situation. It is employed by various software and machines to find the best possible behavior or path it should take in a specific situation. Reinforcement learning differs from supervised learning in a way that in supervised learning the training data has the answer key with it so the model is trained with the correct answer itself whereas in reinforcement learning, there is no answer but the reinforcement agent decides what to do to perform the given task. In the absence of a training dataset, it is bound to learn from its experience.

Reinforcement learning uses algorithms that learn from outcomes and decide which action to take next. After each action, the algorithm receives feedback that helps it determine whether the choice it made was correct, neutral or incorrect. It is a good technique to use for automated systems that have to make a lot of small decisions without human guidance.

Reinforcement learning is an autonomous, self-teaching system that essentially learns by trial and error. It performs actions with the aim of maximizing rewards, or in other words, it is learning by doing in order to achieve the best outcomes.

We model an environment after the problem statement. The model interacts with this environment and comes up with solutions all on its own, without human interference. To push it in the right direction, we simply give it a positive reward if it performs an action that brings it closer to its goal or a negative reward if it goes away from its goal.

To understand reinforcement learning better, consider a dog that we have to house train. Here, the dog is the agent and the house, the environment as shown in figure 12.1



Figure 12.1: Agent and Environment

We can get the dog to perform various actions by offering incentives such as dog biscuits as a reward as shown in figure 12.2

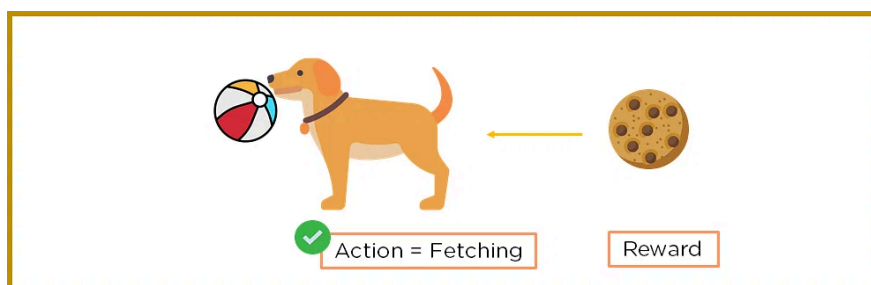


Figure 12.2: Performing an Action and getting Reward

The dog will follow a policy to maximize its reward and hence will follow every command and might even learn a new action, like begging, all by itself as shown in figure 12.3.





Figure 12.3: Learning new actions

The dog will also want to run around and play and explore its environment. This quality of a model is called Exploration. The tendency of the dog to maximize rewards is called Exploitation. There is always a tradeoff between exploration and exploitation, as exploration actions may lead to lesser rewards as shown in figure 12.4.



Figure 12.4: Exploration vs Exploitation

### 12.2.1 Supervised vs Unsupervised vs Reinforcement Learning

| Supervised Learning   | Unsupervised Learning  | Reinforcement Learning   |
|---|--|--|
| Data provided is labeled data, with output values specified | Data provided is unlabeled data, the outputs are not specified, machine makes its own prediction | The machine learns from its environment using rewards and errors |
| Used to solve regression and classification problems        | Used to solve Association and clustering problems  | Used to solve reward based problems                              |
| Labeled data is used  | Unlabeled data is used   | No predefined data is used                                       |
| External Supervision  | No supervision   | No supervision   |
| Solves problems by mapping labeled input to known output    | Solves problem by understanding patterns and discovering output                                  | Follows Trail and Error problem solving approach                 |

### 12.2.2 Important Terms in Reinforcement Learning

- **Agent:** Agent is the model that is being trained via reinforcement learning

- **Environment:** The training situation that the model must optimize to is called its environment
- **Action:** All possible steps that can be taken by the model
- **State:** The current position/ condition returned by the model
- **Reward:** To help the model move in the right direction, it is rewarded/points are given to it to appraise some action
- **Policy:** Policy determines how an agent will behave at any time. It acts as a mapping between Action and present State.

The important terms in Reinforcement Learning is shown in figure 12.5

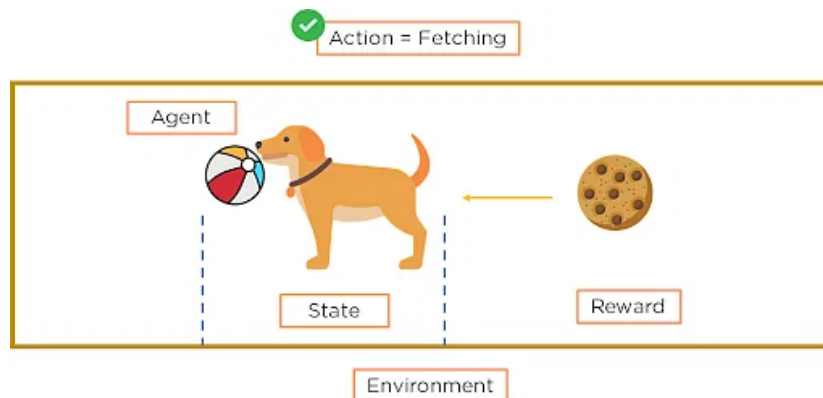


Figure 12.5: Important terms in Reinforcement Learning

There are two types of Reinforcement:

**Positive:** Positive Reinforcement is defined as when an event, occurs due to a particular behavior, increases the strength and the frequency of the behavior. In other words, it has a positive effect on behavior.

**Negative:** Negative Reinforcement is defined as strengthening of behavior because a negative condition is stopped or avoided.

## 12.3 Markov's Decision Process

Markov's Decision Process is a Reinforcement Learning policy used to map a current state to an action where the agent continuously interacts with the environment to produce new solutions and receive rewards as shown in figure 12.6

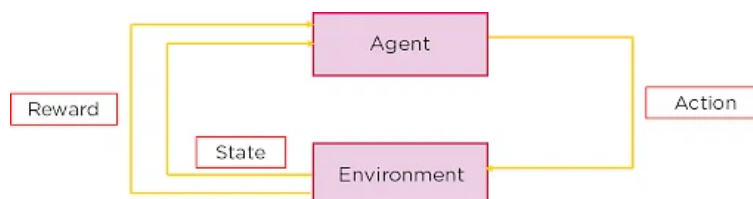



Figure 12.6: Markov's Decision Process



First, let's understand Markov's Process. Markov's Process states that the future is independent of the past, given the present. This means that, given the present state, the next state can be predicted easily, without the need for the previous state.

This theory is used by Markov's Decision Process to get the next action in our machine learning model. Markov's Decision Process (MDP) uses:

- A set of States ( $S$ )
- A set of Models
- A set of all possible actions ( $A$ )
- A reward function that depends on the state and action  $R(S, A)$
- A policy which is the solution of MDP

**State:** A State is a set of tokens that represent every state that the agent can be in.

**Model:** A Model (sometimes called Transition Model) gives an action's effect in a state. In particular,  $T(S, a, S')$  defines a transition  $T$  where being in state  $S$  and taking an action 'a' takes us to state  $S'$  ( $S$  and  $S'$  may be the same). For stochastic actions (noisy, non-deterministic) we also define a probability  $P(S'|S, a)$  which represents the probability of reaching a state  $S'$  if action 'a' is taken in state  $S$ . Markov property states that the effects of an action taken in a state depend only on that state and not on the prior history.

**Action:** An Action  $A$  is a set of all possible actions.  $A(S)$  defines the set of actions that can be taken being in state  $S$ .

**Reward:** A Reward is a real-valued reward function.  $R(S)$  indicates the reward for simply being in the state  $S$ .  $R(S, a)$  indicates the reward for being in a state  $S$  and taking an action 'a'.  $R(S, a, S')$  indicates the reward for being in a state  $S$ , taking an action 'a' and ending up in a state  $S'$ .

**Policy:** A Policy is a solution to the Markov Decision Process. A policy is a mapping from  $S$  to  $a$ . It indicates the action 'a' to be taken while in state  $S$ .

Let us take the example of a grid world:

The policy of Markov's Decision Process aims to maximize the reward at each state. The Agent interacts with the Environment and takes Action while it is at one State to reach the next future State. We base the action on the maximum Reward returned.

In the figure 12.7 shown, we need to find the shortest path between node A and D. Each path has a reward associated with it, and the path with maximum reward is what we want to choose. The nodes; A, B, C, D; denote the nodes. To travel from node to node (A to B) is an action. The reward is the cost at each path, and policy is each path taken.

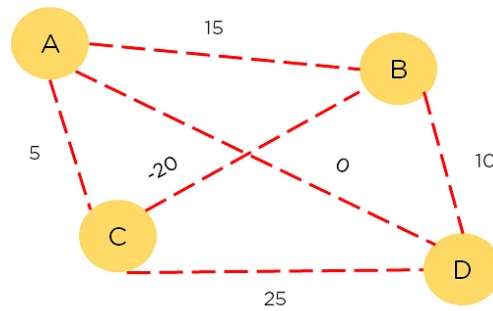


Figure 12.7: Nodes to traverse

The process will maximize the output based on the reward at each step and will traverse the path with the highest reward. This process does not explore but maximizes reward as shown in figure 12.8.

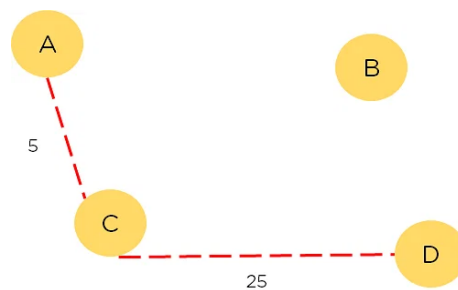


Figure 12.8: Path taken by MDP

## 12.4 Gradient descent reduces the cost function

After training your model, you need to see how well your model is performing. While accuracy functions tell you how well the model is performing, they do not provide you with an insight on how to better improve them. Hence, you need a correctional function that can help you compute when the model is the most accurate, as you need to hit that small spot between an undertrained model and an overtrained model.

A Cost Function is used to measure just how wrong the model is in finding a relation between the input and output. It tells you how badly your model is behaving/predicting

Consider a robot trained to stack boxes in a factory. The robot might have to consider certain changeable parameters, called Variables, which influence how it performs. Let's say the robot comes across an obstacle, like a rock. The robot might bump into the rock and realize that it is not the correct action.

It will learn from this, and next time it will learn to avoid rocks. Hence, your machine uses variables to better fit the data. The outcome of all these obstacles will further optimize the robot and help it perform better. It will generalize and learn to avoid obstacles in general, say

like a fire that might have broken out. The outcome acts as a cost function, which helps you optimize the variable, to get the best variables and fit for the model.

### 12.4.1 Gradient Descent

Gradient Descent is an algorithm that is used to optimize the cost function or the error of the model. It is used to find the minimum value of error possible in your model. Gradient Descent can be thought of as the direction you have to take to reach the least possible error. The error in your model can be different at different points, and you have to find the quickest way to minimize it, to prevent resource wastage.

Gradient Descent can be visualized as a ball rolling down a hill. Here, the ball will roll to the lowest point on the hill. It can take this point as the point where the error is least as for any model, the error will be minimum at one point and will increase again after that.

In gradient descent, you find the error in your model for different values of input variables. This is repeated, and soon you see that the error values keep getting smaller and smaller. Soon you'll arrive at the values for variables when the error is the least, and the cost function is optimized as shown in figure 12.9.

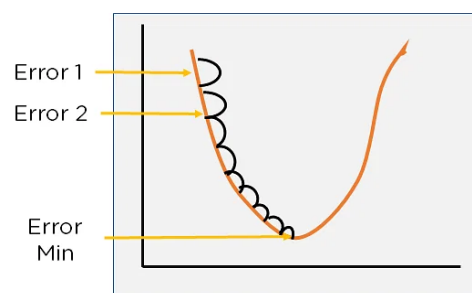


Figure 12.9: Gradient Descent

### 12.4.2 Cost Function For Linear Regression

A Linear Regression model uses a straight line to fit the model. This is done using the equation for a straight line as shown :

$$\text{Output} = a * \text{Input} + b$$

In the equation, you can see that two entities can have changeable values (variable) a, which is the point at which the line intercepts the x-axis, and b, which is how steep the line will be, or slope.

At first, if the variables are not properly optimized, you get a line that might not properly fit the model. As you optimize the values of the model, for some variables, you will get the perfect fit. The perfect fit will be a straight line running through most of the data points while ignoring the noise and outliers. A properly fit Linear Regression model looks as shown in figure 12.10.

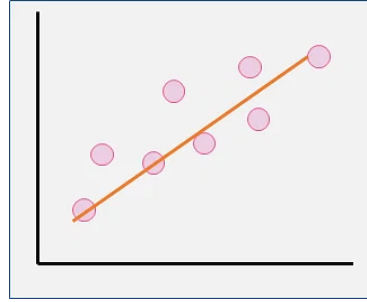


Figure 12.10: Linear regression graph

For the Linear regression model, the cost function will be the minimum of the Root Mean Squared Error of the model, obtained by subtracting the predicted values from actual values. The cost function will be the minimum of these error values.

$$\text{Cost Function (J)} = \frac{1}{n} \sum_{i=0}^n (h_{\theta}(x^i) - y^i)^2$$

By the definition of gradient descent, you have to find the direction in which the error decreases constantly. This can be done by finding the difference between errors. The small difference between errors can be obtained by differentiating the cost function and subtracting it from the previous gradient descent to move down the slope.

$$\text{Gradient Descent } \theta_j = \theta_j - \alpha \frac{\partial J}{\partial \theta}$$

After substituting the value of the cost function (J) in the above equation, a simplified Linear regression gradient descent function is obtained.

$$\text{Gradient Descent} = \theta_j - \frac{\alpha}{n} \sum_{i=0}^n ((h_{\theta}(x^i) - y^i)x^i)^2$$

In the above equations,  $\alpha$  is known as the learning rate. It decides how fast you move down the slope. If alpha is large, you take big steps, and if it is small; you take small steps. If alpha is too large, you can entirely miss the least error point and our results will not be accurate. If it is too small it will take too long to optimize the model and you will also waste computational power. Hence you need to choose an optimal value of alpha as shown in figure 12.11.

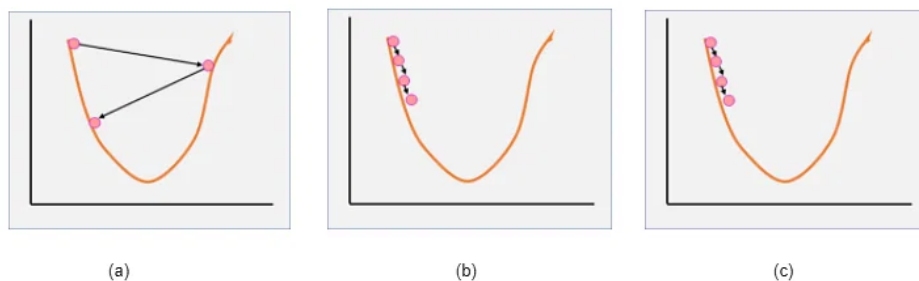


Figure 12.11: (a) Large learning rate, (b) Small learning rate, (c) Optimum learning rate

### 12.4.3 Cost Function for Neural Networks

A neural network is a machine learning algorithm that takes in multiple inputs, runs them through an algorithm, and essentially sums the output of the different algorithms to get the final output. The cost function of a neural network will be the sum of errors in each layer. This is done by finding the error at each layer first and then summing the individual error to get the total error. In the end, it can represent a neural network with cost function optimization as shown in figure 12.12.

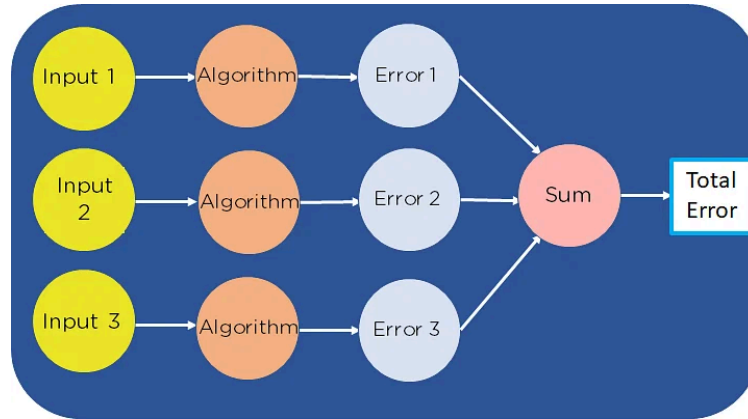


Figure 12.12: Neural network with the error function

For neural networks, each layer will have a cost function, and each cost function will have its own least minimum error value. Depending on where you start, you can arrive at a unique value for the minimum error. You need to find the minimum value out of all local minima. This value is called the global minima as shown in figure 12.13.

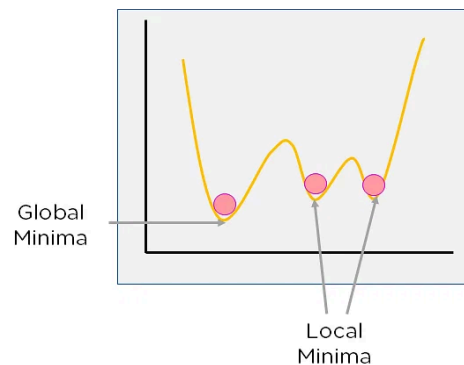


Figure 12.13: Cost function graph for Neural Networks

The cost function for neural networks is given as

$$\text{Cost Function (J)} = \frac{1}{n} \sum_{i=0}^n (y^i - (mx^i + b))^2$$

Gradient descent is just the differentiation of the cost function. It is given as

$$\text{Gradient Descent } \left( \frac{\partial J}{\partial \theta} \right) = \begin{bmatrix} \frac{1}{N} \sum_{i=0}^n (-2x_i(y_i - (mx_i + b))) \\ \frac{1}{N} \sum_{i=0}^n (-2(y_i - (mx_i + b))) \end{bmatrix}$$