Module 3

Multi-Layer Perceptron (MLP)

A Multi-Layer Perceptron (MLP) is one of the most widely used types of neural networks.

Multi-Layer Perceptron (MLP) is an **artificial neural network** widely used for solving **classification** and **regression tasks**. **In this article, we will explore the concept of MLP in-depth and demonstrate how to implement it in Python using the TensorFlow library.**

Multilayer Perceptron

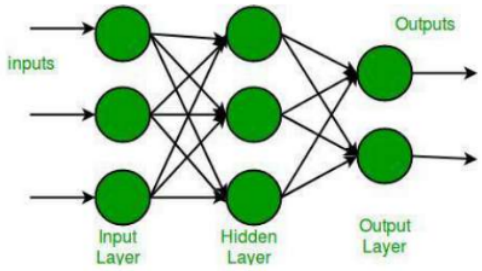
A **Multi-Layer Perceptron (MLP)** consists of fully connected dense layers that transform input data from one dimension to another. It is called "multi-layer" because it contains an input layer, one or more hidden layers, and an output layer. The purpose of an MLP is to model complex relationships between inputs and outputs, making it a powerful tool for various machine learning tasks.

The key components of Multi-Layer Perceptron includes

Input Layer: Each neuron (or node) in this layer corresponds to an input feature. For instance, if you have three input features, the input layer will have three neurons.

Hidden Layers: An MLP can have any number of hidden layers, with each layer containing any number of nodes. These layers process the information received from the input layer.

Output Layer: The output layer generates the final prediction or result. If there are multiple outputs, the output layer will have a corresponding number of neurons.



Working of Multi-Layer Perceptron

Let's delve in to the working of the multi-layer perceptron. The key mechanisms such as forward propagation, loss function, backpropagation, and optimization.

Step 1: Forward Propagation

In **forward propagation**, the data flows from the input layer to the output layer, passing through any hidden layers. Each neuron in the hidden layers processes the input as follows:

Weighted Sum: The neuron computes the weighted sum of the inputs:

$z=\sum iwixi+bz=\sum i \quad wi \quad xi \quad +b$

Where:

xi is the input feature.

wi is the corresponding weight.

b is the bias term.

Activation Function: The weighted sum z is passed through an activation function to introduce non-linearity. Common activation functions include:

Sigmoid: $\sigma(z)=1/1+e-z$

ReLU (Rectified Linear Unit): $f(z)=\max(0,z)$

Tanh (Hyperbolic Tangent): $\tanh(z)=2/(1+e-2z)-1$

Step 2: Loss Function

Once the network generates an output, the next step is to calculate the **loss** using a loss function. In supervised learning, this compares the predicted output to the actual label.

For a classification problem, the commonly used binary cross-entropy loss function is:

$$L = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$$

Where:

yi is the actual label.

y^i is the predicted label.

N is the number of samples.

For regression problems, the mean squared error (MSE) is often used:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i-\hat{y}_i)^2$$

Step 3: Backpropagation

The goal of training an MLP is to minimize the loss function by adjusting the network's weights and biases. This is achieved through backpropagation:

Gradient Calculation: The gradients of the loss function with respect to each weight and bias are calculated using the chain rule of calculus.

Error Propagation: The error is propagated back through the network, layer by layer.

Gradient Descent: The network updates the weights and biases by moving in the opposite direction of the gradient to reduce the loss: w=w–η· ∂L∂ww=w–η· ∂w∂L

Where:

w is the weight.

η is the learning rate.

∂L∂w ∂w∂L is the gradient of the loss function with respect to the weight.

Step 4: Optimization

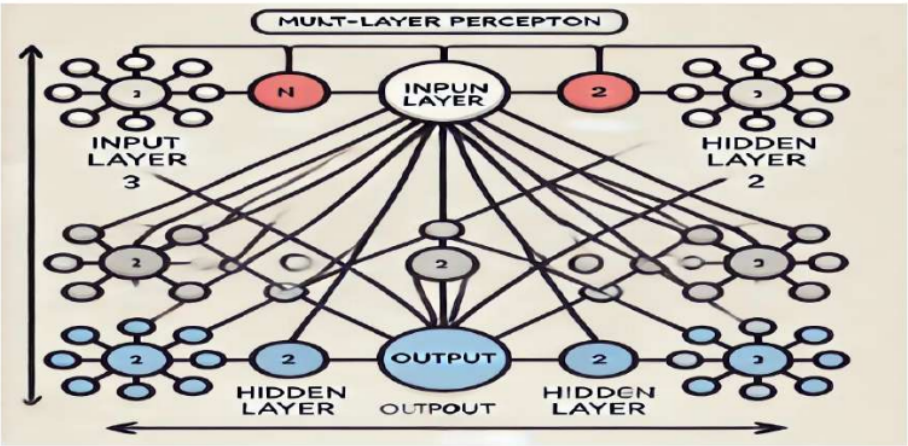
MLPs rely on optimization algorithms to iteratively refine the weights and biases during training. Popular optimization methods include:

Stochastic Gradient Descent (SGD): Updates the weights based on a single sample or a small batch of data: w=w–η· ∂L∂ww=w–η· ∂w∂L

Adam Optimizer: An extension of SGD that incorporates momentum and adaptive learning rates for more efficient training:

- $m_t = \beta_1 m_{t-1} + (1-\beta_1) \cdot g_t$
- $v_t = \beta_2 v_{t-1} + (1-\beta_2) \cdot g_t^2$

Here, gtgt represents the gradient at time tt, and β1,β2β1 ,β2 are decay rates.



Feed-forward Network Mappings

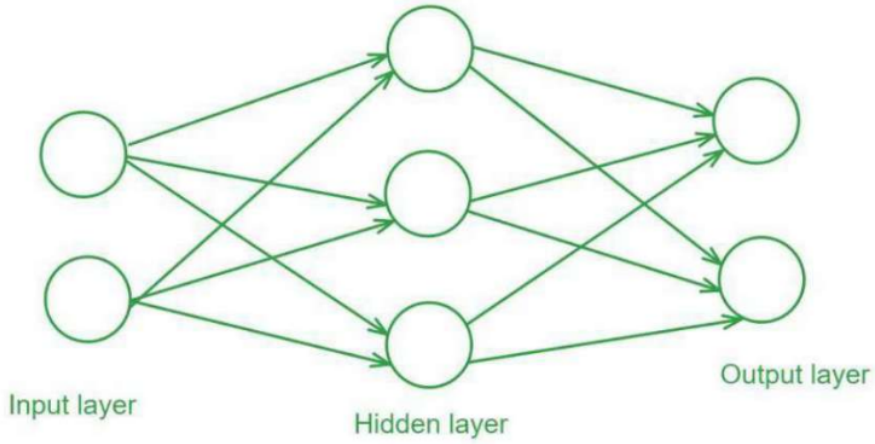
A Feedforward Neural Network (FNN) is a type of artificial neural network where connections between the nodes do not form cycles. This characteristic differentiates it from recurrent neural networks (RNNs). The network consists of an input layer, one or more hidden layers, and an output layer. Information flows in one direction—from input to output—hence the name "feedforward."

Structure of a Feedforward Neural Network

Input Layer: The input layer consists of neurons that receive the input data. Each neuron in the input layer represents a feature of the input data.

Hidden Layers: One or more hidden layers are placed between the input and output layers. These layers are responsible for learning the complex patterns in the data. Each neuron in a hidden layer applies a weighted sum of inputs followed by a non-linear activation function.

Output Layer: The output layer provides the final output of the network. The number of neurons in this layer corresponds to the number of classes in a classification problem or the number of outputs in a regression problem.



Mathematical Representation:

For a given input xx, the network output y=f(Wx+b)

where:

x = input vector, W = weight matrix, b = bias vector

ff = activation function (like Sigmoid, ReLU, etc.)

Activation Functions

Activation functions introduce non-linearity into the network, enabling it to learn and model complex data patterns. Common activation functions include:

Sigmoid: σ(x)=σ(x)=11+e−xσ(x)=1+e−x1

Tanh: tanh(x)=ex−e−xex+e−xtanh(x)=ex+e−xex−e−x

ReLU (Rectified Linear Unit): ReLU(x)=max⁡(0,x)ReLU(x)=max(0,x)

Leaky ReLU: Leaky ReLU(x)=max⁡(0.01x,x)Leaky ReLU(x)=max(0.01x,x)

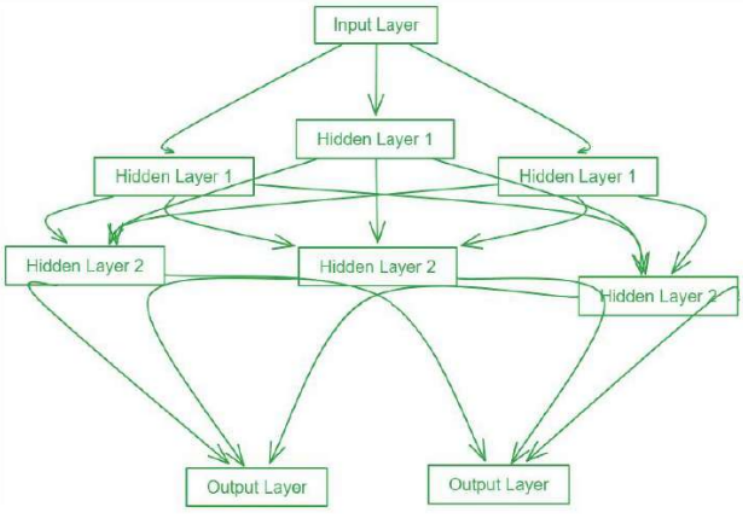
Training a Feed forward Neural Network

Training a Feed forward Neural Network involves adjusting the weights of the neurons to minimize the error between the predicted output and the actual output. This process is typically performed using backpropagation and gradient descent.

Forward Propagation: During forward propagation, the input data passes through the network, and the output is calculated.

Loss Calculation: The loss (or error) is calculated using a loss function such as Mean Squared Error (MSE) for regression tasks or Cross-Entropy Loss for classification tasks.

Backpropagation: In backpropagation, the error is propagated back through the network to update the weights. The gradient of the loss function with respect to each weight is calculated, and the weights are adjusted using gradient descent.



Evaluation of Feedforward neural network

Evaluating the performance of the trained model involves several metrics:

Accuracy: The proportion of correctly classified instances out of the total instances.

Precision: The ratio of true positive predictions to the total predicted positives.

Recall: The ratio of true positive predictions to the actual positives.

F1 Score: The harmonic mean of precision and recall, providing a balance between the two.

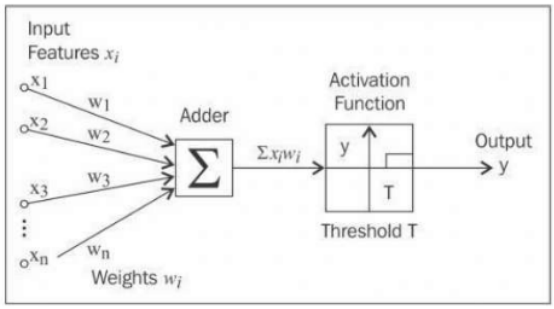
Confusion Matrix: A table used to describe the performance of a classification model, showing the true positives, true negatives, false positives, and false negatives.

Threshold Units

The Threshold Logic Units (TLUs)

Threshold Logic Units (TLUs) also known as Linear Threshold Units (LTU) were initially proposed by Frank Rosenblatt in the late 1950s as the basic units of Perceptrons.

These computational units are also based on a threshold activation function i.e., they apply a step function to the weighted sum of their inputs and then produce outputs that are now numbers (instead of binary on/off values used in McCulloch-Pitts model).



Threshold Logic Unit - Image Source: O'Reily

There are two common step functions used in TLUs: **Heaviside and Sign.**

$$\text{heaviside}\,(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}\,(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

TLUs are capable of defining linear decision boundaries in input space.

In two-dimensional space, for example, a single TLU can separate points into two classes (binary classification) using a straight line.

It simply computes a linear combination of the inputs: if the result exceeds a threshold, it outputs the positive class or else outputs the negative class (just like a Logistic Regression classifier). Training a TLU in this case means finding the right values for weights.

Used in Perceptrons for binary classification.

Limitation: Threshold units are linear, so they can't solve non-linear problems like XOR. This is why MLPs use non-linear activation functions.

Sigmoidal Units

A **sigmoid function** is any mathematical function whose graph has a characteristic S-shaped or **sigmoid curve**.

A sigmoid function is a bounded, differentiable, real function that is defined for all real input values and has a non-negative derivative at each point and exactly one inflection point.

A common example of a sigmoid function is the logistic function, which is defined by the formula:[1]

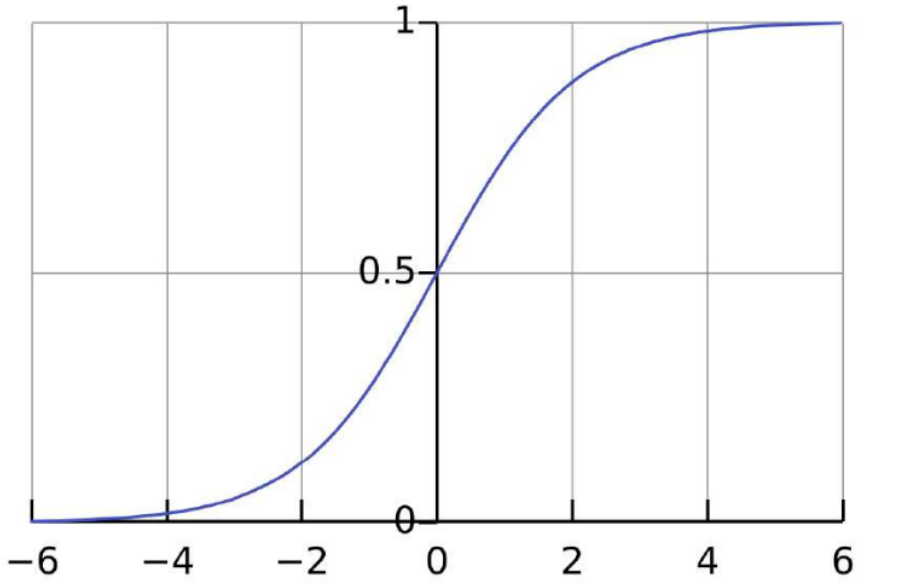
$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x} = 1 - \sigma(-x).$$

Properties:

1. Outputs values between (0, 1).
2. Used in binary classification problems.
3. Unlike threshold units, they are smooth and differentiable, making them useful for gradient-based learning.

In general, a sigmoid function is monotonic, and has a first derivative which is bell shaped. Conversely, the integral of any continuous, non-negative, bell-shaped function (with one local maximum and no local minimum, unless degenerate) will be sigmoidal. Thus the cumulative distribution functions for many common probability

distributions are sigmoidal. One such example is the error function, which is related to the cumulative distribution function of a normal distribution.



A sigmoid function is convex for values less than a particular point, and it is concave for values greater than that point: in many of the examples here, that point is 0.

Examples

Logistic function

$$f(x) = \frac{1}{1 + e^{-x}}$$

Hyperbolic tangent (shifted and scaled version of the logistic function, above)

$$f(x) = \tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Arctangent function

$$f(x) = \arctan x$$

Gudermannian function

$$f(x) = \operatorname{gd}(x) = \int_0^x \frac{dt}{\cosh t} = 2 \arctan\left(\tanh\left(\frac{x}{2}\right)\right)$$

Error function

$$f(x) = \operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

Applications

Many natural processes, such as those of complex system **learning curves,** exhibit a progression from small beginnings that accelerates and approaches a climax over time. When a specific mathematical model is lacking, a sigmoid function is often used.

The **van Genuchten–Gupta model** is based on an inverted S-curve and applied to the response of crop yield to soil salinity.

Examples of the application of the logistic S-curve to the response of crop yield (wheat) to both the soil salinity and depth to water table in the soil are shown in modeling crop response in agriculture.

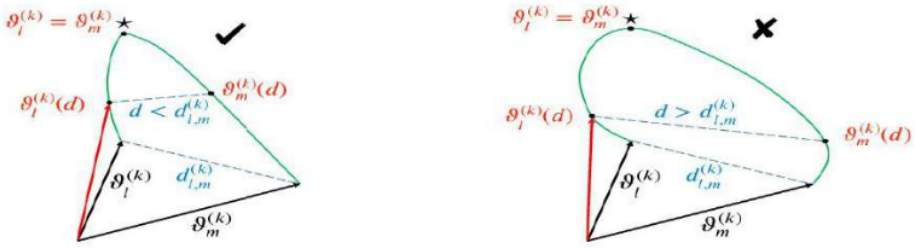
In Artificial neural networks, sometimes non-smooth functions are used instead for efficiency; these are known as hard sigmoids.

Weight-space Symmetries

In the context of deep learning, **weight space symmetry** means that non-identifiable models are invariant to random permutations in their weight layers. This symmetry holds since in deep learning there are generally not enough training samples to rule out all parameter settings but one, there usually exist a large amount of possible weight combinations for a given dataset that yield similar model performance.

Weight-space symmetry is a property of neural network landscapes that describes how permutation symmetries give rise to multiple equivalent global minima in the weight space. This property can have implications for training dynamics, and can also be used to uncover a model's underlying structure

B



Weight-space symmetry can also give rise to first-order saddle points on the path between the global minima.

A challenging problem in machine learning is to process weight-space features, which involves transforming or extracting information from the weights and gradients of a neural network.

The weight space is a concatenation of all the weight and biases.

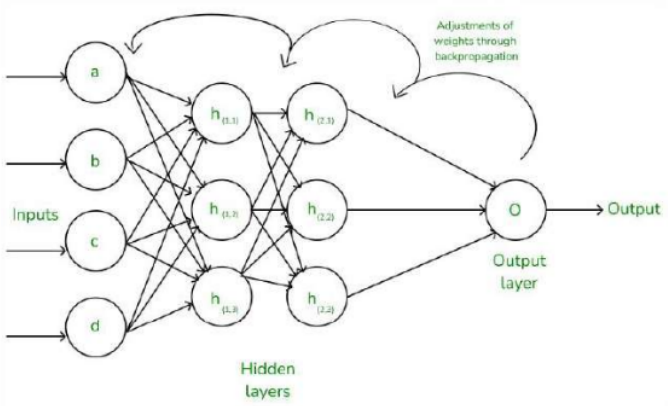
The symmetry group acts on each one of those independently, which is called a direct-sum of representations.

Error Back-propagation

Backpropagation in Neural Network

Backpropagation is a powerful algorithm in deep learning, primarily used to train artificial neural networks, particularly **feed forward networks.** It works iteratively, minimizing the cost function by adjusting weights and biases.

In each epoch, the model adapts these parameters, reducing loss by following the error gradient. Backpropagation often utilizes optimization algorithms like **gradient descent or stochastic gradient descent.** The algorithm computes the gradient using the chain rule from calculus, allowing it to effectively navigate complex layers in the neural network to minimize the cost function.



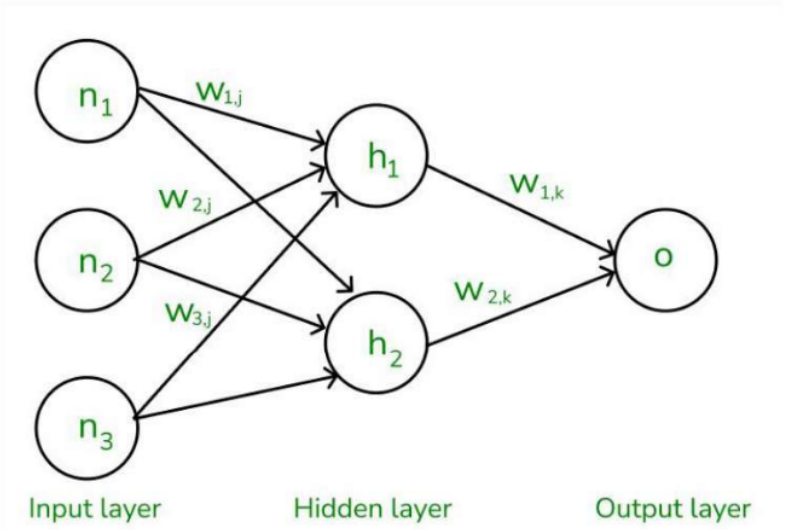
Working of Backpropagation Algorithm

The Backpropagation algorithm involves two main steps: the **Forward Pass** and the **Backward Pass**.

In the **forward pass**, the input data is fed into the input layer. These inputs, combined with their respective weights, are passed to hidden layers.

For example, in a network with two hidden layers (h1 and h2), the output from h1 serves as the input to h2. Before applying an activation function, a bias is added to the weighted inputs.

Each hidden layer applies an activation function like **ReLU (Rectified Linear Unit),** which returns the input if it's positive and zero otherwise. This adds non-linearity, allowing the model to learn complex relationships in the data. Finally, the outputs from the last hidden layer are passed to the output layer, where an activation function, such as **softmax,** converts the weighted outputs into probabilities for classification.



Backward Pass Working

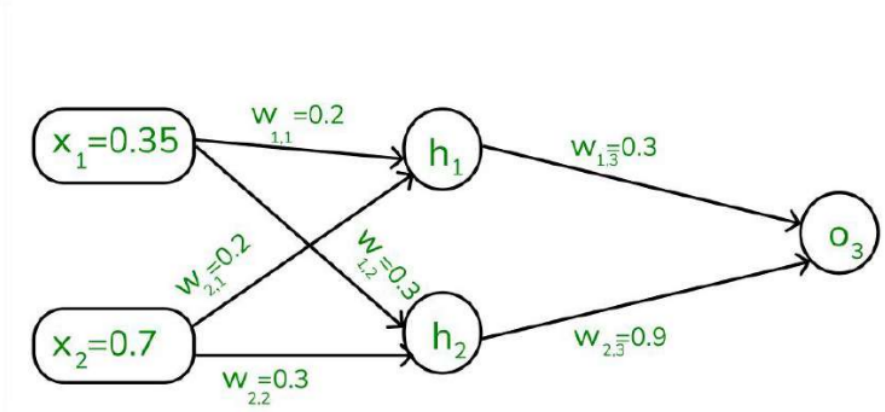
In the backward pass, the error (the difference between the predicted and actual output) is propagated back through the network to adjust the weights and biases. One common method for error calculation is the **Mean Squared Error (MSE),** given by:

MSE=(Predicted Output−Actual Output)2*MSE*=(Predicted Output−Actual Output)2

Once the error is calculated, the network adjusts weights using **gradients**, which are computed with the chain rule. These gradients indicate how much each weight and bias should be adjusted to minimize the error in the next iteration. The backward pass continues layer by layer, ensuring that the network learns and improves its performance. The activation function, through its derivative, plays a crucial role in computing these gradients during backpropagation.

Example of Backpropagation in Machine Learning

Let's walk through an example of backpropagation in machine learning. Assume the neurons use the sigmoid activation function for the forward and backward pass. The target output is 0.5, and the learning rate is 1.



Error Calculation

To calculate the error, we can use the below formula:

Errorj=ytarget−y5*Errorj* =*ytarget* −*y*5

Error=0.5−0.67=−0.17*Error*=0.5−0.67=−0.17

Using this error value, we will be backpropagating.

Backpropagation

1. Calculating Gradients

The change in each weight is calculated as:

Δwij=η×δj×Oj*Δwij* =*η*×*δj* ×*Oj*

Where:

δj*δj* is the error term for each unit,

η*η* is the learning rate.

2. Output Unit Error

For O3:

*δ*5 =*y*5 (1−*y*5)(*ytarget* −*y*5)

=0.67(1−0.67)(−0.17)=−0.0376=0.67(1−0.67)(−0.17)=−0.0376

Advantages of Backpropagation for Neural Network Training

The key benefits of using the backpropagation algorithm are:

Ease of Implementation: Backpropagation is beginner-friendly, requiring no prior neural network knowledge, and simplifies programming by adjusting weights via error derivatives.

Simplicity and Flexibility: Its straightforward design suits a range of tasks, from basic feed forward to complex convolution or recurrent networks.

Efficiency: Backpropagation accelerates learning by directly updating weights based on error, especially in deep networks.

Generalization: It helps models generalize well to new data, improving prediction accuracy on unseen examples.

Scalability: The algorithm scales efficiently with larger datasets and more complex networks, making it ideal for large-scale tasks.

Challenges with Backpropagation

While backpropagation is powerful, it does face some challenges:

Vanishing Gradient Problem: In deep networks, the gradients can become very small during backpropagation, making it difficult for the network to learn. This is common when using activation functions like sigmoid or tanh.

Exploding Gradients: The gradients can also become excessively large, causing the network to diverge during training.

Over fitting: If the network is too complex, it might memorize the training data instead of learning general patterns.

Radial Basis Function Networks (RBFNs)

Radial Basis Function (RBF) **Neural Networks** are a specialized type of Artificial Neural Network (ANN) used primarily for function approximation tasks. Known for their distinct three-layer architecture and universal approximation capabilities, RBF Networks offer faster learning speeds and efficient performance in classification and regression problems.

Radial Basis Functions

Radial Basis Functions (RBFs) are a special category of **feed-forward neural networks** comprising three layers:

Input Layer: Receives input data and passes it to the hidden layer.

Hidden Layer: The core computational layer where RBF neurons process the data.

Output Layer: Produces the network's predictions, suitable for classification or regression tasks.

RBF Networks Working

RBF Networks are conceptually similar to K-Nearest Neighbor (k-NN) models, though their implementation is distinct. The fundamental idea is that an item's predicted target value is influenced by nearby items with similar predictor variable values. Here's how RBF Networks operate:

Input Vector: The network receives an n-dimensional input vector that needs classification or regression.

RBF Neurons: Each neuron in the hidden layer represents a prototype vector from the training set. The network computes the Euclidean distance between the input vector and each neuron's center.

Activation Function: The Euclidean distance is transformed using a Radial Basis Function (typically a Gaussian function) to compute the neuron's activation value. This value decreases exponentially as the distance increases.

Output Nodes: Each output node calculates a score based on a weighted sum of the activation values from all RBF neurons. For classification, the category with the highest score is chosen.

Key Characteristics of RBFs

Radial Basis Functions: These are real-valued functions dependent solely on the distance from a central point. The Gaussian function is the most commonly used type.

Dimensionality: The network's dimensions correspond to the number of predictor variables.

Center and Radius: Each RBF neuron has a center and a radius (spread). The radius affects how broadly each neuron influences the input space.

Architecture of RBF Networks

The architecture of an RBF Network typically consists of three layers:

Input Layer

Function: After receiving the input features, the input layer sends them straight to the hidden layer.

Components: It is made up of the same number of neurons as the characteristics in the input data. One feature of the input vector corresponds to each neuron in the input layer.

Hidden Layer

Function: This layer uses radial basis functions (RBFs) to conduct the non-linear transformation of the input data.

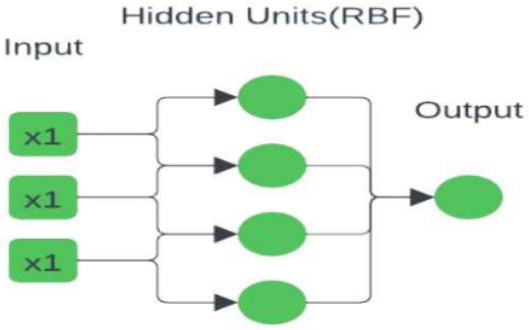
Components: Neurons in the buried layer apply the RBF to the incoming data. The Gaussian function is the RBF that is most frequently utilized.

RBF Neurons: Every neuron in the hidden layer has a spread parameter (σ) and a center, which are also referred to as prototype vectors. The spread parameter modulates the distance between the center of an RBF neuron and the input vector, which in turn determines the neuron's output.

Output Layer

Function: The output layer uses weighted sums to integrate the hidden layer neurons' outputs to create the network's final output.

Components: It is made up of neurons that combine the outputs of the hidden layer in a linear fashion. To reduce the error between the network's predictions and the actual target values, the weights of these combinations are changed during training.



Training Process of radial basis function neural network

An RBF neural network must be trained in three stages: choosing the center's, figuring out the spread parameters, and training the output weights.

Step 1: Selecting the Centers

Techniques for Centre Selection: Centre's can be picked at random from the training set of data or by applying techniques such as **k-means clustering.**

K-Means Clustering: The center's of these clusters are employed as the center's for the RBF neurons in this widely used center selection technique, which groups the input data into k groups.

Step 2: Determining the Spread Parameters

The spread parameter (σ) governs each RBF neuron's area of effect and establishes the width of the RBF.

Calculation: The spread parameter can be manually adjusted for each neuron or set as a constant for all neurons. Setting σ based on the separation between the center's is a popular method, frequently accomplished with the help of a heuristic like dividing the greatest distance between canters by the square root of twice the number of center's

Step 3: Training the Output Weights

Linear Regression: The objective of linear regression techniques, which are commonly used to estimate the output layer weights, is to minimize the error between the anticipated output and the actual target values.

Pseudo-Inverse Method: One popular technique for figuring out the weights is to utilize the pseudo-inverse of the hidden layer outputs matrix