UNIT - II

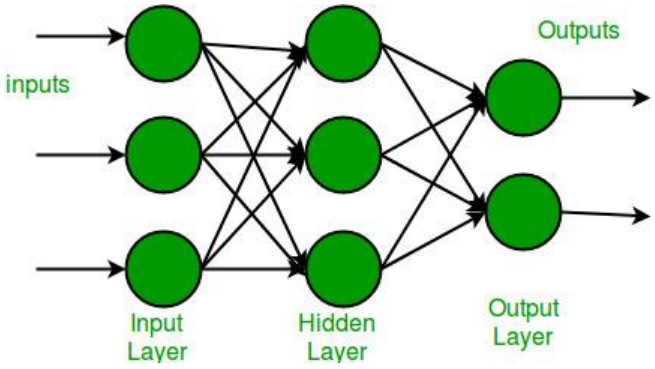
Multi-layer Perceptron– Going Forwards – Going Backwards: Back Propagation Error – Multi-layer Perceptron in Practice – Examples of using the MLP – Overview – Deriving Back-Propagation – Radial Basis Functions and Splines – Concepts – RBF Network – Curse of Dimensionality – Interpolations and Basis Functions – Support Vector Machines

MULTI-LAYER 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. MLP (Multi-Layer Perceptron) is primarily used for supervised learning, as it is a type of artificial neural network that requires labeled data to train and learn relationships between input features and target outputs, making it suitable for tasks like classification and regression.

The key components of Multi-Layer Perceptron include:

- **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.



Every connection in the diagram is a representation of the fully connected nature of an MLP. This means that every node in one layer connects to every node in the next layer. As the data moves through the network, each layer transforms it until the final output is generated in the output layer.

WORKING OF MULTI-LAYER PERCEPTRON

Step 1: Forward Propagation

In **forward propagation**, the data flows from the input layer to the output layer, passing through any hidden layers. In forward propagation, the **MLP computes predictions**, regardless of whether we use **MSE** or **BCE**. The choice of loss function (MSE or BCE) depends on whether the task is **regression or classification**. Each neuron in the hidden layers processes the input as follows:

1. **Weighted Sum**: The neuron computes the weighted sum of the inputs:

   - $z = \sum_i w_i x_i + b$
     - Where:
       - x_i is the input feature.
       - w_i is the corresponding weight.
       - b is the bias term.

2. **Activation Function:** An activation function is a mathematical function applied to the output of a neuron. It introduces non-linearity into the model, allowing the network to learn and represent complex patterns in the data. Without this non-linearity feature, a neural network would behave like a linear regression model, no matter how many layers it has.

   The activation function decides whether a neuron should be activated by calculating the weighted sum of inputs and adding a bias term. This helps the model make complex decisions and predictions by introducing non-linearities to the output of each neuron. Neural networks consist of neurons that operate using **weights**, **biases**, and **activation functions**.

   Without non-linearity, even deep networks would be limited to solving only simple, linearly separable problems. Activation functions empower neural networks to model highly complex data distributions and solve advanced deep learning tasks. Adding non-linear activation functions introduce flexibility and enable the network to learn more complex and abstract patterns from data.

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

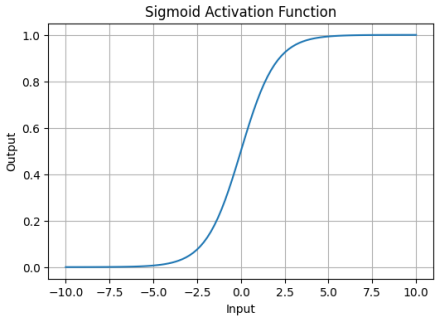
1. **Sigmoid**: Sigmoid function is a mathematical function that has an "S"-shaped curve (sigmoid curve). The sigmoid function is one of the most commonly used activation functions in Machine learning and Deep learning. It is particularly useful in neural networks, where it introduces non-linearity, allowing the model to handle complex patterns in the data.

Sigmoid function is also known as the squashing function, as it takes the input from the previously hidden layer and squeezes it between 0 and 1. So a value fed to the sigmoid function will always return a value between 0 and 1, no matter how big or small the value is fed.

The formula of the sigmoid activation function is:

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

Here, e is the base of the natural logarithm (approximately equal to 2.71828), and x is the input to the function.



2. **ReLU (Rectified Linear Unit)**:

The **Rectified Linear Unit (ReLU)** is one of the most popular activation functions used in neural networks, especially in deep learning models. It has become the default choice in many architectures due to its simplicity and efficiency. The ReLU function is a piecewise linear function that outputs the input directly if it is positive; otherwise, it outputs zero.

In simpler terms, ReLU allows positive values to pass through unchanged while setting all negative values to zero.

The ReLU function can be described mathematically as follows:

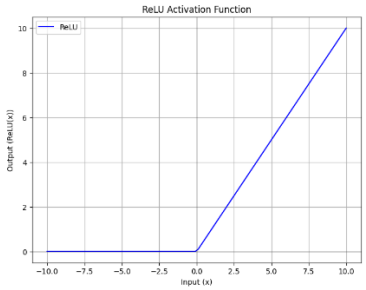
***f*(*x*)=max(0,*x*)**

Where:

- x is the input to the neuron.
- The function returns x if x is greater than 0.
- If x is less than or equal to 0, the function returns 0. In mathematical terms, the ReLU function can be written as:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

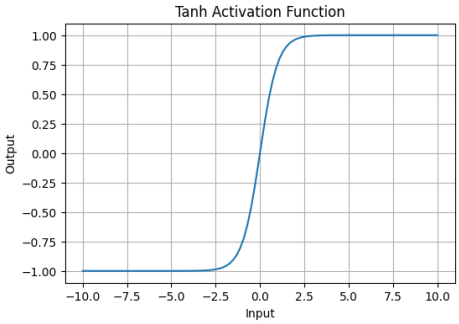
The graph of ReLU activation looks like:



3. **Tanh (Hyperbolic Tangent):** The hyperbolic tangent (tanh) activation function is a mathematical function used in artificial neural networks to transform input values into output values between -1 and 1. The tanh function outputs values in the range of -1 to +1. This means that it can deal with negative values more effectively than the sigmoid function, which has a range of 0 to 1. Tanh is preferred over sigmoid in hidden layers of a neural network, as its zero-centered property often results in faster training.

   The **tanh function** is mathematically similar to the sigmoid function but differs in its output range. It is defined as:

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



Step 2: Loss Function

A **loss function** is a mathematical function that measures how well a model's predictions match the true outcomes. It provides a quantitative metric for the accuracy of the model's predictions, which can be used to guide the model's training process. The goal of a loss function is to guide optimization algorithms in adjusting model parameters to reduce this loss over time. Once the network generates an output, the next step is to calculate the **loss** using a loss 3function. 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:

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

Where:

- y_i is the actual label.
- $\hat{y}_i$ 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**. **Both MSE and BCE can be used in backpropagation**. Backpropagation **computes gradients of the chosen loss function** (MSE or BCE) and updates the network's weights using **gradient descent**.

1. **Gradient Calculation**: The gradients of the loss function with respect to each weight and bias are calculated using the chain rule of calculus.
2. **Error Propagation**: The error is propagated back through the network, layer by layer.
3. **Gradient Descent**: The network updates the weights and biases by moving in the opposite direction of the gradient to reduce the loss

For both **regression (MSE loss)** and **classification (BCE loss)**, the weights are updated using the gradient descent formula:

$$W = W - \eta\frac{\partial L}{\partial W}$$

$$b = b - \eta\frac{\partial L}{\partial b}$$

where:

- η is the **learning rate** (controls the step size).
- $\frac{\partial L}{\partial W}$ is the **gradient of the loss function w.r.t. weights.**
- $\frac{\partial L}{\partial b}$ is the **gradient of the loss function w.r.t. bias.**

$$W = W - \eta\frac{\partial L}{\partial W}$$

$$b = b - \eta\frac{\partial L}{\partial b}$$

Step 4: Iteration

- Forward and backward propagation repeat over multiple epochs until the model converges (i.e., achieves an acceptable error rate).

MLP ALGORITHM:

The **Multi-Layer Perceptron (MLP) Algorithm** is like training a digital brain to learn patterns and make predictions.

1. **Start with Inputs**:
   - Give the MLP some data (like an image or numbers).
2. **Forward Propagation**:
   - Pass the data through each layer.
   - The system adjusts the importance of each input using weights and biases.
3. **Calculate the Loss**:
   - Compare the model's guess (output) to the correct answer (target).
   - If it's wrong, calculate the "error" using a loss function.
4. **Backward Propagation**:
   - Work backward to figure out how to reduce the error.
   - Adjust the weights and biases to improve the next prediction.
5. **Update Weights**:
   - Update the weights and biases to make the model smarter.
6. **Repeat**:
   - Do this many times (epochs) until the system gets good at predicting.

THE MULTI-LAYER PERCEPTRON IN PRACTICE

This section explores practical considerations for using Multi-Layer Perceptrons (MLPs) to solve real-world problems, focusing on three critical aspects: the amount of training data, the number of hidden layers, and when to stop learning.

Amount of Training Data:

• For the MLP with one hidden layer there are $(L + 1) \times M + (M + 1) \times N$ weights, where L,M,N are the number of nodes in the input, hidden, and output layers, respectively.

• The extra +1s come from the bias nodes, which also have adjustable weights

• This is a potentially huge number of adjustable parameters that we need to set during the training phase.

• Setting the values of these weights is the job of the back-propagation algorithm, which is driven by the errors coming from the training data.

• Clearly, the more training data there is, the better for learning, although the time that the algorithm takes to learn increases.

• Unfortunately, there is no way to compute what the minimum amount of data required is, since it depends on the problem.

• A rule of thumb that you should use a number of training examples that is at least 10 times the number of weights.

• This is probably going to be a very large number of examples, so neural network training is a fairly computationally expensive operation, because we need to show the network all of these inputs lots of times.

Number of Hidden Layers:

• Two Choices

- The number of hidden nodes
- The number of hidden layers

• It is possible to show mathematically that one hidden layer with lots of hidden nodes is sufficient. This is known as the Universal Approximation Theorem.

• we will never normally need more than two layers (that is, one hidden layer and the output layer)

When to stop Learning:

• The training of the MLP requires that the algorithm runs over the entire dataset many times, with the weights changing as the network makes errors in each iteration.

• Two options o Predefined number of Iterations o Predefined minimum error reached

• Using both of these options together can help, as can terminating the learning once the error stops decreasing.

• We train the network for some predetermined amount of time, and then use the validation set to estimate how well the network is generalising.

• We then carry on training for a few more iterations, and repeat the whole process.

• At some stage the error on the validation set will start increasing again, because the network has stopped learning about the function that generated the data, and started to learn about the noise that is in the data itself.

• At this stage we stop the training. This technique is called early stopping.

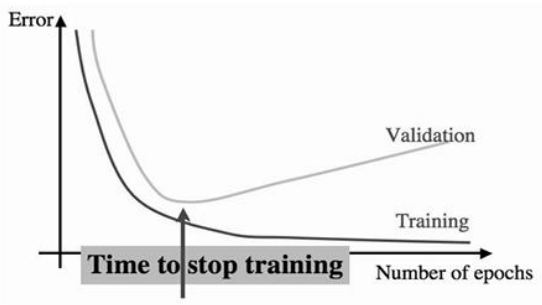


FIGURE 4.11 The effect of overfitting on the training and validation error curves, with the point at which early stopping will stop the learning marked.

EXAMPLES OF USING MLP

• We will then apply MLP to find solutions to four different types of problem: Regression, Classification, Time-series prediction, and Data compression.

Regression:

• Regression is a statistical technique that is used for predicting continuous outcomes.

• If you want to predict a single value, you only need a single output neuron and if you want to predict multiple values, you can add multiple output neurons.

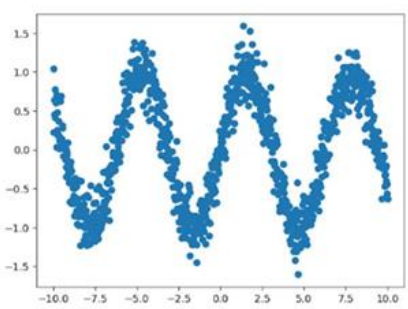
• In general, we don't apply any activation function to the output layer of MLP, when dealing with regression tasks, It just does the weighted sum and sends the output.

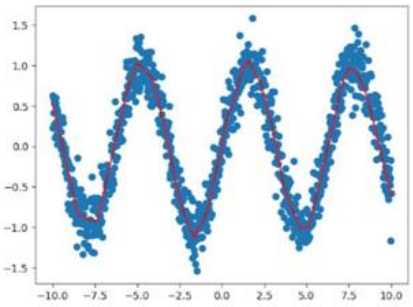
• But, in case you want your value between a given range, for example, -1 or +1 you can use activation like Tanh(Hyperbolic Tangent) function.

• The loss functions that can be used in Regression MLP include Mean Squared Error(MSE) and Mean Absolute Error(MAE).

• MSE can be used in datasets with fewer outliers, while MAE is a good measure in datasets which has more outliers.

• Example: Rainfall prediction, Stock price prediction

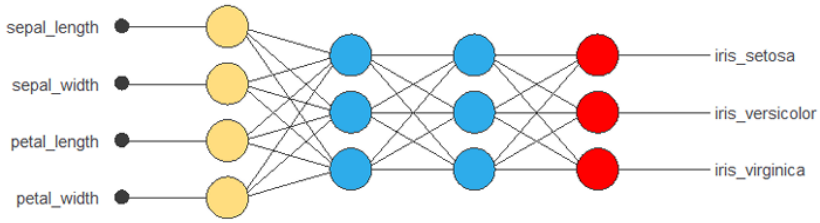




Classification:

- If the output variable is categorical, then we have to use classification for prediction.

Example: Iris Flower classification



- The aim is to classify iris flowers among three species (Setosa, Versicolor, or Virginica) from the sepals' and petals' length and width measurements.

- The above neural network has one input layer, two hidden layers and one output layer.

- In the hidden layers we use sigmoid as an activation function for all neurons.

- In the output layer, we use softmax as an activation function for the three output neurons.

- In this regard, all outputs are between 0 and 1, and their sum is 1.

- The neural network has three outputs since the target variable contains three classes (Setosa, Versicolor, and Virginica).

Time series Prediction:

- There is a common data analysis task known as time-series prediction, where we have a set of data that show how something varies over time, and we want to predict how the data will vary in the future.

- The problem is that even if there is some regularity in the time-series, it can appear over many different scales. For example, there is often seasonal variation in temperatures.

• Example: A typical time-series problem is to predict the ozone levels into the future and see if you can detect an overall drop in the mean ozone level.

Data Compression / Data denoising:

• We train the network to reproduce the inputs at the output layer called auto-associative learning.

• These networks are known as auto encoders.

• The network is trained so that whatever you give as the input is reproduced at the output, which doesn't seem very useful at first, but suppose that we use a hidden layer that has fewer neurons than the input layer.

• This bottleneck hidden layer has to represent all of the information in the input, so that it can be reproduced at the output.

• It therefore performs some compression of the data, representing it using fewer dimensions than were used in the input.

• They are finding a different representation of the input data that extracts important components of the data, and ignores the noise.

• This auto-associative network can be used to compress images and other data.

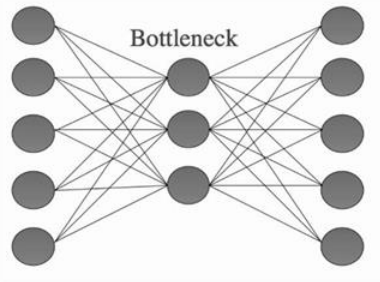


FIGURE 4.17 The auto-associative network. The network is trained to reproduce the inputs at the outputs, passing them through the bottleneck hidden layer that compresses the data.

Advantages of Multi-Layer Perceptron Neural Network

- Multi-Layer Perceptron Neural Networks can easily work with non-linear problems.
- It can handle complex problems while dealing with large datasets.
- Developers use this model to deal with the fitness problem of Neural Networks.
- It has a higher accuracy rate and reduces prediction error by using backpropagation.
- After training the model, the Multilayer Perceptron Neural Network quickly predicts the output.

Disadvantages of Multi-Layer Perceptron Neural Network

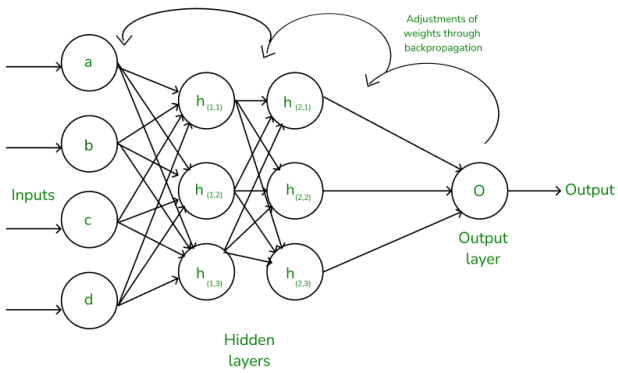
- This Neural Network consists of large computation, which sometimes increases the overall cost of the model.
- The model will perform well only when it is trained perfectly.
- Due to this model's tight connections, the number of parameters and node redundancy increases.

DERIVING BACK-PROPAGATION

Backpropagation is an algorithm used in artificial intelligence and machine learning to train artificial neural networks through error correction. The computer learns by calculating the loss function, or the difference between the input you provided and the output it produced. When you apply backpropagation, you work backward from output nodes to input nodes to reduce the loss function and produce the desired result.

How Does the Backward Pass Work?

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.
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.



Back Propagation Algorithm

The backpropagation algorithm is used in a Multilayer perceptron neural network to increase the accuracy of the output by reducing the error in predicted output and actual output.

According to this algorithm,

- Calculate the error after calculating the output from the Multilayer perceptron neural network.

- This error is the difference between the output generated by the neural network and the actual output. The calculated error is fed back to the network, from the output layer to the hidden layer.
- Now, the output becomes the input to the network.
- The model reduces error by adjusting the weights in the hidden layer.
- Calculate the predicted output with adjusted weight and check the error. The process is recursively used till there is minimum or no error.
- This algorithm helps in increasing the accuracy of the neural network.

Deriving the Backpropagation

Backpropagation is the process of adjusting a neural network's weights and biases **to reduce error**. It does this by:

1. **Calculating the error** (how wrong the model's prediction is).
2. **Finding gradients** (how much each weight contributes to the error).
3. **Updating the weights** using gradient descent to make better predictions.

Backpropagation has 4 main steps:

1. **Forward Propagation** (Make a prediction).
2. **Calculate Loss** (Measure how wrong the prediction is).
3. **Compute Gradients** (Find how much each weight contributed to the error).
4. **Update Weights** (Adjust weights to minimize the error).

Backpropagation for Regression (MSE Loss)

We use **Mean Squared Error (MSE) loss**, which is used when predicting **continuous values** (e.g., predicting house prices).

Step 1: Forward Propagation (Compute Prediction)

Each neuron performs:

$$z = WX + b$$

$$y_{\text{pred}} = f(z) \quad \text{(e.g., Linear or ReLU activation)}$$

where:

- W = weight
- X = input
- b = bias
- y_{pred} = predicted output

Step 2: Compute MSE Loss

$$L = \frac{1}{N} \sum (y_{\text{true}} - y_{\text{pred}})^2$$

where y_{true} is the actual value.

Step 3: Compute Gradients (Find Errors)

1. Compute **error signal**:

$$\delta_{\text{out}} = 2(y_{\text{pred}} - y_{\text{true}})$$

This formula calculates the **error signal**, which tells us how far the predicted output y_{pred} is from the true target y_{true}.

2. Compute **gradient w.r.t weights**:

$$\frac{\partial L}{\partial W} = \delta_{\text{out}} \cdot X$$

This formula calculates how much the **weight W** should be adjusted.

3. Compute **gradient w.r.t bias**:

$$\frac{\partial L}{\partial b} = \delta_{\text{out}}$$

This formula calculates how much the **bias b** should be adjusted.

Step 4: Update Weights

$$W = W - \eta \frac{\partial L}{\partial W}$$

$$b = b - \eta \frac{\partial L}{\partial b}$$

where η is the **learning rate**.

Repeating these steps reduces error over time. By repeating this process, the model gradually improves and learns the correct weight and bias to minimize the error.

Backpropagation for Classification (BCE Loss)

We use **Binary Cross-Entropy (BCE) loss**, which is used for **binary classification** (e.g., Spam vs. Not Spam).

Step 1: Forward Propagation (Compute Prediction)

Each neuron performs:

$$z = WX + b$$

$$y_{\text{pred}} = \sigma(z) = \frac{1}{1 + e^{-z}} \quad \text{(sigmoid activation)}$$

where y_{pred} is the predicted probability.

Step 2: Compute Binary Cross-Entropy (BCE) Loss

$$L = -\frac{1}{N} \sum [y_{\text{true}} \log(y_{\text{pred}}) + (1 - y_{\text{true}}) \log(1 - y_{\text{pred}})]$$

Step 3: Compute Gradients (Find Errors)

1. Compute **error signal**:

$$\delta_{\text{out}} = y_{\text{pred}} - y_{\text{true}}$$

   - If $y_{\text{true}} = 1$ and $y_{\text{pred}} = 0.1$, error is **high**.
   - If $y_{\text{true}} = 1$ and $y_{\text{pred}} = 0.9$, error is **low**.

2. Compute **gradient w.r.t weights**:

$$\frac{\partial L}{\partial W} = \delta_{\text{out}} \cdot X$$

3. Compute **gradient w.r.t bias**:

$$\frac{\partial L}{\partial b} = \delta_{\text{out}}$$

Step 4: Update Weights

$$W = W - \eta \frac{\partial L}{\partial W}$$

$$b = b - \eta \frac{\partial L}{\partial b}$$

Example:

Step 1: Network architecture and Define Input Values and given weights

We assume the network has:

- **1 input neuron** (X)
- **1 hidden neuron** (H) with activation function (Sigmoid)
- **1 output neuron** (y_{pred}) with activation function (Sigmoid)
- **Weights and biases**:
  - W_1 (input to hidden weight)
  - b_1 (hidden layer bias)
  - W_2 (hidden to output weight)
  - b_2 (output layer bias)

We assume:

- **Input:** $X = 1.5$
- **True Output:** $y_{\text{true}} = 1$
- **Initial Weights:**
  - $W_1 = 0.5$ (input to hidden weight)
  - $W_2 = 0.8$ (hidden to output weight)
- **Initial Biases:**
  - $b_1 = 0.1$ (hidden layer bias)
  - $b_2 = 0.2$ (output layer bias)
- **Learning Rate:** $\eta = 0.1$

Step 2: Forward Propagation: We calculate the **hidden layer activation**, then the **output layer activation**.

1.Compute Hidden Layer Activation:

The hidden layer neuron receives input:

$$z_1 = W_1 \cdot X + b_1$$

$$z_1 = (0.5 \times 1.5) + 0.1$$

$$z_1 = 0.75 + 0.1 = 0.85$$

Apply **Sigmoid activation**:

$$H = \sigma(z_1) = \frac{1}{1 + e^{-z_1}}$$

$$H = \frac{1}{1 + e^{-0.85}}$$

$$H \approx 0.7$$

2. **Compute Output Layer Activation**

The output neuron receives input:

$$z_2 = W_2 \cdot H + b_2$$

$$z_2 = (0.8 \times 0.7) + 0.2$$

$$z_2 = 0.56 + 0.2 = 0.76$$

Apply **Sigmoid activation**:

$$y_{\text{pred}} = \sigma(z_2) = \frac{1}{1 + e^{-z_2}}$$

$$y_{\text{pred}} = \frac{1}{1 + e^{-0.76}}$$

$$y_{\text{pred}} \approx 0.68$$

3. Compute Loss (Error)

Using **Mean Squared Error (MSE)**:

$$L = (y_{\text{pred}} - y_{\text{true}})^2$$

$$L = (0.68 - 1)^2$$

$$L = (-0.32)^2 = 0.1024$$

Step 3: Backward Propagation

Now, we compute gradients and update weights/biases.

1. Compute Error Signal for Output Layer

$$\delta_{\text{out}} = 2(y_{\text{pred}} - y_{\text{true}})$$

$$\delta_{\text{out}} = 2(0.68 - 1)$$

$$\delta_{\text{out}} = -0.64$$

Apply derivative of **Sigmoid function**:

$$\frac{d\sigma}{dz_2} = y_{\text{pred}}(1 - y_{\text{pred}})$$

$$\frac{d\sigma}{dz_2} = 0.68(1 - 0.68) = 0.2176$$

$$\delta_2 = \delta_{\text{out}} \times \frac{d\sigma}{dz_2}$$

$$\delta_2 = (-0.64) \times (0.2176) = -0.1393$$

2. Compute Gradient w.r.t. W2

$$\frac{\partial L}{\partial W_2} = \delta_2 \times H$$

$$\frac{\partial L}{\partial W_2} = (-0.1393) \times (0.7)$$

$$\frac{\partial L}{\partial W_2} = -0.0975$$

3. Compute Gradient w.r.t. b2

$$\frac{\partial L}{\partial b_2} = \delta_2$$

$$\frac{\partial L}{\partial b_2} = -0.1393$$

4. Compute Error Signal for Hidden Layer

$$\delta_1 = \delta_2 \times W_2 \times H(1 - H)$$

$$\delta_1 = (-0.1393) \times (0.8) \times (0.7 \times (1 - 0.7))$$

$$\delta_1 = (-0.1393) \times (0.8) \times (0.7 \times 0.3)$$

$$\delta_1 = (-0.1393) \times (0.8) \times (0.21)$$

$$\delta_1 = -0.0234$$

5. Compute Gradient w.r.t. W1

$$\frac{\partial L}{\partial W_1} = \delta_1 \times X$$

$$\frac{\partial L}{\partial W_1} = (-0.0234) \times (1.5)$$

$$\frac{\partial L}{\partial W_1} = -0.0351$$

6. Compute Gradient w.r.t. b1

$$\frac{\partial L}{\partial b_1} = \delta_1$$

$$\frac{\partial L}{\partial b_1} = -0.0234$$

Step 4: Update Weights and Biases

Using gradient descent:

$$W_1' = W_1 - \eta \times \frac{\partial L}{\partial W_1}$$

$$W_2' = W_2 - \eta \times \frac{\partial L}{\partial W_2}$$

$$b_1' = b_1 - \eta \times \frac{\partial L}{\partial b_1}$$

$$b_2' = b_2 - \eta \times \frac{\partial L}{\partial b_2}$$

Substituting values:

$$W_1' = 0.5 - (0.1 \times -0.0351) = 0.5 + 0.0035 = 0.5035$$

$$W_2' = 0.8 - (0.1 \times -0.0975) = 0.8 + 0.00975 = 0.80975$$

$$b_1' = 0.1 - (0.1 \times -0.0234) = 0.1 + 0.00234 = 0.10234$$

$$b_2' = 0.2 - (0.1 \times -0.1393) = 0.2 + 0.01393 = 0.21393$$

Final Updated Values

- Updated W_1: 0.5035
- Updated W_2: 0.80975
- Updated b_1: 0.10234
- Updated b_2: 0.21393

By repeating this process over multiple iterations (epochs), the neural network **learns** the correct weights and biases to reduce the error.

Note: Hidden layers do have their own weights and biases. The **hidden layer does have an input value**, but it comes from the previous layer

Each neuron in a layer is connected to neurons in the previous layer via **weights**. Every layer (except the input layer) has:

- **Weights** for each connection from the previous layer.
- **Biases** added to the weighted sum before applying the activation function.

For a Neural Network with 1 Input, 1 Hidden Layer, and 1 Output Layer:

- **Input Layer → Hidden Layer:**
  - Weight: W1 (connects input to hidden neuron)
  - Bias: b1 (bias for hidden neuron)
- **Hidden Layer → Output Layer:**
  - Weight: W2 (connects hidden neuron to output neuron)
  - Bias: b2(bias for output neuron)

Each layer **learns its own set of weights and biases**.

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 feedforward to complex convolutional 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:

1. **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.
2. **Exploding Gradients**: The gradients can also become excessively large, causing the network to diverge during training.
3. **Overfitting**: If the network is too complex, it might memorize the training data instead of learning general patterns.

RADIAL BASIS FUNCTIONS AND SPLINES

THE RADIAL BASIS FUNCTION (RBF) NETWORK

A radial basis function (RBF) neural network is a type of artificial neural network that uses radial basis functions as activation functions. It typically consists of three layers: an input layer, only one hidden layer, and an output layer. The hidden layer applies a radial basis function, usually a Gaussian function. RBF neural networks are highly versatile and are extensively used in pattern classification tasks, function approximation, and a variety of machine learning applications. They are especially known for their ability to handle non-linear problems effectively.

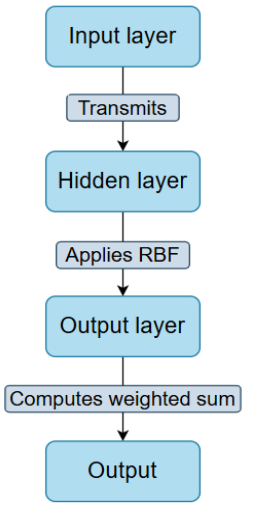
Structure of RBF neural networks

An RBF neural network typically comprises three layers:

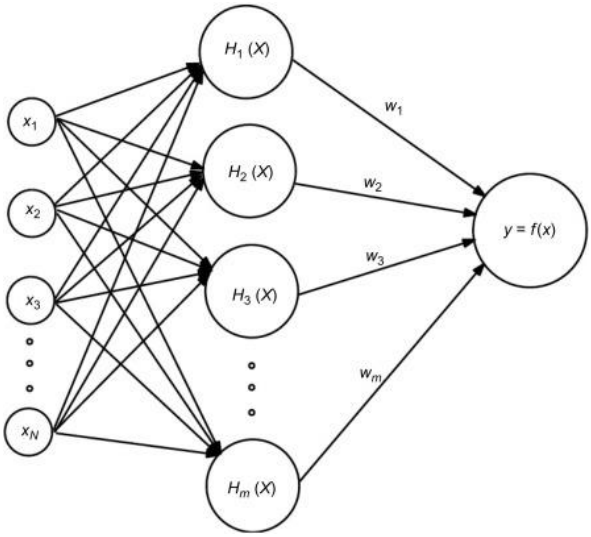
- **Input layer**: This layer simply transmits the inputs to the neurons in the hidden layer.
- **Hidden layer**: Each neuron in this layer applies a radial basis function to the inputs it receives. RBF has strictly one hidden layer.
- **Output layer**: Each neuron in this layer computes a weighted sum of the outputs from the hidden layer, resulting in the final output.

Working of RBF

- When dealing with **non-linear data**, we aim to convert it into **linearly separable data**.
- To achieve this, every hidden layer neuron uses a **non-linear radial basis function** as the activation function, transforming the data into a **higher-dimensional space**.



Structure of RBF neural networks



Types of Radial Basis Functions:

1. **Gaussian RBF** (most common):

$$H(x) = e^{-\frac{(x-c)^2}{r^2}}$$

x = Input
c = Center
r = Radius

2. **Multiquadric RBF**:

$$H(x) = \sqrt{x^2 + (x-c)^2}$$

Algorithm of RBF

Input & Output

- Input: A set of **input vectors** (x1,x2,...,xn)
- Output: **y_n**

Step 1: Initialize Weights

- Assign weights for each connection from **hidden layer** to **output layer**.
- Initially, weights are randomly assigned in the range **[-1,1]**.

Forward Phase

Step 1: Input Layer Computation

- Each node in the **input layer** directly passes its input:

$$I_i = x_i, \quad O_i = I_i$$

  - Input at **Node i**: I_i
  - Output at **Node i**: O_i (same as input)

Step 2: Hidden Layer Computation

- The hidden layer applies the **Radial Basis Function**:

$$H_j(x) = e^{-\frac{(x-c_j)^2}{r^2}}$$

  - x = Input
  - c_j = Center
  - r = Radius

- The distance between **input x** and **center c** determines the activation.

Step 3: Output Layer Computation

- Compute **weighted sum** of hidden layer outputs:

$$F(x) = \sum W_{jk} H_j(x)$$

  - W_{jk} = Weight from hidden neuron j to output neuron k
  - $H_j(x)$ = Output of hidden neuron

- The final output y_o is obtained.
- If the output is **not satisfactory**, we perform **backpropagation**.

Backward Phase (Training)

1. Train the hidden layer using backpropagation.

2. Update weights **between** hidden layer **and** output layer**.**

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.

Advantages of RBF Networks

1. **Universal Approximation**: RBF Networks can approximate any continuous function with arbitrary accuracy given enough neurons.
2. **Faster Learning**: The training process is generally faster compared to other neural network architectures.
3. **Simple Architecture**: The straightforward, three-layer architecture makes RBF Networks easier to implement and understand.

Applications of RBF Networks

- **Classification**: RBF Networks are used in pattern recognition and classification tasks, such as speech recognition and image classification.
- **Regression**: These networks can model complex relationships in data for prediction tasks.
- **Function Approximation**: RBF Networks are effective in approximating non-linear functions.

THE CURSE OF DIMENSIONALITY

The curse of dimensionality is a common machine learning problem that occurs when a dataset has many dimensions. This can make it difficult to analyze, organize, and model the data. The Curse of Dimensionality refers to the various challenges and complications that arise when analyzing and organizing data in high-dimensional spaces (often hundreds or thousands of dimensions). In the realm of machine learning, it's crucial to understand this concept because as the number of features or dimensions in a dataset increases, the amount of data we need to generalize accurately grows exponentially.

What problems does it cause?

1. Data sparsity: As mentioned, data becomes sparse, meaning that most of the high-dimensional space is empty. This makes clustering and classification tasks challenging.
2. Increased computation: More dimensions mean more computational resources and time to process the data.

3. Overfitting: With higher dimensions, models can become overly complex, fitting to the noise rather than the underlying pattern. This reduces the model's ability to generalize to new data.
4. Distances lose meaning: In high dimensions, the difference in distances between data points tends to become negligible, making measures like **Euclidean distance** less meaningful.
5. Performance degradation: Algorithms, especially those relying on distance measurements like **k-nearest neighbors**, can see a drop in performance.
6. Visualization challenges: High-dimensional data is hard to visualize, making exploratory data analysis more difficult.

Why does the curse of dimensionality occur?

It occurs mainly because as we add more features or dimensions, we're increasing the complexity of our data without necessarily increasing the amount of useful information. Moreover, in high-dimensional spaces, most data points are at the "edges" or "corners," making the data sparse.

How to Solve the Curse of Dimensionality

The primary solution to the curse of dimensionality is "dimensionality reduction." It's a process that reduces the number of random variables under consideration by obtaining a set of principal variables. By reducing the dimensionality, we can retain the most important information in the data while discarding the redundant or less important features.

Dimensionality Reduction Methods

Principal Component Analysis (PCA)

PCA is a statistical method that transforms the original variables into a new set of variables, which are linear combinations of the original variables. These new variables are called principal components.

Let's say we have a dataset containing information about different aspects of cars, such as horsepower, torque, acceleration, and top speed. We want to reduce the dimensionality of this dataset using PCA.

Using PCA, we can create a new set of variables called principal components. The first principal component would capture the most variance in the data, which could be a combination of horsepower and torque. The second principal component might represent acceleration and top speed. By reducing the dimensionality of the data using PCA, we can visualize and analyze the dataset more effectively.

Linear Discriminant Analysis (LDA)

LDA aims to identify attributes that account for the most variance between classes. It's particularly useful for classification tasks. Suppose we have a dataset with various features of flowers, such as petal length, petal width, sepal length, and sepal width. Additionally, each flower in the dataset is labeled as either a rose or a lily. We can use LDA to identify the attributes that account for the most variance between these two classes.

LDA might find that petal length and petal width are the most discriminative attributes between roses and lilies. It would create a linear combination of these attributes to form a new variable, which can then be used for classification tasks. By reducing the dimensionality using LDA, we can improve the accuracy of flower classification models.

t-Distributed Stochastic Neighbor Embedding (t-SNE)

t-SNE is a non-linear dimensionality reduction technique that's particularly useful for visualizing high-dimensional datasets. Let's consider a dataset with images of different types of animals, such as cats, dogs, and birds. Each image is represented by a high-dimensional feature vector extracted from a deep neural network.

Using t-SNE, we can reduce the dimensionality of these feature vectors to two dimensions, allowing us to visualize the dataset. The t-SNE algorithm would map similar animals closer together in the reduced space, enabling us to observe clusters of similar animals. This visualization can help us understand the relationships and similarities between different animal types in a more intuitive way.

Autoencoders

These are neural networks used for dimensionality reduction. They work by compressing the input into a compact representation and then reconstructing the original input from this representation. Suppose we have a dataset of images of handwritten digits, such as the MNIST dataset. Each image is represented by a high-dimensional pixel vector.

We can use an **autoencoder**, which is a type of neural network, for dimensionality reduction. The autoencoder would learn to compress the input images into a lower-dimensional representation, often called the latent space. This latent space would capture the most important features of the images. We can then use the autoencoder to reconstruct the original images from the latent space representation. By reducing the dimensionality using autoencoders, we can effectively capture the essential information from the images while discarding unnecessary details.

INTERPOLATION AND BASIS FUNCTIONS

INTERPOLATION:

In machine learning, interpolation **refers to the process of estimating unknown values that fall between known data points.** This can be useful in various scenarios, such as filling in missing values in a dataset or generating new data points to smooth out a curve.

It can be used in a variety of industries, including

- **Geodesy:** Interpolation is used to map out features on Earth's surface, such as mountains or ocean currents, using satellite imagery.

- **Engineering:** Interpolation predicts how materials behave in extreme conditions, such as high temperatures or pressure.

- **Statistical analysis:** Interpolation can be used to smooth out data sets so that they become more evenly distributed. For example, if you have a spike in sales one day, you can use interpolation to smooth out the rest of your sales data for that month so that the overall trend looks smooth instead of erratic.

TYPES OF INTERPOLATION:

- **Linear interpolation:** Linear interpolation is a simple method for estimating unknown values between two known points. It assumes that the data points can be connected by a straight line.
  Formula for Linear Interpolation:

  For two known points (x_0, y_0) and (x_1, y_1), the interpolated value y for a given x is:

  $$y = y_0 + \frac{(x - x_0)(y_1 - y_0)}{x_1 - x_0}$$

  This formula calculates the value of y by assuming a linear relationship between the two points.

- **Polynomial interpolation:** What if we have **more than two** points? Instead of a straight line, we can fit a **curve** using a polynomial. This works like **connecting the dots smoothly** so the estimated values follow the trend of the data. A common method for this is **Lagrange interpolation**.

  If we have three known points: (x0,y0),(x1,y1),(x2,y2) ,we can construct a polynomial P(x) that passes through these points.

Lagrange Interpolation Formula:

$$P(x) = y_0 L_0(x) + y_1 L_1(x) + y_2 L_2(x)$$

where each $L_i(x)$ is called a **Lagrange basis polynomial** and is given by:

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}$$

This means for every $L_i(x)$, we multiply fractions that exclude x_i itself.

- **Spline Interpolation (Smooth Curves):** Spline interpolation is used when we need smooth curves instead of sharp turns. The most common type is **cubic spline interpolation**, which fits a cubic polynomial between each pair of points.

  The general form of a cubic spline is:

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$$

where a_i, b_i, c_i, d_i are coefficients determined by solving a system of equations.

Spline interpolation is especially useful in graphics, motion planning, and scientific computing.

Basis function

Instead of using a single equation to represent a function, we **combine multiple small functions** (called **basis functions**) to form the final function. It means **a function breaks into small parts** using **basis functions** so that a machine learning model can learn patterns better.

$$f(x) = \sum_{i=1}^{n} \alpha_i \Phi_i(x)$$

- $\Phi_i(x)$ are small functions that help build the final function.
- α_i are numbers (weights) that the model learns.

Think of it like building a house with Lego blocks—each basis function is a Lego piece.

This method is used in **splines** and **radial basis functions (RBFs)** to make models that can fit complex patterns.

THE CUBIC SPLINE

A **cubic spline** is a smooth curve made up of **cubic polynomials** that are joined together at specific points called **knotpoints**.

The key idea is:

- The function is made up of **different cubic equations** for different sections.
- These cubic equations **connect smoothly** at the knotpoints.
- The function and its **first two derivatives (slope & curvature) must match** at each knotpoint.
- A **basis function** is like a small building block that helps us construct the final curve.

A cubic spline has **four basic basis functions**:

$$\Phi_1(x) = 1, \quad \Phi_2(x) = x, \quad \Phi_3(x) = x^2, \quad \Phi_4(x) = x^3$$

These functions allow the curve to take different shapes.

For every **knotpoint** x_i, we add **extra basis functions**:

$$\Phi_{4+i}(x) = (x - x_i)_+^3$$

where $(x - x_i)_+$ means:

- $(x - x_i)^3$ if $x > x_i$
- 0 if $x \leq x_i$

This ensures the curve behaves properly near each knotpoint.

Once you have knotpoints, you need to choose how the function behaves in each section.

1. Constant Basis Function (Blocky Steps)

- Imagine a staircase: each step is flat and has a fixed height.
- In this case, each section (between knotpoints) has a constant value.
- This is a **piecewise constant function** (it looks like a blocky step graph).

Problem: The function is not smooth—it jumps from one level to another without a transition.

2. Linear Basis Function (Straight Line Segments)

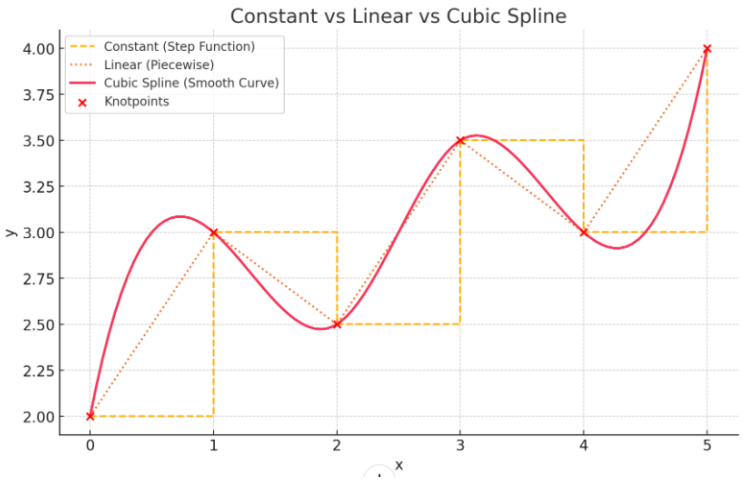
- Instead of keeping each section flat, we allow it to **increase or decrease linearly**.
- This creates a piecewise **linear function** (like a zigzag pattern).
- The function now smoothly transitions between points.

Problem: If you just use straight lines, they may not connect smoothly at **knotpoints**—meaning there might be sharp corners.

3. Cubic Basis Function (Smooth Curves)

- To avoid sharp corners, we use **cubic splines**.
- Instead of straight lines, each section is a **cubic equation**.
- This ensures that the **function, slope, and curvature** match at knotpoints.
- The result is a smooth, flowing curve.

Best Choice for Smoothness: Cubic splines! They create **smooth curves** that don't have sharp edges or abrupt changes.

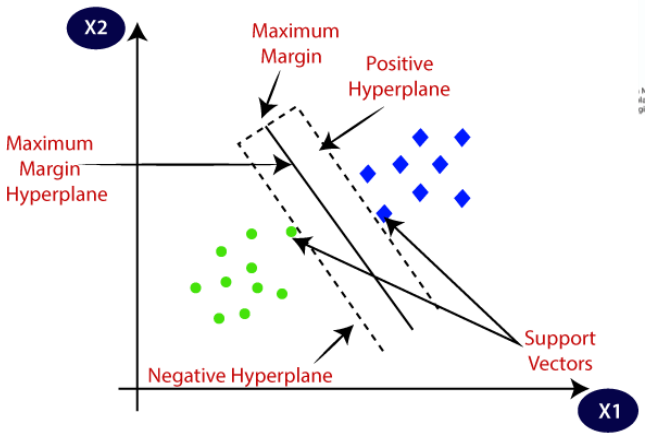


SUPPORT VECTOR MACHINE

Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.

SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called as support vectors, and hence algorithm is termed as Support Vector Machine. Consider the below diagram in which there are two different categories that are classified using a decision boundary or hyperplane:



SVM algorithm can be used for **Face detection, image classification, text categorization,** etc.

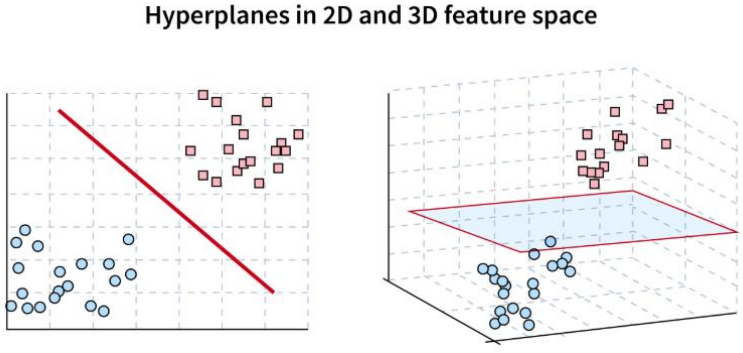
Types of SVM

SVM can be of two types:

- **Linear SVM:** Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier is used called as Linear SVM classifier.
- **Non-linear SVM:** Non-Linear SVM is used for non-linearly separated data, which means if a dataset cannot be classified by using a straight line, then such data is termed as non-linear data and classifier used is called as Non-linear SVM classifier.

Hyperplane and Support Vectors in the SVM algorithm:

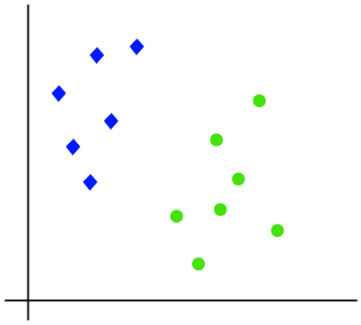
Hyperplane: There can be multiple lines/decision boundaries to segregate the classes in n-dimensional space, but we need to find out the best decision boundary that helps to classify the data points. This best boundary is known as the hyperplane of SVM.



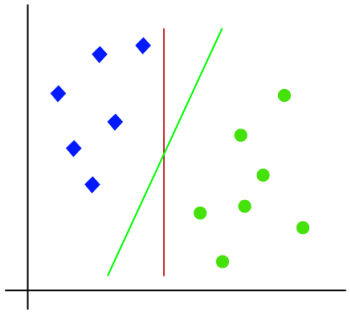
Support Vectors: The data points or vectors that are the closest to the hyperplane and which affect the position of the hyperplane are termed as Support Vector. Since these vectors support the hyperplane, hence called a Support vector.

Linear SVM:

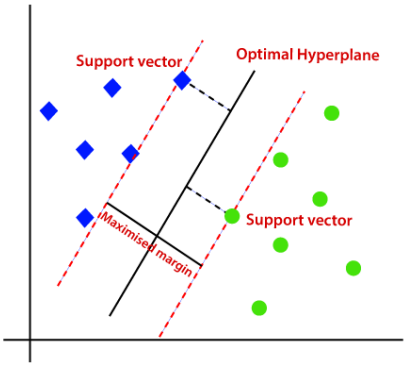
The working of the SVM algorithm can be understood by using an example. Suppose we have a dataset that has two tags (green and blue), and the dataset has two features x1 and x2. We want a classifier that can classify the pair(x1, x2) of coordinates in either green or blue. Consider the below image:



So as it is 2-d space so by just using a straight line, we can easily separate these two classes. But there can be multiple lines that can separate these classes. Consider the below image:

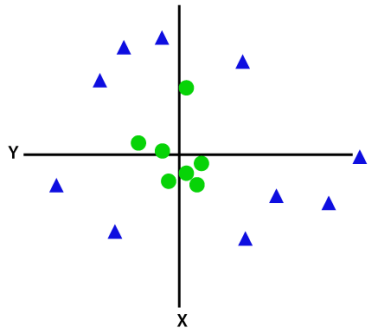


Hence, the SVM algorithm helps to find the best line or decision boundary; this best boundary or region is called as a **hyperplane**. SVM algorithm finds the closest point of the lines from both the classes. These points are called support vectors. The distance between the vectors and the hyperplane is called as **margin**. And the goal of SVM is to maximize this margin. The **hyperplane** with maximum margin is called the **optimal hyperplane**.



Kernel or Non-Linear SVM:

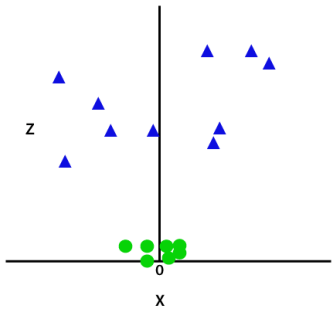
If data is linearly arranged, then we can separate it by using a straight line, but for non-linear data, we cannot draw a single straight line. Consider the below image:



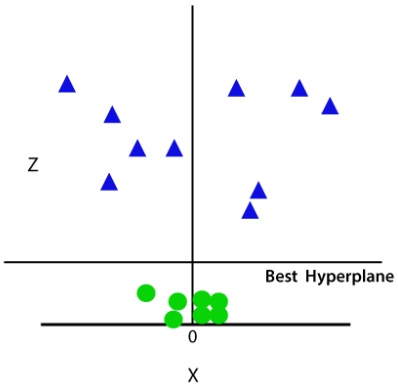
So, to separate these data points, we need to add one more dimension. For linear data, we have used two dimensions x and y, so for non-linear data, we will add a third-dimension z. It can be calculated as:

$$z=x^2 +y^2$$

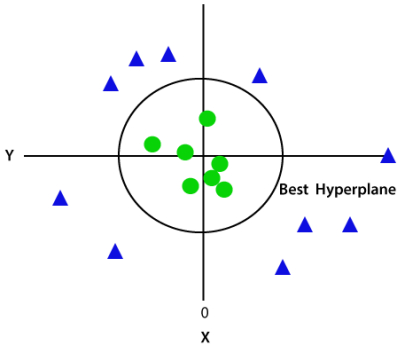
By adding the third dimension, the sample space will become as below image:



So now, SVM will divide the datasets into classes in the following way. Consider the below image:



Since we are in 3-d Space, hence it is looking like a plane parallel to the x-axis. If we convert it in 2d space with z=1, then it will become as:



Hence, we get a circumference of radius 1 in case of non-linear data.

SVM Algorithm

1. **Goal**:
    - Find the best line (or hyperplane in higher dimensions) that separates two classes of data points.
2. **Steps**:
    - **Step 1: Collect Data**:
        - Gather your data with features (e.g., height, weight) and labels (e.g., cat or dog).
    - **Step 2: Plot Data**:
        - Visualize the data points on a graph (if possible).
    - **Step 3: Find the Best Line**:
        - Draw a line that separates the two classes.
        - Make sure the line is as far as possible from the closest data points of both classes (these closest points are called **support vectors**).
    - **Step 4: Handle Non-Linear Data**:
        - If the data isn't linearly separable (you can't draw a straight line), use a trick called the **kernel trick** to transform the data into a higher dimension where a line can separate the classes.
    - **Step 5: Make Predictions**:

How does Support Vector Machine Algorithm Work?

1. **Plot the Data:** Each data point is represented in n-dimensional space (n = number of features). For example, if you have two features, you can plot the data on a 2D graph.

    Each data point is represented as x_i in an n-dimensional space:

    $$D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$$

    Where:

    - $x_i \in \mathbb{R}^n$ is the feature vector for the i-th data point.
    - $y_i \in \{-1, +1\}$ is the label (class) of the data point.

    For example:

    - In 2D: $x_i = (x_{i1}, x_{i2})$.
    - In 3D: $x_i = (x_{i1}, x_{i2}, x_{i3})$.

2. **Find the Hyperplane:** SVM finds the hyperplane (a straight line in 2D, a flat plane in 3D, or more generally, an n-dimensional plane) that separates the two classes of data points with the **maximum margin**.

- **Maximum Margin:** This is the largest possible distance between the hyperplane and the nearest data points from both classes.
- These closest points are called **support vectors** because they "support" the hyperplane.

The hyperplane is defined as:

$$w \cdot x + b = 0$$

Where:

- w: Normal vector of the hyperplane.
- b: Bias term (offset from the origin).

Maximum Margin:

The margin is the distance between the hyperplane and the closest data points (called **support vectors**).
The formula for the margin is:

$$\text{Margin} = \frac{2}{\|w\|}$$

We maximize this margin by solving the following optimization problem:

$$\min \frac{1}{2}\|w\|^2$$

Subject to:

$$y_i(w \cdot x_i + b) \geq 1 \quad \forall i$$

3. **Separate the Classes:** The hyperplane divides the data into two regions, each representing one class. For example:
   - One side of the line = Class A.
   - Other side = Class B.

The hyperplane divides the data into two regions:

- One side of the hyperplane: $w \cdot x + b > 0$ ($y = +1$, Class A).
- Other side: $w \cdot x + b < 0$ ($y = -1$, Class B).

For classification, the decision rule is:

$$f(x) = \text{sign}(w \cdot x + b)$$

4. **Non-Linearly Separable Data:** If the data cannot be separated with a straight line (e.g., spiral data), SVM uses something called a **kernel trick** to transform the data into a higher dimension where it becomes linearly separable.
   - **Kernel Functions:** Mathematical functions like polynomial, RBF (Radial Basis Function), etc., are used to transform the data.

The kernel function $K(x_i, x_j)$ replaces $w \cdot x$, allowing calculations in the transformed space without explicitly transforming the data.

Common kernel functions:

1. **Linear Kernel:**

$$K(x_i, x_j) = x_i \cdot x_j$$

2. **Polynomial Kernel:**

$$K(x_i, x_j) = (x_i \cdot x_j + c)^d$$

3. **RBF (Gaussian) Kernel:**

$$K(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$$

The decision function with kernels becomes:

$$f(x) = \text{sign}\left(\sum_{i=1}^{n} \alpha_i y_i K(x_i, x) + b\right)$$

Where α_i are the weights learned from the training process.

Advantages and Disadvantages of Support Vector Machine (SVM)

1. **High-Dimensional Performance**: SVM excels in high-dimensional spaces, making it suitable for image classification and gene expression analysis.
2. **Nonlinear Capability**: Utilizing kernel functions like RBF and polynomial, SVM effectively handles nonlinear relationships.
3. **Outlier Resilience**: The soft margin feature allows SVM to ignore outliers, enhancing robustness in spam detection and anomaly detection.
4. **Binary and Multiclass Support**: SVM is effective for both binary classification and multiclass classification, suitable for applications in text classification.
5. **Memory Efficiency**: SVM focuses on support vectors, making it memory efficient compared to other algorithms.

Disadvantages of Support Vector Machine (SVM)

1. **Slow Training**: SVM can be slow for large datasets, affecting performance in SVM in data mining tasks.
2. **Parameter Tuning Difficulty**: Selecting the right kernel and adjusting parameters like C requires careful tuning, impacting SVM algorithms.
3. **Noise Sensitivity**: SVM struggles with noisy datasets and overlapping classes, limiting effectiveness in real-world scenarios.
4. **Limited Interpretability**: The complexity of the hyperplane in higher dimensions makes SVM less interpretable than other models.
5. **Feature Scaling Sensitivity**: Proper feature scaling is essential; otherwise, SVM models may perform poorly.