UNIT-II

Multi-Layer Perceptron: Going Forwards, Going Backwards, Back Propagation Error, Multi-layer perceptron in practice, Examples of using the MLP, Deriving Back-propagation.

Radial Basis Functions and Splines: Concepts, RBF Network, Curse of Dimensionality, Interpolations and Basis Functions, Support Vector Machine

Multi-Layer Perceptron (MLP):

- The Multilayer Perceptron is a neural network where the mapping between inputs and output is non-linear.
- A Multilayer Perceptron has input and output layers, and one or more hidden layers with many neurons stacked together.
- Multilayer Perceptron can use any arbitrary activation function.
- Multilayer Perceptron falls under the category of feedforward algorithms, because inputs are combined with the initial weights in a weighted sum and subjected to the activation function just like in the Perceptron.
- But the difference is that each linear combination is propagated to the next layer.
- Each layer is feeding the next one with the result of their computation, their internal representation of the data. This goes all the way through the hidden layers to the output layer.

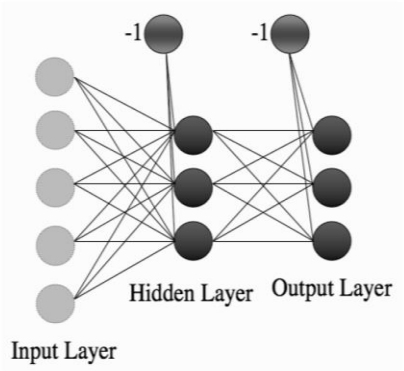


FIGURE 4.1 The Multi-layer Perceptron network, consisting of multiple layers of connected neurons.

Example - XOR Problem:

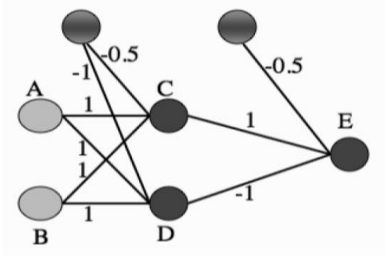


FIGURE 4.2 A Multi-layer Perceptron network showing a set of weights that solve the XOR problem.

Input: (0,1)

A=1, B=0

At Neuron C:

1x1+0x1+1x(-0.5)=1+0-0.5=0.5 > Threshold 0

Neuron C Fires, so output is 1

At Neuron D:

1x1+0x1+1x(-1)=1+0-1=0

Neuron D does not fire, so output is 0

At Neuron E:

1x1+0x(-1)+1x(-0.5)=1-0-0.5=0.5 > Threshold 0

Neuron E fires, so output is 1.

Going Forwards:

- Training the MLP consists of two parts: working out what the outputs are for the given inputs and the current weights, and then updating the weights according to the error, which is a function of the difference between the outputs and the targets.
- These are generally known as going forwards and backwards through the network.
- Each neuron in the network (whether it is a hidden layer or the output) has one extra input, with fixed value is called bias.

Going Backwards- Back Propagation of Error:

- Back-propagation of error makes it clear that the errors are sent backwards through the network.
- It is a form of gradient descent.
- The problem is that when we try to adapt the weights of the Multi-layer Perceptron, we have to work out which weights caused the error.
- This could be the weights connecting the inputs to the hidden layer, or the weights connecting the hidden layer to the output layer.
- We use sum-of-squares error function, which calculates the difference between y and t for each node, squares them, and adds them all together.

$$E(\mathbf{t},\mathbf{y}) = \frac{1}{2}\sum_{k=1}^{N}(y_k - t_k)^2.$$

Where y is output, t is target and N is the number of output nodes.

- If we differentiate a function, then it tells us the gradient of that function, which is the direction along which it increases and decreases the most.
- If we differentiate an error function, we get the gradient of the error.
- The purpose of learning is to minimise the error, we follow the error function downhill.

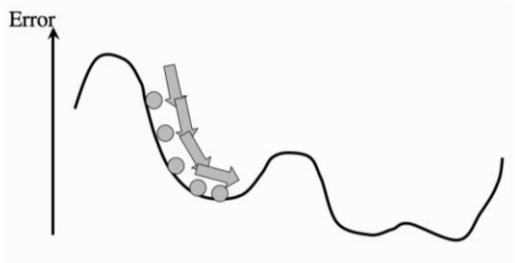


FIGURE 4.3 The weights of the network are trained so that the error goes downhill until it reaches a local minimum, just like a ball rolling under gravity.

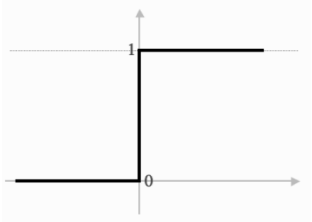


FIGURE 4.4 The threshold function that we used for the Perceptron. Note the discontinuity where the value changes from 0 to 1.

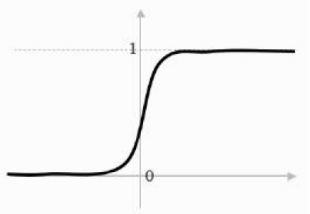


FIGURE 4.5 The sigmoid function, which looks qualitatively fairly similar, but varies smoothly and differentiably.

- We need an activation function that looks like a threshold function but is differentiable so that we can compute the gradient.

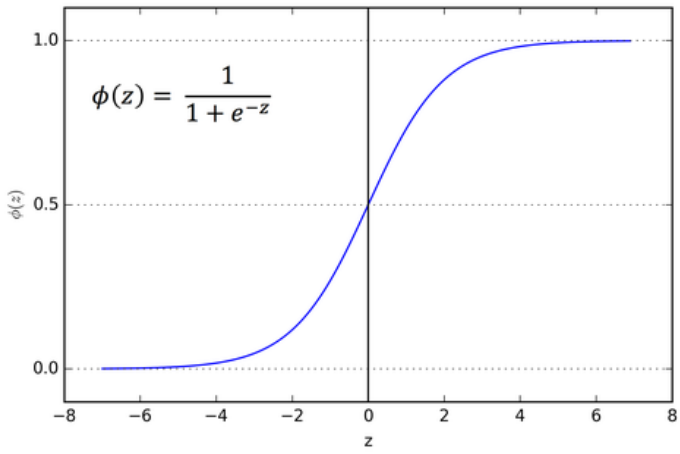
Activation Functions:

- The activation function basically decides whether a neuron should be activated or not.
- The activation function is a non-linear transformation that we do over the input before sending it to the next layer of neurons or finalizing it as output.

Sigmoid Function:

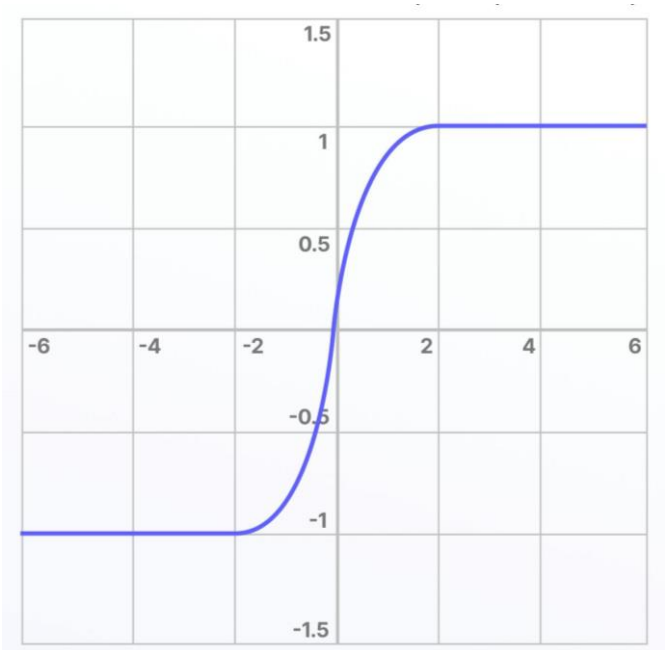
- The Sigmoid activation function, also known as the logistic activation function, takes inputs and turns them into outputs ranging between 0 and 1.
- For this reason, sigmoid is referred to as the "squashing function" and is differentiable.
- Larger, more positive inputs should produce output values close to 1.0, with smaller, more negative inputs producing outputs closer to 0.0.

$$a = g(h) = \frac{1}{1 + \exp(-\beta h)}. \qquad f(x) = \frac{1}{1 + e^{-x}}$$



Hyperbolic Tangent Function:

- Tanh function is very similar to the sigmoid/logistic activation function, and even has the same S-shape with the difference in output range of -1 to 1.
- In Tanh, the larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to -1.0.



$$a = g(h) = \tanh(h) = \frac{\exp(h) - \exp(-h)}{\exp(h) + \exp(-h)}, \qquad f(x) = \frac{\left(e^{x} - e^{-x}\right)}{\left(e^{x} + e^{-x}\right)}$$

Multi-Layer Perceptron Algorithm:

1. An input vector is put into the input nodes

2. The inputs are fed forward through the network

   - The inputs and the first-layer weights (here labelled as v) are used to decide whether the hidden nodes fire or not.

- The activation function g(·) is the sigmoid function
- The outputs of these neurons and the second-layer weights (labelled as w) are used to decide if the output neurons fire or not

3. The error is computed as the sum-of-squares difference between the network outputs and the targets

4. This error is fed backwards through the network in order to

- First update the second-layer weights and then afterwards, the first-layer weights

The Multi-layer Perceptron Algorithm

- **Initialisation**
  - initialise all weights to small (positive and negative) random values
- **Training**
  - repeat:
    - for each input vector:

      Forwards phase:

      - compute the activation of each neuron j in the hidden layer(s) using:

$$h_\zeta = \sum_{i=0}^{L} x_i v_{i\zeta} \tag{4.4}$$

$$a_\zeta = g(h_\zeta) = \frac{1}{1+\exp(-\beta h_\zeta)} \tag{4.5}$$

      - work through the network until you get to the output layer neurons, which have activations (although see also Section 4.2.3):

$$h_\kappa = \sum_j a_j w_{j\kappa} \tag{4.6}$$

$$y_\kappa = g(h_\kappa) = \frac{1}{1+\exp(-\beta h_\kappa)} \tag{4.7}$$

      Backwards phase:

      - compute the error at the output using:

$$\delta_o(\kappa) = (y_\kappa - t_\kappa)\, y_\kappa (1 - y_\kappa) \tag{4.8}$$

      - compute the error in the hidden layer(s) using:

$$\delta_h(\zeta) = a_\zeta (1 - a_\zeta) \sum_{k=1}^{N} w_\zeta \delta_o(k) \tag{4.9}$$

      - update the output layer weights using:

$$w_{\zeta\kappa} \leftarrow w_{\zeta\kappa} - \eta \delta_o(\kappa) a_\zeta^{\text{hidden}} \tag{4.10}$$

      - update the hidden layer weights using:

$$v_\iota \leftarrow v_\iota - \eta \delta_h(\kappa) x_\iota \tag{4.11}$$

    - (if using sequential updating) randomise the order of the input vectors so that you don't train in exactly the same order each iteration
  - until learning stops (see Section 4.3.3)
- **Recall**
  - use the Forwards phase in the training section above

Improvements for MLP Algoritthm:

Initializing the weights:

- The MLP algorithm suggests that the weights are initialised to small random numbers, both positive and negative.
- A common trick is to set the weights in the range $-1/\sqrt{n} < w < 1/\sqrt{n},$ where n is the number of nodes in the input layer to those weights.
- We use random values for the initialisation so that the learning starts off from different places for each run, and we keep them all about the same size because we want all of the weights to reach their final values at about the same time. This is known as uniform learning

Different Output Activation Functions:

- Sigmoid activation function in the output layer is fine for Binary classification problems.
- For regression problems, we use linear activation function in the output layer.
- For multi class classification, we use softmax activation function.

$$y_\kappa = g(h_\kappa) = \frac{\exp(h_\kappa)}{\sum_{k=1}^{N} \exp(h_k)}.$$

Problem Type	Output Type	Final Activation Function	Loss Function
Regression	Numerical value	Linear	Mean Squared Error (MSE)
Classification	Binary outcome	Sigmoid	Binary Cross Entropy
Classification	Single label, multiple classes	Softmax	Cross Entropy
Classification	Multiple labels, multiple classes	Sigmoid	Binary Cross Entropy

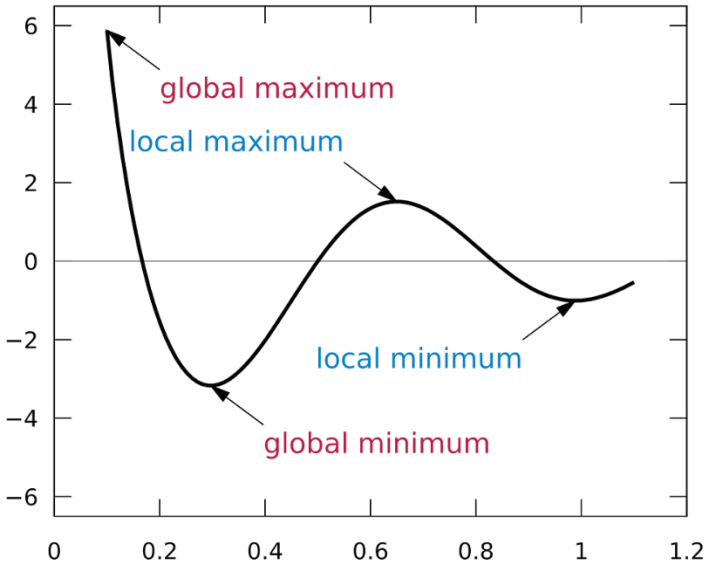
Sequential and Batch Training:

- In Batch Training, we only update the weights once for each iteration of the algorithm, which means that the weights are moved in the direction that most of the inputs want them to move, rather than being pulled around by each input individually.
- The batch method performs a more accurate estimate of the error gradient, and will thus converge to the local minimum more quickly.
- In Sequential Training, where the errors are computed and the weights updated after each input.
- This is not guaranteed to be as efficient in learning, but it is simpler to program.
- Since it does not converge as well, it can also sometimes avoid local minima, thus potentially reaching better solutions.

Local Minima:

- A loss function is a function that measures the error between a model's predictions and the ground truth.
- The goal of machine learning is to find a model that minimizes the loss function.

- A local minimum is a point in the parameter space where the loss function is minimized in a local neighborhood.
- A global minimum is a point in the parameter space where the loss function is minimized globally.



Picking Up Momentum:

- Momentum in neural networks is a parameter optimization technique that accelerates gradient descent by adding a fraction of the previous weight update to the current weight update.

$$\Delta w_{ij} = (\eta * \frac{\partial E}{\partial w_{ij}})$$

weight increment — learning rate — weight gradient

$$\Delta w_{ij} = (\eta * \frac{\partial E}{\partial w_{ij}}) + (\gamma * \Delta w_{ij}^{t-1})$$

momentum factor — weight increment, previous iteration

Minibatches and Stochastic Gradient Descent:

- MiniBatch method is to find a middle way between Batch Algorithm and Sequential Algorithm, by splitting the training set into random batches, estimating the gradient based on one of the subsets of the training set, performing a weight update, and then using the next subset to estimate a new gradient and using that for the weight update, until all of the training set have been used.
- If the batches are small, then there is often a reasonable degree of error in the gradient estimate, and so the optimisation has the chance to escape from local minima.
- In Stochastic Gradient Descent method, a single input vector is chosen from the training set, and the output and hence the error for that one vector computed, and this is used to estimate the gradient and so update the weights.
- A new random input vector is then chosen and the process repeated. This is known as stochastic gradient descent.

Multi-layer perceptron in practice:

- Here we can discuss choices that can be made about the network in order to use it for solving real problems.

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
    - Predefined number of Iterations
    - 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.

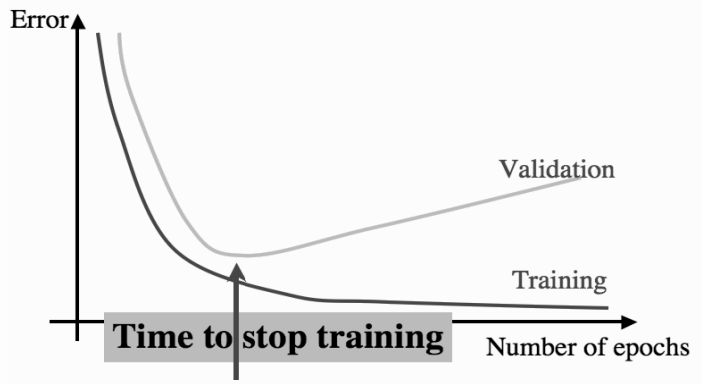


FIGURE 4.11 The effect of overfitting on the training and validation error curves, with the point at which early stopping will stop the learning marked.

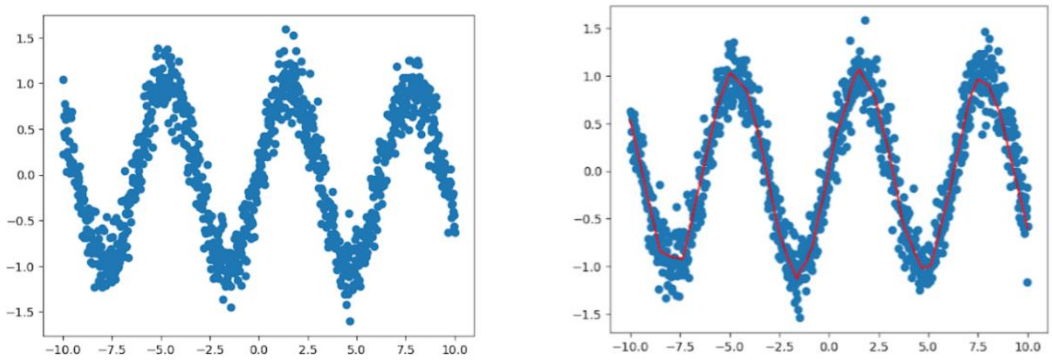
Examples of using the MLP:

- 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.

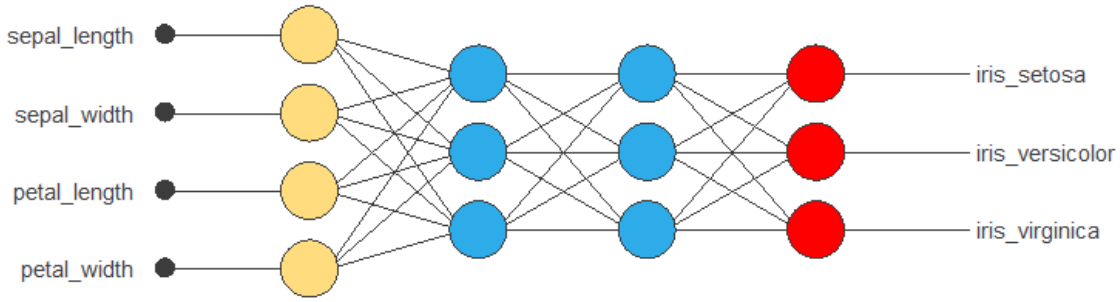
- The loss functions that can be used in Regression MLP include Mean Squared Error(MSE) and Mean Absolute Error(MAE).
- MSE can be used in datasets with fewer outliers, while MAE is a good measure in datasets which has more outliers.
- Example: Rainfall prediction, Stock price prediction



Classification:

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

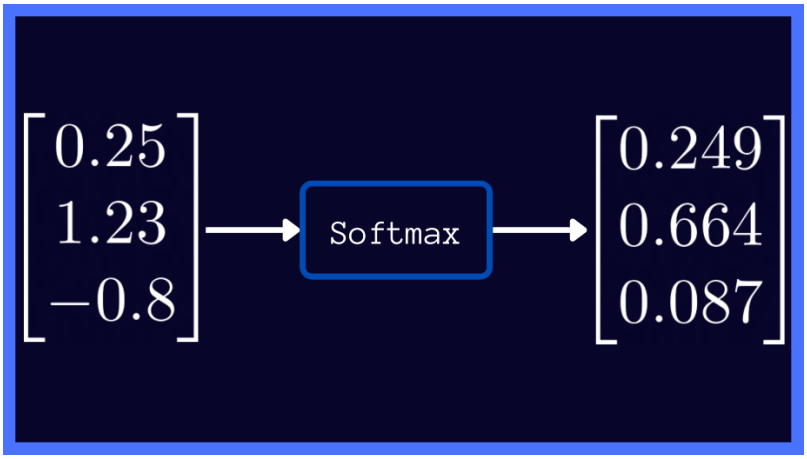
Example: Iris Flower classification



- 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).

$$softmax(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

Working of Softmax:



- The input object belongs to Class 2 (66.4%)

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.

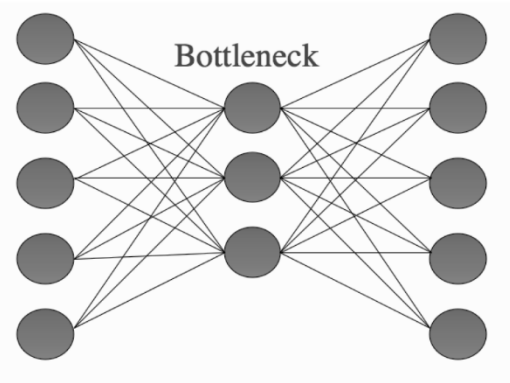


FIGURE 4.17 The auto-associative network. The network is trained to reproduce the inputs at the outputs, passing them through the bottleneck hidden layer that compresses the data.

Deriving Back-propagation:

Things to know:

1. Derivative of ½ x^2 is x
2. Chain rule:

$$\frac{dy}{dx} = \frac{dy}{dt}\frac{dt}{dx}$$

3. Derivative of constant is zero. (Not a function of x is zero)

$$\frac{dy}{dx} = 0$$

The Network output and the Error:

- The output of the neural network (the end of the forward phase of the algorithm) is a function of three things:
    - the current input (x)
    - the activation function g(·) of the nodes of the network
    - the weights of the network (v for the first layer and w for the second)
- We can't change the inputs, since they are what we are learning about, nor can we change the activation function as the algorithm learns.
- So the weights are the only things that we can vary to improve the performance of the network, i.e., to make it learn.

    - Here, x represents inputs
    - v represents first set of weights
    - w represents second set of weights
    - y represents outputs

- Note that i is an index over the input nodes, j is an index over the hidden layer neurons, and k is an index over the output neurons.

The Error of the Network:

- Error function E(v,w) remind us that the only things that we can change are the weights v and w.
- We will choose sum of squared error function

$$\begin{aligned} E(\mathbf{w}) &= \frac{1}{2}\sum_{k=1}^{N}(y_k - t_k)^2 \\ &= \frac{1}{2}\sum_{k=1}^{N}\left[g\left(\sum_{j=0}^{M} w_{jk}a_j\right) - t_k\right]^2 \end{aligned}$$

- We are going to use a gradient descent algorithm that adjusts each weight.
- The gradient that we want to know is how the error function changes with respect to the different weights

$$\begin{aligned} \frac{\partial E}{\partial w_{\iota\kappa}} &= \frac{\partial}{\partial w_{\iota\kappa}}\left(\frac{1}{2}\sum_{k=1}^{N}(y_k - t_k)^2\right) \\ &= \frac{1}{2}\sum_{k=1}^{N} 2(y_k - t_k)\frac{\partial}{\partial w_{\iota\kappa}}\left(y_k - \sum_{i=0}^{L} w_{i\kappa}x_i\right) \end{aligned}$$

Now t_k is not a function of any of the weights, since it is a value given to the algorithm, so $\frac{\partial t_k}{\partial w_{\iota\kappa}} = 0$ for all values of k, ι, κ, and the only part of $\sum_{i=0}^{L} w_{i\kappa}x_i$ that is a function of $w_{\iota\kappa}$ is when $i = \iota$, that is $w_{\iota\kappa}$ itself, which has derivative 1. Hence:

$$\frac{\partial E}{\partial w_{\iota\kappa}} = \sum_{k=1}^{N}(t_k - y_k)(-x_\iota). \tag{4.25}$$

Now the idea of the weight update rule is that we follow the gradient downhill, that is, in the direction $-\frac{\partial E}{\partial w_{\iota\kappa}}$. So the weight update rule (when we include the learning rate η) is:

$$w_{\iota\kappa} \leftarrow w_{\iota\kappa} + \eta(t_\kappa - y_\kappa)x_\iota, \tag{4.26}$$

Requirements of an Activation Function:

In order to model a neuron we want an activation function that has the following properties:

- it must be differentiable so that we can compute the gradient
- it should saturate (become constant) at both ends of the range, so that the neuron either fires or does not fire
- it should change between the saturation values fairly quickly in the middle

There is a family of functions called sigmoid functions because they are S-shaped that satisfy all those criteria perfectly.

$$a = g(h) = \frac{1}{1+\exp(-\beta h)}, \tag{4.27}$$

where β is some positive parameter. One happy feature of this function is that its derivative has an especially nice form:

$$\begin{aligned} g'(h) = \frac{dg}{dh} &= \frac{d}{dh}(1+e^{-\beta h})^{-1} \\ &= -1(1+e^{-\beta h})^{-2}\frac{de^{-\beta h}}{dh} \\ &= -1(1+e^{-\beta h})^{-2}(-\beta e^{-\beta h}) \\ &= \frac{\beta e^{-\beta h}}{(1+e^{-\beta h})^2} \\ &= \beta g(h)(1-g(h)) \\ &= \beta a(1-a) \end{aligned}$$

Back propagation of Error:

- Now we need chain rule as follows**.**

$$\frac{\partial E}{\partial w_{\zeta\kappa}} = \frac{\partial E}{\partial h_\kappa}\frac{\partial h_\kappa}{\partial w_{\zeta\kappa}},$$

Let's think about the second term first (in the third line we use the fact that $\frac{\partial w_{j\kappa}}{\partial w_{\zeta\kappa}} = 0$ for all values of j except $j = \zeta$, when it is 1):

$$\begin{aligned} \frac{\partial h_\kappa}{\partial w_{\zeta\kappa}} &= \frac{\partial \sum_{j=0}^{M} w_{j\kappa}a_j}{\partial w_{\zeta\kappa}} && (4.35) \\ &= \sum_{j=0}^{M}\frac{\partial w_{j\kappa}a_j}{\partial w_{\zeta\kappa}} && (4.36) \\ &= a_\zeta. && (4.37) \end{aligned}$$

Now we can worry about the $\frac{\partial E}{\partial h_\kappa}$ term. This term is important enough to get its own term, which is the error or delta term:

$$\delta_o(\kappa) = \frac{\partial E}{\partial h_\kappa}. \tag{4.38}$$

since we don't know much about the inputs to a neuron, we just know about its output. That's fine, because we can use the chain rule again

$$\delta_o(\kappa) = \frac{\partial E}{\partial h_\kappa} = \frac{\partial E}{\partial y_\kappa}\frac{\partial y_\kappa}{\partial h_\kappa}. \tag{4.39}$$

Now the output of output layer neuron κ is

$$y_\kappa = g(h_\kappa^{\text{output}}) = g\left(\sum_{j=0}^{M} w_{j\kappa} a_j^{\text{hidden}}\right), \tag{4.40}$$

$$\begin{aligned}
\delta_o(\kappa) &= \frac{\partial E}{\partial g\left(h_\kappa^{\text{output}}\right)}\frac{\partial g\left(h_\kappa^{\text{output}}\right)}{\partial h_\kappa^{\text{output}}} \\
&= \frac{\partial E}{\partial g\left(h_\kappa^{\text{output}}\right)} g'\left(h_\kappa^{\text{output}}\right) \\
&= \frac{\partial}{\partial g\left(h_\kappa^{\text{output}}\right)}\left[\frac{1}{2}\sum_{k=1}^{N}\left(g(h_k^{\text{output}}) - t_k\right)^2\right] g'\left(h_\kappa^{\text{output}}\right) \\
&= \left(g(h_\kappa^{\text{output}}) - t_\kappa\right) g'(h_\kappa^{\text{output}}) \\
&= (y_\kappa - t_\kappa) g'(h_\kappa^{\text{output}}),
\end{aligned}$$

where $g'(h_\kappa)$ denotes the derivative of g with respect to h_κ. This will change depending upon which activation function we use for the output neurons, so for now we will write the update equation for the output layer weights in a slightly general form and pick it up again at the end of the section:

$$\begin{aligned}
w_{\zeta\kappa} &\leftarrow w_{\zeta\kappa} - \eta\frac{\partial E}{\partial w_{\zeta\kappa}} \\
&= w_{\zeta\kappa} - \eta\delta_o(\kappa) a_\zeta.
\end{aligned} \tag{4.46}$$

where we are using the minus sign because we want to go downhill to minimise the error.

$$\begin{aligned}
\delta_h(\zeta) &= \sum_{k=1}^{N}\frac{\partial E}{\partial h_k^{\text{output}}}\frac{\partial h_k^{\text{output}}}{\partial h_\zeta^{\text{hidden}}} \\
&= \sum_{k=1}^{N}\delta_o(k)\frac{\partial h_k^{\text{output}}}{\partial h_\zeta^{\text{hidden}}},
\end{aligned}$$

The important thing that we need to remember is that inputs to the output layer neurons come from the activations of the hidden layer neurons multiplied by the second layer weights:

$$h_\kappa^{\text{output}} = \sum_{j=0}^{M} w_{j\kappa} g\left(h_j^{\text{hidden}}\right), \tag{4.49}$$

which means that:

$$\frac{\partial h_\kappa^{\text{output}}}{\partial h_\zeta^{\text{hidden}}} = \frac{\partial g\left(\sum_{j=0}^{M} w_{j\kappa} h_j^{\text{hidden}}\right)}{\partial h_j^{\text{hidden}}}. \tag{4.50}$$

We can now use a fact that we've used before, which is that $\frac{\partial h_\zeta}{\partial h_j} = 0$ unless $j = \zeta$, when it is 1. So:

$$\frac{\partial h_\kappa^{\text{output}}}{\partial h_\zeta^{\text{hidden}}} = w_{\zeta\kappa} g'(a_\zeta). \tag{4.51}$$

The hidden nodes always have sigmoidal activation functions, so that we can use the derivative that we computed in Equation (4.33) to get that $g'(a_\zeta) = \beta a_\zeta (1 - a_\zeta)$, which allows us to compute:

$$\delta_h(\zeta) = \beta a_\zeta (1 - a_\zeta) \sum_{k=1}^{N} \delta_o(k) w_\zeta. \tag{4.52}$$

This means that the update rule for v_ι is:

$$\begin{aligned} v_\iota &\leftarrow v_\iota - \eta \frac{\partial E}{\partial v_\iota} \\ &= v_\iota - \eta a_\zeta (1 - a_\zeta) \left(\sum_{k=1}^{N} \delta_o(k) w_\zeta \right) x_\iota. \end{aligned} \tag{4.53}$$

The output Activation Functions:

- For regression case, we use linear activation function.
- For multiclass classification , we use softmax activation function.

Linear $y_\kappa = g(h_\kappa) = h_\kappa$

Sigmoidal $y_\kappa = g(h_\kappa) = 1/(1 + \exp(-\beta h_\kappa))$

Soft-max $y_\kappa = g(h_\kappa) = \exp(h_\kappa) / \sum_{k=1}^{N} \exp(h_k)$

For each of these we need the derivative with respect to each of the output weights so that we can use Equation (4.45).

This is easy for the first two cases, and tells us that for linear outputs $\delta_o(\kappa) = (y_\kappa - t(\kappa)) y_\kappa$, while for sigmoidal outputs it is $\delta_o(\kappa) = \beta (y_\kappa - t(\kappa)) y_\kappa (1 - y_\kappa)$.

Derivative of softmax function is as follows:

$$\delta_o(\kappa) = (y_\kappa - t_\kappa) y_\kappa (\delta_{\kappa K} - y_K).$$

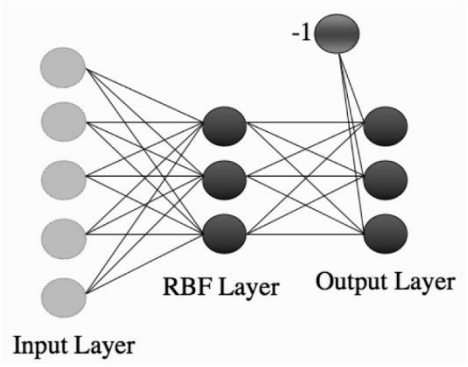
Radial Basis Functions and Splines:

Receptive Fields:

- The Receptive Field (RF) is defined as the size of the area in the input that creates the feature.
- It is essentially a measure of the relationship of an output feature (of any layer) with the input area (patch).

Radial Basis Function(RBF) Network:

- Radial Basis Function (RBF) Networks are a specialized type of Artificial Neural Network (ANN) used primarily for function approximation tasks.
- Radial Basis Function (RBF) Networks 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/RBF 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 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 (center, radius/spread) 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.

$$g(\mathbf{x}, \mathbf{w}, \sigma) = \exp\left(\frac{-\|\mathbf{x} - \mathbf{w}\|^2}{2\sigma^2}\right). \quad (5.1)$$

The Radial Basic Function Algorithm

- Position the RBF centres by either:
  - using the k-means algorithm to initialise the positions of the RBF centres OR
  - setting the RBF centres to be randomly chosen datapoints
- Calculate the actions of the RBF nodes using Equation (5.1)
- Train the output weights by either:
  - using the Perceptron OR
  - computing the pseudo-inverse of the activations of the RBF centres (this will be described shortly)

Advantages of RBF Networks:

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

Interpolation:

- Interpolation is estimating or measuring an unknown quantity between two known quantities.
- Interpolation is a mathematical function that takes the values of nearby points and uses them to predict the value of the unknown end.

- The process of interpolation involves creating a smooth curve between two data points.
- The curve is created by plotting the point on the graph at which the distance between two points is equal to half of their difference in y-coordinates.
- It is important because it ensures that your data points are evenly spaced along your line.
- The interpolation formula is as follows:

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

Example: if a child's height was measured at age 5 and age 6, interpolation could be used to estimate the child's height at age 5.5.

Basis Function:

- Radial basis functions and several other machine learning algorithms can be written in this form:

$$f(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i \Phi_i(\mathbf{x}), \tag{5.3}$$

where $\Phi_i(\mathbf{x})$ is some function of the input value $\mathbf{x}$ and the α_i are the parameters we can solve for in order to make the model fit the data. We will consider the input being scalar values x rather than vector values $\mathbf{x}$ in what follows. The $\Phi_i(x)$ are known as basis functions and they are parameters of the model that are chosen. The first thing we need to think

The Cubic Spline:

- We can continue to make the functions more complicated, with the important point being how many degrees of continuity we require at the boundaries between the points.
- These functions are known as splines, and the most common one to use is the cubic spline.
- Cubic Spline Interpolation: This type of interpolation creates a curved line to connect two points in a graph. It's also called "quadratic spline interpolation" or "quadratic smoothing."

We can carry on adding extra powers of x, but it turns out that the cubic spline is generally sufficient. This has four basic basis functions ($\Phi_1(x) = 1, \Phi_2(x) = x, \Phi_3(x) = x^2, \Phi_4(x) = x^3$), and then as many extras as there are knotpoints, each of the form $\Phi_{4+i}(x) = (x - x_i)_+^3$. This function constrains the function itself and also its first two derivatives to meet at each knotpoint. Notice that while the Φs are not linear, we are simply adding up a weighted sum of them, and so the model is linear in them. We can then produce curves like Figure 5.8, which represent the data very well.

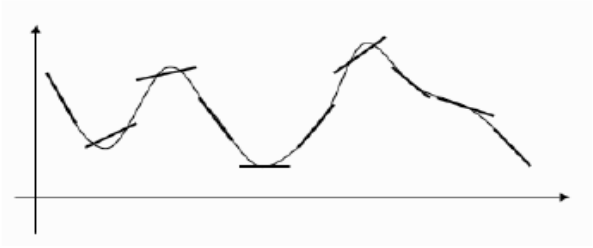


FIGURE 5.6 Representing the points by straight lines that aren't necessarily horizontal (so that their first derivative matches at the point) gives a better approximation.

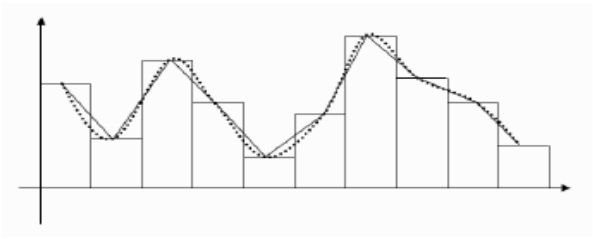


FIGURE 5.7 Making the straight lines meet so that the function is continuous gives a better approximation.

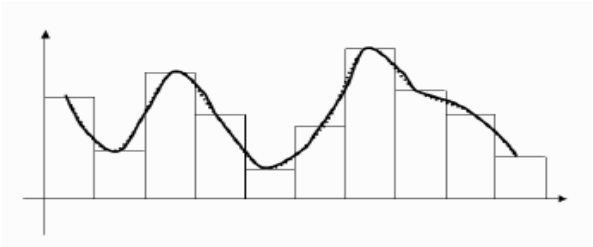


FIGURE 5.8 Using cubic functions to connect the points gives an even better approximation, and the curve is also continuous at the points where the sections join up (known as knotpoints).

Curse of Dimensionality:

- The Curse of Dimensionality refers to the phenomenon where the efficiency and effectiveness of algorithms deteriorate as the dimensionality of the data increases exponentially.
- It is 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.
- Dimensions refer to the features or attributes of data.
- For instance, if we consider a dataset of houses, the dimensions could include the house's price, size, number of bedrooms, location, and so on.

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.

How to Solve the Curse of Dimensionality?.

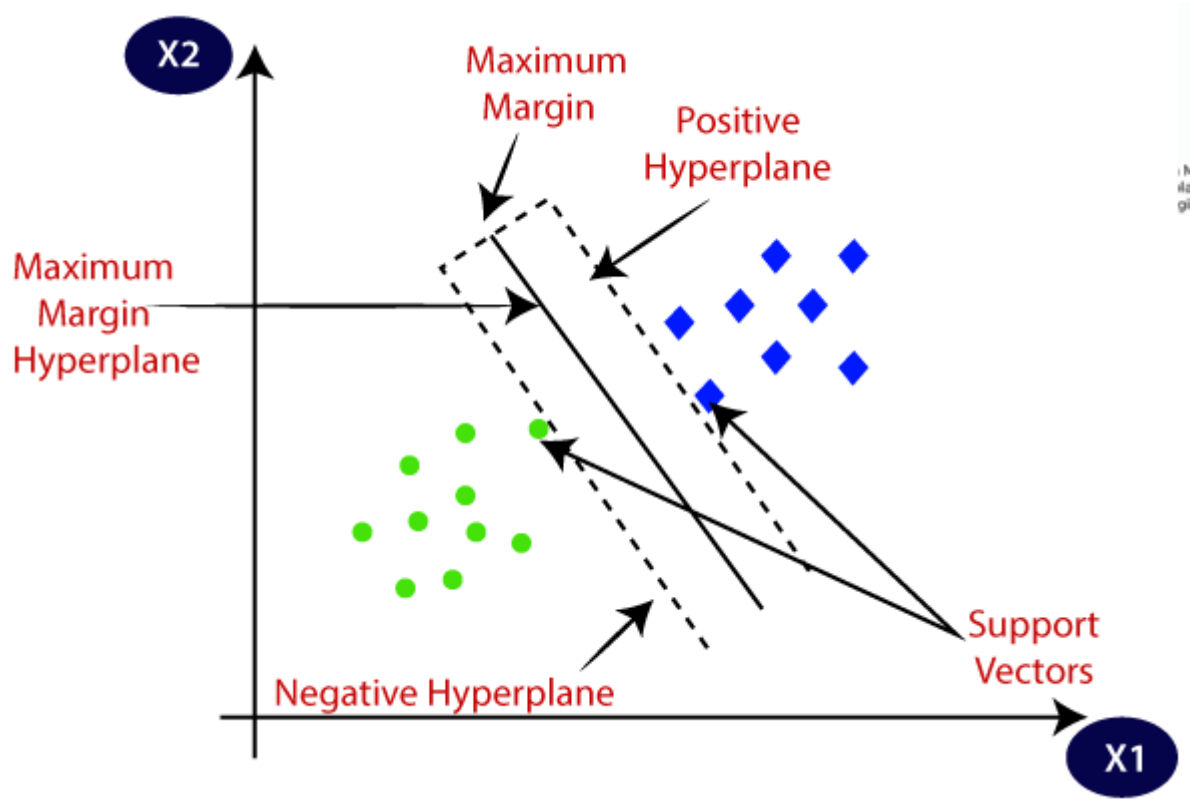
- The primary solution to the curse of dimensionality is "dimensionality reduction."
- It's a process that reduces the number of random variables under consideration by obtaining a set of principal variables.
- By reducing the dimensionality, we can retain the most important information in the data while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

Dimensionality Reduction Methods:

- Principal Component Analysis (PCA)
- Linear Discriminant Analysis (LDA)
- Factor Analysis
- Independent Component Analysis(ICA)

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.



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

Types of SVM:

- **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:

- 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.
- The dimensions of the hyperplane depend on the features present in the dataset, which means if there are 2 features, then hyperplane will be a straight line.
- And if there are 3 features, then hyperplane will be a 2-dimension plane.

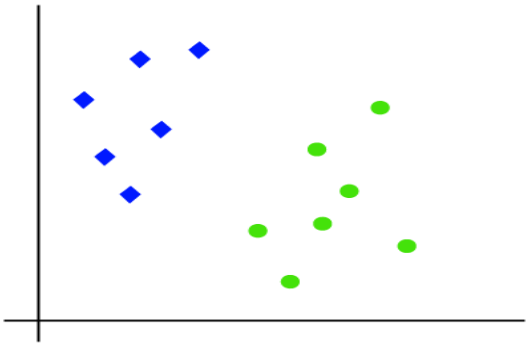
- We always create a hyperplane that has a maximum margin, which means the maximum distance between the data points.

Support Vectors:

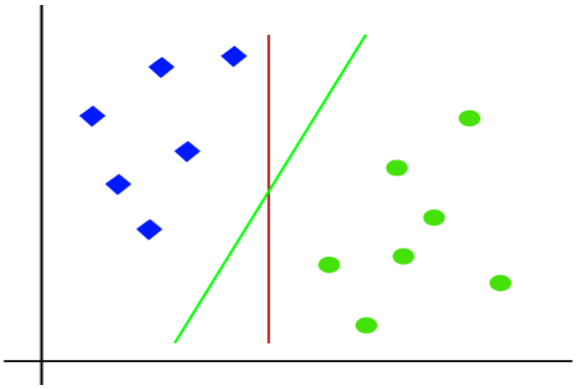
- The data points or vectors that are the closest to the hyperplane and which affect the position of the hyperplane are termed as Support Vector. Since these vectors support the hyperplane, hence called a Support vector.

Linear SVM:

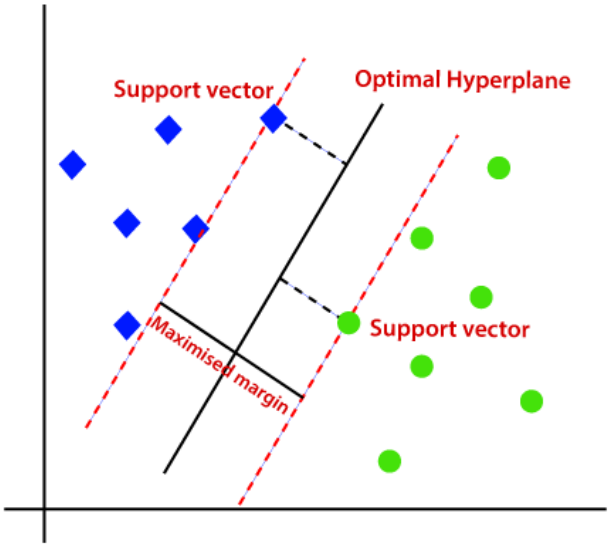
- Suppose we have a dataset that has two tags (green and blue), and the dataset has two features x1 and x2. We want a classifier that can classify the pair(x1, x2) of coordinates in either green or blue. Consider the below image:



- 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:

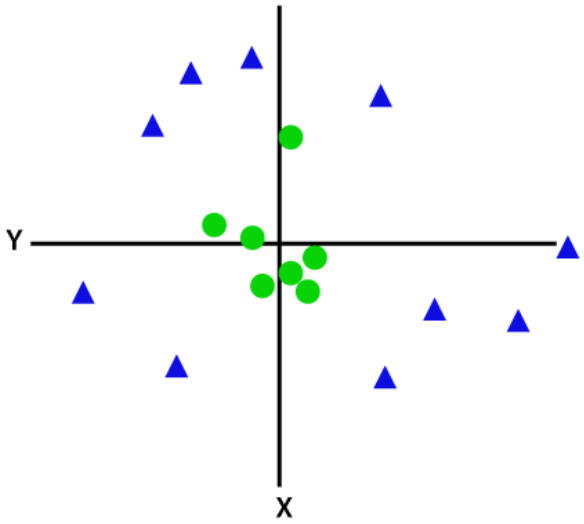


- Hence, the SVM algorithm helps to find the best line or decision boundary; this best boundary or region is called as a **hyperplane**.
- SVM algorithm finds the closest point of the lines from both the classes. These points are called support vectors.
- The distance between the vectors and the hyperplane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



Non-Linear SVM:

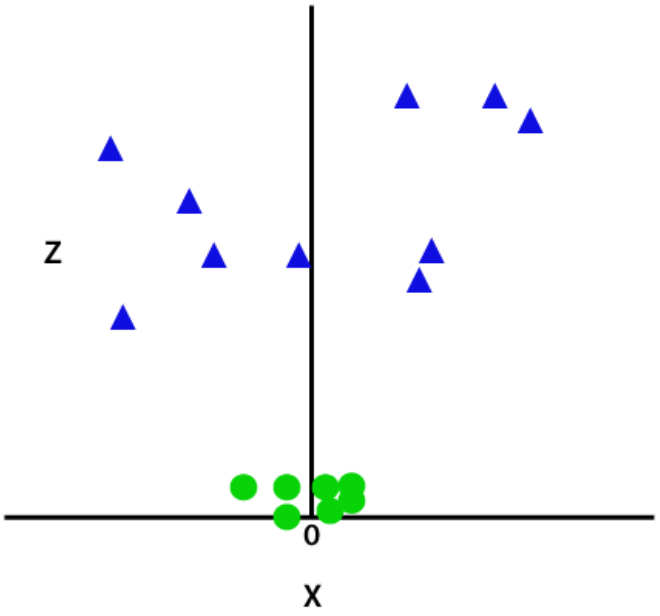
- If data is linearly arranged, then we can separate it by using a straight line, but for non-linear data, we cannot draw a single straight line. Consider the below image:



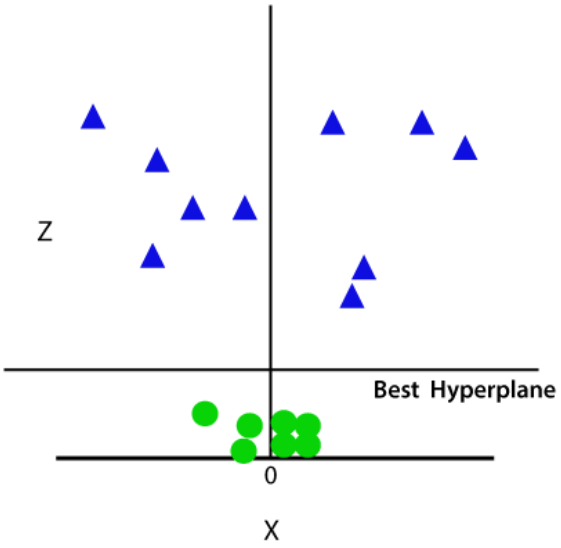
- So to separate these data points, we need to add one more dimension. For linear data, we have used two dimensions x and y, so for non-linear data, we will add a third dimension z. It can be calculated as:

  $Z=x^2+y^2$

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

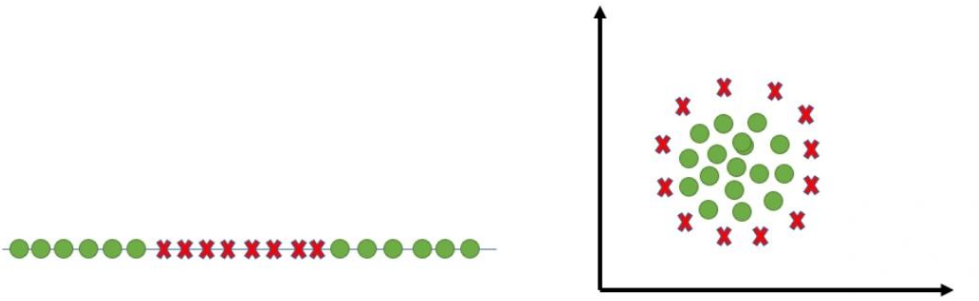


- So now, SVM will divide the datasets into classes in the following way. Consider the below image:

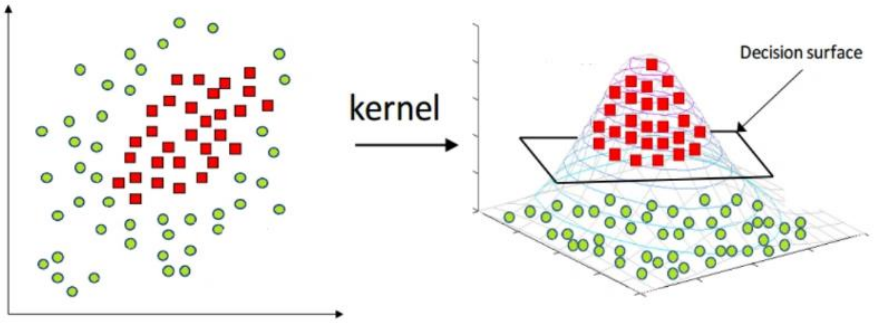


Kernels:

- The most interesting feature of SVM is that it can even work with a non-linear dataset and for this, we use "Kernel Trick" which makes it easier to classifies the points. Suppose we have a dataset like this:



- Here we see we cannot draw a single line or say hyperplane which can classify the points correctly.
- So we convert this lower dimension space to a higher dimension space using some quadratic functions which will allow us to find a decision boundary that clearly divides the data points.
- The functions which help us to do this are called Kernels and which kernel to use is purely determined by hyperparameter tuning.



Different Kernel functions:

- **Polynomial Kernel**

$$f(X1, X2) = (X1^T . X2 + 1)^d$$

  - Here d is the degree of the polynomial, which we need to specify manually.
  - Suppose we have two features X1 and X2 and output variable as Y, so using polynomial kernel we can write it as:

$$X1^T . X2 = \begin{bmatrix} X1 \\ X2 \end{bmatrix} . [X1 \quad X2]$$

$$= \begin{bmatrix} X1^2 & X1.X2 \\ X1.X2 & X2^2 \end{bmatrix}$$

  - So we basically need to find X_1^2, X_2^2 and X1.X2, and now we can see that 2 dimensions got converted into 5 dimensions.

- **Sigmoid Kernel**

$$f(x1, x2) = \tanh(\alpha x^T y + x)$$

  - It is just taking your input, mapping them to a value of 0 and 1 so that they can be separated by a simple straight line.
- **RBF Kernel**

- It creates non-linear combinations of our features to lift your samples onto a higher-dimensional feature space where we can use a linear decision boundary to separate your classes
- It is the most used kernel in SVM classifications, the following formula explains it mathematically:

$$f(x1, x2) = e^{\frac{-||(x1 - x2)||^2}{2\sigma^2}}$$

where,

1. 'σ' is the variance and our hyperparameter
2. $||X_1 - X_2||$ is the Euclidean Distance between two points X_1 and X_2

The Support Vector Machine Algorithm:

The Support Vector Machine Algorithm

- **Initialisation**
  - for the specified kernel, and kernel parameters, compute the kernel of distances between the datapoints
    - the main work here is the computation $\mathbf{K} = \mathbf{X}\mathbf{X}^T$
    - for the linear kernel, return $\mathbf{K}$, for the polynomial of degree d return $\frac{1}{\sigma}\mathbf{K}^d$
    - for the RBF kernel, compute $\mathbf{K} = \exp(-(\mathbf{x} - \mathbf{x}')^2/2\sigma^2)$
- **Training**
  - assemble the constraint set as matrices to solve:

$$\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T t_i t_j \mathbf{K}\mathbf{x} + \mathbf{q}^T\mathbf{x} \text{ subject to } \mathbf{G}\mathbf{x} \leq \mathbf{h}, \mathbf{A}\mathbf{x} = b$$

  - pass these matrices to the solver

– identify the support vectors as those that are within some specified distance of the closest point and dispose of the rest of the training data

– compute b* using equation

$$b^* = \frac{1}{N_s} \sum_{\text{support vectors } j} \left(t_j - \sum_{i=1}^{n} \lambda_i t_i \mathbf{x}_i^T \mathbf{x}_j \right).$$

- Classification
  - for the given test data $\mathbf{z}$, use the support vectors to classify the data for the relevant kernel using:
    - compute the inner product of the test data and the support vectors
    - perform the classification as $\sum_{i=1}^{n} \lambda_i t_i \mathbf{K}(\mathbf{x}_i, \mathbf{z}) + b^*$, returning either the label (hard classification) or the value (soft classification)

Advantages of SVM:

- SVM works better when the data is Linear
- It is more effective in high dimensions
- With the help of the kernel trick, we can solve any complex problem
- SVM is not sensitive to outliers
- Can help us with Image classification

Disadvantages of SVM:

- Choosing a good kernel is not easy
- It doesn't show good results on a large dataset
- The SVM hyperparameters are Cost -C and gamma. It is not that easy to fine-tune these hyper-parameters. It is hard to visualize their impact.

******