Assignment

B ANIRUDH SRINIVASAN
COE17B019

1. Decision Tree

1.1 Introduction

Decision Tree Mining is a type of data mining technique that is used to build Classification Models. It builds classification models in the form of a tree-like structure. This type of mining belongs to supervised class learning. Decision trees can be used for both categorical and numerical data.

Decision Tree has three parts,

- An inner node which represents an attribute.
- An edge representing the test on the father node.
- And a leaf node representing one of the classes.

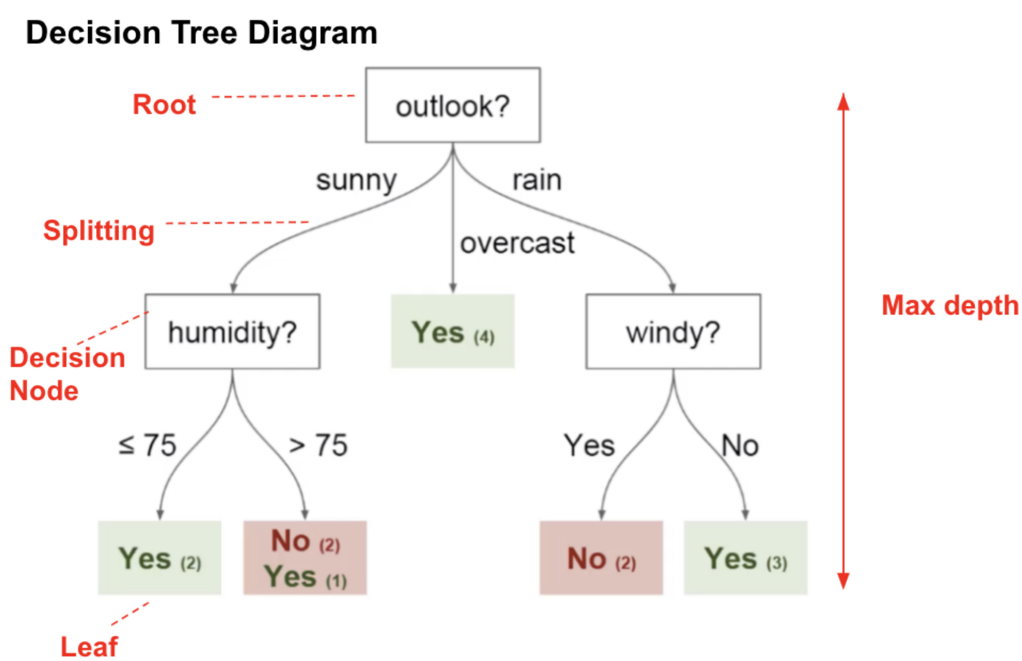


Figure 1: figure explaining different part of a decision tree

1.2 Construction of a decision tree with example

Example Dataset

Outlook	Temperature	Humidity	Windy	Play Golf
Rainy	Hot	High	FALSE	No
Rainy	Hot	High	TRUE	No
Overcast	Hot	High	FALSE	Yes
Sunny	Mild	High	FALSE	Yes
Sunny	Cool	Normal	FALSE	Yes
Sunny	Cool	Normal	TRUE	No
Overcast	Cool	Normal	TRUE	Yes
Rainy	Mild	High	FALSE	No
Rainy	Cool	Normal	FALSE	Yes
Sunny	Mild	Normal	FALSE	Yes
Rainy	Mild	Normal	TRUE	Yes
Overcast	Mild	High	TRUE	Yes
Overcast	Hot	Normal	FALSE	Yes
Sunny	Mild	High	TRUE	No

Figure 2: Example dataset

Step 1: Determine the Decision Column

Since decision trees are used for classification, you need to determine the classes which are the basis for the decision.

In this case, it it the last column, that is Play Golf column with classes Yes and No.

To determine the rootNode we need to compute the entropy. To do this, we create a frequency table for the classes (the Yes/No column).

Play Golf(14)	
Yes	No
9	5

Figure 3: Frequency table for Play Golf attribute

Step 2: Calculating Entropy for the classes

In this step, you need to calculate the entropy for the Decision column Formula for entropy is :

$$Entropy(Attribute) = -\sum_{i \in c} p_i \log_2 p_i \tag{1}$$

where,

- c is set of all classes for the attribute.
- p_i is the probability associated with that class.

The calculation of Play Golf attribute is :

$$\boldsymbol{E(PlayGolf)} = E(5,9)$$

$$= -\left(\frac{9}{14}\log_2\frac{9}{14}\right) - \left(\frac{5}{14}\log_2\frac{5}{14}\right)$$

$$= -(0.357\ \log_2 0.357) - (0.643\ \log_2 0.643)$$

$$\boldsymbol{= 0.94}$$

Figure 4: Calculating Entropy

Step 3: Calculate Entropy for Other Attributes After Split

We need to calculate the entropy of the resultant tree, wrt each attribute and use it to calculate *Gain*. The entropy for two variables is :

$$Entropy(A1, A2) = \sum_{i \in c} P(i)E(i) \tag{2}$$

where,

- c is set of all classes for the attribute A2.
- P(i) is the probability associated with the $i^t h$ class of attribute A2.
- E($i^t h$) is the entropy of the $i^t h$ class of attribute A2.

The calculation of Play Golf attribute is :
E(PlayGolf, Outloook)
E(PlayGolf, Temperature)
E(PlayGolf, Humidity)
E(PlayGolf,windy)

		PlayGolf(14)		
		Yes	No	
Outlook	Sunny	3	2	5
	Overcast	4	0	4
	Rainy	2	3	5

Figure 5: Table for Outlook and Play Golf

E(PlayGolf, Outlook) = P(Sunny) E(Sunny) + P(Overcast) E(Overcast) + P(Rainy) E(Rainy)

Which is same as:

E(PlayGolf, Outlook) = P(Sunny) E(3,2) + P(Overcast) E(4,0) + P(rainy) E(2,3)

E(x,y) = E(y,x)
Therefore, E(2,3) = E(3,2)

$$\boldsymbol{E(Sunny)} = E(3,2)$$
$$= -\left(\frac{3}{5}\log_2\frac{3}{5}\right) - \left(\frac{2}{5}\log_2\frac{2}{5}\right)$$
$$= -(0.60\ \log_2 0.60) - (0.40 \log_2 0.40)$$
$$= -(0.60 * 0.737) - (0.40 * 0.529)$$
$$\boldsymbol{= 0.971}$$

Figure 6: Calculating E(3, 2)

$$\boldsymbol{E(Overcast)} = E(4,0)$$
$$= -\left(\frac{4}{4}\log_2\frac{4}{4}\right) - \left(\frac{0}{4}\log_2\frac{0}{4}\right)$$
$$= -(0) - (0)$$
$$\boldsymbol{= 0}$$

Figure 7: Calculating E(4, 0)

Therefore, the E(Play Golf, Outlook) is :

$$\begin{aligned} E(PlayGolf, Outlook) &= \frac{5}{14}E(3,2) + \frac{4}{14}E(4,0) + \frac{5}{14}E(2,3) \\ &= \frac{5}{14}0.971 + \frac{4}{14}0.0 + \frac{5}{14}0.971 \\ &= 0.357 * 0.971 + 0.0 + 0.357 * 0.971 \\ &= \mathbf{0.693} \end{aligned}$$

Figure 8: Calculating Entropy

Similarly,

		PlayGolf(14)		
		Yes	No	
Temperature	Hot	2	2	4
	Cold	3	1	4
	Mild	4	2	6

Figure 9: Table for Temperature and Play Golf

E (PlayGolf, Temperature) = **4/14 * E**(Hot) + **4/14 * E**(Cold) + **6/14 * E**(Mild)

E (PlayGolf, Temperature) = **4/14 * E**(2, 2) + **4/14 * E**(3, 1) + **6/14 * E**(4, 2)

E (PlayGolf, Temperature) = 4/14 * -(2/4 log 2/4) – (2/4 log 2/4)
+ 4/14 * -(3/4 log 3/4) – (1/4 log 1/4)
+ 6/14 * -(4/6 log 4/6) – (2/6 log 2/6)

E (PlayGolf, Temperature) = 5/14 * 1.0
+ 4/14 * 1.811
+ 5/14 * 0.918
= 0.911

Figure 10: Calculating E(Play Golf, Temperature)

		PlayGolf(14)		
		Yes	No	
Humidity	High	3	4	7
	Normal	6	1	7

Figure 11: Table for Humidity and Play Golf

E (PlayGolf, Humidity) = **7/14 * E**(High) + **7/14 * E**(Normal)

E (PlayGolf, Humidity) = **7/14 * E**(3, 2) + **7/14 * E**(4, 0)

E (PlayGolf, Humidity) = 7/14 * -(3/7 log 3/7) – (4/7 log 4/7)

+ 7/14 * -(6/7 log 6/7) – (1/7 log 1/7)

E (PlayGolf, Humidity) = 7/14 * 0.985

+ 7/14 * 0.592

= 0.788

Figure 12: Calculating E(Play Golf, Humidity)

		PlayGolf(14)		
		Yes	No	
Windy	TRUE	3	3	6
	FALSE	6	2	8

Figure 13: Table for windy and Play Golf

E (PlayGolf, Windy) = **6/14 * E**(True) + **8/14 * E**(False)

E (PlayGolf, Windy) = **6/14 * E**(3, 3) + **8/14 * E**(6, 2)

E (PlayGolf, Windy) = 6/14 * -(3/6 log 3/6) – (3/6 log 3/6)
+ 8/14 * -(6/8 log 6/8) – (2/8 log 2/8)

E (PlayGolf, Windy) = 6/14 * 1.0
+ 8/14 * 0.811
= **0.892**

Figure 14: Calculating E(Play Golf, y)

So now that we have all the entropies for all the four attributes, let's go ahead to summarize them as shown in below:

E(PlayGolf, Outloook) = 0.693
E(PlayGolf, Temperature) = 0.911
E(PlayGolf, Humidity) = 0.788
E(PlayGolf,windy) = 0.892

Step 4: Calculating Information Gain for Each Split

The next step is to calculate the information gain for each of the attributes. The information gain is calculated from the split using each of the attributes. Then the attribute with the largest information gain is used for the split.

The information gain is calculated using the formula:

$$Gain(S,T) = Entropy(S) – Entropy(S,T) \tag{3}$$

where,

- Entropy(S) is the entropy associated with attribute S.
- Entropy(S, T) is the joint entropy of attributes, S and T.

For example, the information gain after splitting using the Outlook attribute is given by:

Gain(PlayGolf, Outlook) = Entropy(PlayGolf) – Entropy(PlayGolf, Outlook)
= 0.94 – 0.693
= 0.247

Gain(PlayGolf, Temperature) = Entropy(PlayGolf) – Entropy(PlayGolf, Tempara-ture)
= 0.94 – 0.911
= 0.029

Gain(PlayGolf, Humidity) = Entropy(PlayGolf) – Entropy(PlayGolf, Humidity)
= 0.94 – 0.788
= 0.152

Gain(PlayGolf, windy) = Entropy(PlayGolf) – Entropy(PlayGolf, windy)
= 0.94 – 0.892
= 0.048

Having calculated all the information gain, we now choose the attribute that gives the highest information gain after the split.

Step 5: Perform the First Split

Now that we have all the information gain, we then split the tree based on the attribute with the highest information gain.

From our calculation, the highest information gain comes from Outlook. Therefore the split will look like this:

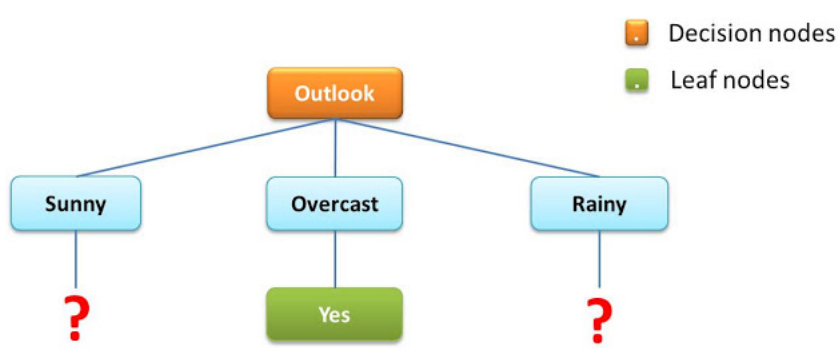


Figure 15: Decision Tree after first split

Now that we have the first stage of the decison tree, we see that we have one leaf node. But we still need to split the tree further.

To do that, we need to also split the original table to create sub tables. This sub tables are given in below.

Outlook	Temperature	Humidity	Windy	Play Golf
Sunny	Mild	Normal	FALSE	Yes
Sunny	Mild	High	FALSE	Yes
Sunny	Cool	Normal	FALSE	Yes
Sunny	Cool	Normal	TRUE	No
Sunny	Mild	High	TRUE	No
Overcast	Hot	High	FALSE	Yes
Overcast	Mild	High	TRUE	Yes
Overcast	Hot	Normal	FALSE	Yes
Overcast	Cool	Normal	TRUE	Yes
Rainy	Hot	High	FALSE	No
Rainy	Hot	High	TRUE	No
Rainy	Mild	High	FALSE	No
Rainy	Cool	Normal	FALSE	Yes
Rainy	Mild	Normal	TRUE	Yes

Figure 16: Sub Tables

Step 6: Perform Further Splits

Now, we need to recalculate the entropies and the gain for the remaining attributes, and split with the attribute that has the maximum gain.

Note that, an attribute wont repeat in the subtree with that attribute as node.
Splitting for Rainy

From the above table its clear that Humidity gives an homogeneous split, i.e Entropy(Outlook = Rainy, Humidity) = 0
Therefore, lets split Rainy with Humidity

Similarly, when the Outlook is Sunny, the attribute windy gives an homogeneous split, thus we split Sunny with windy.

Step 7: Complete the Decision Tree

Keep on splitting the decision tree, until no other attribute is left, or if we get an homogeneous split.
Therefore, the final table is :

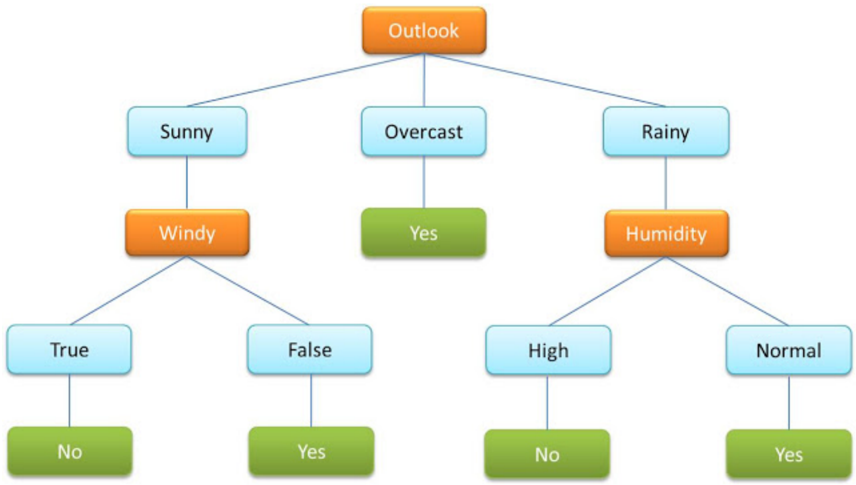


Figure 17: Sub Tables

1.3 Pseudo-Code for Decision Tree construction

```
Begin
Load learning sets  and create decision tree root node(rootNode), add learning
     set S into root not as its subset
For rootNode, compute Entropy(rootNode.subset) first
If Entropy(rootNode.subset) == 0 (subset is homogenious)
      return a leaf node

If Entropy(rootNode.subset)!= 0 (subset is not homogenious)
     compute Information Gain for each attribute left (not been used for
    spliting)
     Find attibute A with Maximum(Gain(S, A))
     Create child nodes for this root node and add to rootNode in the decision
     tree

For each child of the rootNode
   Apply ID3(S, A, V)
   Continue until a node with Entropy of 0 or a leaf node is reached
End
```

1.4 Additional Notes

1. For numerical attributes,

   Step 1: Sort the Attribute in ascending order.

   Step 2: Calculate the avg value for two adjacent datapoints.

   Step 3: Now consider each avg value to be a class, and calculate the Gain. Choose the average value with the most gain.

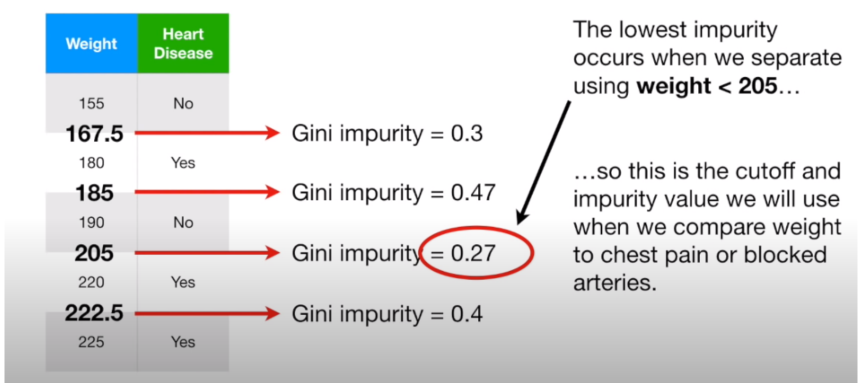


Figure 18: Illustration of the above steps. Note that here gini impurity is used and accordingly the one with the least impurity is chosen.

2. For attributes with rank as their value, we try to split based on each possible rank.

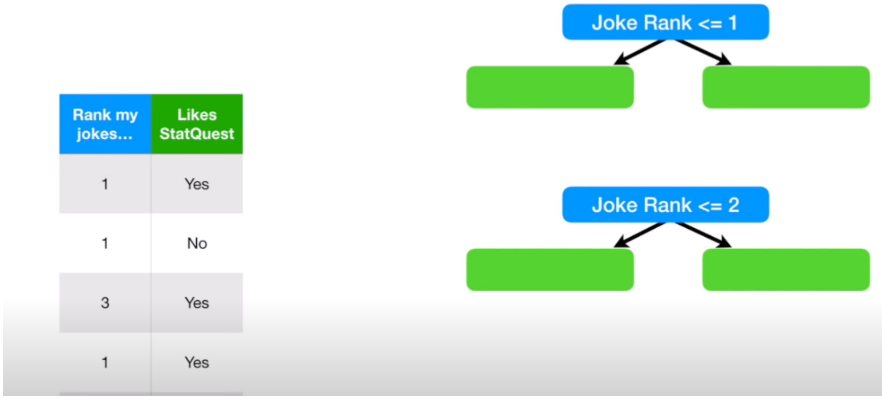


Figure 19: Illustration of the above steps. Note that the highest rank is avoided as all other ranks are less than it.

3. We can use other parameters to identify the attribute which splits the tree.
   The commonly used statistical measures are :

   - **Entropy and Information Gain**
     Entropy quantifies how much information there is in a random variable, or more specifically its probability distribution. A skewed distribution has a low entropy, whereas a distribution where events have equal probability has a larger entropy. Information gain is the reduction in entropy or surprise by transforming a dataset and is often used in training decision trees.

     $$Entropy(Attribute) = -\sum_{i\in c} p_i \log_2 p_i \tag{4}$$

     where,
     - c is set of all classes for the attribute.
     - p_i is the probability associated with that class.

     And then we find the information gain, and choose the attribute with the highest gain.

     $$Gain(S,T) = Entropy(S) – Entropy(S,T) \tag{5}$$

     where,
     - Entropy(S) is the entropy associated with attribute S.
     - Entropy(S, T) is the joint entropy of attributes, S and T.

- **Gini Impurity and Gini Gain**

  Gini Impurity is a measurement of the likelihood of an incorrect classification of a new instance of a random variable, if that new instance were randomly classified according to the distribution of class labels from the data set.

  $$G(Attribute) = \sum_{i \in c} p_i * (1 - p_i) \tag{6}$$

  where,

  - c is set of all classes for the attribute.
  - p_i is the probability associated with that class.

  And then we find the gini gains, similar to information gain, and choose the attribute with the highest gain.

  $$GG(Attribute) = 1 - Remainder_Impurity(attribute) \tag{7}$$

  $$RemainderImpurity(attribute) = \sum_{i \in B} p_i * (G(i)) \tag{8}$$

  where,

  - B is the, set of all branches for the attribute.
  - p_i is the probability associated with that branch.

  It seems that gini impurity and entropy are often interchanged in the construction of decision trees. Neither metric results in a more accurate tree than the other.

  All things considered, a slight preference might go to gini since it doesn't involve a more computationally intensive log to calculate.

- **Gain Ratio**

  In decision tree learning, Information gain ratio is a ratio of information gain to the intrinsic information.

  Information gain ratio biases the decision tree against considering attributes with a large number of distinct values. So it solves the drawback of information gain—namely, information gain applied to attributes that can take on a large number of distinct values might learn the training set too well. For example, suppose that we are building a decision tree for some data describing a business's customers. Information gain is often used to decide which of the attributes are the most relevant, so they can be tested near the root of the tree. One of the input attributes might be the customer's credit card number. This attribute has a high information gain, because it uniquely identifies each customer, but we do not want to include it in the decision tree: deciding how to treat a customer based on their credit card number is unlikely to generalize to customers we haven't seen before.

$$IV(E_x, a) = -\sum_{v \in values(a)} \frac{|\{x \in E_x | value(x, a) = v\}|}{|E_x|} * log_2 \frac{|\{x \in E_x | value(x, a) = v\}|}{|E_x|} \tag{9}$$

where,

- IV is the Intrinsic Inforamtion.
- E_x is the dataset.
- value(x, a) returns the value of attribute a for the datapoint x.
- a A, where A is the set of all attributes of E_x.

$$GainRatio(S, A) = \frac{Gain(S, A)}{IV(S, A)}) \tag{10}$$

where,

- S is the Target Variable.
- A is an attribute.

2. Neural Network Classifier (Back Propagation Network(BPN))

2.1 Introduction

A neural network is a network or circuit of neurons, or in a modern sense, an artificial neural network, composed of artificial neurons or nodes. A positive weight reflects an excitatory connection, while negative values mean inhibitory connections. All inputs are modified by a weight and summed. This activity is referred to as a linear combination. Finally, an activation function controls the amplitude of the output. For example, an acceptable range of output is usually between 0 and 1, or it could be 1 and 1.

In general terms, a neural network is a set of connected input/output units in which each connection has a weight associated with it. The weights are adjusted during the learning phase to help the network predict the correct class label of the input tuples.

These artificial networks may be used for predictive modeling, adaptive control and applications where they can be trained via a dataset. Self-learning resulting from experience can occur within networks, which can derive conclusions from a complex and seemingly unrelated set of information.

Some terms before starting BPN

- **Layer**

  Layer is a general term that applies to a collection of 'nodes' operating together at a specific depth within a neural network.

  There are three type of layer

  - The input layer is contains your raw data.
  - The hidden layer(s) are where the black magic happens in neural networks.
  - And a output layer which gives output.

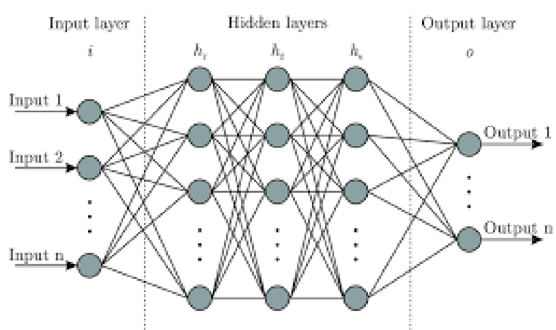
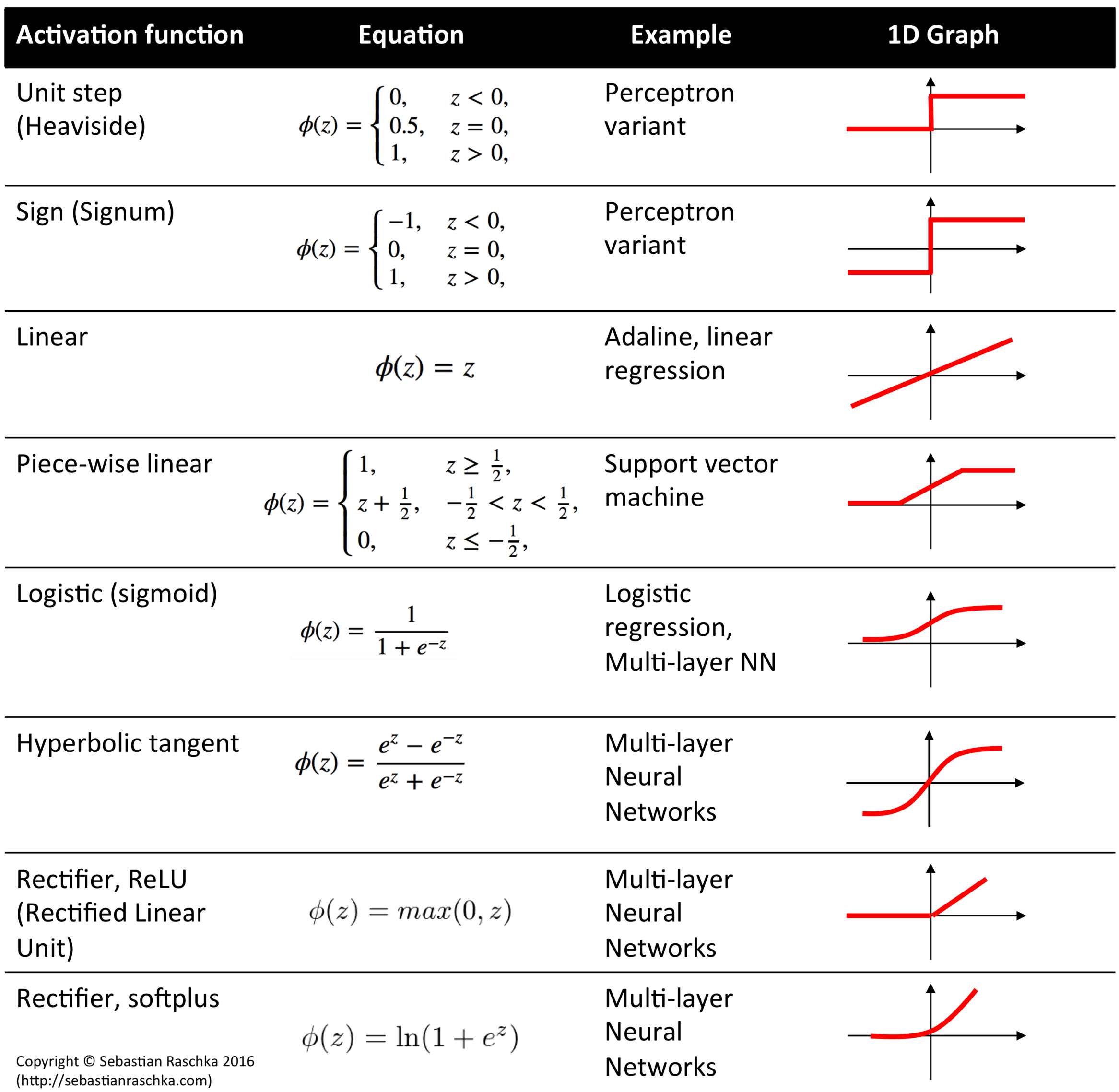


Figure 20: figure explaining different types of a layer in neural network

- **Activation function**
  In artificial neural networks, the activation function of a node defines the output of that node given an input or set of inputs. A standard integrated circuit can be seen as a digital network of activation functions that can be "ON" (1) or "OFF" (0), depending on input. This is similar to the behavior of the linear perceptron in neural networks. However, only nonlinear activation functions allow such networks to compute nontrivial problems using only a small number of nodes.



Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^{z})$	Multi-layer Neural Networks	

Figure 21: Some commonly used activation function

- **Weights**
  Weights(Parameters) — A weight represent the strength of the connection between units. If the weight from node 1 to node 2 has greater magnitude, it means that neuron 1 has greater influence over neuron 2. A weight brings down the importance of the input value.

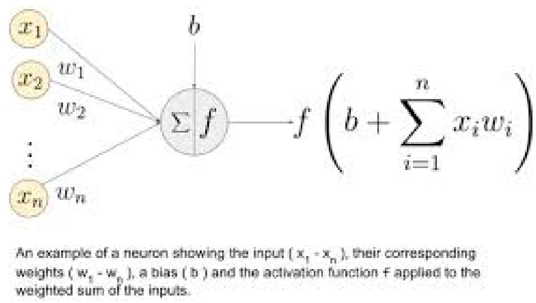


Figure 22: weights and the formula to calculate the value of a node.

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

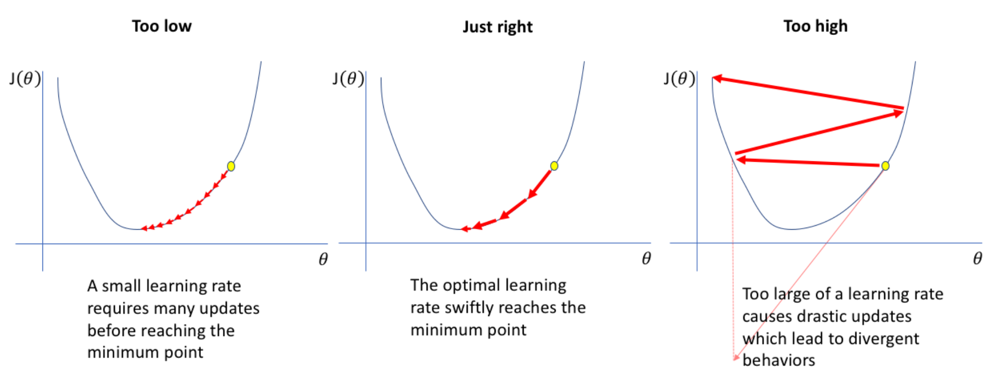


Figure 23: Effects of different Learning rate on convergence.

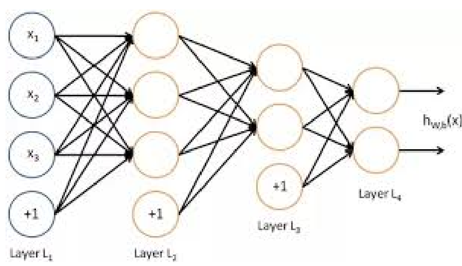
- **Bias**

  The bias neuron is a special neuron added to each layer in the neural network, which simply stores the value of 1. This makes it possible to move or "translate" the activation function left or right on the graph.

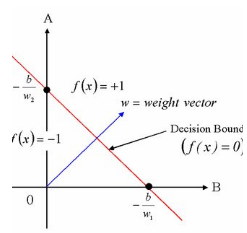
  Without a bias neuron, each neuron takes the input and multiplies it by a weight, with nothing else added to the equation. So, for example, it is not possible to input a value of 0 and output 2. In many cases, it is necessary to move the entire activation function to the left or right to generate the required output values—this is made possible by the bias.

  Although neural networks can work without bias neurons, in reality, they are almost always added, and their weights are estimated as part of the overall model.

  Note that, typically all layer (except the output layer) will have a bias neuron.



(i) Subfigure 1



(ii) Subfigure 2

Figure 24: Subfigure 1 shows a neural network model with with bias.
Subfigure 2 shows the effect of bias on the decision boundary.

- **Architecture**

  The number of layers and node in each layer constitutes the architecture of a neural network.

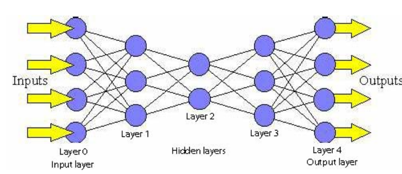


Figure 25: Illustration of architecture

- **Epoch**

  An epoch is a measure of the number of times all of the training vectors are used once to update the weights.

  For batch training all of the training samples pass through the learning algorithm simultaneously in one epoch before weights are updated.

2.2 Back Propagation Neural Network

Back-propagation is the essence of neural net training. It is the method of fine-tuning the weights of a neural net based on the error rate obtained in the previous epoch (i.e., iteration). Proper tuning of the weights allows you to reduce error rates and to make the model reliable by increasing its generalization.

Backpropagation is a short form for "backward propagation of errors." It is a standard method of training artificial neural networks. This method helps to calculate the gradient of a loss function with respects to all the weights in the network.

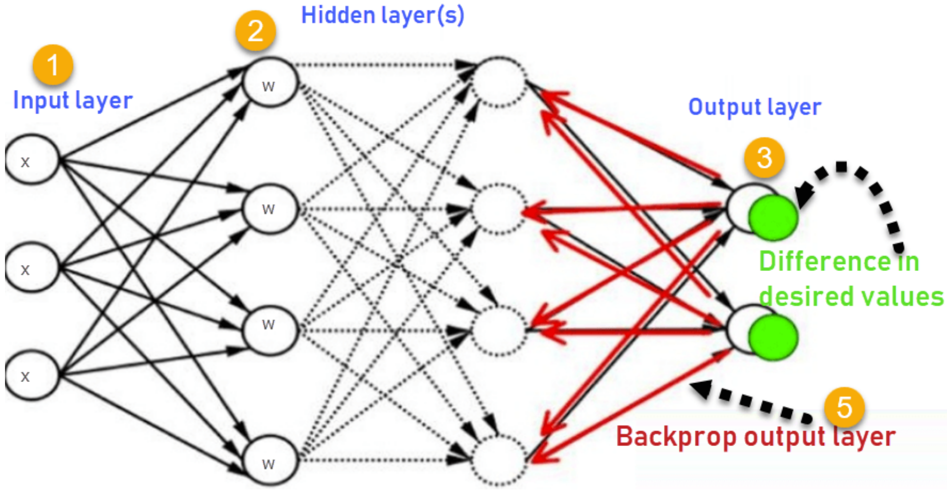


Figure 26: Simple working of BPN

Most prominent advantages of Backpropagation are:

- Backpropagation is fast, simple and easy to program t has no parameters to tune apart from the numbers of input
- It is a flexible method as it does not require prior knowledge about the network
- It is a standard method that generally works well
- It does not need any special mention of the features of the function to be learned.

Types of BPNs
Two Types of Backpropagation Networks are:

- **Static back-propagation**
  It is one kind of backpropagation network which produces a mapping of a static input for static output. It is useful to solve static classification issues like optical character recognition.
- **Recurrent Backpropagation**
  Recurrent backpropagation is fed forward until a fixed value is achieved. After that, the error is computed and propagated backward.

The main difference between both of these methods is that the mapping is rapid in static back-propagation while it is non-static in recurrent backpropagation.

2.3 Classifying using BPN with example

Example Dataset

Table 1: Example dataset

class	x	y
class 1 (w1)	2	2
class 1 (w1)	-1	2
class 1 (w1)	1	3
class 1 (w1)	-1	-1
class 1 (w1)	0.5	0.5
class 2 (w2)	-1	-3
class 2 (w2)	0	-1
class 2 (w2)	1	-2
class 2 (w2)	-1	-2
class 2 (w2)	0	-2

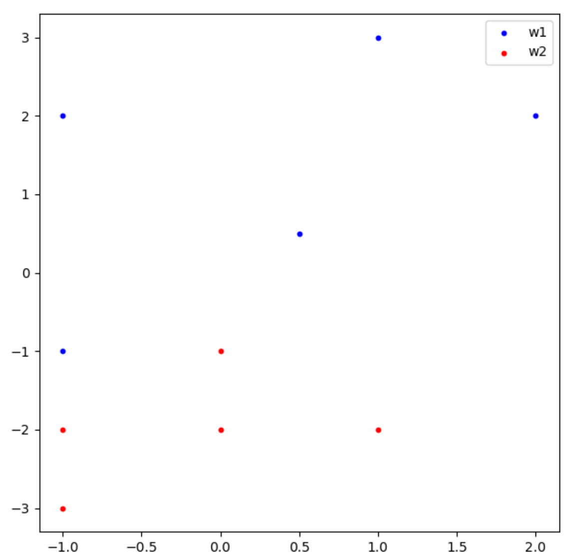


Figure 27: Visualisation of the dataset

Step 1: Determine the architecture of the neural network

The architecture of the neural network will depend on the problem. For linearly separable problem, having one hidden layer is enough. For non linearly separable and more complex problem, having more hidden layers would be better.

The number of nodes in the input layer would be the dimension of the data points + 1 (for bias). There is no constraint on the number of nodes in the input layer, but its better to have them equal to or greater than the nodes in the input layer. The number of nodes in the output layer would be the number of distinct class for the classification problem.

For this example, we are going with a neural network with 3 layer (1 - input, 1 - hidden, 1 - output) with 2 + 1 (bias) nodes for input, 2 + 1 (bias) nodes for hidden and 2 nodes for output.

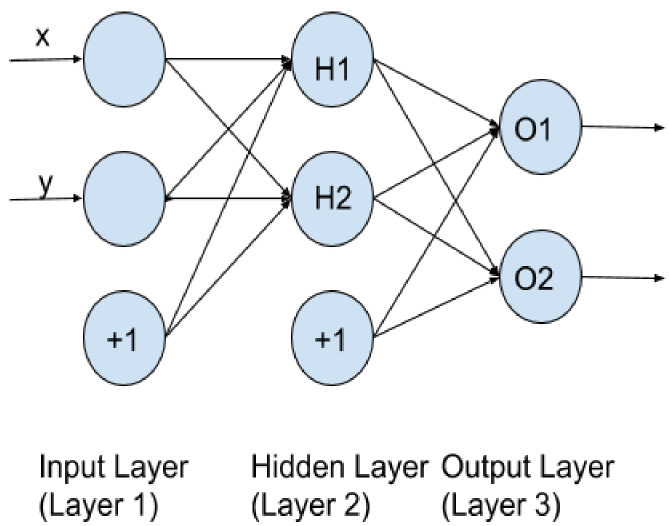


Figure 28: Architecture of the Neural Network

Step 2: Deciding the weights, learning rate, activation function and number of epoch

We need to assign value to weights, learning rate, activation function and tha max epoch value. The value of learning rate is depends upon the problem we are solving. The best way to assign weights is to randomly assign values between 0 and 1.

Here, we are using learning rate to be 0.5, running the neural network for 1000 epoch and sigmoid function as activation function.(problem dependent)

A little notation,

W^i_{jk} means the weight that is present at the j^{th} node of i^{th} layer to the k^{th} node of $(i+1)^{th}$ layer.

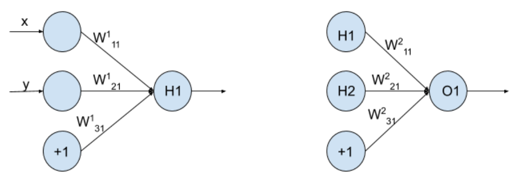


Figure 29: Illustration of the above notation

Table 2: Value of weights

Weights	Value
W^1_{11}	0.13436424411240122
W^1_{21}	0.8474337369372327
W^1_{31}	0.763774618976614
W^1_{12}	0.2550690257394217
W^1_{22}	0.49543508709194095
W^1_{32}	0.4494910647887381
W^2_{11}	0.651592972722763
W^2_{21}	0.7887233511355132
W^2_{31}	0.0938595867742349
W^2_{12}	0.02834747652200631
W^2_{22}	0.8357651039198697
W^2_{32}	0.43276706790505337

Step 3: Forward propagation

Let Out_j be the output of j^th node of a layer.
Let W_{ij} denote the weight connecting i_{th} node of a layer to the j_{th} node of the next layer.
These conventions make it easier to talk about two successive layers.
We need to forward propagate the neural network to get the ouptput value given by our network.

Forward is nothing but the calculation of output nodes for the input vector.

Initially the output would be erroneous and we need to propagate this error to update weights at each level/ layer.

Value at a node is given by,

$$N_k^{i+1} = f(\sum_{j=1}^{n} W_{jk}^i * N_j^i) \tag{11}$$

where,

- N_y^x is the x^{th} node in y^{th} layer (it also includes the bias node).
- n is the total number of nodes in i^{th} layer.
- f(.) is the activation function.

Therefore, if we use the above formula, we get, N_1^1 (H_1) = f($W^1$11 * x + $W^1$21 * y + $W^1$31 * 1)
N_2^1 (H_2) = f($W^1$12 * x + $W^1$22 * y + $W^1$32 * 1)
N_1^2 (O_1) = f($W^2$11 * H_1 + $W^2$21 * H_2 + $W^2$31 * 1)
N_2^2 (O_2) = f($W^2$12 * H_1 + $W^2$22 * H_2 + $W^2$32 * 1)

When we plug in the values for the first value of the dataset i.e. (2, 2), We get,

H_1 = f(0.13436424411240122 * 2 + 0.8474337369372327 * 2 + 0.763774618976614 * 1)
=f(2.7273705810758817)
= 0.9386225302230806

H_2 = f(0.2550690257394217 * 2 + 0.49543508709194095 * 2 + 0.4494910647887381 * 1)
= f(1.9504992904514635)
= 0.8755010741482664

O_1 = f(0.651592972722763 * 0.9386225302230806 + 0.7887233511355132 * 0.8755010741482664 + 0.0938595867742349 * 1)
= f(1.3959875726318156)
= 0.8015464048450159

O_2 = f(0.02834747652200631 * 0.9386225302230806 + 0.8357651039198697 * 0.8755010741482664 + 0.43276706790505337 * 1)
= f(1.1910878942610617)
= 0.7669355767878618

Step 4: Calculating the Error and Back Propagating it

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

$$Err_j = Out_j(1 - Out_j) * (T_j - Out_j) \tag{12}$$

where,

- Out_j(1 - Out_j) is the derivative of the logistic (sigmoid) function.
- T_j is the target value of the O_j node.

To compute the error of a hidden layer unit j, the weighted sum of the errors of the units connected to unit j in the next layer are considered. The error of a hidden layer unit j is,

$$Err_j = Out_j(1 - Out_j) * \sum_k Err_k * W_{jk} \tag{13}$$

where,

- O_j(1 - O_j) is the derivative of the logistic (sigmoid) function.
- W_{jk} is the weight of the connection from unit j to a unit k in the next higher layer.
- Err_k is the is the error of unit k.

The weights and biases are updated to reflect the propagated errors. Weights are updated by the following equations, where Δ W_{ij} is the change in weight W_{ij}:

$$\Delta W_{ij} = (l) * Err_j * Out_i \tag{14}$$
$$W_{ij} = W_{ij} + \Delta W_{ij} \tag{15}$$

where,

- l is the learning rate.

In our example, the error for Err_1 for the output layer is,
Err_i = 0.8015464048450159 * (1 - 0.8015464048450159) * (1 - 0.8015464048450159)
= 0.03156796688859639

The change in weight for Hidden Layer is,
ΔW_{11} = (0.5) * (0.03156796688859639) * (0.9386225302230806)
= 0.014815202477486385

W_{11} = 0.651592972722763 + 0.014815202477486385
= 0.6664081752002493

Similarly, the error for Err_1 for the Hidden Layer is,
Err_i = 0.9386225302230806 * (1 - 0.9386225302230806) *
(0.03156796688859639 * 0.651592972722763 +
(0.7669355767878618) * (1 - 0.7669355767878618) * (0 - 0.7669355767878618) * 0.02834747652200631)
= 0.0009611362816286504

The change in weight for Hidden Layer is,
ΔW_{11} = (0.5) * (0.0009611362816286504) * (2)
= 0.0009611362816286504

W_{11} = 0.13436424411240122 + 0.0009611362816286504
= 0.13532538039402986

By doing the same for all weights, we get,

Table 3: Updated value of weights

Weights	Updated Value
W_{11}^1	0.13532538039402986
W_{21}^1	0.8483948732188613
W_{31}^1	0.7642551871174283
W_{12}^1	0.24529471187779883
W_{22}^1	0.4856607732303181
W_{32}^1	0.4446039078579267
W_{11}^2	0.6664081752002493
W_{21}^2	0.8025422455953347
W_{31}^2	0.10964357021853309
W_{12}^2	-0.03598862367938312
W_{22}^2	0.7757555441456759
W_{32}^2	0.3642239655078642

Step 5: Forward Propagate and Repeat

We now, just forward propagated and back propagated for one data-point. We need to do that for all datapoints in our data set, and we need to do these max_epoch number of times.

We have a max_epoch, as it is unlikely we get all errors as zero.

Step 6: Output Visualisation (Additional step)

Here are the weights and decision boundary for our neural network after 1000 epoch.

Table 4: Updated value of weights

Weights	Updated Value
W^1_{11}	-4.165736617684192
W^1_{21}	4.494165872980581
W^1_{31}	2.064357120666933
W^1_{12}	-2.924097575618484
W^1_{22}	3.3401443204537666
W^1_{32}	1.4433168050534642
W^2_{11}	5.429646386040146
W^2_{21}	4.037845396293455
W^2_{31}	-4.237515602055247
W^2_{12}	-5.892609497304456
W^2_{22}	-3.4770990687653325,
W^2_{32}	4.209278121692298

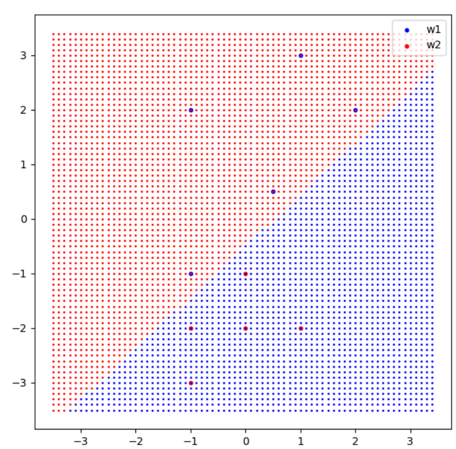


Figure 30: Classification using our neural network

2.4 Pseudo-Code for BPN

```
Begin
Initialize all weights and biases in network;
while terminating condition is not satisfied {
 for each training tuple X in D {
 // Propagate the inputs forward:
     for each input layer unit j {
         O_j = I_j; // output of an input unit is its actual input value
         for each hidden or output layer unit j {
         I_j = sum_i w_ij O_i + θ_j; //compute the net input of unit j with respect to the previous layer, i
         O_j = 1/(1 + e^(-I_j)) ; // compute the output of each unit j
     // Backpropagate the errors:
    for each unit j in the output layer
         Err_j = O_j(1 - O_j)(T_j - O_j); // compute the error
    for each unit j in the hidden layers, from the last to the first hidden layer
         Err_j = O_j(1 - O_j) sum_k Err_k * w_jk; // compute the error with respect to the next higher layer, k
    for each weight w_ij in network {
         Δw_ij = (l) * Err_j * O_i;
        // weight increment
        w_ij = w_ij + Δw_ij; // weight update
    for each bias θ_j in network {
        Δθ_j = (l) * Err_j; // bias increment
        θ_j = θ_j + Δθ_j;
        } // bias update
 } }
```

2.5 Additional Notes

Disadvantages

- Knowledge Representation

  A major disadvantage of neural networks lies in their knowledge representation. Acquired knowledge in the form of a network of units connected by weighted links is difficult for humans to interpret. This factor has motivated research in extracting the knowledge embedded in trained neural networks and in representing that knowledge symbolically. Methods include extracting rules from networks and sensitivity analysis.

  Various algorithms for rule extraction have been proposed. The methods typically impose restrictions regarding procedures used in training the given neural network, the network topology, and the discretization of input values.

  Fully connected networks are difficult to articulate. Hence, often the first step in extracting rules from neural networks is network pruning. This consists of simplifying the network structure by removing weighted links that have the least effect on the trained network. For example, a weighted link may be deleted if such removal does not result in a decrease in the classification accuracy of the network.

  Once the trained network has been pruned, some approaches will then perform link, unit, or activation value clustering. In one method, for example, clustering is used to find the set of common activation values for each hidden unit in a given trained twolayer neural network (Figure 9.6). The combinations of these activation values for each hidden unit are analyzed. Rules are derived relating combinations of activation values with corresponding output unit values. Similarly, the sets of input values and activation

values are studied to derive rules describing the relationship between the input layer and the hidden "layer units"? Finally, the two sets of rules may be combined to form IF-THEN rules. Other algorithms may derive rules of other forms, including M-of-N rules (where M out of a given N conditions in the rule antecedent must be true for the rule consequent to be applied), decision trees with M-of-N tests, fuzzy rules, and finite automata.

Sensitivity analysis is used to assess the impact that a given input variable has on a network output. The input to the variable is varied while the remaining input variables are fixed at some value. Meanwhile, changes in the network output are monitored. The knowledge gained from this analysis form can be represented in rules such as "IF X decreases 5

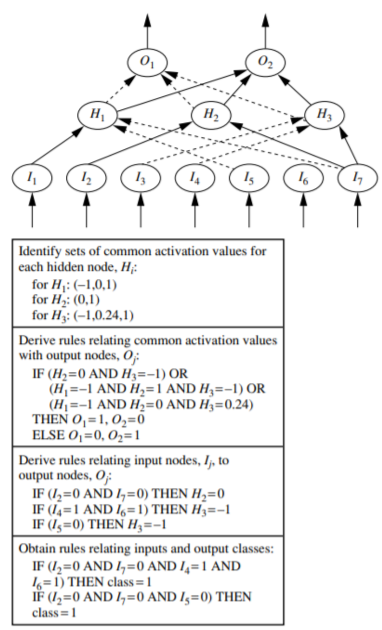


Figure 31: Knowledge representation of a neural network.

- The actual performance of backpropagation on a specific problem is dependent on the input data.
- Backpropagation can be quite sensitive to noisy data
- You need to use the matrix-based approach for backpropagation instead of mini-batch.

3. Support Vector Machine (SVM)

3.1 Introduction

SVM is a method for the classification of both linear and nonlinear data. In a nutshell, an SVM is an algorithm that works as follows. It uses a nonlinear mapping to transform the original training data into a higher dimension. Within this new dimension, it searches for the linear optimal separating hyperplane (i.e., a “decision boundary” separating the tuples of one class from another). With an appropriate nonlinear mapping to a sufficiently high dimension, data from two classes can always be separated by a hyperplane. The SVM finds this hyperplane using support vectors (“essential” training tuples) and margins (defined by the support vectors). Although the training time of even the fastest SVMs can be extremely slow, they are highly accurate, owing to their ability to model complex nonlinear decision boundaries. They are much less prone to overfitting than other methods. The support vectors found also provide a compact description of the learned model. SVMs can be used for numeric prediction as well as classification. They have been applied to a number of areas, including handwritten digit recognition, object recognition, and speaker identification, as well as benchmark time-series prediction tests.

Intuition and Math behind SVM

For linearly separable data, a hyper plane divides the hyperspace in such a way that, points to one side of hyperplane is class 1 and the points on the other side of the hyperplane is class 2.

In mathematical sense, if

$$w^t x + b = 0 \tag{16}$$

is the equation of the hyperplane, then, $w^t x$ + b ¿ 0, all x satisfying this equation, will be of one class and the points satisfying the eq
$w^t x$ + b ¡ 0
will be another class.

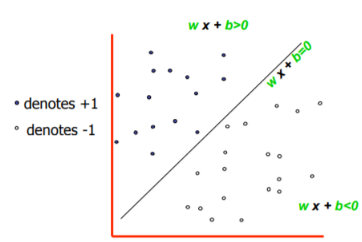


Figure 32: Illustration of how the hyperplane separates points.

We can easily separate the datapoints with many hyperplanes, For example,

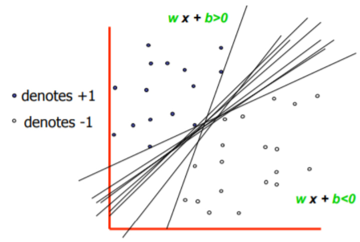


Figure 33: This problem has infinitely many solutions.

But only one of those hyerplane classify the data optimally, and we use svm to idenitify them.

The optimal plane is the one in which the distance between the plane and the class points are maximum. Moreover, the minimum distance between points in class A and the hyperplane and points in class B and the hyperplane must be same, i.e. if the minimum distance of points class A with the hyperplane is greater than that of minimum distance of points in classe B with the hyper plane, then the hyperplane wouldnt be optimum as we tend to classify points more towards class A than class B. For example, consider this one dimensional data set,

Table 5: Example dataset

class	x
class A	1
class A	2
class A	3
class B	5
class B	6
class B	7

The optimum plane (or point) in this case would be x = 4, which is equally spaced between both the classes.

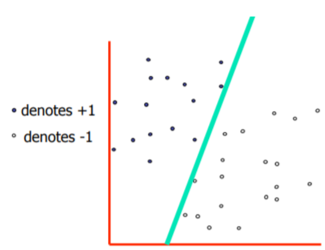


Figure 34: Optimal hyper plane

Support vectors are those planes which run parallel to the decision boundary and pass through the points of both the classes, which have minimum distance with the hyperplane.
The equation of the support vectors is,

$$w^t x + b = \pm 1 \tag{17}$$

The optimal decision boundary is the one in which the margin (distance between the support vectors) is maximum.

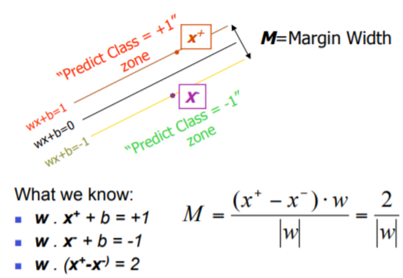


Figure 35: Optimal hyper conditions

From above figure, we can tell that,

$$w^t x + b \geqslant 1 if x \in classA \tag{18}$$
$$w^t x + b \geqslant -1 if x \in classB \tag{19}$$

We can create a label such that, points in class B have label -1 and point in class A has label 1.
Therefore, the above equation becomes,

$$w^t x_i + b \geqslant 1 if y_i \in classA \tag{20}$$
$$w^t x_i + b \geqslant -1 if y_i \in classB \tag{21}$$

We can combine them into one equation,

$$y_i(w^t x_i + b) \geqslant 1 \tag{22}$$

Moreover, we know that margin width,

$$M = \frac{2}{|w|} \tag{23}$$

Maximising M, is same as minimising —w—.
Thus, we minimise,

$$|w| \tag{24}$$
$$and\forall(y_i, x_i) : y_i(w^t x_i + b \geqslant 1) \tag{25}$$

This minimisation problem can be solved by associating Lagrange multiplier with each constraint of this problem.

Cases of SVM

There are three cases of SVM. They are:

- **Hard Margin**

  Here, we try to find a decision boundary that classifies all training points correctly.

  Extremely useful in the case of linearly seperable, but performs poorly in case of non linearly separable and is susceptible to noises.

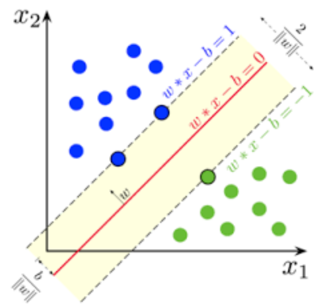


Figure 36: Example of Hard Margin.

- **Soft Margin**

  Here, we try to find a decision boundary that classifies most of the points correctly. We allow some points to be misclassified. The amount of error allowed is set by us.

  This is useful to classify noisy data and some non-linearly separable data.

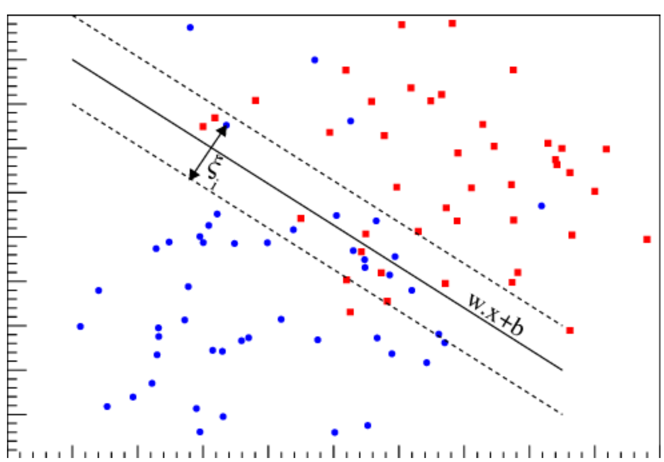


Figure 37: Example of Soft Margin.

- **Kernal**
  Here, we transform the points to a higher dimension (where they are linearly separable) and find a linear hyperplane that classifies the points accurately in that dimension and we plot the contours of that hyperplane on our lower dimension to get a decision boundary.

  This is extremely useful to classify non-linearly seperable data.

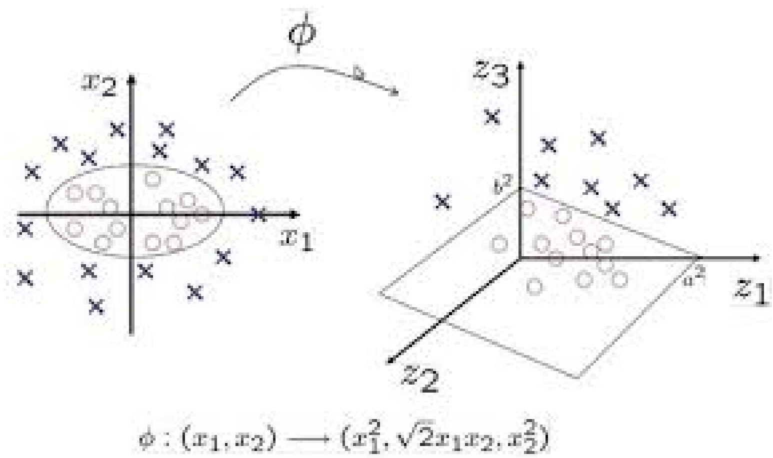


Figure 38: Example of the use of kernal function to transform the input to higher dimension, where they are linearly separable.

3.2 Classifying using SVM with example

Example Dataset

Table 6: Example dataset

class	x	y
class 1 (w1)	2	2
class 1 (w1)	-1	2
class 1 (w1)	1	3
class 1 (w1)	-1	-1
class 1 (w1)	0.5	0.5
class 2 (w2)	-1	-3
class 2 (w2)	0	-1
class 2 (w2)	1	-2
class 2 (w2)	-1	-2
class 2 (w2)	0	-2

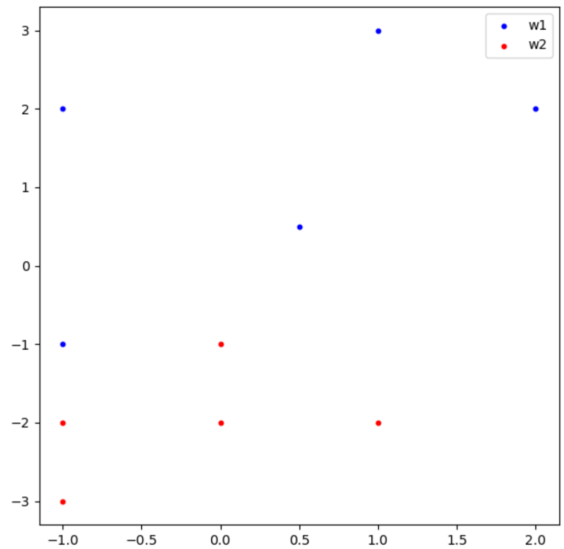


Figure 39: Visualisation of the dataset

Case 1 : Hard Margin

Step 1: Decide the initial weights of the plane.

The Equation of the plane that separates the classes is :

$$w^t x + b = 0 \tag{26}$$

where,

- w is the normal vector of the hyperplane that separates the data. It is represented as a matrix. Hence w^t means transpose of w matrix.
- x is a variable vector.
- b is the bias.

The z matrix would be a matrix where, z_i = 1 if x_i in class 1
z_i = -1 if x_i in class 2

Step 2: Finding the Lagrange Multipliers

We need to find the lagrange multipliers for our hyperplane, which inturn would help us to identify the equation of the hyperplane.

$$L(w, b, \alpha) = \frac{||w||^2}{2} - \sum_{k=1}^{n} \alpha_k (1 - z_k(w^t x_k + b)) \tag{27}$$

where,

- w^t is the transpose of weight matrix, and $||w||$ is the magnitude of the normal vector of the hyper plane.
- b is the bias.
- α is a matrix of all the lagrange multipliers. There are totally n lagrange multiplier, where n is the number of datapoints.
- z_k is the class value of a data point x_k. Note that z_k takes either 1 or -1 and not 1 or 0.

After finding few relation between α and weight matrix and bias, the above equation reduces to,

$$L(\alpha) = \sum_{k=1}^{n} \alpha_k - \frac{1}{2} \sum_{k,j}^{n} \alpha_k \alpha_j z_k z_j x_j^t x_k \tag{28}$$

We should, find the minimum value of L(α), which satisfy the following constraint,

(a) $\sum_{k=1}^{n} z_k \alpha_k = 0$
(b) $\alpha_k \geqslant 0, k = 1, 2, ..., n$

For our example, we got,

Step 3: Finding Weight Vector and Bias

The weight vector is given by,

$$w = \sum_{k=1}^{n} \alpha_i z_i x_i \tag{29}$$

We know that,
$z_i(w^t x_i + b) = 1$
is the equation of the support planes.

And these support planes would pass through the corner points(that are to the other class end) of each class.

Table 7: $\alpha values$

α	Value
α_1	1.26183502 x 10^{-1}
α_2	1.03087027 x 10^{-14}
α_3	2.58278642 x 10^{-15}
α_4	3.45312250 x 10^{0}
α_5	4.21384434 x 10^{1}
α_6	1.64516838 x 10^{-15}
α_7	3.01092476 x 10^{0}
α_8	1.82009356 x 10^{-14}
α_9	9.89765678 x 10^{1}
α_10	8.76371458 x 10^{-15}

Therefore, we need to find the points.

α_i would be negligible for all non - other points/ points through which the support vector wont pass through.

Therefore, with this we can locate the corner point and take avgerage to find avgerage b value.

In our example we get,

w = [-2.0002976 2.00039283] and b = 1.0001476102615467.

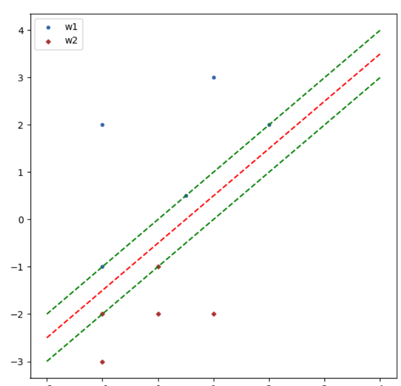


Figure 40: The decision boundary for the example dataset. The red line is the actual decision boundary with green line as the support vectors.

Case 2 : Soft Margin

The steps are same as that of Hard Margin, we only change the equation to allow error.

$$w^t x + b = 1 - \varepsilon \tag{30}$$

$$L(w, b, \varepsilon, \alpha, \gamma) = \frac{||w||^2}{2} + C * \sum_{k=1}^{n} \varepsilon_k - \sum_{k=1}^{n} \alpha_k (z_k (w^t x_k + b) - 1 + \varepsilon_k) - \sum_{i=k}^{n} \gamma_k \varepsilon_k \tag{31}$$

where,

- ε is the error in our classifier.
- $\alpha and \gamma$ are two lagrange multiplier.
- C is a regularisation parameter, which tells us how much error to be present. If C = 0, we get hard margin, and if C is a large value, then we allow a lot of miscalculation as a the error term in $eq(31)$ increases and dominates the equation.

This may seem difficult to solve, but they give the same equation,but differ in constraint.

$$L(\alpha) = \sum_{k=1}^{n} \alpha_k - \frac{1}{2} \sum_{k,j}^{n} \alpha_k \alpha_j z_k z_j x_j^t x_k \tag{32}$$

We should, find the minimum value of L(α), which satisfy the following constraint,

(a) $\sum_{k=1}^{n} z_k \alpha_k = 0$
(b) $C \geqslant \alpha_k \geqslant 0, k = 1, 2, ..., n$

The weight vector is given by,

$$w = \sum_{k=1}^{n} \alpha_i z_i x_i \tag{33}$$

The error or slack is giver by,

$$\varepsilon_i = \frac{\alpha_i}{C} \tag{34}$$

Similarly, corresponding α of non corner point will be negligible. With this we find the corner points.

Same way we substitute the corner points in eq(32) to find bias .

Case 3 : Kernal method

Here we transform the input data to a higher dimension using kernal functions.

$$y = \Phi(x) \tag{35}$$

where,

- y is the transformed data.
- x is the input data points.
- $\phi(.)$ is a transformation or kernal function.

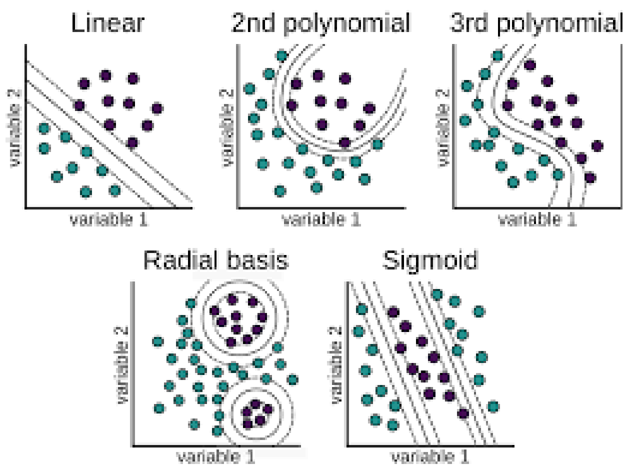


Figure 41: Different kernal functions

After transforming the input data, follow hard margin algorithm to get the equation of plane in the higher dimension.

To make the computation at higher dimension easier, we use a special property of kernal functions. i.e. the dot product of the function is conserved throughout the dimensions.

The contours of that plane on the input dimension would be our decision tree.

3.3 Pseudo-Code SVM

Hard Margin

```
Begin
Initialize all weights to 0;
for each training tuple X in D {
z_i = classlabel(X)
}
n = len(X)
L(α) = \sum_{k=1}^{n} α_k − \frac{1}{2} \sum_{k,j}^{n} α_k α_j z_k z_j x_j^t x_k
α = α : L(α) is maximum,
          α_i ⩾ 0, and
          \sum_{i=1}^{n} α_i z_i = 0

Weights = \sum_{i=1}^{n} z_i α_i x_i

for i in len(α){
if(alpha_i ⩾ 0.001)
  b = \dfrac{1}{z_i} − w^t x_i
  add b to the set bias
}
bias = bias.mean()
```

Soft Margin SVM

```
Begin
Initialize all weights to 0;
for each training tuple x in X {
z_i = classlabel(x)
}
initialise the desired value of c.
n = len(X)
L(α) = \sum_{k=1}^{n} α_k − \frac{1}{2} \sum_{k,j}^{n} α_k α_j z_k z_j x_j^t x_k
α = α : L(α) is maximum,
          c ⩾ alpha_i ⩾ 0, and
          \sum_{i=1}^{n} α_i z_i = 0

Weights = \sum_{i=1}^{n} z_i α_i x_i
for i in len(X){
 ε_i = \dfrac{α_i}{c}
}
for i in len(X){
if(alpha_i ⩾ 0.001)
  b = \dfrac{1−ε_i}{z_i} − w^t x_i
  add b to the set bias
}
bias = bias.mean()
```

kernal trick SVM

```
Begin
Initialize all weights to 0;
for each training tuple x in X {
z_i = classlabel(x)
}
```

```
Y = ψ(X) //transforming the input into higher dimensions.

weight, bias = HardMargin(Y)
```

3.4 Additional Notes

More about Kernal functions

Consider the following example. A 3-D input vector X = (x1, x2, x3) is mapped into a 6-D space, Z, using the mappings ϕ_1(X) = x1, ϕ_2(X) = x2, ϕ_3(X) = x3, ϕ_4(X) = $(x1)^2$, ϕ_5(X) = x1.x2, and ϕ_6(X) = x1.x3. A decision hyperplane in the new space is

$$d(Z) = W^t Z + b \tag{36}$$

where, W and Z are vectors. This is linear. We solve for W and b and then substitute back so that the linear decision hyperplane in the new (Z) space corresponds to a nonlinear second-order polynomial in the original 3-D input space:

But there are some problems. First, how do we choose the nonlinear mapping to a higher dimensional space? Second, the computation involved will be costly. Refer to

Given the test tuple, we have to compute its dot product with every one of the support vectors. In training, we have to compute a similar dot product several times in order to find the MMH. This is especially expensive. Hence, the dot product computation required is very heavy and costly.

Luckily, we can use another math trick. It so happens that in solving the quadratic optimization problem of the linear SVM (i.e., when searching for a linear SVM in the new higher dimensional space), the training tuples appear only in the form of dot products, ϕ(Xi)$\cdot$ ϕ(Xj), where ϕ(X) is simply the nonlinear mapping function applied to transform the training tuples. Instead of computing the dot product on the transformed data tuples, it turns out that it is mathematically equivalent to instead apply a kernel function, K(Xi, Xj), to the original input data. That is,

$$K(Xi, Xj) = \phi(Xi).\phi(Xj) \tag{37}$$

In other words, everywhere that ϕ(Xi)$\cdot$ ϕ(Xj) appears in the training algorithm, we can replace it with K(Xi ,Xj). In this way, all calculations are made in the original input space, which is of potentially much lower dimensionality! We can safely avoid the mapping—it turns out that we don't even have to know what the mapping is! We will talk more later about what kinds of functions can be used as kernel functions for this problem.

After applying this trick, we can then proceed to find a maximal separating hyperplane.

kernel functions that could be used to replace the dot product scenario just described,

$$Polynomialkernelofdegreeh : K(Xi, Xj) = (Xi\Delta Xj + 1)^h \tag{38}$$

$$Gaussianradialbasisfunctionkernel : K(Xi, Xj) = e^{-}\frac{||kXiXjk||^2}{2\sigma^2} \tag{39}$$

$$Sigmoidkernel : K(Xi, Xj) = tanh(\kappa XiXj - \delta) \tag{40}$$