Module 4 – Artificial Neural Networks

Objectives

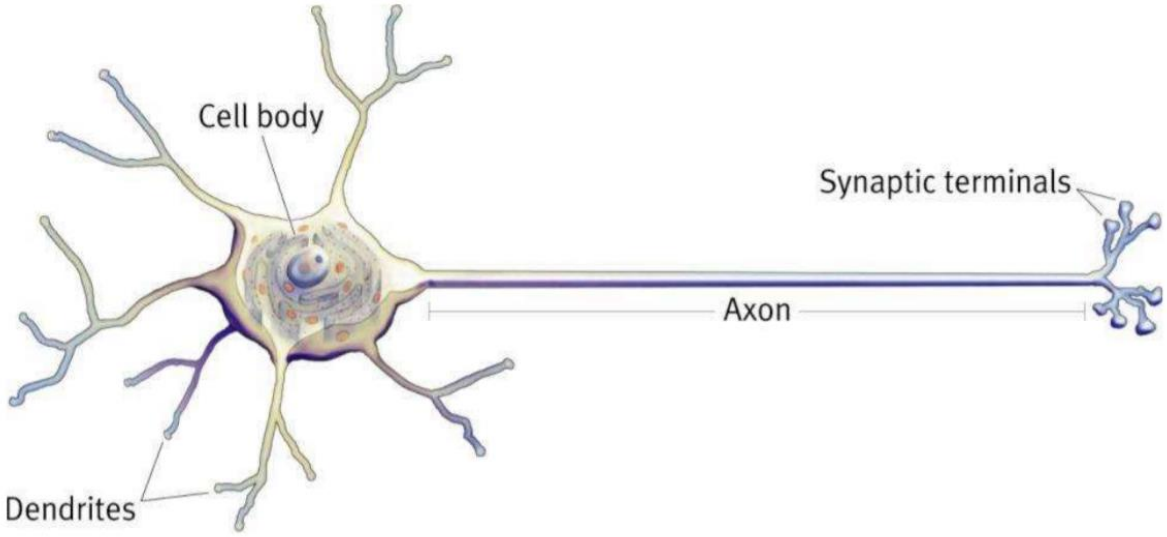
At the end of this module, you should be able to:

- Understand Artificial Neural Networks
- Understand the need of Activation Functions
- Neural Network Architecture
- Types of Neural Networks
- Applications, Advantages and Limitations

Introduction to Neurons and Artificial Neural Networks

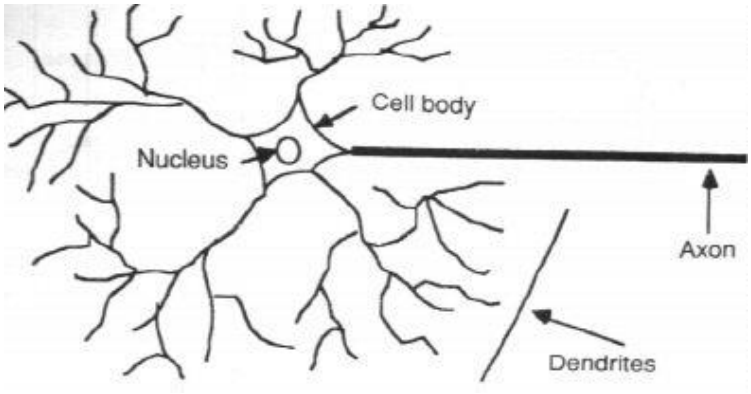
Artificial Neural Networks are inspired from the working of human brain , The human brain is complex computational system and makes use of more than 86 billion nerve cells called neurons. All the Neurons are connected to other thousand cells by Axons.

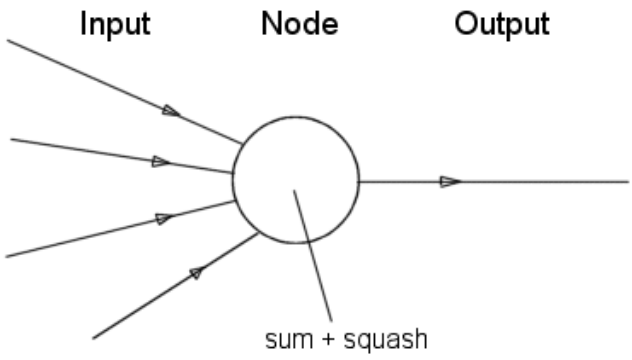
Neurons carry information from input organs to billions of neurons inside human brain and then after processing passes the actions to the output neurons so that the body can take action.



Above is the image of a single neuron which has axon as output part, dendrites as input part.

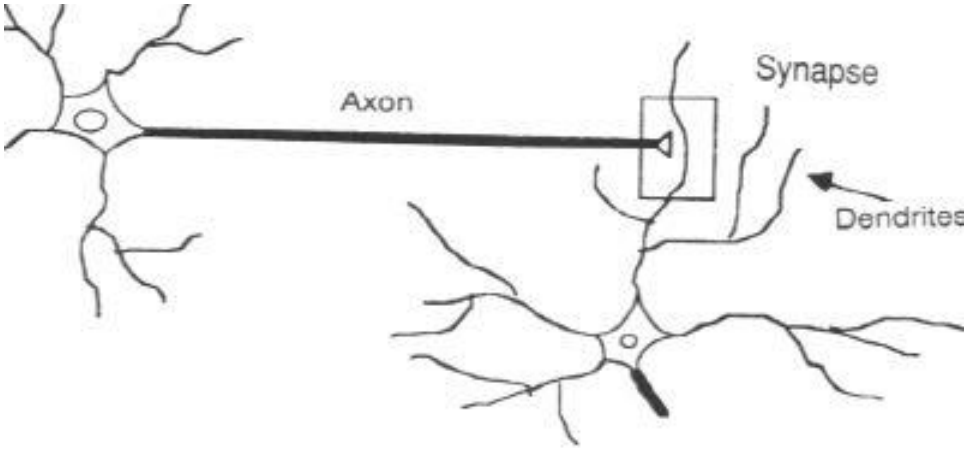
Similar to the natural neuron, we use mathematical neuron or node in Artificial Neural Networks.





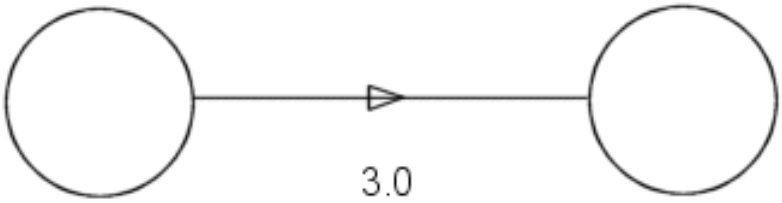
Similar to natural neuron the mathematical neuron also takes some inputs and does some mathematical process and output the value.

Artificial Neural Networks are composed of multiple nodes, which copy the way biological neurons of human brain function. Axon is the output portion of every neuron and dendrites is for taking input inside neuron, axon of neuron passes information to dendrite of another neuron. The nodes can take input data and perform simple operations on the data. The result of these operations is passed to other neurons. The output at each node is basically known as its activation or node value.



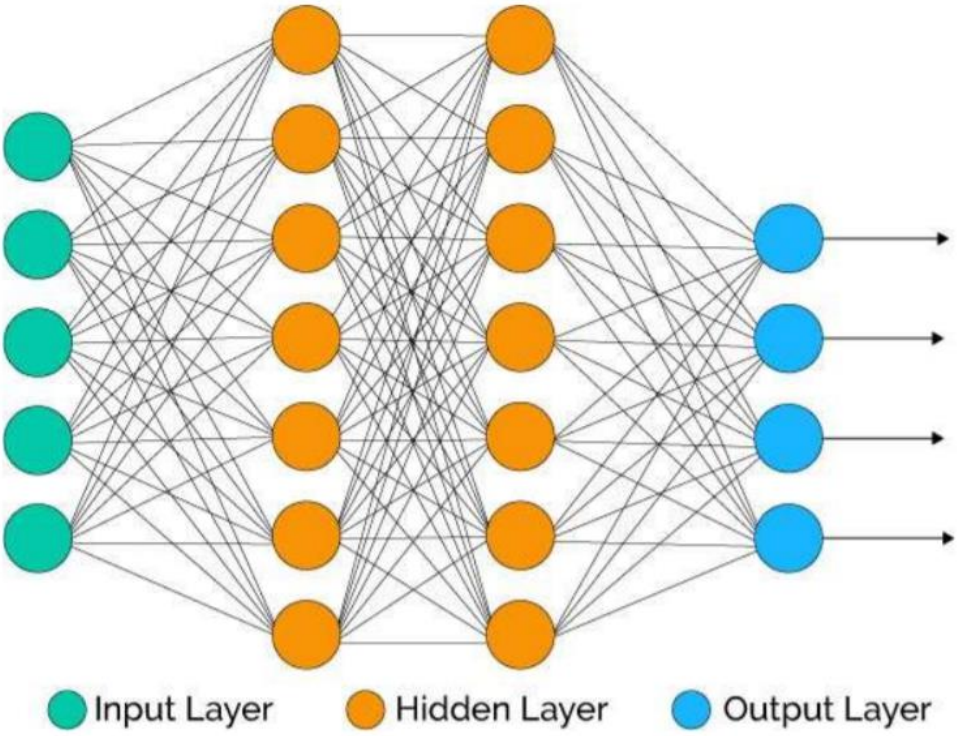
Above is how the two natural neuron connects and pass information from one to other.

Similarly mathematical nodes pass information in forward direction.



The two mathematical neurons combines by weight which shows weightage of that neuron for making certain decisions.

The following illustration shows a simple ANN −



There are three types of layers in a general Neural Network – input layer, hidden layer and the output layer. Neural Network without hidden layer can be considered as a linear Regression model or a linear model. As much as hidden layers are added to the neural network more of non linear pattern are extracted from the data. More than required number of hidden layer sometimes results into the problem of overfitting.

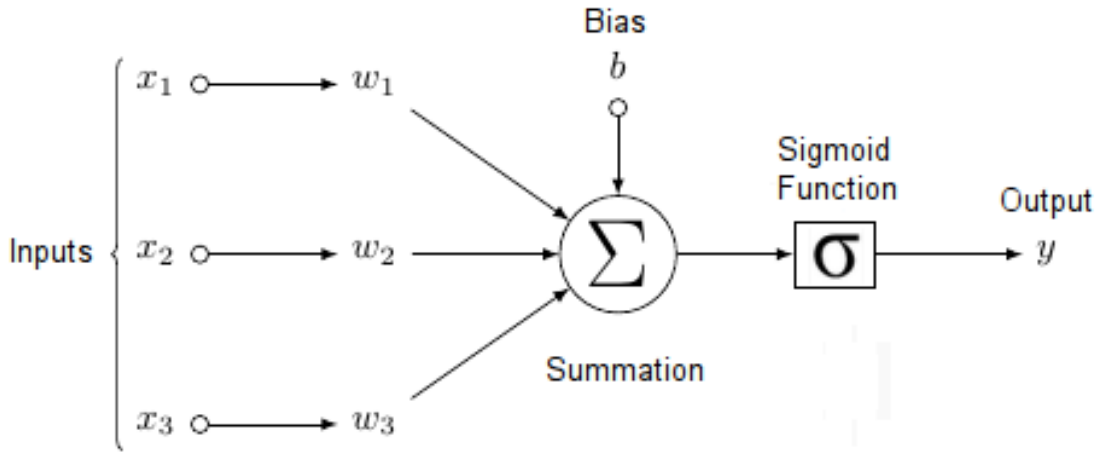
The choice of number of hidden layers and the number of neurons in those hidden layers as to be made by the developer.

In the topology diagrams shown, each arrow represents a connection between two neurons and indicates the process for the flow of information. Each connection between the two neurons has a weight, an integer number that controls the signal between the two neurons which further gets updated while network is exposed to training data for training of the network.

If the neural network provides desired output, there is no need to adjust the weights further. You can also use the tolerance as a stopping criterion. Otherwise if the error is still there or sufficient performance is not achieved then further updating of weights has to be done using some solver. E.g. Gradient Descent Algorithm.

Single layer perceptron

Single Layer perceptron can be defined as a simplest type of Neural Network which has a single layer of weights connecting the inputs and output. In this way, it can be considered the simplest kind of feed-forward network. In a feed forward network, the information always moves in one direction; it never goes backwards.



2 rules of Neural Networks

1. Input layer neurons are not computational neurons, the job of input layer of neurons is to load the features in the neural network.

2. At every computational neuron i.e. every neuron at hidden layer and the output layer two operations occur –
   - Weighted sum of inputs and weights
   - Applying activation function on weighted sum to get the final output.

$$z = x_1 w_1 + x_2 w_2 + x_3 w_3 + b$$

$$yhat = f(x)$$

where yhat is the predicted output

Why is bias added in the neural network ?

The Bias (w) is very much similar to the intercept term in linear regression.

The work of bias in a neural network is to improve the accuracy of prediction by shifting the decision boundary along Y axis.

Bias in a Neural Network helps in making the weight obtain smaller value which makes the network to be less sensitive to the input data.

What is an activation function?

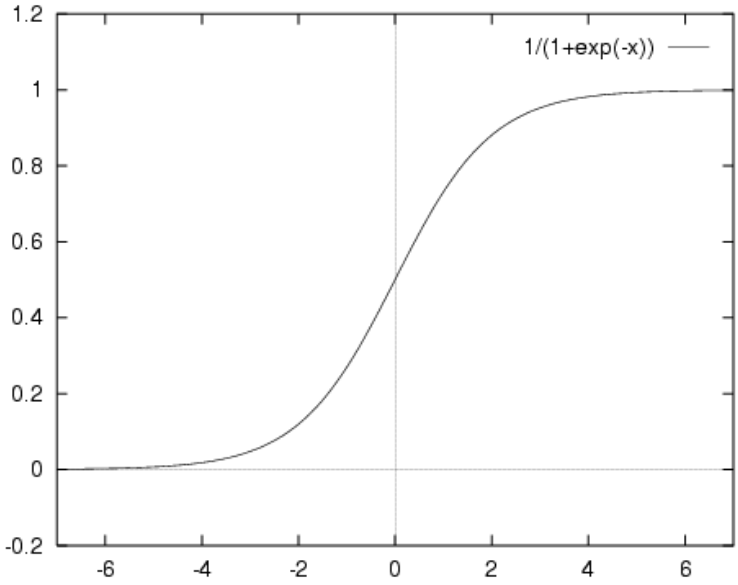
Activation Function takes the sum of weighted input (w1*x1 + w2*x2 + w3*x3 + 1*b) as an argument and return the output of the neuron.

The activation function is mostly used to make a non-linear transformation which allows us to fit nonlinear hypotheses or to estimate the complex functions. There are multiple activation functions, like: “Sigmoid”, “Tanh”, ReLu and many other.

Sigmoid Function

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

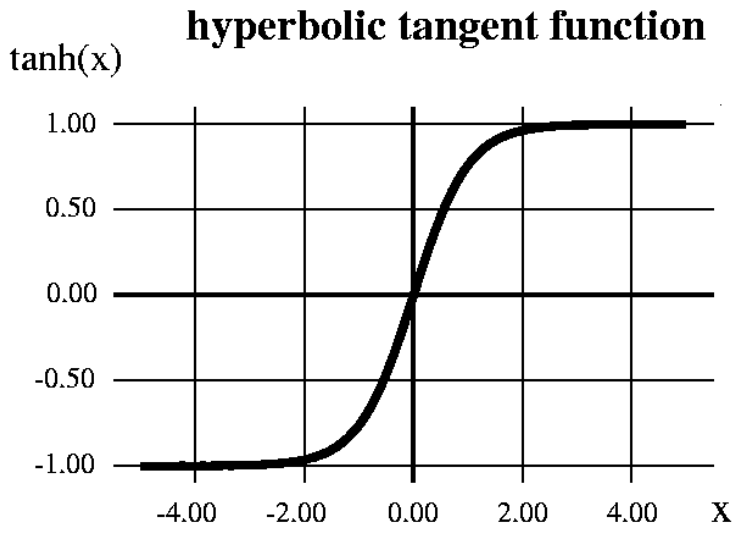
- The function is **differentiable**. That means, we can find the slope of the sigmoid curve at any two points.
- The function is **monotonic** but function's derivative is not.
- The logistic sigmoid function can cause a neural network to get stuck at the training time.
- The **softmax function** is a more generalized logistic activation function which is used for multiclass classification.



Hyperbolic Tangent function

$$\boldsymbol{tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}}$$

- The function is **differentiable**.
- The function is **monotonic** while its **derivative is not**.
- The tanh function is mainly used classification between two classes.
- output is zero centered because its range in between -1 to 1 i.e -1 < output < 1 . Hence optimization is *easier* in this method hence in practice it is always preferred over Sigmoid function .



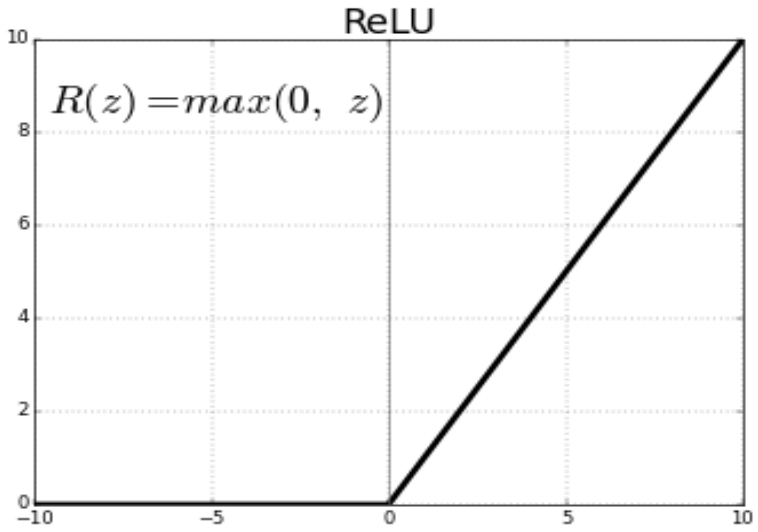
ReLu- Rectified Linear units

$$f(x) = \max(x, 0)$$

The function and its derivative **both are monotonic**.

It should only be used within Hidden layers of a Neural Network Model.

Hence for output layers we should use a **Softmax** function for a Classification problem to compute the probabilites for the classes , and for a regression problem it should simply use a **linear** function.

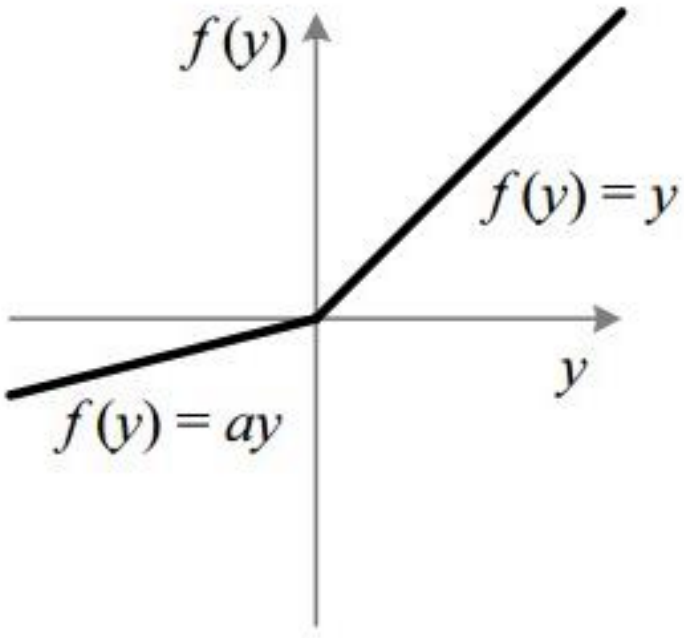


Leaky ReLu

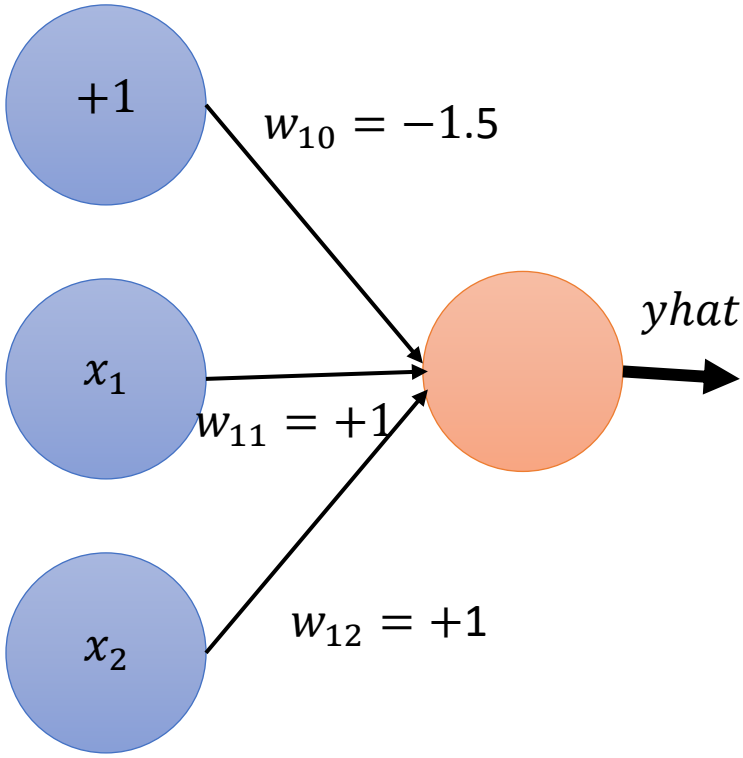
$$f(x) = \begin{cases} x & if\ x < 0 \\ 0.01x & otherwise \end{cases}$$

- The problem with ReLu is that some gradients can be fragile during training and can die. It can cause a weight update which will makes it never activate on any data point again. Simply saying that ReLu could result in Dead Neurons.

- To fix the problem of dying neurons another modification was introduced called Leaky ReLu. It introduces a small slope to keep the updates alive.

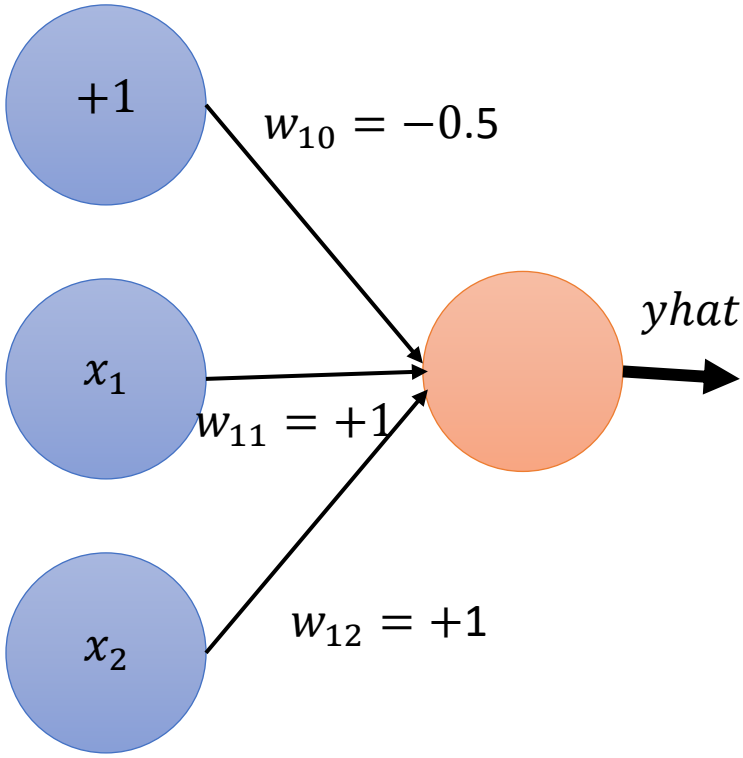


Example – AND Function



x_1	x_2	Yhat=F(x,w)
0	0	yhat=g{1*-1.5+0*1+0*1}=g{-1.5}=0
0	1	yhat=g{1*-1.5+0*1+1*1}=g{-0.5}=0
1	0	yhat=g{1*-1.5+1*1+0*1}=g{-0.5}=0
1	1	yhat=g{1*-1.5+1*1+1*1}=g{+0.5}=1

OR Function



x_1	x_2	Yhat=F(x,w)
0	0	yhat=g{1*-0.5+0*1+0*1}=g{-0.5}=0
0	1	yhat=g{1*-0.5+0*1+1*1}=g{+0.5}=1
1	0	yhat=g{1*-0.5+1*1+0*1}=g{+0.5}=1
1	1	yhat=g{1*-0.5+1*1+1*1}=g{+1.5}=1

Decision Boundaries in Two Dimensions

For simple logic gate problems, it is easy to visualize what the neural network is doing. It is forming ***decision boundaries*** between classes. Remember, the network output is:

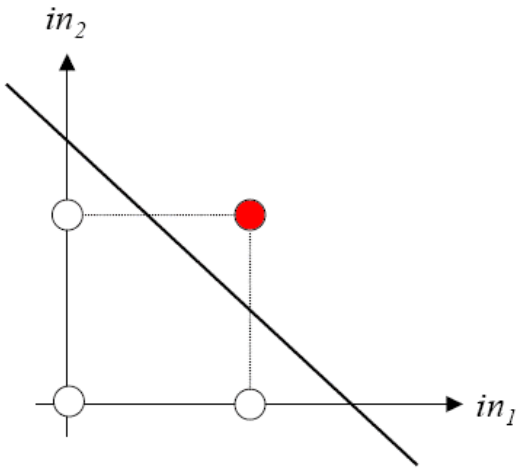
The decision boundary (between *out* = 0 and *out* = 1) is at

$$w_1x_1 + w_2x_2 - b = 0$$

The **decision boundary** is the line that separates the area where y = 0 and where y = 1. It is created by our current model.

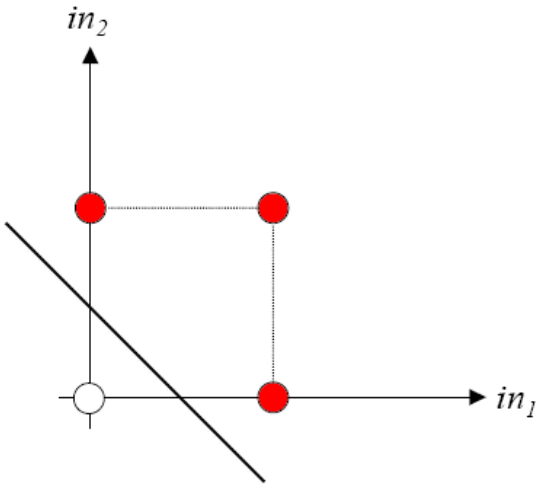
AND

$w_1 = 1,\ w_2 = 1,\ \theta = -1.5$

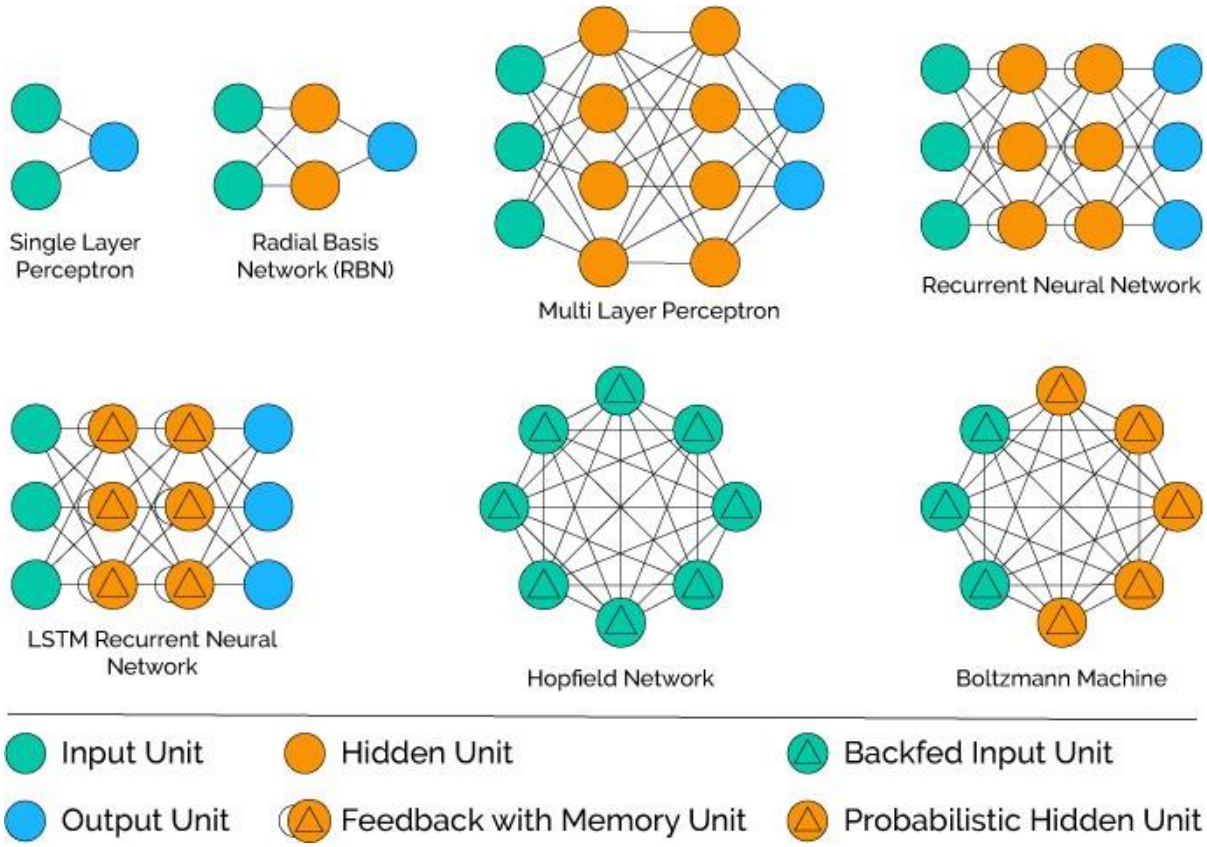


OR

$w_1 = 1,\ w_2 = 1,\ \theta = -0.5$

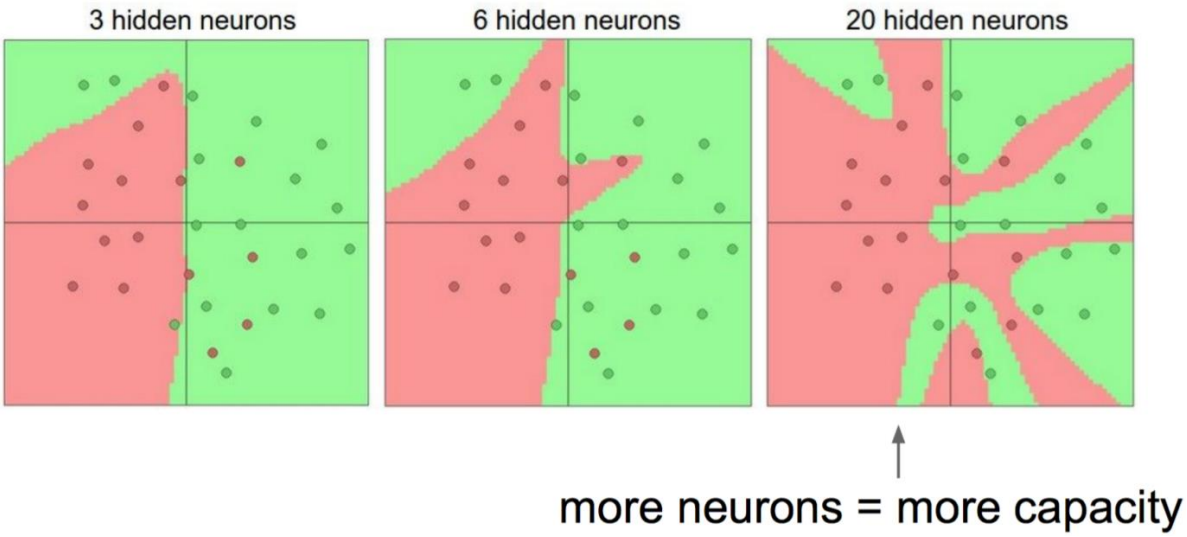


Neural Network Architecture



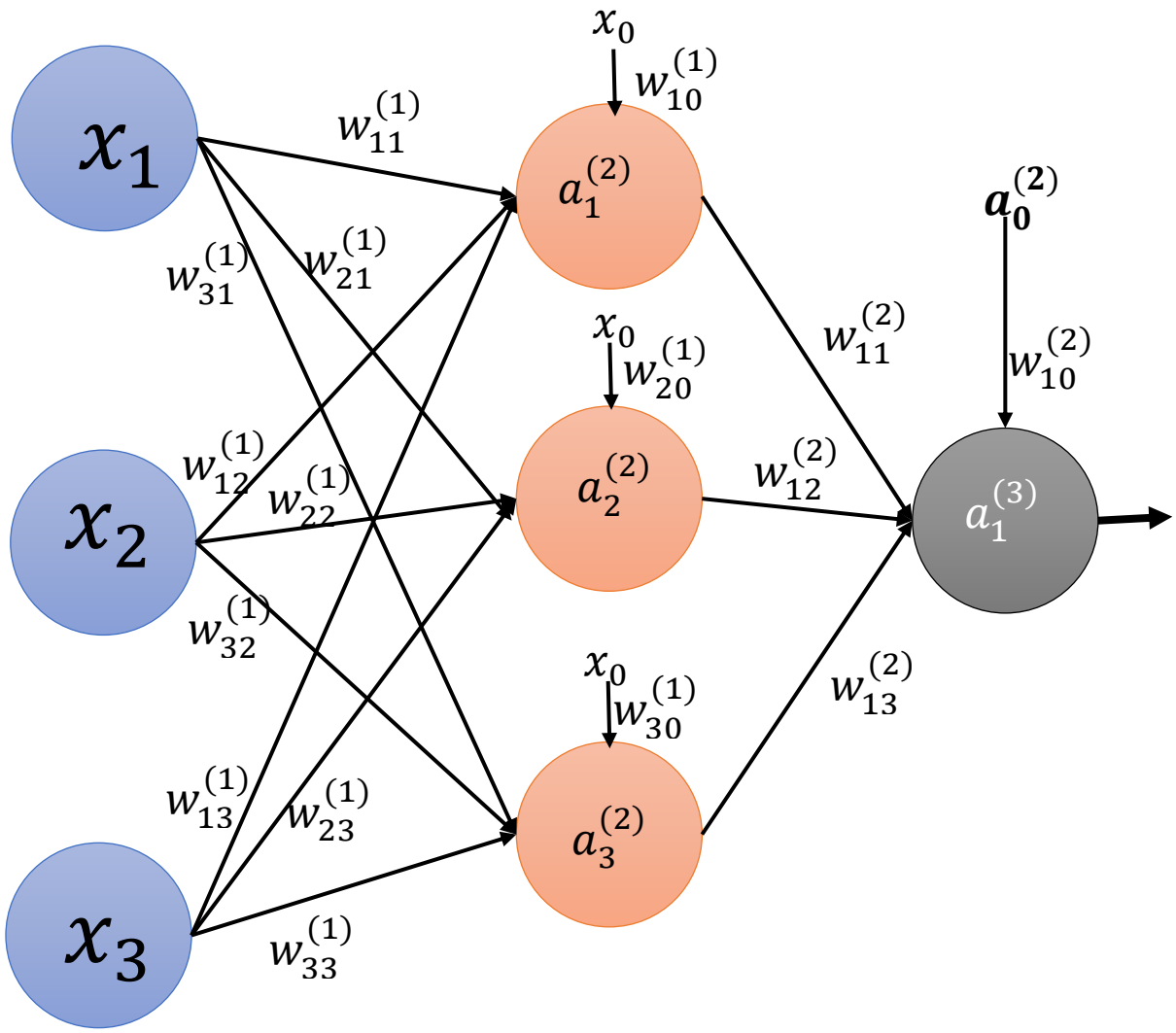
Above are couple of Neural Network Architectures which are generally used for different purposes. Recurrent Neural Networks and LTMS are used for Handling Time series application and also for Text Based Applications as well.

How many hidden Neurons?



More the hidden layer neurons generally are used to introduce more non-linearity into the neural network pattern detection.

Mathematical Modelling of Multilayer Neural Network



$$a_1^{(2)} = g\{w_{10}^{(1)}x_0 + w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3\}$$

$$a_2^{(2)} = g\{w_{20}^{(1)}x_0 + w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3\}$$

$$a_3^{(2)} = g\{w_{30}^{(1)}x_0 + w_{31}^{(1)}x_1 + w_{32}^{(1)}x_2 + w_{33}^{(1)}x_3\}$$

$$yhat = a_1^{(3)} = g\{w_{10}^{(2)}a_0^{(2)} + w_{11}^{(2)}a_1^{(2)} + w_{12}^{(2)}a_2^{(2)} + w_{13}^{(2)}a_3^{(2)}\}$$

Learning Rate lr

Learning Rate lr controls how big step we take while updating our parameter w.

- If lr is too small, gradient descent can be slow.
- If lr is too big, gradient descent can overshoot the minimum, it may fail to converge

Variant of Gradient Descent algorithms

Vanilla Gradient Descent

Vanilla Gradient Descent or pure Gradient Descent is the simplest form of gradient descent technique. The feature of VGD is that we take small steps in the direction of the minima by taking gradient of the cost function.

```
parameters = parameters - learning_rate * gradient_of_parameters
```

Gradient Descent with Momentum

Here, we tweak the above algorithm in such a way that we pay heed to the prior step before taking the next step.

In this case the update pattern is the same as that of vanilla gradient descent. But we introduce a new term called velocity, which considers the previous update and a constant, the constant here is called as momentum.

```
velocity = previous_update * momentum


parameter = parameter + velocity – learning_rate * gradient
```

ADAGRAD

ADAGRAD version of Gradient Descent makes use of adaptive technique for learning rate updating.

In case of ADAGRAD, on the basis of how the gradient has been changing for all the previous iterations we try to change the learning rate along with it.

```
grad_component = previous_grad_component + (gradient * gradient)


rate_change = square_root(grad_component) + epsilon


adapted_learning_rate = learning_rate * rate_change


parameter = parameter – adapted_learning_rate * gradient
```

ADAM

There is one more adaptive technique which builds on ADAGRAD and further reduces it downside called as ADAM.

We can also call it as momentum + ADAGRAD.

```
adapted_gradient = previous_gradient +((gradient – previous_gradient)*(1– beta1))

gradient_component = (gradient_change – previous_learning_rate)

adapted_learning_rate= previous_learning_rate +(gradient_component * (1 – beta2))

parameter = parameter – adapted_learning_rate * adapted_gradient
```

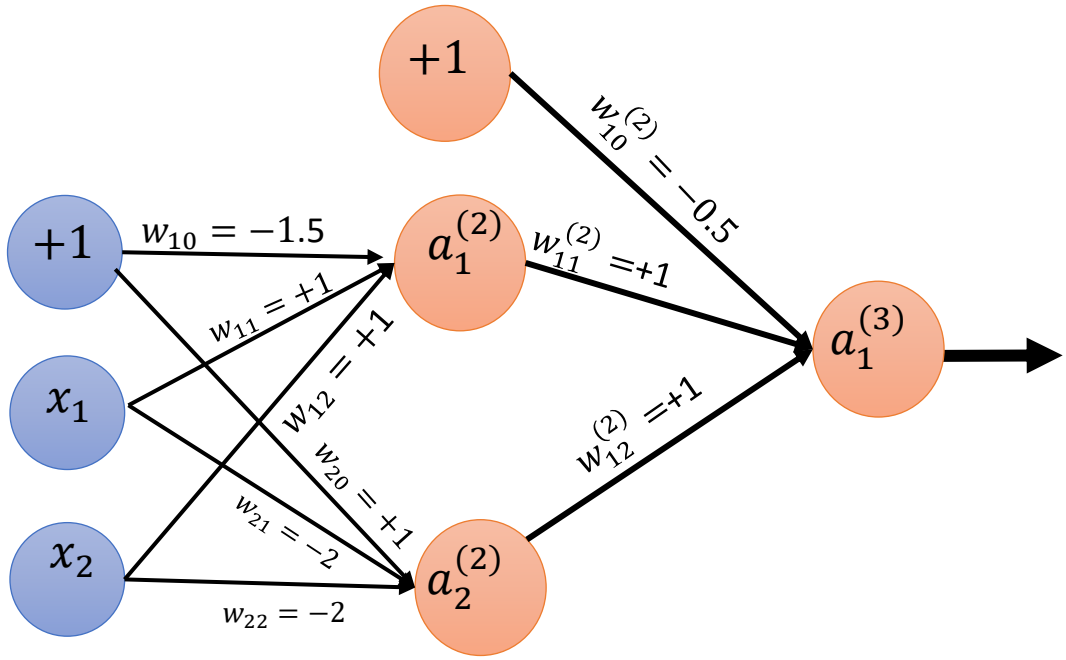
Here beta1 and beta2 are constants to keep changes in gradient and learning rate in check

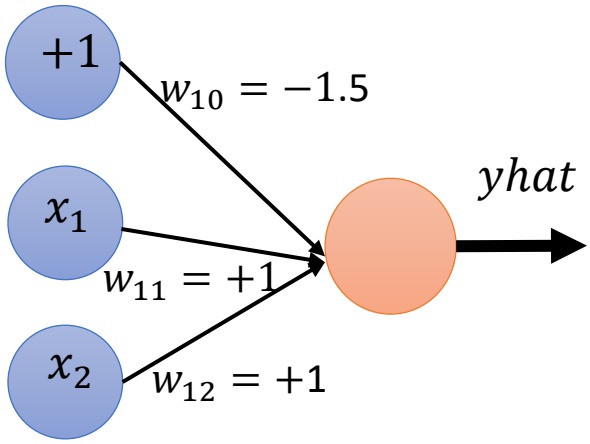
There are also second order differentiation method like l-BFGS.

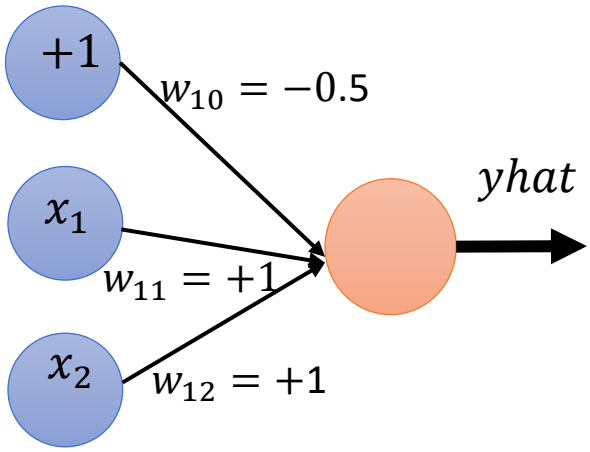
When applying gradient descent, you can look at these points which might be helpful in circumventing the problem:

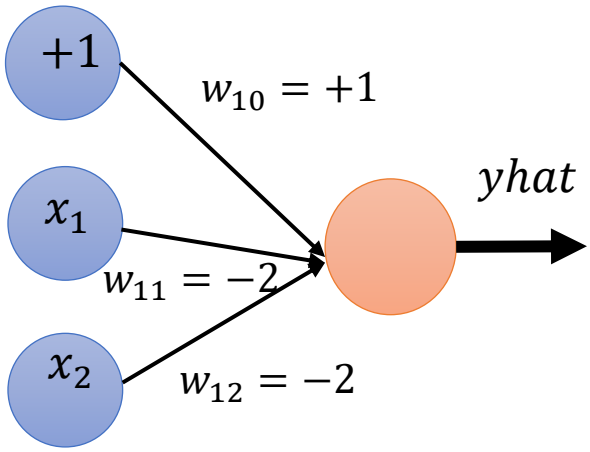
- Error rates – while training process we should always check the training and testing error after specific iterations and make sure both errors decreases.
- Gradient flow in hidden layers – we should also check if the network doesn't show a vanishing gradient problem or exploding gradient problem during the training process.
- Learning rate – which you should check when using adaptive techniques.

Example: XOR Function

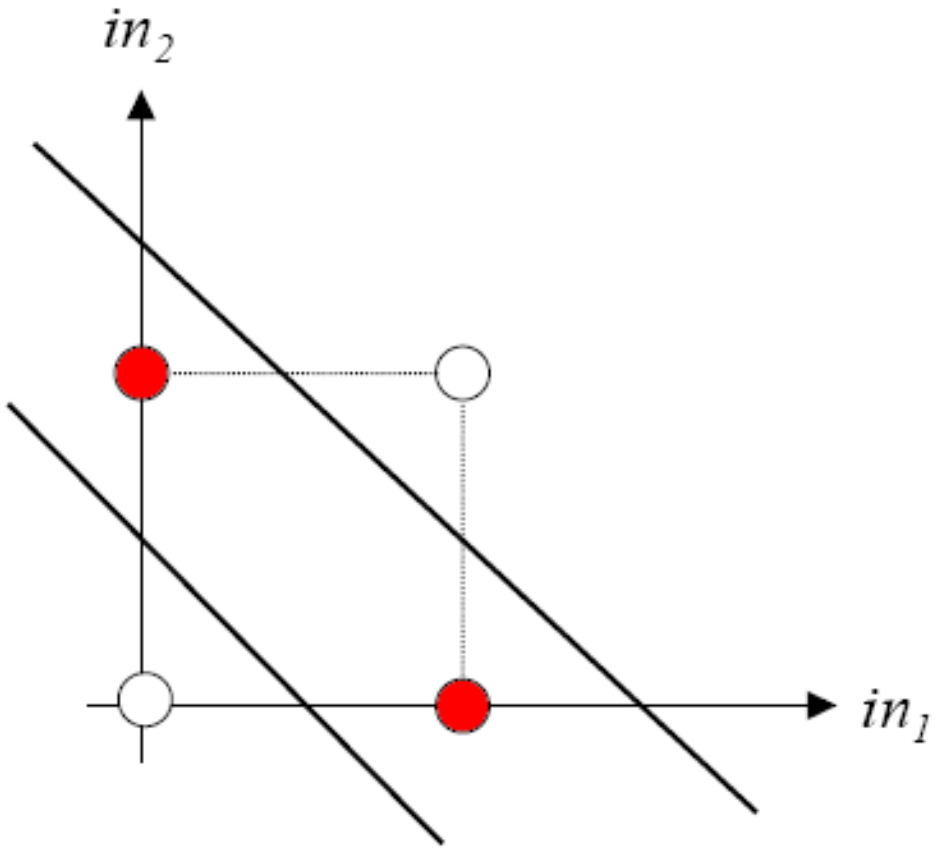








x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	$a_1^{(3)}$
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1



Decision Hyperplanes and Linear Separability

If we have two inputs, then the weights define a decision boundary that is a one dimensional straight line in the two dimensional ***input space*** of possible input values

If we have *n* inputs, the weights define a decision boundary that is an *n-1* dimensional ***hyperplane*** in the *n* dimensional input space:

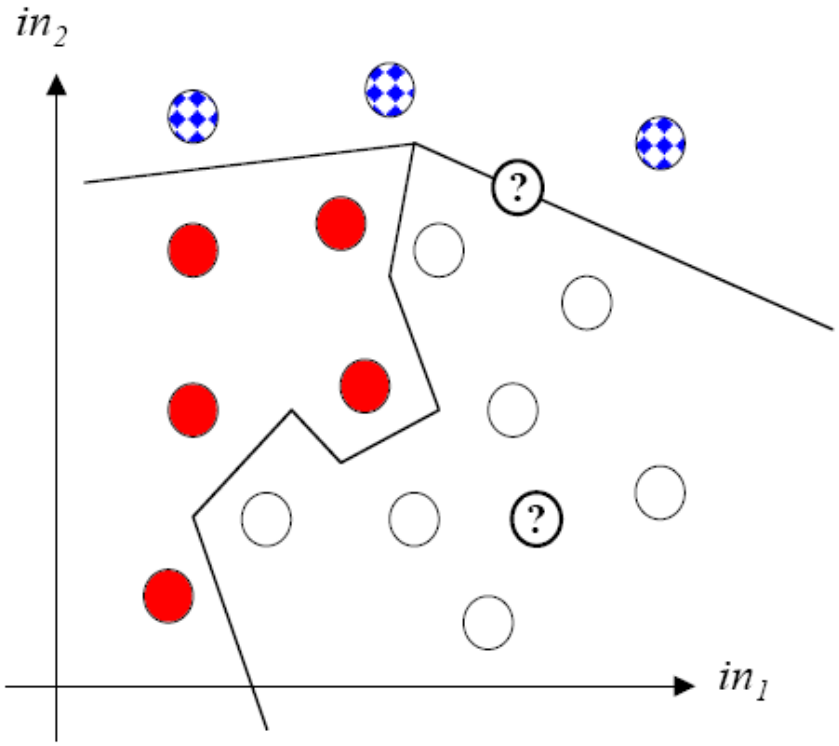
$w_1 in_1 + w_2 in_2 + \ldots + w_n in_n - \vartheta = 0$

This hyperplane is clearly still linear (i.e. straight/flat) and can still only divide the space into two regions. We still need more complex transfer functions, or more complex networks, to deal with XOR type problems

Problems with input patterns which can be classified using a single hyperplane are said to be ***linearly separable***. Problems (such as XOR) which cannot be classified in this way are said to be ***non-linearly separable***.

Generally, we will want to deal with input patterns that are not binary, and expect our neural networks to form complex decision boundaries

We may also wish to classify inputs into many classes (such as the three shown here)



Back Propagation Algorithm

Phase 1: Propagation

Each propagation involves the following steps:

1. Forward propagation of a training pattern's input through the neural network in order to generate the propagation's output activations.

2. Backward propagation of the propagation's output activations through the neural network using the training pattern target in order to generate the deltas of all output and hidden neurons.

Phase 2: Weight update

For each weight-synapse follow the following steps:

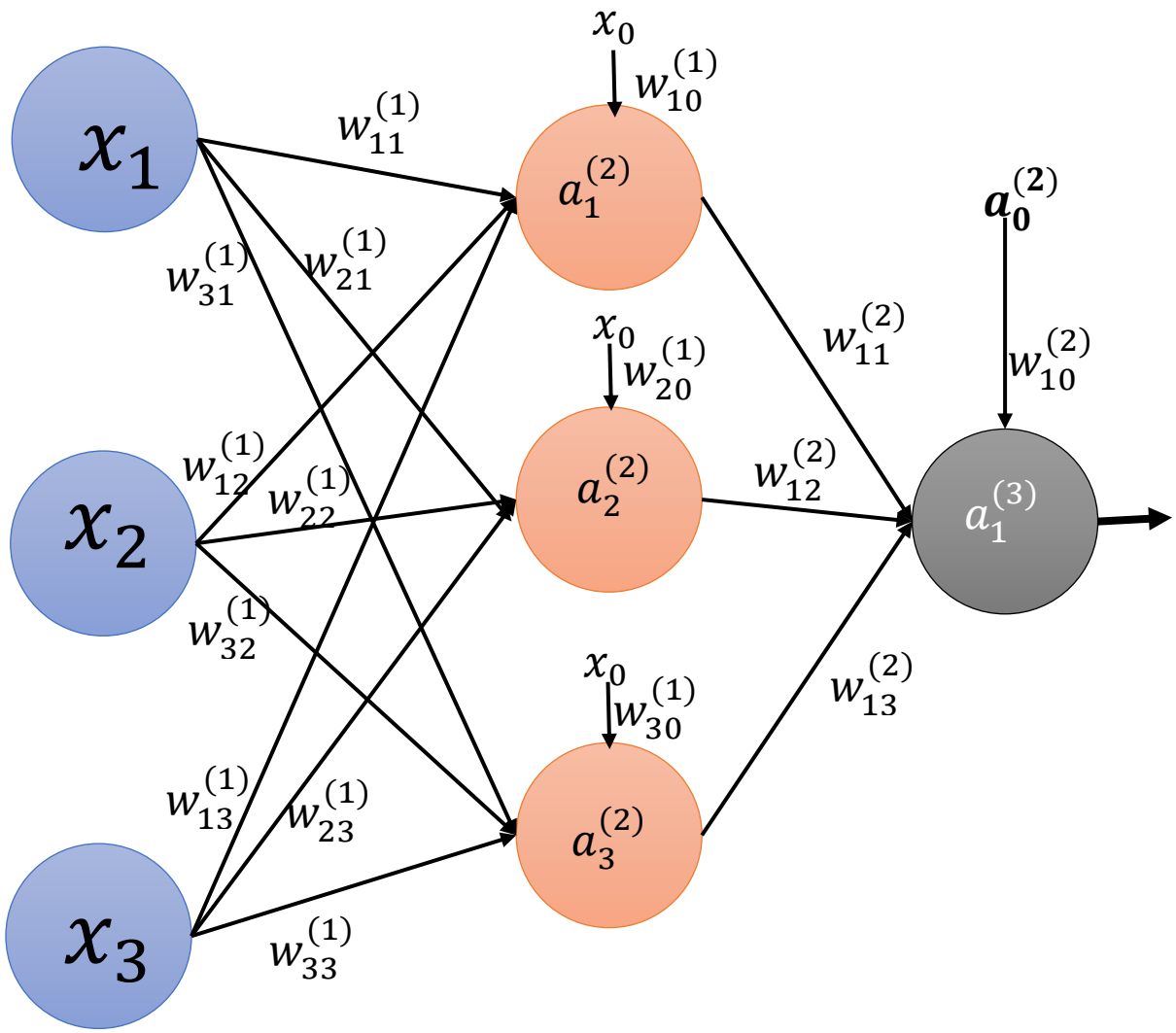
1. Multiply its output delta and input activation to get the gradient of the weight.

2. Subtract a ratio (percentage) of the gradient from the weight.

This ratio (percentage) influences the speed and quality of learning; it is called the learning rate. The greater the ratio, the faster the neuron trains; the lower the ratio, the more accurate the training is. The sign of the gradient of a weight indicates where the error is increasing, this is why the weight must be updated in the opposite direction.

Repeat phase 1 and 2 until the performance of the network is satisfactory.

Consider the following Neural Network



Step 1 Cost Function

$$(Error)E = \frac{1}{2}(t - yhat)^2$$

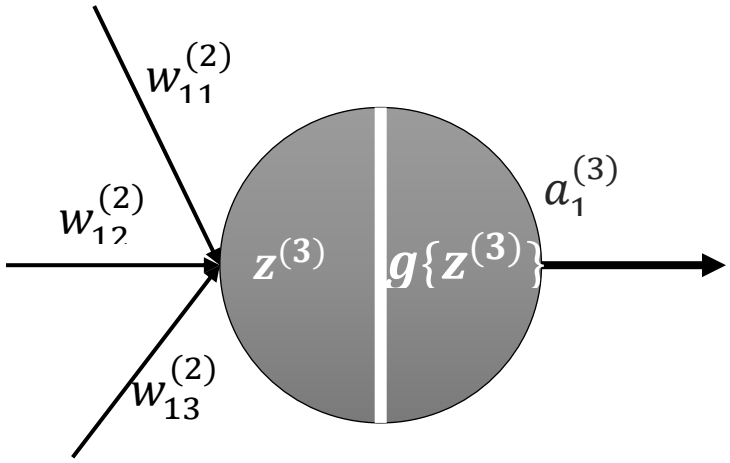
E = mean square error

t = target value (Actual Value)

yhat = predicted output of the output neuron

Step 2 finding the derivative of the error with respect to weights

$$\frac{\partial E}{\partial w_{ij}^{(l)}} = Gradient\ of\ error\ w.r.t.\ weight$$



$$\boldsymbol{E = \frac{1}{2}(t - yhat)^2 = \frac{1}{2}(t - a_1^{(3)})^2}$$

$$\boldsymbol{\frac{\partial E}{\partial w_{ij}^{(l)}} = \frac{\partial E}{\partial a_1^{(3)}} \frac{\partial a_1^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial w_{ij}^{(l)}}}$$

$$\boldsymbol{\frac{\partial z^{(3)}}{\partial w_{ij}^{(l)}} = \frac{\partial}{\partial w_{ij}^{(l)}} \left(\sum_{k=1}^{n} (w_{kj}^{(l)} * a_k^{(l)}) \right) = a_k^{(l)}}$$

$$\frac{\partial a_1^{(3)}}{\partial z^{(3)}} = \frac{\partial g\{z^{(3)}\}}{\partial z^{(3)}} = g\{z^{(3)}\}\left(1 - g\{z^{(3)}\}\right)$$

$$= a_1^{(3)}(1-a_1^{(3)})$$

$$\frac{\partial E}{\partial a_1^{(3)}} = \frac{\partial}{\partial a_1^{(3)}}\frac{1}{2}(t-y)^2 = \frac{\partial}{\partial a_1^{(3)}}\frac{1}{2}(t-a_1^{(3)})^2 = (a_1^{(3)} - \mathrm{t})$$

putting it altogether,

$$\frac{\partial E}{\partial w_{ij}^{(l)}} = a_1^{(3)}(1- a_1^{(3)}) *(a_1^{(3)} - \mathrm{t})* a_k^{(2)}$$

$$\frac{\partial E}{\partial w_{ij}^{(l)}} = \delta^{(3)} * a_k^{(2)}$$

$$where\ \delta^{(3)} = a_1^{(3)}(1-a_1^{(3)}) *(a_1^{(3)} - \mathrm{t})$$

Backpropagation Steps

Step 1 Operate Feed Forward Network to find the y actual value of network and calculate the Error term on output neuron.

Step 2 Calculate δ for output neuron and hidden neurons, δ will not be calculated for input neuron

$$\boldsymbol{\delta^{(3)}} = \boldsymbol{a_1^{(3)}(1-a_1^{(3)}) * \left(a_1^{(3)} - \mathrm{t}\right)}$$

$$\boldsymbol{\delta^{(2)}} = \boldsymbol{\delta^{(3)} w_{kj}^{(2)} a_1^{(2)}(1-a_1^{(2)})}$$

Step 3 Update the weights so as to minimize the error term

$$\frac{\boldsymbol{\partial E}}{\boldsymbol{\partial w_{ij}^{(l)}}} = \boldsymbol{\delta^{(3)} * a_k^{(2)}}$$

$$\boldsymbol{\Delta w_{ij}^{(2)}} = \boldsymbol{\alpha} \frac{\boldsymbol{\partial E}}{\boldsymbol{\partial w_{ij}^{(l)}}} = \boldsymbol{\alpha * \delta^{(3)} * a_k^{(2)}}$$

$$\boldsymbol{wn_{ij}^{(2)}} = \boldsymbol{w_{ij}^{(2)}} + \boldsymbol{\Delta w_{ij}^{(2)}}$$

Learning with an adaptive learning rate

To accelerate the convergence and yet avoid the

danger of instability, we can apply two heuristics:

Heuristic

If the error is decreasing the learning rate α, should be increased.

If the error is increasing or remaining constant the learning rate α, should be decreased.

- Adapting the learning rate requires some changes in the back-propagation algorithm.
- If the sum of squared errors at the current epoch exceeds the previous value by more than a predefined ratio (typically 1.04), the learning rate parameter is decreased (typically by multiplying by 0.7) and new weights and thresholds are calculated.
- If the error is less than the previous one, the learning rate is increased (typically by multiplying by 1.05).

Practical considerations for Training Neural Networks

There a number of important practical/implementational considerations that must be taken into account when training neural networks:

1. Do we need to pre-process the training data? If so, how?
2. How may hidden units do we need?
3. Are some activation functions better than others?
4. How do we choose the initial weights from which we start the training?
5. Should we have different learning rates for the different layers?
6. How do we choose the learning rates?
7. Do we change the weights after each training pattern, or after the whole set?
8. How do we avoid flat spots in the error function?
9. How do we avoid local minima in the error function?
10. When do we stop training? In general, the answers to these questions are highly problem dependent.

Applications of Neural Networks

They can perform tasks that are easy for a human but difficult for a machine −

- Neural Networks are widely used in Military, Automotive and Aerospace for applications like Autopilot aircrafts, aircraft fault detection in aerospace, Automobile guidance systems in automotive and weapon orientation and steering, target tracking, object discrimination, facial recognition, signal/image identification in several military applications.

- Financial Industries also make use of Neural Networks for applications like real estate appraisal, loan advisor, Credit Risk Management, Churn prediction portfolio trading program, corporate financial analysis, mortgage screening, currency value prediction, corporate bond rating, document readers, credit application evaluators.

- For industrial applications, neural networks are used in manufacturing process control, product design and analysis, quality inspection systems, etc.

- Neural Networks also have applications like welding quality analysis, paper quality prediction, chemical product design analysis, dynamic modeling of chemical process systems, machine maintenance analysis, project bidding, planning, and management.

- Electronics is one of the field where neural networks have been rapidly adapted over few year for applications like Code sequence prediction, IC chip layout, chip failure analysis, machine vision, voice synthesis, etc.

- For applications like cancer cell analysis, EEG and ECG analysis, prosthetic design, transplant time optimizer Artificial Neural Networks are used in Medical Science.

- In the field of speech recognition, speech classification, text to speech conversion also Neural Network perform much better than the traditional techniques.

- Telecommunications and Transport Industry also makes use of Neural Networks for image and data compression, automated information services, real-time spoken language translation, truck Brake system diagnosis, vehicle scheduling, routing systems etc.

- Software − Pattern Recognition in facial recognition, optical character recognition, etc.

- Time Series Prediction applications like sales prediction, customer volume prediction, prediction on stocks, weather prediction, cryptocurrency prices prediction as well also makes use of Artificial Neural Networks.

- Signal Processing − Neural networks can be trained to process an audio signal and filter it appropriately in the hearing aids.

- Application like Credit Card Fraud Detection, Gmail Security Detection makes use of technique called Anomaly Detection which is also backed by Artificial Neural Networks.

Advantages of Neural Networks

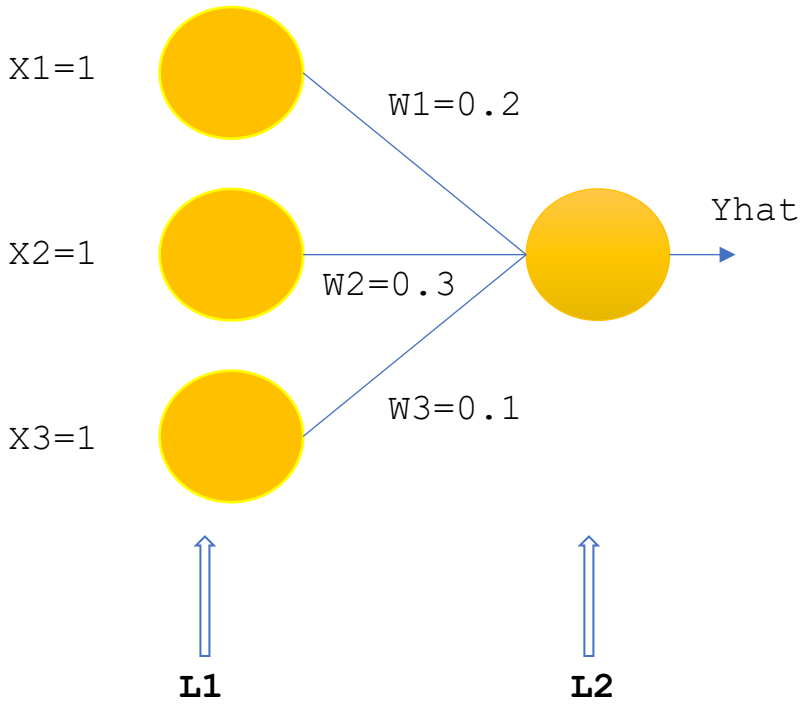
- A neural network can perform tasks that a linear program can not.
- When an element of the neural network fails, it can continue without any problem by their parallel nature.
- A neural network learns and does not need to be reprogrammed.
- It can be implemented in any application.
- It can be performed without any problem.

Limitations of Neural Networks

- The neural network needs the training to operate.
- The architecture of a neural network is different from the architecture of microprocessors, therefore, needs to be emulated.
- Requires high processing time for large neural networks.

Exercise 1 – Single Layer Perceptron model

Model description




```
# Single layer Neural Network

# x1,x2=,x3=1,1,1

# w1,w2,w3=0.2,0.3,0.1

# Activation function --> fx=2*x
```

Importing Libraries and taking the data as per the problem

```
import numpy

x=numpy.array([1.0,1.0,1.0]).reshape(1,-1)

t=numpy.array([30.0])

w=numpy.array([0.2,0.3,0.1]).reshape(3,1)
```

Programming the process involved in single layer perceptron model.

```
for i in range(30):

    l1=x #loading the data into the network - input layer

      #weighted sum of inputs from l1 layer & weights

      z=numpy.dot(l1,w)

      l2=2*z #activation function

    #GDA - wn=w-LR*dj/dw

    djdw=(l2-t)*2*l1

    w=w-0.01*djdw.T

    print(l2)

print(w)
```

Exercise 2 - Multilayer Perceptron Model with Backpropagation

Importing libraries and data

```
import numpy

xdata=numpy.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]])

ydata=numpy.array([[0],[1],[1],[0]])

numpy.random.seed(1) #for getting uniformly distributed
values

w0=numpy.random.random((3,3))

w1=numpy.random.random((3,1))
```

Training & Backpropagation

```
for i in range(5000):

   #feedforward part of the algorithm

   l0=xdata;

   l1=1/(1+numpy.exp(-(numpy.dot(l0,w0))))

   l2=1/(1+numpy.exp(-(numpy.dot(l1,w1))))

   e=l2-ydata #here we are done with forward propagation
part#here starts the backpropagation part

   delta2=e*(l2*(1-l2)) #backpropagating this delta to l1
layer

   delta1=delta2.dot(w1.T)*l1*(1-l1)

   dedw2=l1.T.dot(delta2)

   dedw1=l0.T.dot(delta1)

   w1=w1-0.8*dedw2

   w0=w0-0.8*dedw1

   if (i%100)==0:

      print("Error= "+str(numpy.mean(numpy.abs(e))))

print(l2)
```

MLP using Scikit-learn

Exercise 3 - Optical Recognition of Handwritten Digits Data Set

==========

Data Set Characteristics:

Number of Instances: 1797

Number of Attributes: 64

Attribute Information: 8x8 image of integer pixels in the range 0..16.

Missing Attribute Values: None

Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)

Date: July; 1998

This is a copy of the test set of the UCI ML hand-written digits datasets

http://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

The data set contains images of hand-written digits: 10 classes where each class refers to a digit.

Preprocessing programs made available by NIST were used to extract normalized bitmaps of handwritten digits from a preprinted form. From a total of 43 people, 30 contributed to the training set and different 13 to the test set. 32x32 bitmaps are divided into nonoverlapping locks of 4x4 and the number of on pixels are counted in each block. This generates an input

matrix of 8x8 where each element is an integer in the range 0..16. This reduces dimensionality and gives invariance to small distortions.

For info on NIST preprocessing routines, see M. D. Garris, J. L. Blue, G. T. Candela, D. L. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C.L. Wilson, NIST Form-Based Handprint Recognition System, NISTIR 5469,1994.

Import the libraries

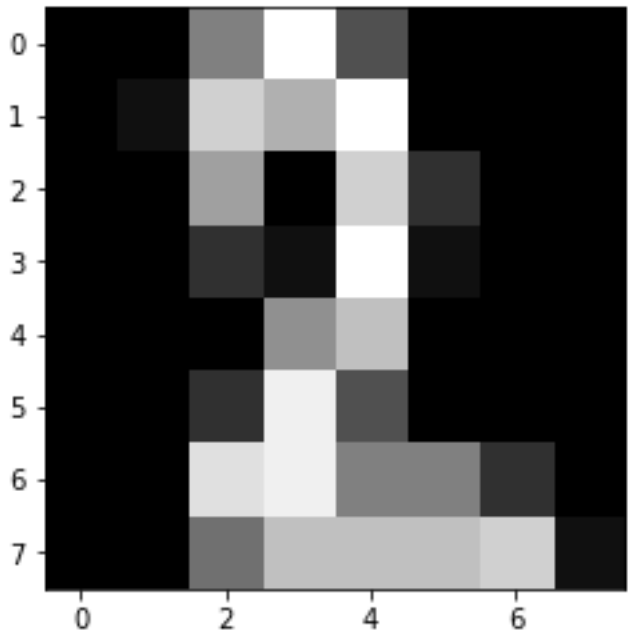
```
from sklearn import datasets

from sklearn import model_selection

from sklearn import neural_network

import matplotlib.pyplot as plt
```

Import dataset

```
digits=datasets.load_digits()

x=digits.data

y=digits.target

im=digits.images
```

Visualize the images

```
plt.imshow(im[22],cmap='gray',interpolation='nearest')

plt.show()
```



Train Test Split

```
#breaking the dataset into train and test

xtrain,xtest,ytrain,ytest,imtrain,imtest=model_selection.train
_test_split(x,y,im,test_size=0.2,random_state=5)
```

```
print(xtrain.shape)

print(xtest.shape)
```

(1437, 64)

(360, 64)

Train the algorithm using Neural Network

Hidden Layer =3 with 20 neurons in each.

Optimizer=SGD

Learning Rate = 0.001 (default)

```
nn=neural_network.MLPClassifier(hidden_layer_sizes=(20,20,20),
solver='sgd')

nn.fit(xtrain,ytrain)
```

Performance

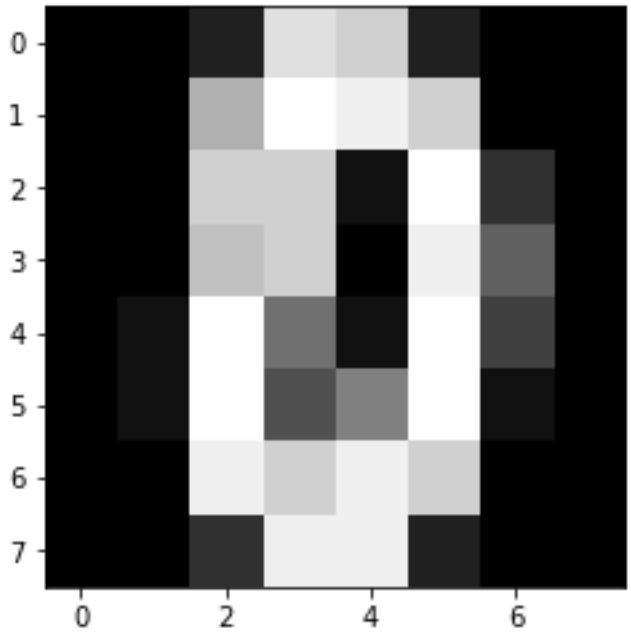
```
print('accuracy is  ')

print(nn.score(xtest,ytest))

print('the actual value is '+str(ytest[22]))#right value

print('prediction is ' + str(nn.predict(xtest[22].reshape(1,-
1)))) #predicted value

plt.imshow(imtest[22],cmap='gray') #main image

plt.show()
```

accuracy is

0.9583333333333334

the actual value is 0

prediction is [0]


0
1
2
3
4
5
6
7
0
2
4
6