**Assignment #1**

**Neural Network and Deep Learning**

**Professor Ngom**

**Students:**

**Amangel Bhullar**

**Patrick Devaney 104277922**

**Jayanthi Raghavan 104913356**

**Jess Zamani 110053408**

**Question #1** In this question you are to create some simulated data sets and then use the Adaline neuron and the Sigmoid to perform some prediction. Use whatever programming language you want to use.

* 1. Using the rnorm() function in R (or equivalent in Matlab or Python or etc), create a vector, ***x***, containing 5000 observations drawn from a Gaussian distribution *N*(0, 1) [ie, a normal distribution with mean 0 and variance 1]. This vector ***x*** represents your set of inputs *x*.

**Code: generating x**

Text

Description automatically generated with medium confidence

We are using python3 on jupyter notebook to implement question 1. First, we create a vector ***x***, containing 5000 observations drawn from a normal distribution with mean 0 and variance 1. This will represent a vector of a set of ***x*** inputs.

* 1. Using the rnorm() function in R (or equivalent in Matlab or Python or etc), create a vector, ***eps***, containing 5000 observation drawn from a *N*(0, 0.25) distribution; ie, a normal distribution with mean 0 and variance 0.25.

**Code: generating eps**

Text

Description automatically generated with medium confidence

Secondly, we are creating vector ***eps*** containing 5000 observations drawn from a normal distribution with mean 0 and variance 0.25.

* 1. Using vectors ***x*** and ***eps***, generate a vector ***y*** according to the model

*y = -*1 *+* 0.5*x –* 2*x*2 *+* 0.3*x*3 *+ eps*.

Your 5000 data-points (*x*, *y*) are generated upon completion of this Part-c. Note that the true function is a cubic function with true weight vector being ***wtrue =*** (*-*1, *+*0.5, *-*2, *+*0.3).

**Code: generating y from x and eps using model**

A picture containing graphical user interface

Description automatically generated

Now, we are generating vector y according to model ***y = -*1 *+* 0.5*x –* 2*x*2 *+* 0.3*x*3 *+ eps***using ***x*** and ***eps*** vectors. We are storing these vector ***y*** points in variable y and writing them in a file.

Calendar

Description automatically generated with low confidence

* 1. Implement the Adaline neuron learning algorithm using (i) batch gradient descent and (ii) stochastic gradient descent, and test and compare them on linear regression over your synthetic data-points. You need not to perform a cross-validation scheme here; only use the whole data set as your training set. Though if you wish, you may perform cross-validation (LOOCV or 10-fold-cv) but without a test set. The initializations, the learning rate, and the stopping criterion are left for you to explore. Think about the reasons why you use a particular strategy.

Here in Q1(d), we are implementing Adaline (Adaptive neuron) neuron learning algorithm. We Initially started weights to be very small numbers (i.e., close to 0) and for each training session we computed the prediction and then we updated the weights. We updated the weights **∆𝑊 = 𝑥𝜆 (𝑦−y^)** (delw variable in code for ∆𝑊). In Stochastic/Iterative gradient descent, we input one training example at a time and carry out forward and back propagation to update the weights.  However, In Batch gradient descent, we input the entire training dataset and then carry out forward and back propagation. The perceptron will fire if the weighted sum of its inputs is greater than threshold. There will always be a sudden change in the decision from 0 to 1 when crosses the threshold.

**Code: divide into test and training set**

Text

Description automatically generated

**Code: implementation of Adaline algorithm with batch gradient descent and stochastic gradient descent.**

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text

Description automatically generated with medium confidence

Using a degree 3 polynomial with a learning rate of 1e-10 and a maximum of 1000 iterations, batch gradient descent converges on the vector [-0.69094918 0.13415115 0.46779276 -0.1136261 ], which gives a MSE of 12.53. Iterative gradient descent converges on the vector [ 0.69288998 0.60079732 0.87249862 -0.02489512], which gives a MSE of 20.27.

* 1. Repeat Part-d, but with a Sigmoid neuron of your choice (logistic or tanh).

To smooth this sudden change in the decision (i.e. 0 or 1), we are using the sigmoid function. The sigmoid function is a family of functions and one of them is called the logistic function.

Logistic function = ,

**Code: implementation of Q1(d) Adaline algorithm with batch gradient descent and stochastic gradient descent combined with sigmoid logistic function.**

Graphical user interface, text, application, email

Description automatically generated

Using a degree 3 polynomial with a learning rate of 1e-10 and a maximum of 1000 iterations, the sigmoid neuron gives the vector [-0.00041132 0.00020512 -0.00126763 0.00103869], which has a MSE of 10.26.

* 1. Repeat Part-d and Part-e with cross-validation method of your own choice (LOOCV or 10-fold-cv) to find the best degree *d*. You must first randomly create a test set of size between 20% and 30% drawn from your original full data set. If this is correctly done, your methods should not only find *d =* 3 to be the degree, but they should also find the best weight vector, ***wbest***, to be as close as possible to ***wtrue =*** (*-*1, *+*0.5, *-*2, *+*0.3).

Text

Description automatically generated

degree d *=* 3, Mean Square Error calculated with Batch Gradient Descent, stochastic/Iterative Gradient Descent and logistic sigmoid function. As we observe from the table below, the mean square error is least with Logistic sigmoid function compared to Batch Gradient Descent, stochastic/Iterative Gradient Descent.

|  |  |  |  |
| --- | --- | --- | --- |
| Test set | Batch Gradient Descent  (Mean Square Error) | Stochastic/Iterative Gradient Descent  (Mean Square Error) | Sigmoid (Mean Square Error) |
| 20% | 15.461761166878377 | 13.656968655092966 | 9.559370335538583 |
| 25% | 17.062839729113954 | 9.558280000895515 | 8.184903804211087 |
| 30% | 14.090065846168693 | 18.905321624884138 | 9.284334418865136 |
| Best weights | [0.47629954, 0.08687094, 0.57626535, 0.11145471] | [-0.96264035, -0.49044357, 0.58539094, -0.62965457] | [-0.00044045, 0.00021043, -0.00140316, 0.00117031] |

g. Comparing different learning rates, with a maximum of 1000 iterations

|  |  |  |  |
| --- | --- | --- | --- |
| Learning Rate | Batch Gradient Descent  (Mean Square Error) | Stochastic/Iterative Gradient Descent  (Mean Square Error) | Sigmoid (Mean Square Error) |
| 1e-10 | 2.3230422589178654 | 19.928130877396747 | 9.464297748461169 |
| 1e-11 | 6.930133285800313 | 18.94820807503174 | 9.478738805284111 |
| 1e-12 | 16.640769747532524 | 6.735043510303182 | 9.480184079246671 |
| 1e-13 | 11.184037897147949 | 6.1926763709367485 | 9.48032861832573 |

A clear pattern that emerges is that decreasing the learning rate seems to increase the error of batch gradient descent after 1000 iterations. However, the opposite is true of iterative gradient descent – error decreases as the learning rate decreases. This makes sense, because it is making decisions based on each individual vector instead of all of them at once. While the error of sigmoid very slightly increases as the learning rate decreases, it seems fairly robust to changes in the learning rate.

**Question #2** The accuracy you get for the q2 code will depend on the dataset you generate, the learning rate, and the maximum iterations you allow

In general, if you decrease the learning rate (e.g. changing 2e-10 to 1e-11) you will probably need to increase the maximum number of iterations (the fourth function parameter) to get the same or better accuracy

1. In this question, first we created 5000 data points which are linearly separable by the line mx+b.

|  |
| --- |
| import numpy as np |
|  | from sys import argv |
|  |  |
|  | # Take line info as command arguments |
|  | m = int(argv[1]) |
|  | b = int(argv[2]) |
|  | # Name of file to write to |
|  | fn = argv[3] |
|  |  |
|  | with open(fn, 'w+') as f: |
|  | for i in range(2500): |
|  | # Generate + instance |
|  | plusx = np.random.uniform(-1000,1000) |
|  | # Subtract negative y-adjustment to get around half-closed interval |
|  | plusy = m \* plusx + b - np.random.uniform(-1000,0) |
|  | # Generate - instance |
|  | minusx = np.random.uniform(-1000,1000) |
|  | minusy = m \* minusx + b + np.random.uniform(-1000,0) |
|  | # Write generated points to file |
|  | f.write(str(plusx) + " " + str(plusy) + " +1\n") |
|  | f.write(str(minusx) + " " + str(minusy) + " -1\n") |

These data points are divided into two different classes, class +1 has 2500 points and class -1 has 2500 points. These data points are supposed to be separated linearly and we considered line m\*x+b to be the classifier. The accuracy of models depends on the data set generated.

1. Comparing Perceptron with Adaline and Logistic function in same data points

Comparing the Sign function (Perceptron) with Sigmoid function (Logistic regression in this case) shows that with sigmoid we can achieve better accuracy, for example for one run in same data set, 10-fold cross validation, and training size of 0.75, the accuracy of Perceptron was 0.80 but the accuracy of sigmoid model was 0.85. In fact, we expected to get better accuracy with sigmoid model because it has a better activation function.

**First run:** Mean Accuracy of Perceptron model: 0.928

Mean Accuracy of Sigmoid model: 0.938

**Second run:** Mean Accuracy of Perceptron model: 0.936

Mean Accuracy of Sigmoid model: 0.955

Third run: Mean Accuracy of Perceptron model: 0.947

Mean Accuracy of Sigmoid model: 0.961

As you can see, in same situation and on the same data points sigmoid classifier acts better than the perceptron.

In the other step, we compared Adaline model with Perceptron. For Adaline, both batch and stochastic gradient descent were implemented. In one run, MSE for Perceptron we got was 0.4998871 but with Adaline using batch GD the loop did not converge, we increased the number of iterations from 500 to 3000 but it did not converge. MSE criterion for Adaline using stochastic GD was 39.526843.

Iteration = 1500, rate = 1e-12: Perceptron MSE=0.49, SGD Adaline=39.5, BGD adaline NaN

Iteration = 1000, rate = 1e-13: Perceptron MSE=0.43, SGD Adaline=12.28, BGD adaline NaN

In this implementation, batch GD with adaline is not converging. Comparing MSE of Perceptron and SGD Adaline shows that Perceptron acts better in this case.

**Training size:** When we change the size of training and testing set, the accuracy of the classification algorithm changes relatively. For example, with the training size 80%, the accuracy of both pocket and perceptron algorithm was 99.9% but when we changed it to 50% we got 96.5% accuracy of perceptron and pocket algorithm. When increasing the training size to 95%, the accuracy reduced to 98.2%. The best accuracy that we could get was when the size of training set is 80% and test set 20%.

**The learning rate:** By decreasing the learning rate, the accuracy of classification decreases accordingly, but is converges slowly and it takes more time to converge. On the other hand, the mean square error decreases as well.

When you increase the learning rate, the MSE increases drastically and it does not converge. On the other hand, reducing the learning rate leads to low speed of algorithm. For instance, when learning rate is 1e-5, MSE equals to 0.29 but with learning rate 1e-10 MSE is 0.491597.

Rate= 1e-13: Accuracy of Perceptron = 98.64

Rate = 1e-11: Accuracy of Perceptron = 98.96

Another parameter that affect the result is maximum iteration. Changing from 500 to 1500 led to almost same accuracy with more MSE. The accuracy in this case was around 99.6%. MSE is 0.31 for iteration=500, but in 1500 iteration it increased to 0.45.

1. Comparing Perceptron with Pocket algorithm on non-linear separable data points

For this part, we generated to class of data which were non-linearly separable (we use a second degree polynomial to separate data points as follow).

plusx = np.random.uniform(-1000,1000)

# Subtract negative y-adjustment to get around half-closed interval

plusy = m \* plusx \*\* 2 + b \* plusx + c - np.random.uniform(-1000,0)

# Generate - instance

minusx = np.random.uniform(-1000,1000)

minusy = m \* plusx \*\* 2 + b \* plusx + c + np.random.uniform(-1000,0)

Since for this part, we had to approximate a non-linear classifier with linear classifiers (perceptron and pocket), in both cases we got same accuracy which was less than the time that data points were linearly separable. In this case, we got accuracy 56.72% which has been reduced drastically. In fact, since this practice was to approximate second degree polynomial with a line, the inevitable consequence was losing the accuracy. Accuracy of Pocket algorithm was exactly like Perceptron. The reason for this is that in practice, this never runs long enough for values to be different and it gets kicked off by maximum iteration counter instead. In different learning rates and different iterations, accuracy of both models were the same. The best accuracy that we could get was on learning rate 1e-12 and iteration 1500 which was 56.72%.

Perceptron, Adaline and Pocket algorithm codes:

import math

import numpy as np

from sys import argv

fn = 'my new file'

# Partition training and test data

def partition(w, size = 0.75):

# Randomly shuffle the array and split it into two parts

p = np.random.permutation(w)

c = round(size \* len(w))

return p[:c], p[c:]

# Determine sign of dot product

def sign(w,x):

return 1 if np.dot(w.transpose(),x) > 0 else -1

# Perceptron learning with optional pocket feature (returns both)

# Arguments: data set, learning rate, error after which to stop, iterations after which to stop

def perceptron(dataset, rate = 0.1, cutoff = 0.01, maxiter = math.inf):

best\_err = math.inf

best\_w = None

err = math.inf

m = len(dataset)

r\_m = 1/m

# Randomly initialize weights

#w = np.array([np.random.uniform(-1,1),np.random.uniform(-1,1),np.random.uniform(-1,1)])

# Initialize weights as zeros

w = np.array([0.0, 0.0,0.0])

count = 0

while err > cutoff and count < maxiter:

err = 0

for ex in dataset:

ex\_sign = sign(w,ex[:3])

ex\_err = 0.5 \* (ex[3] - ex\_sign)

err += r\_m \* abs(ex\_err)

w += rate \* ex\_err \* ex[:3]

# Keep track of the lowest seen error for the pocket algorithm

if err < best\_err:

best\_err = err

best\_w = w[:]

count += 1

return w, best\_w

# Gradient descent (Adaline)

# Arguments: data set, learning rate, error after which to stop, iterations after which to stop,

# boolean option for + (True) or incremental/stochastic (False)

def gradientDescent(dataset, rate = 0.1, cutoff = 0.01, maxiter = math.inf, batch = True):

err = math.inf

w = np.array([np.random.uniform(-1.0,1.0),np.random.uniform(-1.0,1.0),np.random.uniform(-1.0,1.0)])

count = 0

while err > cutoff and count < maxiter:

delw = np.array([0.0,0.0,0.0])

err = 0

for ex in dataset:

y = np.dot(w.transpose(), ex[:3])

# Update weights based on the mode we're using

if batch:

delw += rate \* (ex[3] - y) \* ex[:3]

else:

w += rate \* (ex[3] - y) \* ex[:3]

err += 0.5 \* (ex[3] - np.dot(w.transpose(), ex[:3])) \*\* 2

if batch:

w += delw

count += 1

return w

# Determine the % accuracy of weights on a dataset

def accuracy\_perceptron(dataset, w):

count = 0

for ex in dataset:

if sign(w,ex[:3]) == ex[3]:

count += 1

return count / len(dataset)

# Determine the MSE for gradient descent

def err\_gd(dataset, w):

m = len(dataset)

err = 0.0

for ex in dataset:

err += 0.5 \* (ex[3] - np.dot(w.transpose(), ex[:3])) \*\* 2

return 1/m \* err

# Read the file to sample into memory

with open(fn) as f:

contents = f.read()

# Split into lines

contents = contents.split('\n')

# Break each valid line into points and make a list

pts = []

for line in contents:

line = [1] + line.split(" ")

if len(line) == 4:

pts.append(np.array([float(x) for x in line]))

# Randomly select test/validation sets

test, val = partition(pts,0.75)

# Test perceptron and pocket perceptron

w, pocket\_w = perceptron(test, 1e-10, 0, 1500)

print('weights = ', w)

print('MSE = ', err\_gd(val,w))

print(' weights of pocket algorithm : ', pocket\_w)

print('accuracy of perceptron: ' , accuracy\_perceptron(val,w))

print('accuracy of pocket algorithm: ', accuracy\_perceptron(val,pocket\_w))

# Test batch gradient descent

# Note: 2e-10 seems to be the highest learning rate that gives reasonable results

# 1e-10 seems about the highest that converges

w = gradientDescent(test, 2e-10, 0, 1000, True)

print(w)

print(err\_gd(val,w))

# Test iterative/stochastic gradient descent

w = gradientDescent(test, 2e-10, 0, 1000, False)

print(w)

print(err\_gd(val,w))

Sigmoid code:

import math

import numpy as np

from sys import argv

fn = 'my new file'

# Partition training and test data

def partition(w, size = 0.75):

# Randomly shuffle the array and split it into two parts

p = np.random.permutation(w)

c = round(size \* len(w))

return p[:c], p[c:]

# Determine logistic regression function

def sigmoid(w,x):

sig = 1 / (1 + math.exp(-(np.dot(w.transpose(),x))))

if sig > 0.5:

return 1

else:

return -1

# Sigmoid neuron

# Arguments: data set, learning rate, error after which to stop, iterations after which to stop

def NeuralNetwork(dataset, rate = 0.1, cutoff = 0.01, maxiter = 100):

err = math.inf

m = len(dataset)

r\_m = 1/m

# Randomly initialize weights

#w = np.array([np.random.uniform(-1,1),np.random.uniform(-1,1),np.random.uniform(-1,1)])

# Initialize weights as zeros

w = np.array([0.0, 0.0,0.0])

count = 0

while err > cutoff and count < maxiter:

err = 0

for ex in dataset:

ex\_net = sigmoid(w,ex[:3])

ex\_err = 0.5 \* (ex[3] - ex\_net)

err += r\_m \* abs(ex\_err)

w += rate \* ex\_err \* ex[:3]

count += 1

return w

# Gradient descent (sigmoid)

# Arguments: data set, learning rate, error after which to stop, iterations after which to stop,

# boolean option for batch (True) or incremental/stochastic (False)

def gradientDescent(dataset, rate = 0.1, cutoff = 0.01, maxiter = math.inf, batch = True):

err = math.inf

w = np.array([np.random.uniform(-1.0,1.0),np.random.uniform(-1.0,1.0),np.random.uniform(-1.0,1.0)])

count = 0

while err > cutoff and count < maxiter:

delw = np.array([0.0,0.0,0.0])

err = 0

for ex in dataset:

y = sigmoid(w, ex[:3])

# Update weights based on the mode we're using

if batch:

delw += rate \* (ex[3] - y) \* y \* (1-y) \* ex[:3]

else:

w += rate \* (ex[3] - y)\* y \* (1-y) \* ex[:3]

err += 0.5 \* (ex[3] - y) \*\* 2

if batch:

w += delw

count += 1

return w

# Determine the % accuracy of weights on a dataset

def accuracy\_sig(dataset, w):

count = 0

for ex in dataset:

if sigmoid(w,ex[:3]) == ex[3]:

count += 1

return count / len(dataset)

# Determine the MSE for gradient descent

def err\_gd(dataset, w):

m = len(dataset)

err = 0.0

for ex in dataset:

err += 0.5 \* (ex[3] - np.dot(w.transpose(), ex[:3])) \*\* 2

return 1/m \* err

# Read the file to sample into memory

with open(fn) as f:

contents = f.read()

# Split into lines

contents = contents.split('\n')

# Break each valid line into points and make a list

pts = []

for line in contents:

line = [1] + line.split(" ")

if len(line) == 4:

pts.append(np.array([float(x) for x in line]))

# Randomly select test/validation sets

test, val = partition(pts,0.75)

# Test perceptron and pocket perceptron

w, pocket\_w = NeuralNetwork(test, 1e-10, 0, 1000)

print('weights = ', w)

print(' weights of pocket algorithm : ', pocket\_w)

print('accuracy of perceptron: ' , accuracy\_sig(val,w))

print('accuracy of pocket algorithm: ', accuracy\_sig(val,pocket\_w))

# Test batch gradient descent

# Note: 2e-10 seems to be the highest learning rate that gives reasonable results

# 1e-10 seems about the highest that converges

w = gradientDescent(test, 2e-10, 0, 1000, True)

print(w)

print(err\_gd(val,w))

# Test iterative/stochastic gradient descent

w = gradientDescent(test, 2e-10, 0, 1000, False)

print(w)

print(err\_gd(val,w))

Q3- **What is XOR?**

Exclusive OR (XOR) is a logical operation, which produces the high output for dissimilar inputs. The truth tabel of XOR operation is given below in tabel 1.

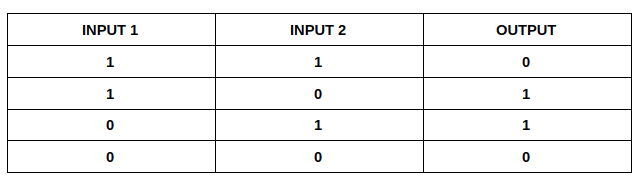


Table 1:The truth tabel of XOR

**What is a XOR Classification Problem?**

The graphical representation, of input and output relation ship of XOR classification problem is given by the following Fig .1. From the figure, it is clear that single linear equation can not separate the different outputs. This problem is called linearly inseparable.

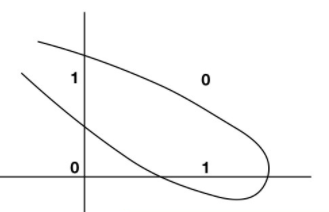


Fig.1 Linearly inseparable

**Solutions to Linearly Inseparable Problem:**

There are two solutions available to resolve the linearly inseparable problem.

* Applying Kernel Trick
* Introducing additional layer i.e., making use of Multi-Layer Perceptron (MLP) architecture.

**Kernel Trick to handle of linearly inseparable problem**

Kernel trick is a concept which makes classification task easy and quick to compute when the data is not linearly separable.

Both KPCA and KLDA are designed to handle the non-linearity in binary-class as well as multi-class problems as shown in Fig.2 (a), (b), (c). Both are extensions of conventional PCA and LDA that use kernel methods to project the data onto a high-dimensional feature space by using the “kernel trick” approach [1]. In KPCA and KLDA, the points are projected to non-linear feature space instead of linear space using polar coordinates to deal with circle. After projecting onto nonlinear feature space, in KPCA principal components are extracted and KLDA projections are linearly separable with very good variation between the classes and also very less variation within the classes.

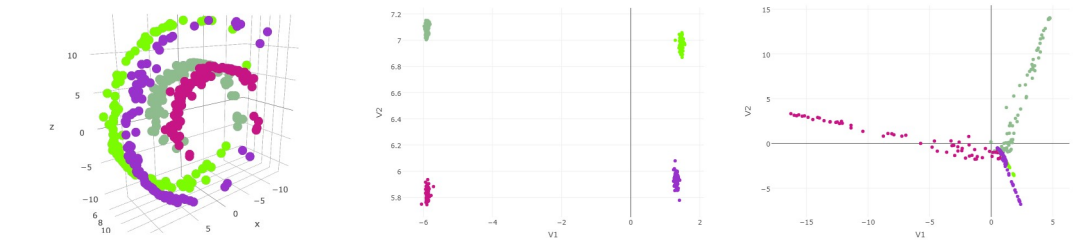


Fig.2 (a) Fig.2 (b) Fig.2 (c)

Fig.2 (a) Original Data (b) KLDA Projections in 2D (c) KPCA Projections in 2D

**Introducing additional layer i.e., making use of Multi -Layer Perceptron (MLP)**

XOR linearly inseparable problem can also be resolved by making changes in existing Perceptron architecture. By introducing additional layers called hidden layers, it is possible to handle the linearly inseparable problem. XOR can be realized as a combination of NOR and AND. XOR can be realized as follows.

**XOR) = NOR((NOR(,AND)),** which can be proved by using truth table.

|  |  |  |
| --- | --- | --- |
| **)** | **))** | **) = XOR** |
| (0,0) | (1,0) | 0 |
| (0,1) | (0,0) | 1 |
| (1,0) | (0,0) | 1 |
| (1,1) | (0,1) | 0 |

Table 2:The truth tabel of XOR realized using NOR((NOR(,AND))

From the truth table, it is clear that XOR can be implemented in two steps. The implementation of the XOR is given below in the Fig 3. There are totally nine weights must be used for this MLP.

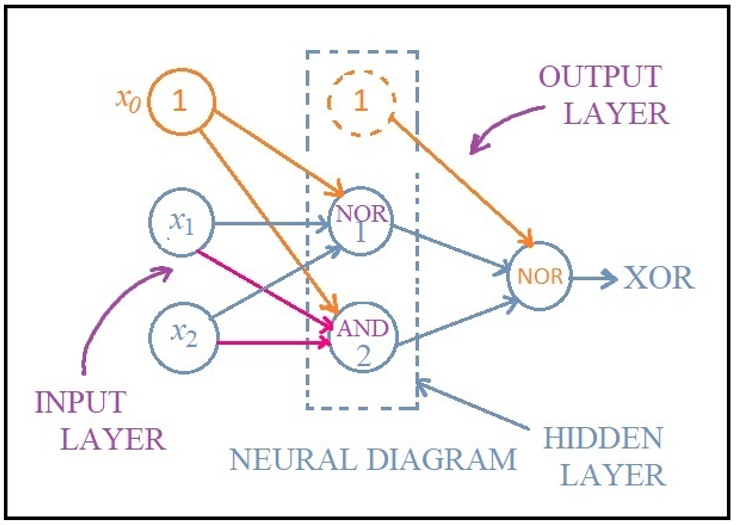


Fig.3 XOR realized using NOR((NOR(,AND))

From the figure, it is clear that, by introducing the additional layer called the hidden layer, the problem of XOR gate can be resolved.

There are totally nine weights needed to represent the XOR gates. The weights applied in each layer are given below. There are some notations to be followed for weights applied in each layer. The super script denotes the layer number. The weights are given below.

**(NOR)** = 1 = - 2, = -2

1st Layer Weights

**(AND)** = -3 = 2, = 2

**(FINAL NOR)** = 1 = - 2, = -2 2nd Layer Weights

By adding the additional layer called hidden layer, introduces non-linearity. Hence it is possible to solve the linearly in separable problem using MLP architecture shown in Fig.3.

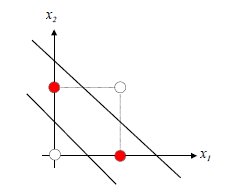


Fig.4.Two linear equation needed for classification