ASSIGNMENT 7

TITLE: BACKPROPAGATION NEURAL NETWORK

PROBLEM STATEMENT: -

Write a python program to show Back Propagation Network for XOR function with Binary Input and Output

OBJECTIVE:

1. To Learn and understand the concepts of types of neural network
2. To learn and understand back propagation network
3. To understand implementation of XOR function with binary input using python.

PREREQUISITE: -

1 Basic of Python Programming

2 Concept of Artificial Neural Network

THEORY:

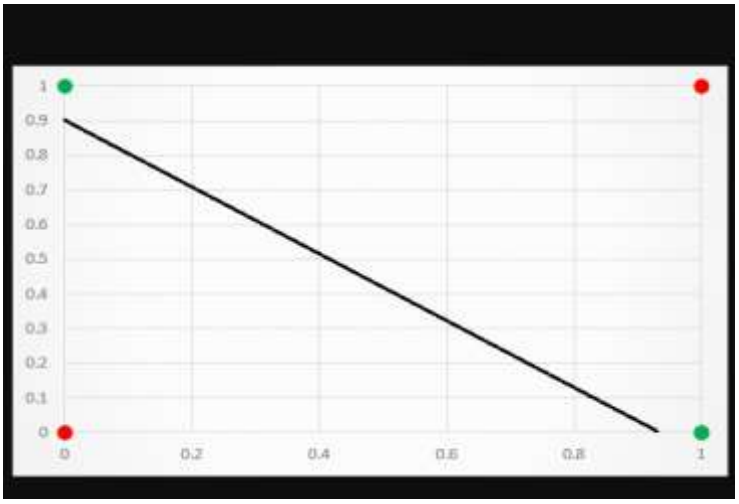
XOR is a problem that cannot be solved by a single layer perceptron, and therefore requires a multi-layer perceptron or a deep learning model. A single-layer perceptron can only learn linearly separable patterns, whereas a straight line or hyperplane can separate the data points. However, the XOR problem requires a non-linear decision boundary to classify the inputs accurately. This means that a single-layer perceptron fails to solve the XOR problem, emphasizing the need for more complex neural networks.

The XOR function is a binary function that takes two binary inputs and returns a binary output. The output is true if the number of true inputs is odd, and false otherwise. In other words, it returns true if exactly one of the inputs is true, and false otherwise.

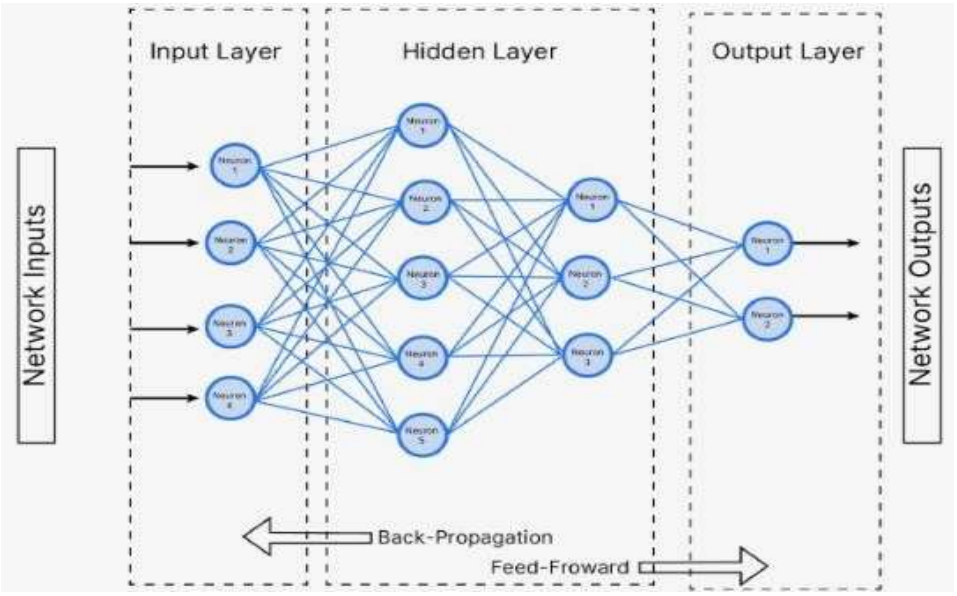
Input A	Input B	Output (A XOR B)
0	0	0
0	1	1
1	0	1
1	1	0

The XOR function is not linearly separable, which means we cannot draw a single straight

line to separate the inputs that yield different outputs.

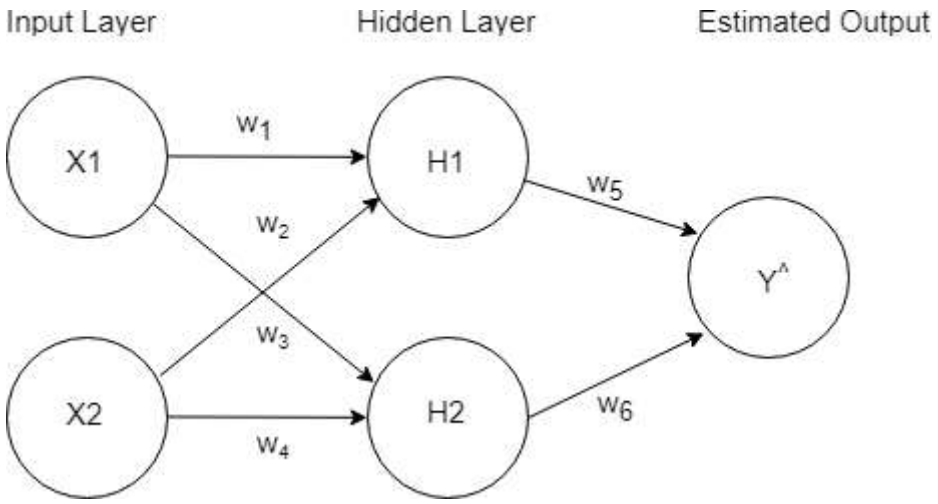


To solve the XOR problem, we need to introduce multi-layer perceptrons (MLPs) and the backpropagation algorithm. MLPs are neural networks with one or more hidden layers between the input and output layers. These hidden layers allow the network to learn non-linear relationships between the inputs and outputs.



The backpropagation algorithm is a learning algorithm that adjusts the weights of the neurons in the network based on the error between the predicted output and the actual output. It works by propagating the error backwards through the network and updating the weights using gradient descent.In addition to MLPs and the backpropagation algorithm, the choice of activation functions also plays a crucial role in solving the XOR problem. Activation functions introduce non-linearity into the network, allowing it to learn complex patterns. Popular

activation functions for solving the XOR problem include the sigmoid function and the hyperbolic tangent function.



Inputs

$$x_1 = [1,1]^T, \quad y_1 = +1$$
$$x_2 = [0,0]^T, \quad y_2 = +1$$
$$x_3 = [1,0]^T, \quad y_3 = -1$$
$$x_4 = [0,1]^T, \quad y_4 = -1$$

2 hidden neurons are used, each takes two inputs with different weights. After each forward pass, the error is back propogated. I have used sigmoid as the activation function at the hidden layer.

At hidden layer:

$$H_1 = x_1 w_1 + x_2 w_2$$
$$H_2 = x_1 w_3 + x_2 w_4$$

At output layer:

$$Y^{\wedge} = \sigma(H_1) w_5 + \sigma(H_2) w_6$$

here, σ represents sigmoid function.

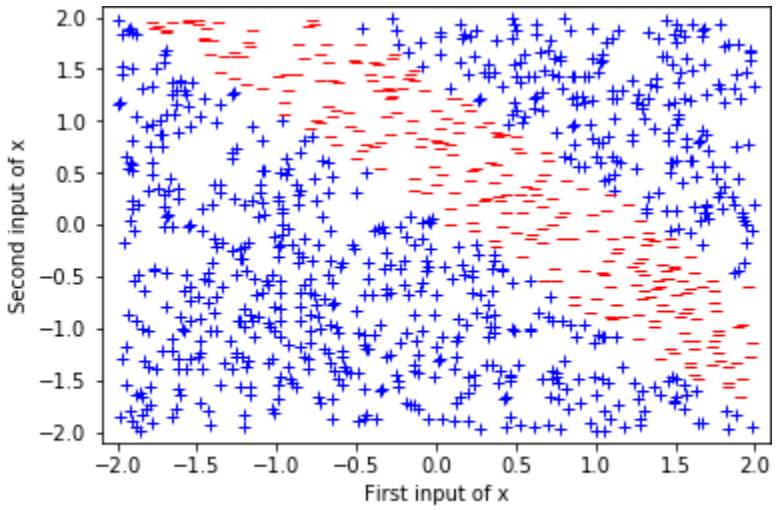
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Loss function:

$$\frac{1}{2}(Y - Y^{\wedge})^2$$

Results

Classification WITHOUT gaussian noise




```
(731, 269)
```

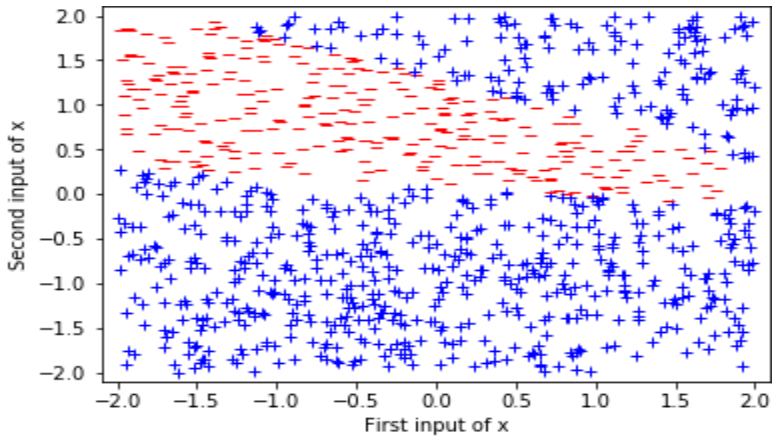
Observation: To classify, I generated 1000 x 2 random floats in range -2 to 2. Using weights from trained model, I classified each input & plotted it on 2-D space. Out of 1000, 731 points were classified as "+1" and 269 points were classified as "-1" It is clearly seen, classification region is not a single line, rather the 2-D region is separated by "-1" class. I did the same for classifications with Gaussian noise.

Classification WITH Gaussian noise

In real applications, we almost never work with data without noise. Now instead of using the above points generate Gaussian random noise centered on these locations.

$$\boldsymbol{x_1} \sim \boldsymbol{\mu_1} = [1,1]^T, \Sigma_1 = \Sigma \quad y_1 = +1$$
$$\boldsymbol{x_2} \sim \boldsymbol{\mu_1} = [0,0]^T, \Sigma_2 = \Sigma \quad y_1 = +1$$
$$\boldsymbol{x_3} \sim \boldsymbol{\mu_1} = [1,0]^T, \Sigma_3 = \Sigma \quad y_1 = -1$$
$$\boldsymbol{x_4} \sim \boldsymbol{\mu_1} = [0,1]^T, \Sigma_4 = \Sigma \quad y_1 = -1$$
$$\Sigma = \begin{bmatrix} \sigma & 0 \\ 0 & \sigma \end{bmatrix}$$

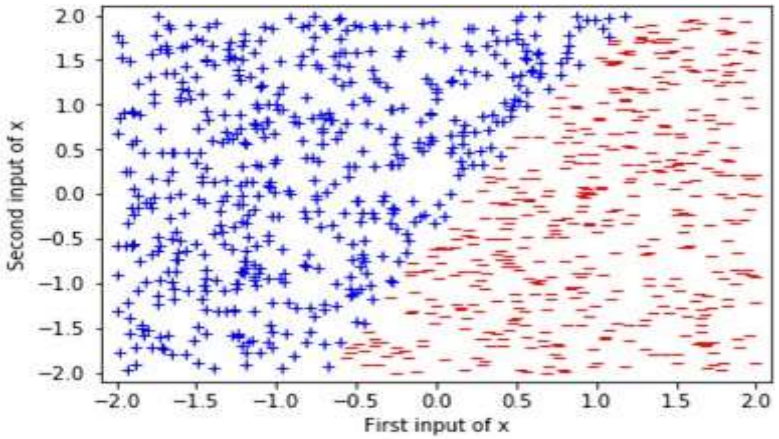
σ = 0.5




```
(702, 298)
```

Observation : We see a shift in classification regions. Here, classification is still not separated by a line

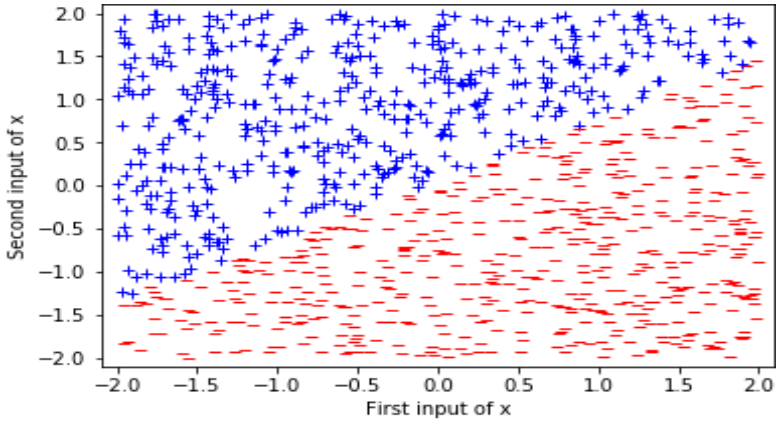
σ = 1.0




```
(538, 462)
```

Observation : We observe classifications being divided into two distinct regions.

σ = 2.0




```
(467, 533)
```

Observation : We see a shift in classification regions (compared to of σ = 1). Here also, we observe two distinct regions of classification.

CONCLUSION: We have studied and implemented Back Propagation Network for XOR function with Binary Input and Output

ASSIGNMENT 8
TITLE: ART NEURAL NETWORK

PROBLEM STATEMENT: -

. Write a python program to illustrate ART neural network.

OBJECTIVE:

1. To learn and understand the concepts of ART neural network
2. To understand implementation of XOR function with binary input using python.

PREREQUISITE: -

3. Basic of ART neural network
4. Concept of ART neural network

THEORY:

Adaptive Resonance Theory (ART) Adaptive resonance theory is a type of neural network technique that uses unsupervised learning technique. The term “adaptive” and “resonance” used in this suggests that they are open to new learning (i.e. adaptive) without discarding the previous or the old information (i.e. resonance).

The ART networks are known to solve the stability-plasticity dilemma i.e., stability refers to their nature of memorizing the learning and plasticity refers to the fact that they are flexible to gain new information.

Due to this the nature of ART they are always able to learn new input patterns without forgetting the past. ART networks implement a clustering algorithm. Input is presented to the network and the algorithm checks whether it fits into one of the already stored clusters. If it fits then the input is added to the cluster that matches the most else a new cluster is formed.

Types of Adaptive Resonance Theory (ART) Carpenter and Grossberg developed different ART architectures as a result of 20 years of research. The ARTs can be classified as follows:

- **ART1 –** It is the simplest and the basic ART architecture. It is capable of clustering binary input values.
- **ART2 –** It is extension of ART1 that is capable of clustering continuous-valued input data.
- **Fuzzy ART –** It is the augmentation of fuzzy logic and ART.

- **ARTMAP –** It is a supervised form of ART learning where one ART learns based on the previous ART module. It is also known as predictive ART.
- **FARTMAP –** This is a supervised ART architecture with Fuzzy logic included.

Basic of Adaptive Resonance Theory (ART) Architecture The adaptive resonant theory is a type of neural network that is self-organizing and competitive. It can be of both types, the unsupervised ones(ART1, ART2, ART3, etc) or the supervised ones(ARTMAP). Generally, the supervised algorithms are named with the suffix "MAP". But the basic ART model is unsupervised in nature and consists of :

- F1 layer or the comparison field(where the inputs are processed)
- F2 layer or the recognition field (which consists of the clustering units)
- The Reset Module (that acts as a control mechanism)

The F1 layer accepts the inputs and performs some processing and transfers it to the F2 layer that best matches with the classification factor. There exist two sets of weighted interconnection for controlling the degree of similarity between the units in the F1 and the F2 layer. The F2 layer is a competitive layer. The cluster unit with the large net input becomes the candidate to learn the input pattern first and the rest F2 units are ignored. The reset unit makes the decision whether or not the cluster unit is allowed to learn the input pattern depending on how similar its top-down weight vector is to the input vector and to the decision. This is called the vigilance test. Thus we can say that the vigilance parameter helps to incorporate new memories or new information. Higher vigilance produces more detailed memories, lower vigilance produces more general memories.

CONCLUSION: We have studied and implemented Back Propagation Network for XOR function with Binary Input and Output

ASSIGNMENT 9

TITLE: BACK PROPAGATION FEED-FORWARD NEURAL NETWORK

PROBLEM STATEMENT: -

. Write a python program in python program for creating a Back Propagation Feed-forward neural network

OBJECTIVE:

1. To Learn and understand the Back Propagation Feed-forward neural network
2. To learn and understand back propagation network

PREREQUISITE: -

1. Basic of Python Programming
2. Concept of Artificial Neural Network

THEORY:

Backpropagation neural network is used to improve the accuracy of neural network and make them capable of self-learning. The backpropagation algorithm is one of the algorithms responsible for updating network weights with the objective of reducing the network error.

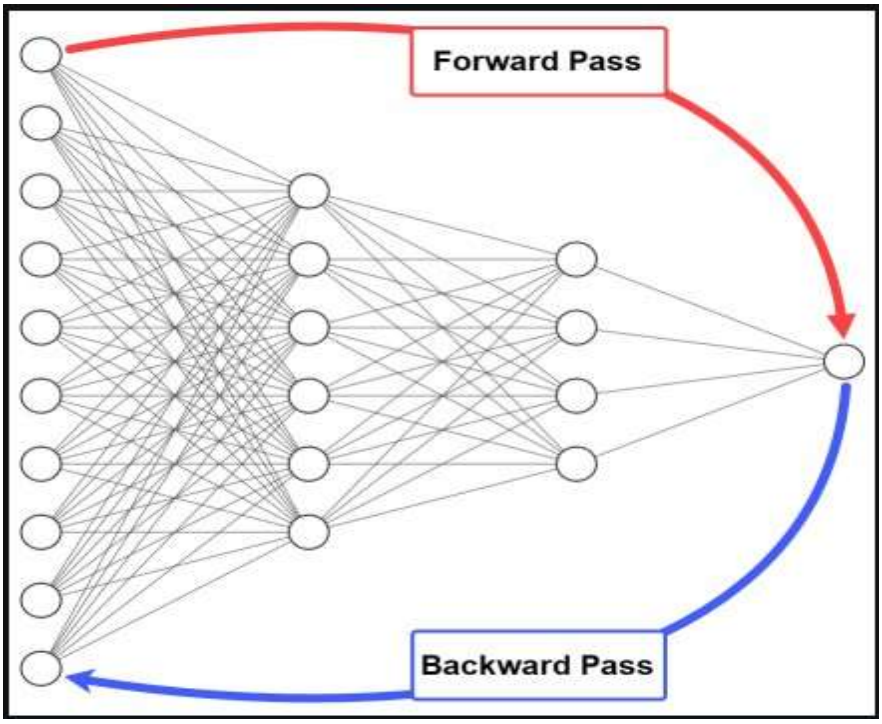
Backpropagation means "backward propagation of errors". Here error is spread into the reverse direction in order to achieve better performance. Backpropagation is an algorithm for supervised learning of artificial neural networks that uses the gradient descent method to minimize the cost function. It searches for optimal weights that optimize the mean-squared distance between the predicted and actual labels.

The backpropagation algorithm consists of two phases:

The forward pass where our inputs are passed through the network and output predictions obtained (also known as the propagation phase).

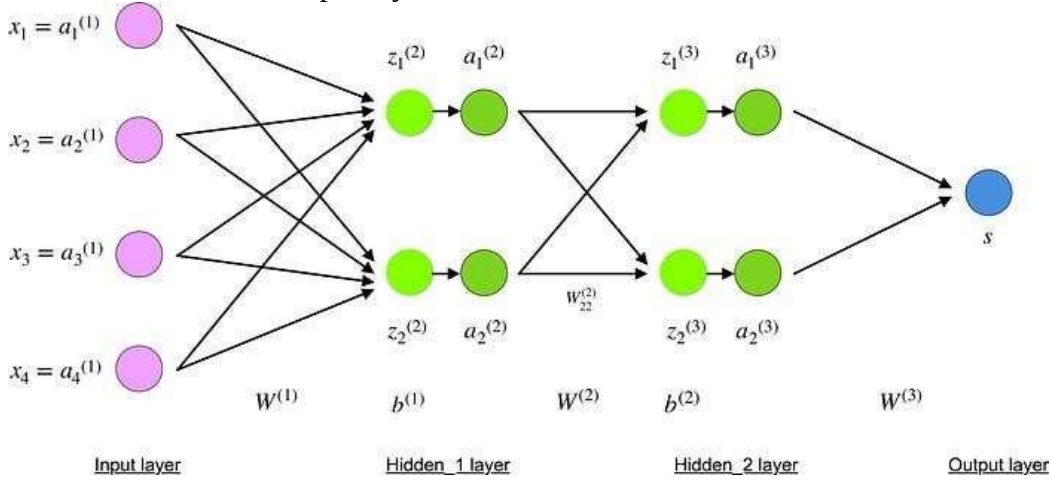
The backward pass where we compute the gradient of the loss function at the final layer (i.e., predictions layer) of the network and use this gradient to recursively apply the chain rule

to update the weights in our network (also known as the weight update phase).



At the end of the forward pass, the network error is calculated, and should be as small as possible. If the current error is high, the network didn't learn properly from the data. What does this mean? It means that the current set of weights isn't accurate enough to reduce the network error and make accurate predictions. As a result, we should update network weights to reduce the network error.

The 4-layer neural network consists of 4 neurons for the input layer, 4 neurons for the hidden layers and 1 neuron for the output layer.



Input layer

The neurons, colored in **purple**, represent the input data. These can be as simple as scalars or more complex like vectors or multidimensional matrices.

$$x_i = a_i^{(1)}, i \in 1,2,3,4$$

Equation for input x_i

The first set of activations (*a*) are equal to the input values. *NB: "activation" is the neuron's value after applying an activation function. See below.*

Hidden layers

The final values at the hidden neurons, colored in **green**, are computed using *z^l* — weighted inputs in layer *l*, and *a^l*— activations in layer *l*. For layer 2 and 3 the equations are:

- *l = 2*

$$z^{(2)} = W^{(1)}x + b^{(1)}$$

$$a^{(2)} = f(z^{(2)})$$

Equations for z^2 and a^2

- *l = 3*

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$

$$a^{(3)} = f(z^{(3)})$$

Equations for z^3 and a^3

W^2 and W^3 are the weights in layer 2 and 3 while b^2 and b^3 are the biases in those layers.

Activations a^2 and a^3 are computed using an activation function f. Typically, this **function f is non-linear** (e.g. sigmoid, ReLU, tanh) and allows the network to learn complex patterns in data. We won't go over the details of how activation functions work, but, if interested, I strongly recommend reading this great article.Looking carefully, you can see that all of x, z^2, a^2, z^3, a^3, W^1, W^2, b^1 and b^2 are missing their subscripts presented in the 4-layer network illustration above. **The reason is that we have combined all parameter values in matrices, grouped by layers.** This is the standard way of working with neural networks and one should be comfortable with the calculations. However, I will go over the equations to clear out any confusion.

Let's pick layer 2 and its parameters as an example. The same operations can be applied to any layer in the network. W^1 is a weight matrix of shape (n, m) where n is the number of output neurons (neurons in the next layer) and m is the number of input neurons (neurons in the previous layer). For us, $n = 2$ and $m = 4$.

$$W^{(1)} = \begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} & W_{13}^{(1)} & W_{14}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} & W_{23}^{(1)} & W_{24}^{(1)} \end{bmatrix}$$

Equation for W^1

NB: The first number in any weight's subscript matches the index of the neuron in the next layer (in our case this is the *Hidden_2 layer*) **and the second number matches the index of the neuron in previous layer** (in our case this is the *Input layer*).

- x is the input vector of shape $(m, 1)$ where m is the number of input neurons. For us, $m = 4$.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

Equation for x

- b^1 is a bias vector of shape $(n , 1)$ where n is the number of neurons in the current layer.

  For us, $n = 2$.

$$b^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix}$$

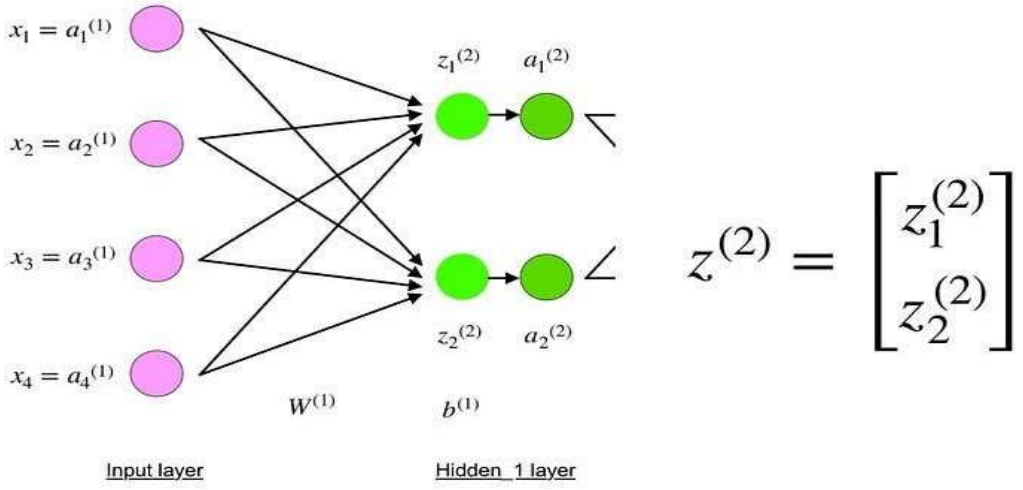
Equation for b^1

Following the equation for z^2, we can use the above definitions of W^1, x and b^1 to derive "*Equation for z^2*":

$$z^{(2)} = \begin{bmatrix} W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + W_{14}^{(1)}x_4 \\ W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + W_{24}^{(1)}x_4 \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix}$$

Equation for z^2

Now carefully observe the neural network illustration from above.



$$z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \end{bmatrix}$$

Input and Hidden_1 layers

You will see that z^2 can be expressed using $(z\_1)^2$ and $(z\_2)^2$ where $(z\_1)^2$ and $(z\_2)^2$ are the sums of the multiplication between every input x_i with the corresponding weight $(W\_ij)^1$. This leads to the same "*Equation for z^2*" and proofs that the matrix representations for z^2, a^2, z^3 and a^3 are correct.

Output layer

The final part of a neural network is the output layer which produces the predicated value. In our simple example, it is presented as a single neuron, colored in blue and evaluated as follows:

$$s = W^{(3)}a^{(3)}$$

Equation for output s

Again, we are using the matrix representation to simplify the equation. One can use the above techniques to understand the underlying logic.

1. Forward propagation and evaluation

The equations above form network's forward propagation. Here is a short overview:

$$x = a^{(1)} \quad Input\ layer$$

$$z^{(2)} = W^{(1)}x + b^{(1)} \quad neuron\ value\ at\ Hidden_1\ layer$$

$$a^{(2)} = f(z^{(2)}) \quad activation\ value\ at\ Hidden_1\ layer$$

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)} \quad neuron\ value\ at\ Hidden_2\ layer$$

$$a^{(3)} = f(z^{(3)}) \quad activation\ value\ at\ Hidden_2\ layer$$

$$s = W^{(3)}a^{(3)} \quad Output\ layer$$

Overview of forward propagation equations colored by layer

The final step in a forward pass is to evaluate the **predicted output** *s* against an **expected output** *y*.

The output *y* is part of the training dataset *(x, y)* where *x* is the input (as we saw in the previous section).

Evaluation between *s* and *y* happens through a **cost function**. This can be as simple as MSE (mean squared error) or more complex like cross-entropy.

We name this cost function *C* and denote it as follows:

$$C = cost(s, y)$$

Equation for cost function C were *cost* can be equal to MSE, cross-entropy or any other cost function.Based on *C*'s value, the model "knows" how much to adjust its parameters in order to get closer to the expected output *y*. This happens using the backpropagation algorithm.

1. Backpropagation and computing gradients

According to the paper from 1989, backpropagation: repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. And the ability to create useful new features distinguishes back-propagation from earlier, simpler methods…

In other words, **backpropagation aims to minimize the cost function by adjusting network's weights and biases.** The level of adjustment is determined by the gradients of the cost function with respect to those parameters.

One question may arise — **why computing gradients**?

To answer this, we first need to revisit some calculus terminology:

- *Gradient of a function C(x_1, x_2, ..., x_m) in point x is a vector of the partial derivatives of C in x.*

$$\frac{\partial C}{\partial x} = [\frac{\partial C}{\partial x_1}, \frac{\partial C}{\partial x_2}, \ldots, \frac{\partial C}{\partial x_m}]$$

Equation for derivative of C in x

- *The derivative of a function C measures the sensitivity to change of the function value (output value) with respect to a change in its argument x (input value). In other words, the derivative tells us the direction C is going.*

- *The gradient shows how much the parameter x needs to change (in positive or negative direction) to minimize C.*

Compute those gradients happens using a technique called chain rule.

For a single weight *(w_jk)^l*, the gradient is:

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} \qquad chain\ rule$$

$$z_j^l = \sum_{k=1}^{m} w_{jk}^l a_k^{l-1} + b_j^l \qquad by\ definition$$

$$m\ -\ number\ of\ neurons\ in\ l-1\ layer$$

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1} \qquad by\ differentiation\ (calculating\ derivative)$$

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}a_k^{l-1} \qquad final\ value$$

Equations for derivative of C in a single weight *(w_jk)^l*

Similar set of equations can be applied to *(b_j)^l*:

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} \qquad chain\ rule$$

$$\frac{\partial z_j^l}{\partial b_j^l} = 1 \qquad by\ differentiation\ (calculating\ derivative)$$

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}1 \qquad final\ value$$

Equations for derivative of C in a single bias *(b_j)^l* The common part in both equations is often called *"local gradient"* and is expressed as follows:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \qquad local\ gradient$$

Equation for local gradient

The *"local gradient"* can easily be determined using the chain rule. I won't go over the process now but if you have any questions, please comment below.

The gradients allow us to optimize the model's parameters:

while (termination condition not met)

$$w := w - \epsilon \frac{\partial C}{\partial w}$$

$$b := b - \epsilon \frac{\partial C}{\partial b}$$

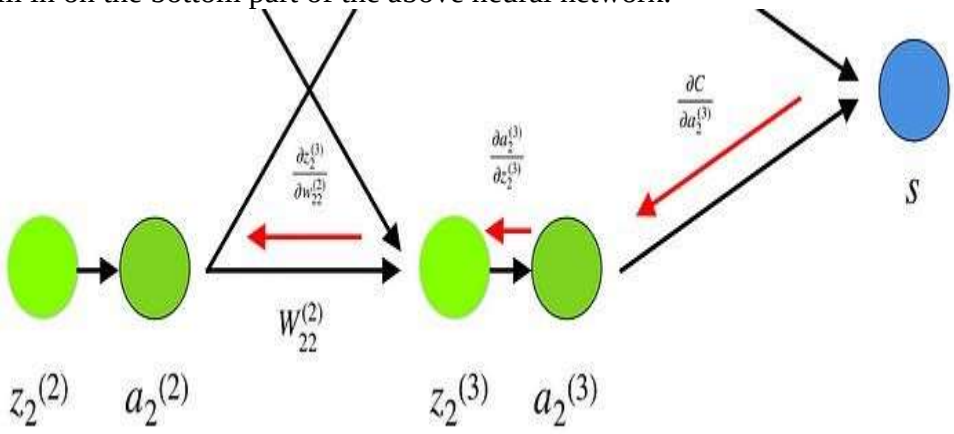
end

Algorithm for optimizing weights and biases (also called "Gradient descent")

- Initial values of *w* and *b* are randomly chosen.
- Epsilon (*e*) is the learning rate. It determines the gradient's influence.
- *w* and *b* are matrix representations of the weights and biases. Derivative of *C* in *w* or *b* can be calculated using partial derivatives of *C* in the individual weights or biases.
- Termination condition is met once the cost function is minimized.

I would like to dedicate the final part of this section to a simple example in which we will calculate the gradient of *C* with respect to a single weight *(w_22)²*.

Let's zoom in on the bottom part of the above neural network:



Visual representation of backpropagation in a neural network

Weight *(w_22)²* connects *(a_2)²* and *(z_2)²*, so computing the gradient requires applying the chain rule through *(z_2)³* and *(a_2)³:*

$$\frac{\partial C}{\partial w_{22}^{(2)}} = \frac{\partial C}{\partial z_2^{(3)}} \cdot \frac{\partial z_2^{(3)}}{\partial w_{22}^{(2)}} = \frac{\partial C}{\partial a_2^{(3)}} \cdot \frac{\partial a_2^{(3)}}{\partial z_2^{(3)}} \cdot a_2^{(2)} = \frac{\partial C}{\partial a_2^{(3)}} \cdot f'(z_2^{(3)}) \cdot a_2^{(2)}$$

Equation for derivative of C in $(w\_22)^2$

Calculating the final value of derivative of *C* in $(a\_2)^3$ requires knowledge of the function *C*. Since *C* is dependent on $(a\_2)^3$, calculating the derivative should be fairly straightforward.

CONCLUSION: We have studied and implemented Back Propagation Network for XOR function with Binary Input and Output

ASSIGNMENT 10

TITLE: HOPFIELD NETWORK

PROBLEM STATEMENT: -

. Write a python program to design a Hopfield Network which stores 4 vectors

OBJECTIVE:

4. To Learn and understand the concepts of types of neural network
5. To learn and understand back propagation network
6. To understand implementation of XOR function with binary input using python.

PREREQUISITE: -

3 Basic of Python Programming

4 Concept of Artificial Neural Network

THEORY:

The Hopfield Neural Networks, invented by Dr John J. Hopfield consists of one layer of 'n' fully connected recurrent neurons. It is generally used in performing auto-association and optimization tasks. It is calculated using a converging interactive process and it generates a different response than our normal neural nets.

It is a fully interconnected neural network where each unit is connected to every other unit. It behaves in a discrete manner, i.e. it gives finite distinct output, generally of two types:

- **Binary (0/1)**
- **Bipolar (-1/1)**

The weights associated with this network are symmetric in nature and have the following properties.

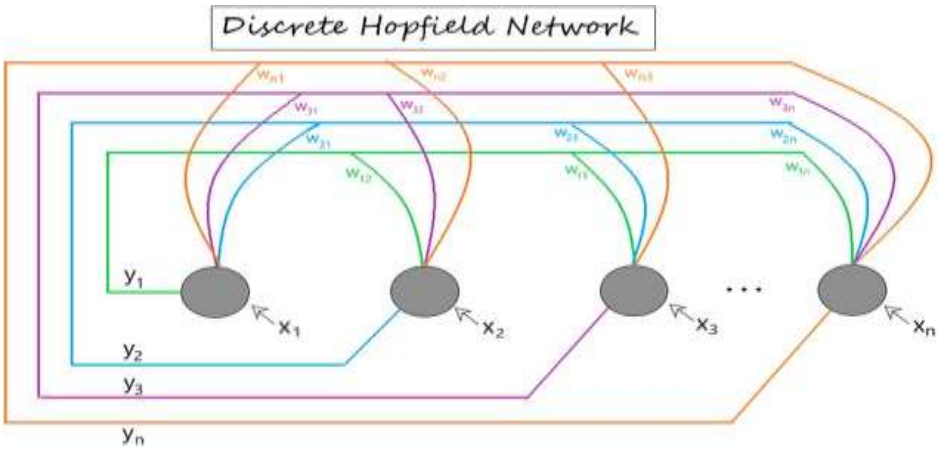
$$1.\ w_{ij} = w_{ji}$$
$$2.\ w_{ii} = 0$$

- **Structure & Architecture of Hopfield Network**

Each neuron has an inverting and a non-inverting output.

Being fully connected, the output of each neuron is an input to all other neurons but not the

self.The below figure shows a sample representation of a Discrete Hopfield Neural Network architecture having the following elements.



Discrete Hopfield Network Architecture

[x1 , x2 , … , xn] -> Input to the n given neurons.
[y1 , y2 , … , yn] -> Output obtained from the n given neurons
W_{ij} -> weight associated with the connection between the i^{th} and the j^{th} neuron.

Training Algorithm

For storing a set of input patterns S(p) [p = 1 to P], where **S(p) = S_1(p) … S_i(p) … S_n(p)**, the weight matrix is given by:

- **For binary patterns**

$$w_{ij} = \sum_{p=1}^{P}[2s_i(p) - 1][2s_j(p) - 1] \ (w_{ij} \ for \ all \ i \neq j)$$

- **For bipolar patterns**

$$w_{ij} = \sum_{p=1}^{P}[s_i(p)s_j(p)] \ (where \ w_{ij} = 0 \ for \ all \ i = j)$$

(i.e. weights here have no self-connection)

Steps Involved in the training of a Hopfield Network are as mapped below:

- Initialize weights **(w_{ij})** to store patterns **(using training algorithm)**.
- For each input vector y_i, perform **steps 3-7**.
- Make the initial activators of the network equal to the external input vector x.

$$y_i = x_i : (for \ i = 1 \ to \ n)$$

- For each vector y_i, perform **steps 5-7**.
- Calculate the total input of the network **y_{in}** using the equation given below.

$$y_{in_i} = x_i + \sum_j [y_j w_{ji}]$$

- Apply activation over the total input to calculate the output as per the equation given below:

$$y_i = \begin{cases} 1 \text{ if } y_{in} > \theta_i \\ y_i \text{ if } y_{in} = \theta_i \\ 0 \text{ if } y_{in} < \theta_i \end{cases}$$

(where θ_i (threshold) and is normally taken as 0)

- Now feedback the obtained output y_i to all other units. Thus, the activation vectors are updated.
- Test the network for convergence.

Continuous Hopfield Network

Unlike the discrete Hopfield networks, here the time parameter is treated as a continuous variable. So, instead of getting binary/bipolar outputs, we can obtain values that lie between 0 and 1. It can be used to solve constrained optimization and associative memory problems. The output is defined as:

$$v_i = g(u_i)$$

where,

- v_i = *output from the continuous hopfield network*
- u_i = *internal activity of a node in continuous hopfield network.*

CONCLUSION: We have studied and implemented Back Propagation Network for XOR function with Binary Input and Out.