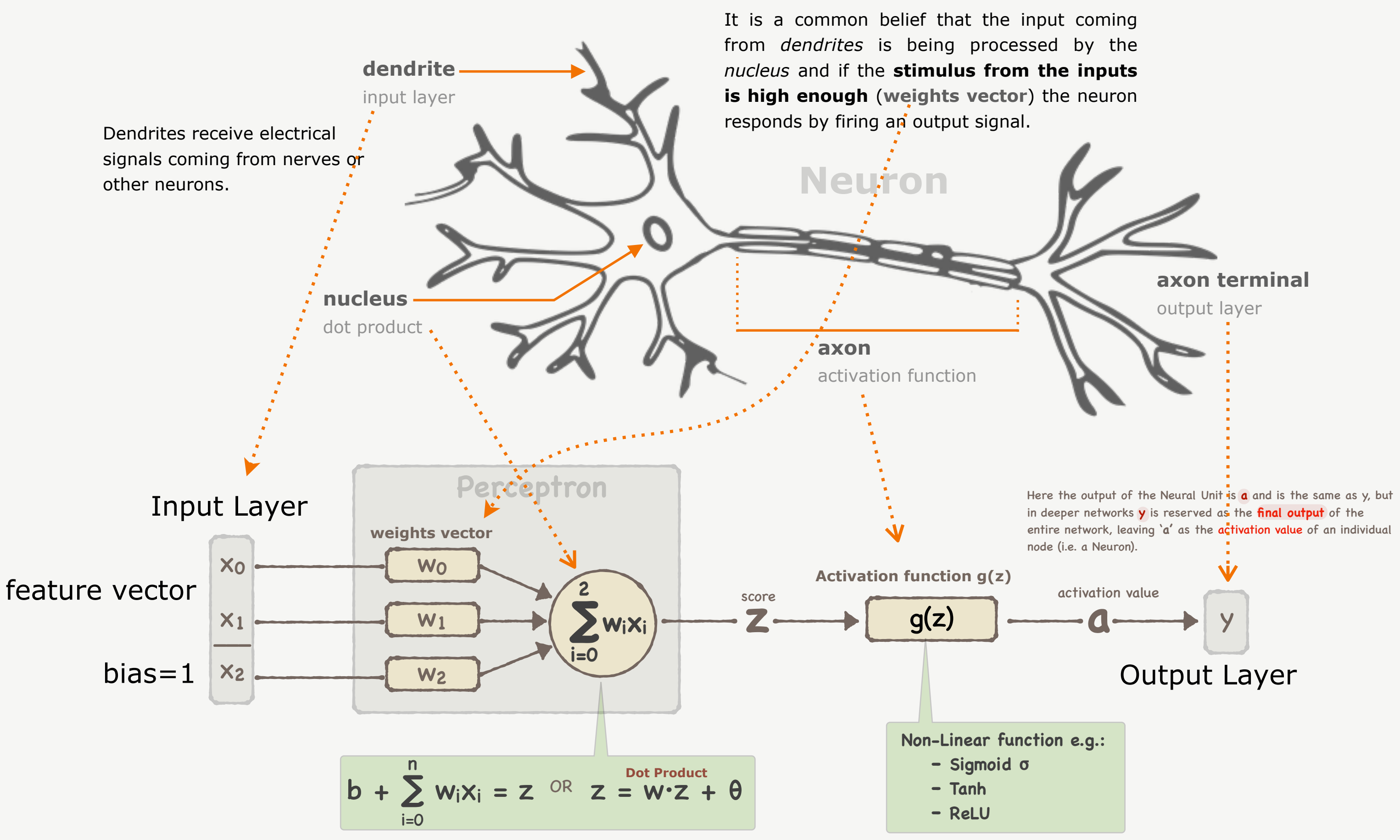
Modeling a Neuron

The human brain is a biological network made out of neurons. Each neuron is said to perform a very moderate cognitive function but collectively contributing to much larger cognitive tasks.



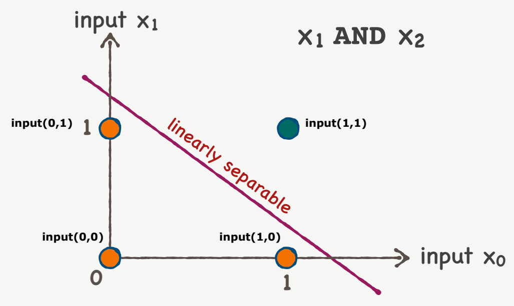

```cpp
//  Perceptron.hpp
//  Neural_Network_FeedForward
//
//  Created by Martin Gregory Sendrowicz on 4/26/23.
//

#ifndef _Perceptron_H_
#define _Perceptron_H_

#include <iostream>
#include <vector>
#include <numeric>            // std::inner_product()
#include <cmath>
#include <random>             // rand(), srand()
#include <time.h>             // time(0) i.e. the current time that seeds srand()
#include <algorithm>          // needed for all STL algorithms

enum class ACTIVATION{ Sigmoid, TanH, ReLu } ;

//||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
class Perceptron {

        friend class MultiLayerPerceptron ;

        // ALL values must be Real values (NO Integers)
        /* Input Layer i.e. the vector of features describing the input data. Note that
        input.size() will give you the number of features in the input vector excluding
        bias. */
        std::vector< double > input ;

        /* vector of weights --i.e. the vector containing values that the Neuron must learn on its
        own via backpropagation */
        std::vector< double > weights ;

        /* Activation Function: The 'activation_flag' determines which activation function to apply
        to the Dot Product. It will also signal the backpropagation algorithm which activation
        function was used (by the given neuron) so that the proper derivative can be computed */
        ACTIVATION activation_flag ;

        /* Output Layer: Neuron's output after passing via activation function. Note that each
        neuron outputs ONLY a single output value */
        double output ;

public:
        Perceptron( int input_size , ACTIVATION flag=ACTIVATION::Sigmoid ) ;

        double run( std::vector< double > features ) ;

        void set_weights( std::vector< double > init_vec ) ;
        std::vector<double> & get_weights() ;
        void print_weights() ;
};//||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

#endif /* Perceptron_hpp */
```

"A journey of a thousand miles begins with a single step" a Chinese proverb.

Building a neural network may seem like a task too large to overcome. When facing a difficult task, it is a sound strategy to break it down into its constituent components and tackle those instead. Let’s use our neuron model to act like a logic gate.

AND Gate

Given four points (0,0), (0,1), (1,0), (1,1) it is quite easy to separate them in a linear fashion i.e. using a straight line.

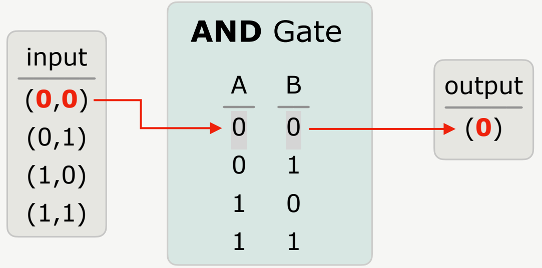


the AND Gate looks like this:

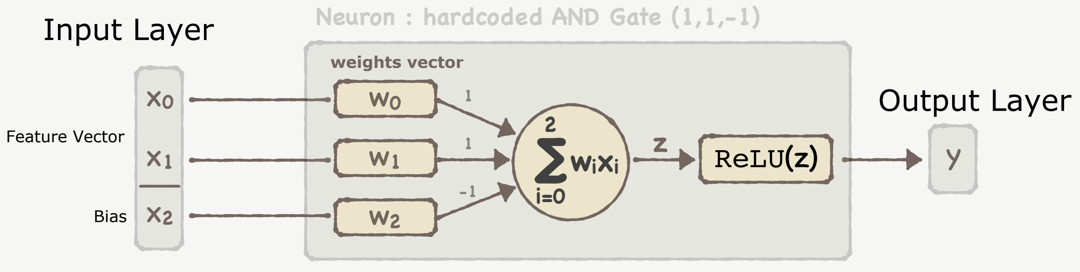
AND Gate

x_0	x_1	output
0	0	0
0	1	0
1	0	0
1	1	1

So, when inputing e.g. (0,0), one must expect that the Perceptron model (given the appropriate set of weights) will output (0).



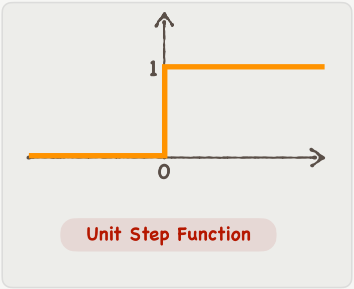
Let’s model a Perceptron model that does just that. Also let’s use ReLU function as model’s activation function.



Below is the working definition of **ReLU** and its derivative:

$$\mathrm{ReLU}(z) = \begin{cases} 0, & \text{if } z < 0 \\ z, & \text{otherwise} \end{cases}$$

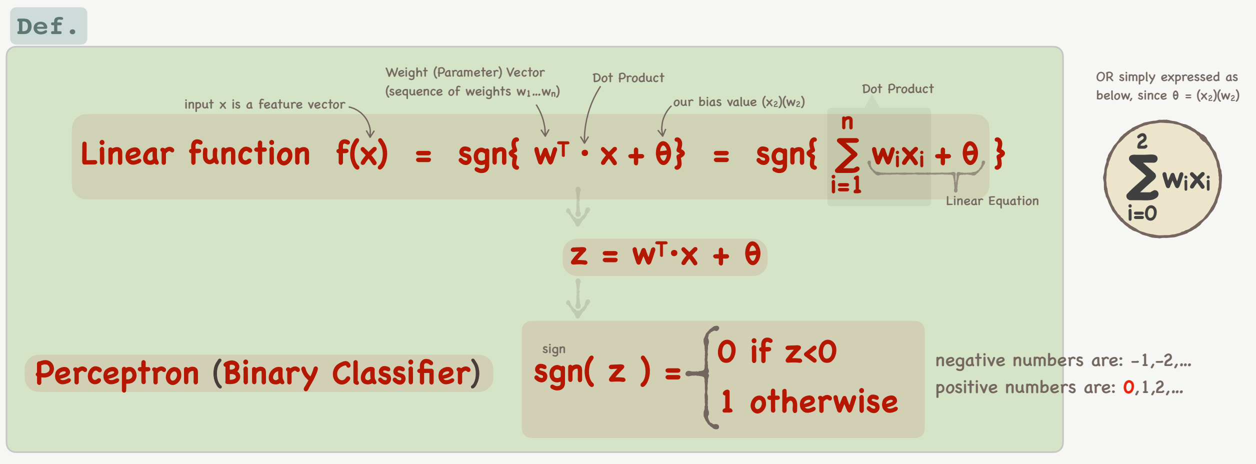
$$\frac{d}{dz}\mathrm{ReLU}(z) = \begin{cases} 0, & \text{if } z < 0 \\ 1, & \text{otherwise} \end{cases}$$



Note that the derivative of ReLU is the **unit step function** which is NOT a non-linear function—however, in our case, it does NOT have to be since our data is linearly separable. Let’s call our unit step function as ‘**sgn**’ (short for + or − sign).

The slope-intercept equation of a line with slope **m** and y-intercept **b** is: **y = mx + b**
where in our situation:

- **x** : is the input $\mathbf{x_i}$, for $0 \le i \le 1$
- **m** : is the weight $\mathbf{w_i}$ (applied to input $\mathbf{x_i}$)
- **b** : is the bias term $\mathbf{x_2}$
- **y** : is the decision boundary (i.e. the line that separates our input data)



First, I tested the model using the hardcoded set of weights that are guaranteed to produce the same output as the AND Gate, i.e. $(w_0,w_1,w_2) = (1,1,-1)$.

given input (x_0,x_1) and bias $x_2 = 1$

$(x_0,x_1,x_2)\cdot(w_0,w_1,w_2) = (x_0w_0)+(x_1w_1)+(x_2w_2) = z$ → **sgn(z)** = output
$(1,1,1)\cdot(1,1,-1) = (1+1+(-1)) = 1$ → **sgn(1)** = 1
$(1,0,1)\cdot(1,1,-1) = (1+0+(-1)) = 0$ → **sgn(0)** = 0
$(0,1,1)\cdot(1,1,-1) = (0+1+(-1)) = 0$ → **sgn(0)** = 0
$(0,0,1)\cdot(1,1,-1) = (0+0+(-1)) = -1$ → **sgn(-1)** = 0

Below is the program's output given the hardcoded weights (1,1,-1)

```
AND GATE:
0 AND 0 : 0
0 AND 1 : 0
1 AND 0 : 0
1 AND 1 : 1
Program ended with exit code: 0
```

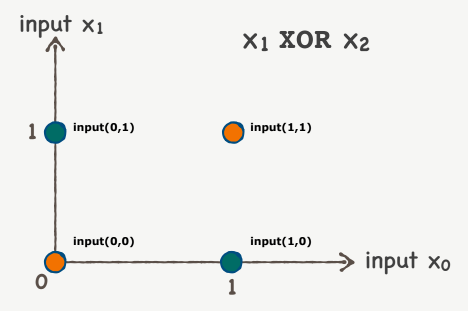
After running the network using backpropagation (i.e. making the model learn its own set of weights), it just happened so that the model, given ReLU as its activation function, learned the same exact set of weights as I provided in the hardcoded test above. I run the model for 100 epochs (iterations). Notice how rapidly the mean squared error (MSE) (i.e. the difference between the model’s output and the ground truth) is diminishing to ≈ zero.

```
Epoch: 0 MSE: 0.25
Epoch: 10 MSE: 0.00989249
Epoch: 20 MSE: 3.28619e-05
Epoch: 30 MSE: 1.04298e-07
Epoch: 40 MSE: 3.30755e-10
Epoch: 50 MSE: 1.04889e-12
Epoch: 60 MSE: 3.32627e-15
Epoch: 70 MSE: 1.05483e-17
Epoch: 80 MSE: 3.34509e-20
Epoch: 90 MSE: 1.06079e-22

Below are the trained/learned weights:
Layer-1 : Input Layer
Layer-2 : Neuron-1: [ 1 1 -1 ]

AND GATE:
0 AND 0 : 0
0 AND 1 : 0
1 AND 0 : 0
1 AND 1 : 1
Program ended with exit code: 0
```

The purpose of using non-linear activation functions is to be able to solve non-linearly separable problems. Simulating XOR Gate is such a task.

XOR Gate

Given four points (0,0), (0,1), (1,0), (1,1) it is NOT possible to separate them in a linear fashion i.e. using a straight line (go ahead try for yourself).

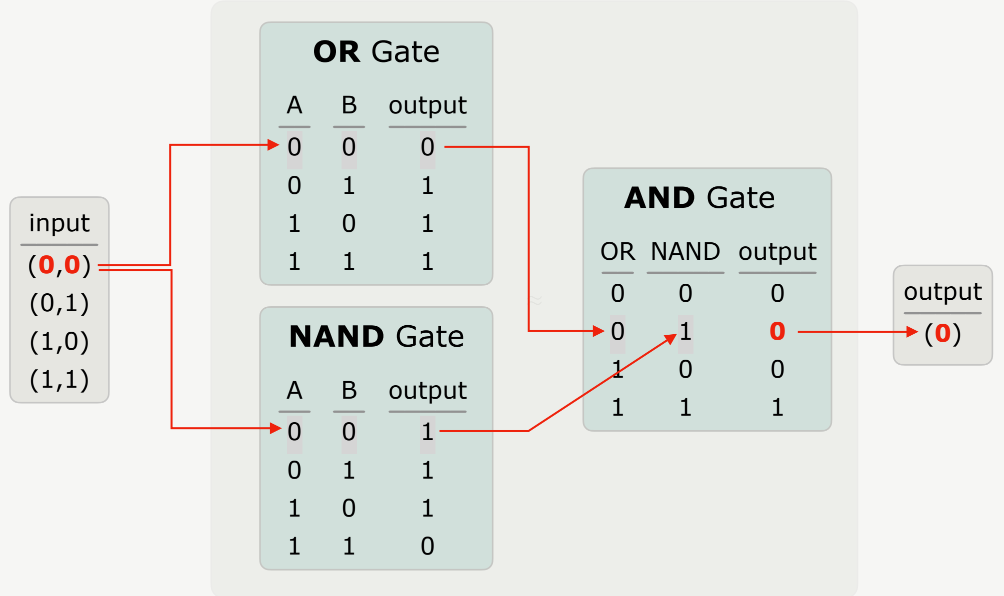


the XOR Gate looks like this:

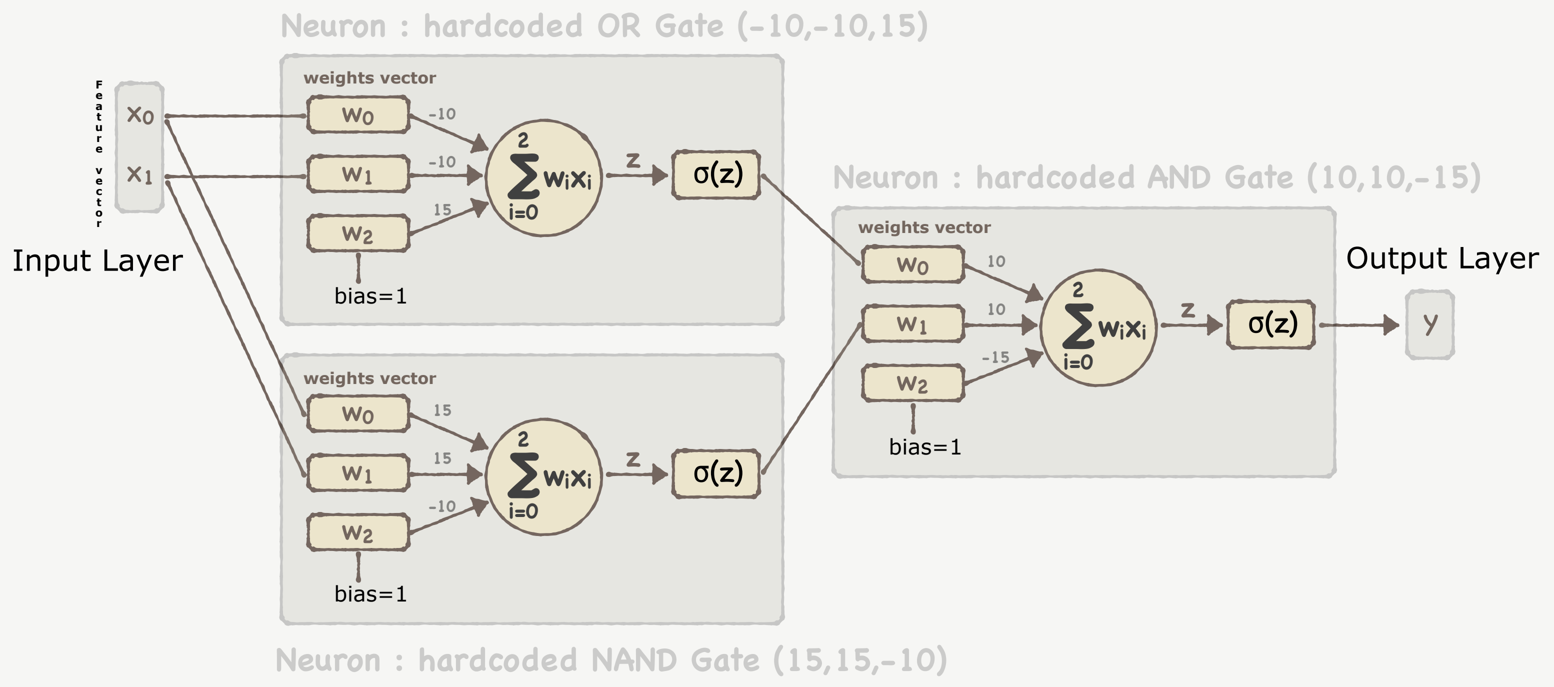
AND Gate

x_0	x_1	output
0	0	0
0	1	1
1	0	1
1	1	0

It is a common approach to implement the XOR gate as a composition of **OR**, **NAND** and **AND** gates. For example given input (0,0) we expect the output (given the appropriate set of weights for each gate) to be (0).



Let's model a Perceptron model to do just that. But this time, let's use either Sigmoid or TanH as model's activation function. YES—we'll need three neurons.



Given the hardcoded set of weights, the model did perform as expected:

```
Layer-1 : Input Layer
Layer-2 : Neuron-1: [ -10 -10 15 ]
Layer-2 : Neuron-2: [ 15 15 -10 ]
Layer-3 : Neuron-1: [ 10 10 -15 ]

XOR GATE:
0 XOR 0 : 0
0 XOR 1 : 1
1 XOR 0 : 1
1 XOR 1 : 0
```

Next I run the model for 3000 epochs using Sigmoid and 1000 epochs using TanH. Below are the result of training/learning the necessary weights; as well as the resulting output of the model while using those weights.

Sigmoid

```
Epoch: 0 MSE: 0.268227
Epoch: 100 MSE: 0.345708
Epoch: 200 MSE: 0.340668
Epoch: 300 MSE: 0.324038
Epoch: 400 MSE: 0.266766
Epoch: 500 MSE: 0.13656
Epoch: 600 MSE: 0.0553266
Epoch: 700 MSE: 0.0292137
Epoch: 800 MSE: 0.0187378
Epoch: 900 MSE: 0.0134525
Epoch: 1000 MSE: 0.0103572
Epoch: 1100 MSE: 0.00835549
Epoch: 1200 MSE: 0.00696778
Epoch: 1300 MSE: 0.00595526
Epoch: 1400 MSE: 0.00518714
Epoch: 1500 MSE: 0.00458628
Epoch: 1600 MSE: 0.00410452
Epoch: 1700 MSE: 0.00371034
Epoch: 1800 MSE: 0.0033823
Epoch: 1900 MSE: 0.00310536
Epoch: 2000 MSE: 0.00286866
Epoch: 2100 MSE: 0.00266419
Epoch: 2200 MSE: 0.00248588
Epoch: 2300 MSE: 0.00232912
Epoch: 2400 MSE: 0.00219028
Epoch: 2500 MSE: 0.0020665
Epoch: 2600 MSE: 0.00195551
Epoch: 2700 MSE: 0.00185545
Epoch: 2800 MSE: 0.0017648
Epoch: 2900 MSE: 0.00168232

Below are the trained/learned weights:
Layer-1 : Input Layer
Layer-2 : Neuron-1: [ 5.20853 -5.4265 -2.92406 ]
Layer-2 : Neuron-2: [ -5.66349 5.50986 -3.10858 ]
Layer-3 : Neuron-1: [ 8.09232 8.02519 -3.9763 ]

XOR GATE:
0 XOR 0 : 0
0 XOR 1 : 1
1 XOR 0 : 1
1 XOR 1 : 0
Program ended with exit code: 0
```

TanH

```
Epoch: 0 MSE: 0.895266
Epoch: 100 MSE: 0.0183869
Epoch: 200 MSE: 0.00350065
Epoch: 300 MSE: 0.0017806
Epoch: 400 MSE: 0.00225495
Epoch: 500 MSE: 0.00181779
Epoch: 600 MSE: 0.00076275
Epoch: 700 MSE: 0.000384027
Epoch: 800 MSE: 0.000291134
Epoch: 900 MSE: 0.000252203

Below are the trained/learned weights:
Layer-1 : Input Layer
Layer-2 : Neuron-1: [ -2.37207 -2.34574 0.882676 ]
Layer-2 : Neuron-2: [ 1.63743 1.63038 -2.14476 ]
Layer-3 : Neuron-1: [ -2.23538 -2.14497 -0.513896 ]

XOR GATE:
0 XOR 0 : 0
0 XOR 1 : 1
1 XOR 0 : 1
1 XOR 1 : 0
Program ended with exit code: 0
```

Using the ReLU whose derivative is NOT a non-linear function, the model could not correctly simulate the XOR gate.

ReLU

```
Epoch: 0 MSE: 0.447224
Epoch: 100 MSE: 0.533333
Epoch: 200 MSE: 0.533333
Epoch: 300 MSE: 0.533333
Epoch: 400 MSE: 0.533333
Epoch: 500 MSE: 0.533333
Epoch: 600 MSE: 0.533333
Epoch: 700 MSE: 0.533333
Epoch: 800 MSE: 0.533333
Epoch: 900 MSE: 0.533333
Epoch: 1000 MSE: 0.533333
Epoch: 1100 MSE: 0.533333
Epoch: 1200 MSE: 0.533333
Epoch: 1300 MSE: 0.533333
Epoch: 1400 MSE: 0.533333
Epoch: 1500 MSE: 0.533333
Epoch: 1600 MSE: 0.533333
Epoch: 1700 MSE: 0.533333
Epoch: 1800 MSE: 0.533333
Epoch: 1900 MSE: 0.533333
Epoch: 2000 MSE: 0.533333
Epoch: 2100 MSE: 0.533333
Epoch: 2200 MSE: 0.533333
Epoch: 2300 MSE: 0.533333
Epoch: 2400 MSE: 0.533333
Epoch: 2500 MSE: 0.533333
Epoch: 2600 MSE: 0.533333
Epoch: 2700 MSE: 0.533333
Epoch: 2800 MSE: 0.533333
Epoch: 2900 MSE: 0.533333

Below are the trained/learned weights:
Layer-1 : Input Layer
Layer-2 : Neuron-1: [ -42.1103 -1.06484 -0.00181035 ]
Layer-2 : Neuron-2: [ -104.48 -0.38306 -0.906524 ]
Layer-3 : Neuron-1: [ 0.0707896 0.177088 0.4 ]

XOR GATE:
0 XOR 0 : 0
0 XOR 1 : 0
1 XOR 0 : 0
1 XOR 1 : 0
Program ended with exit code: 0
```