

# MULTILAYER PERCEPTRON LEARNING

## 1. 프로그램 목적

기존의 AND gate, OR gate 의 경우 이전의 과제였던 1-layer perceptron 으로 모양 구분이 가능했지만 XOR gate, 도넛모양 구분의 경우에는 데이터를 linear seperate 하려면 기존의 1-layer perceptron 으로 불가능했다. 이 문제를 해결하기 위해 여러개의 레이어가 중첩된 multi-layer perceptron 으로 XOR gate 와 도넛모양을 구분할 수 있다. 입력값의 갯수, 출력값의 갯수, 은닉층의 갯수, 은닉층 노드의 갯수를 모두 변수입력을 통해 자유롭게 설계할 수 있는 다층 퍼셉트론을 통해 이전의 과제에 제시된 AND gate, OR gate 뿐만 아니라 XOR gate, 도넛모양 구분실험까지 구현할 수 있었다.

## 2. 실행환경

- OS 버전: macOS Mojave(10.14.6)
- IDE 환경: xcode version11.0(11A420a)
- C++ Standard Library: libc++(LLVM C++ Standard Library with C++11 Support)
- Octave 버전: GNU Octave version 4.4.1

## 3. 자료구조

다층 퍼셉트론을 구현하기 위해 필요한 층의 종류는 크게 3 가지가 있다. 입력층, 은닉층, 출력층이다. 세개의 층을 구현하기 위해 입력층, 은닉층, 출력층을 각각 구조체로 선언하였다. 각각의 층에서 구조체 멤버는 다음과 같다.

- 입력층: 출력값
- 은닉층: 델타값, 가중치, 바이어스, net, 출력값
- 출력층: 델타값, 가중치, 바이어스, net, 출력값, 목표값, err

위의 내용을 간단한 표로 그려보면 다음과 같다.

입력층(InputLayer)	은닉층(HiddenLayer)	출력층(OutputLayer)
출력값(output)	델타(delta)	델타(delta)
	가중치(weight)	가중치(weight)
	바이어스(bias)	바이어스(bias)
	net	net
	출력값(output)	출력값(output)
		목표값(target)
		error

위의 표는 은닉층이 1 개일 때만 해당하는데, 은닉층을 사용자의 지정에 의해 원하는 대로 늘리는 것이 프로그램의 목적이다. 따라서 은닉층을 구조체 배열을 통해서 원하는 대로 늘리면 다음과 같다.

입력층 (InputLayer)	은닉층(Hidden Layer)-1	은닉층(Hidden Layer)-2		은닉층(Hidden Layer)-n	출력층(Output Layer)	
출력값(output)	델타(delta)	델타(delta)	...	델타(delta)	델타(delta)	
	가중치(weight)	가중치(weight)		가중치(weight)	가중치(weight)	
	바이어스(bias)	바이어스(bias)		바이어스(bias)	바이어스(bias)	
	net	net		net	net	
	출력값(output)	출력값(output)		출력값(output)	출력값(output)	
				목표값(target)	목표값(target)	
				error	error	

구조체 배열을 통해 원하는 대로 레이어를 구성할 수 있게 되었다. 다음으로는 레이어별 구조체 멤버에 대해 볼 것이다. 레이어별 구조체 멤버들은 필요한 경우 행렬의 곱셈을 편하게 하기 위해 1 차원 배열도 2 차원 배열로 구성하였다.

- 입력층: 입력층의 출력값은 multilayer perceptron 의 입력값을 그대로 은닉층에 전달하는 역할만을 수행하므로 입력값의 갯수 = 입력층의 출력값의 갯수 = 입력층의 노드수이다. 따라서 입력값이 i 개 있다고 할 때 입력층의 출력값 배열은  $output[i][0]$  으로 정할 수 있다. 이를 표로 나타내면 다음과 같다.

입력층 (InputLayer)	
총 i개의 입력값 (i개의 입력층 노드 수)	output[0][0]
	output[1][0]
	output[2][0]
	output[3][0]
	...
	output[i-2][0]
	output[i-1][0]

- 은닉층: 입력층과 유사한 방법으로 구조체 멤버의 배열을 선언한다. 은닉층 구조체 멤버를 두가지 부류로 분류할 수 있다. 일차원 배열을 전치한 형태의 1 개의 열만 갖는 2 차원 배열과 2 개 이상의 열을 갖는 2 차원 배열이다.
- 델타, 바이어스, net, output: 은닉층의 노드수가 h 라고 할때 모두  $A[h][0]$ 의 2 차원 배열이다.

은닉층 (delta, bias, net, output)	
n개의 노드 수에 따른 배열	A[0][0]
	A[1][0]
	A[2][0]
	A[3][0]
	...
	A[n-2][0]
	A[n-1][0]

- 가중치: 현재 레이어의 노드수가 m, 이전 레이어의 노드 수가 n 이라고 할때  $w[m][n]$ 의 2 차원 배열이다. 가중치배열의 경우 이전 레이어의 노드수에 영향을 받기때문에 맨첫번째 은닉층의 가중치 배열은 다른 은닉층의 가중치 배열과 다르다.(맨 첫번째 은닉층은 n = 입력층의 노드수, 다른 은닉층의 경우 m = n = 은닉층의 노드 수)

은닉층 (weight)	
은닉층의 노드수 = m, 이전 레이어의 노드수 = n일 경우 가중치의 배열	w[0][0]
	w[1][0]
	w[2][0]
	w[3][0]
	...
	w[m-2][0]
	w[m-1][0]
	w[0][n-1]
...	
...	
w[m-2][n-1]	
w[m-1][n-1]	

- 출력층: 은닉층과 같은 방법으로 구조체 멤버의 배열을 선언한다. Error의 경우 1 개의 값만 나오는 값이므로 배열 보다는 변수로 선언해준다.

## 4. 프로그램 OVERVIEW

메인함수에서 training 할 input 배열, target 배열을 선언해준다., 필요한 입력층, 은닉층, 출력층의 노드 수와 은닉층의 갯수를 사용자의 입력을 받아 설정해주고 tolerance 또한 설정해준다. 위 내용을 통해 모든 입력층, 은닉층, 출력층을 통해 output, error 를 구한다(feedfoward). error 를 바탕으로 역전파를 시행하면서 가중치를 업데이트 시킨다(backpropagation). Feedforward 와 backpropagation 을 무한 반복하면서 에러가 일정 수준에 도달하면 프로그램 실행을 중지하고 결과를 출력한다.

프로그램의 에러를 나타내는 방법에는 다양한 방법이 있지만 이 프로그램에서는 Mean Squared Error 를 사용하였다.

활성함수는 Sigmoid 를 사용하였다. 이 프로그램을 실행하면서 vanishing gradient 문제가 발생하지 않았기 때문에 Sigmoid 를 그대로 사용하였다.

## 5. 필요한 초기 입력값

### A. 가중치, 바이어스

가중치의 경우 -1 과 1 사이의 랜덤값으로 초기화 하였고 바이어스의 경우 learning 과정을 조금 더 빠르게 하기 위해 0 과 1 사이의 값으로 초기화 하였다.

## B. Tolerance

Mean Squared Error 의 tolerance 값은 0.00001 로 설정하였다. 0.00001 로 설정할 경우 훈련을 마친 후 target 과 output 값의 차이가 보통 0.0044 정도이다. 입력값을 linearly seperable 하기에 충분한 정도의 에러값을 제공했다.

# 6. 클래스 및 함수 설명

## A. 클래스

multiLayerPerceptron 의 이름을 갖는 단일 클래스로 구성하였다. 입력층, 은닉층, 출력층의 구조체 배열을 쉽게 접근하기 위함이다.

## B. 함수

- i. activationFunc: net 값을 인자로 전달할 경우 활성함수를 거쳐서 output 값을 반환해주는 함수이다.
- ii. def\_activationFunc: input 값을 함수의 인자로 전달할 경우 활성함수에서의 미분값을 반환하는 함수이다.
- iii. mulMatrix: 두개의 행렬을 인자로 전달 받은 후 행렬의 곱셈을 수행하고 결과를 반환하는 함수이다.
- iv. netCalc: input, weight 을 인자로 전달 받은 후 mulMatrix 함수를 통해 net 값을 반환하는 함수이다.
- v. outputCalc: net 값을 입력받아 활성함수를 통과한 후 output 값을 반환하는 함수이다.
- vi. errCalc: 출력층에서의 output 값, target 값을 입력받아 Mean Squared Error 를 계산하는 함수이다.
- vii. updateWeight: 업데이트하려는 가중치값, 입력값, 델타값을 입력받아 가중치를 업데이트하는 함수이다.
- viii. UpdateBias: 업데이트하려는 바이어스, 델타값을 입력받아 바이어스를 업데이트하는 함수이다.

- ix. feedForward: 은닉층의 갯수를 함수의 인자로 입력 받는다. 모든 층의 net, output 을 계산 한 후 출력층의 error 까지 계산하는 함수이다.
- x. backPropagation: 은닉층의 갯수를 함수의 인자로 입력 받는다. 일단 모든 층의 델타를 역전파 방식으로 업데이트 한 후, 가중치를 차례대로 업데이트 한다.
- xi. Learning: main 함수와의 유일한 접점이다. 입력값, 목표값, 모든 층의 차원, 은닉층의 갯수, tolerance 를 입력으로 받은 후 feedForward 와 backPropagation 을 수행하는 함수이다.

## 7. 파일의 출력

파일의 출력은 weightTxt() 함수로 이루어 진다. 원하는 출력 위치에서 함수를 사용하며 인자로 레이어의 구조체 배열을 받는다. 레이어의 갯수마다 각 레이어의 가중치, 바이어스, 출력값의 행렬을 텍스트파일에 입력한 후 텍스트 파일을 생성한다.

이름	▲	수정일	크기	종류
1번째 히든레이어의 weight,output_학습전		오늘 오후 4:22	119바이트	텍스트
1번째 히든레이어의 weight,output_학습중		오늘 오후 4:22	188바이트	텍스트
1번째 히든레이어의 weight,output_학습후		오늘 오후 4:22	184바이트	텍스트
2번째 히든레이어의 weight,output_학습전		오늘 오후 4:22	217바이트	텍스트
2번째 히든레이어의 weight,output_학습중		오늘 오후 4:22	329바이트	텍스트
2번째 히든레이어의 weight,output_학습후		오늘 오후 4:22	315바이트	텍스트
3번째 히든레이어의 weight,output_학습전		오늘 오후 4:22	217바이트	텍스트
3번째 히든레이어의 weight,output_학습중		오늘 오후 4:22	335바이트	텍스트
3번째 히든레이어의 weight,output_학습후		오늘 오후 4:22	321바이트	텍스트
ai_finished_file		오늘 오후 4:22	297KB	Unix 파일
error.txt		오늘 오후 4:22	427KB	일반 텍스트

위와 같이 텍스트 파일이 생성된다. 한 파일을 예시로 들어보면

```
3번째 히든레이어의 weight,output_학습후
-1.35789 1.42827 -2.48757 -1.5482 0.170492 0.675544 0.0953624
-0.643909 -0.148945 -0.489235 -1.13194 -0.90778 -0.856608 0.0576331
-3.57481 2.02061 -4.09808 -2.15681 1.85896 2.05975 0.0487848
-2.67709 1.26186 -2.49995 -0.586483 0.903568 0.755962 0.0914373
-0.211111 -1.14905 -1.13511 -0.77319 -1.05824 -0.442031 0.0528238
```

이 파일의 구조는

	w1	...	wn	bias	output
노드1					
...					
노드k					

위와 같다.

## 8. 그래프 작성

위의 텍스트 파일을 받아서 GNU Octave 로 그래프를 그렸다. 이를 위해 작성한 함수는 총 3 개이다.

- error\_plot.m: iteration 별 에러값을 그래프로 그렸다.

```
1 clear all
2 data = load("error.txt")
3 [m,n] = size(data)
4
5 err = data(:,1)
6
7 plot(err,'lineWidth',2)
8
9 xlabel('iteration')
10 ylabel('error')
11 set(gca, 'FontSize', 30)
12
13
```

- weight\_plot\_for.m: 가중치에 대한 직선을 그래프로 그리는 함수이다.

```

1 clear all
2 data = load("*.txt")
3 [m,n] = size(data)
4
5 w1 = data(:,1)
6 w2 = data(:,2)
7 bias = data(:,3)
8
9 for i=1:m
10    x = [-1:0.01:2]
11    y = (-w1(i)*x-bias(i))/w2(i)
12    plot(x,y,'lineWidth',2);
13    hold on
14    xlabel('X1')
15    ylabel('X2')
16    set(gca, 'FontSize', 30)
17 end
18
19 axis([-0.5 1.5 -0.5 1.5])
20 grid on
21
22 plot(0,0,'-s','MarkerSize',10,'MarkerFaceColor','red','lineWidth',0.00001)
23 hold on
24 plot(0,1,'-s','MarkerSize',10,'MarkerFaceColor','red','lineWidth',0.00001)
25 hold on
26 plot(1,0,'-s','MarkerSize',10,'MarkerFaceColor','red','lineWidth',0.00001)
27 hold on
28 plot(1,1,'-s','MarkerSize',10,'MarkerFaceColor','red','lineWidth',0.00001)
29 hold on
30

```

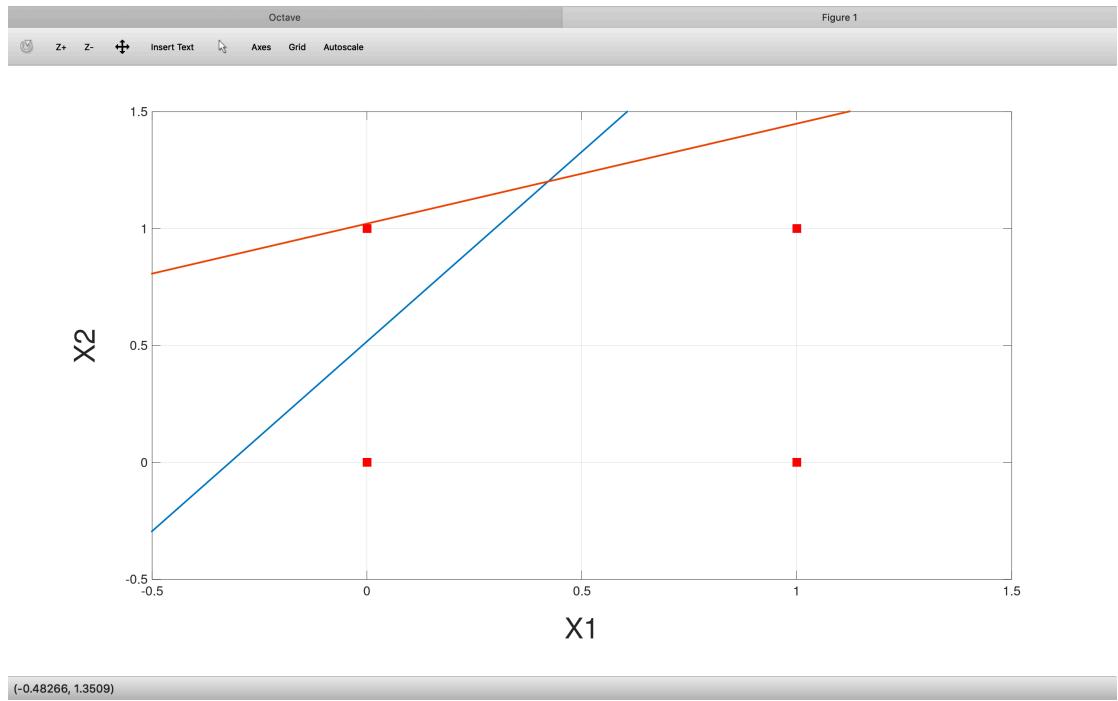
## 9. 모양구분 실험

아래의 모양구분 실험은 학습과정을 잘 드러내기 위해서 입력층노드 2 개, 은닉층 1 개, 은닉층 노드 2 개, 출력층 1 개, 출력층 노드 1 개로 구성하였다.

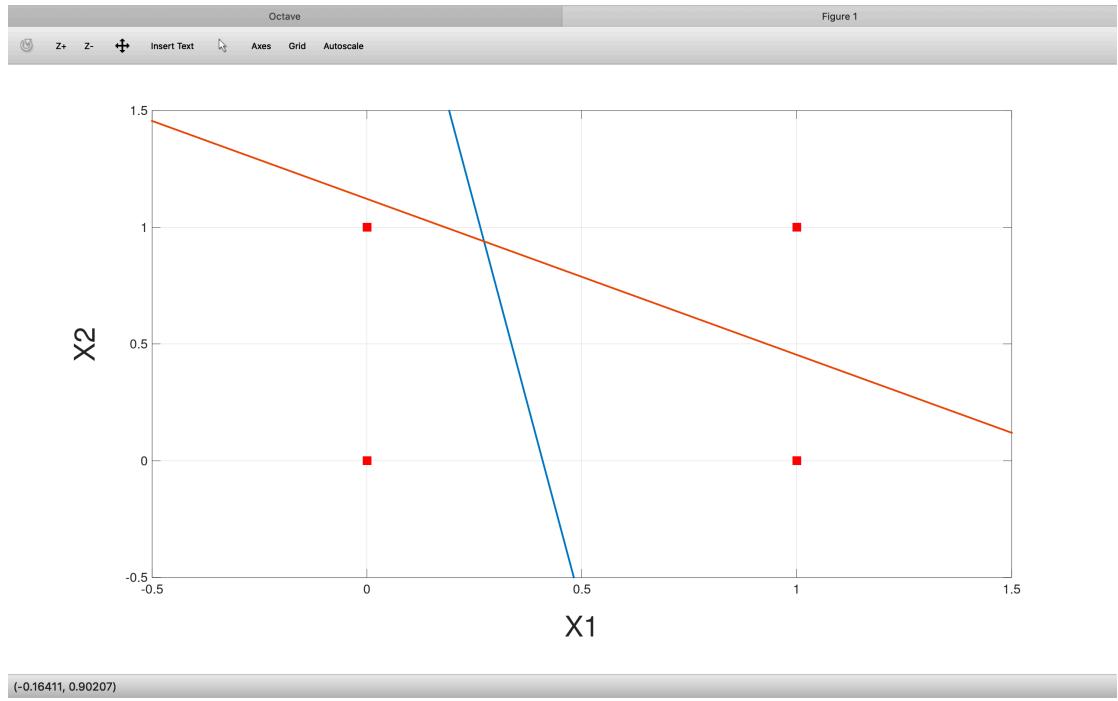
에러를 구할때 각 입력값 별로 mean squared error 를 계산하였으며  
에러그래프는 입력값의 갯수에 따른 평균을 그렸다.

## A. AND gate

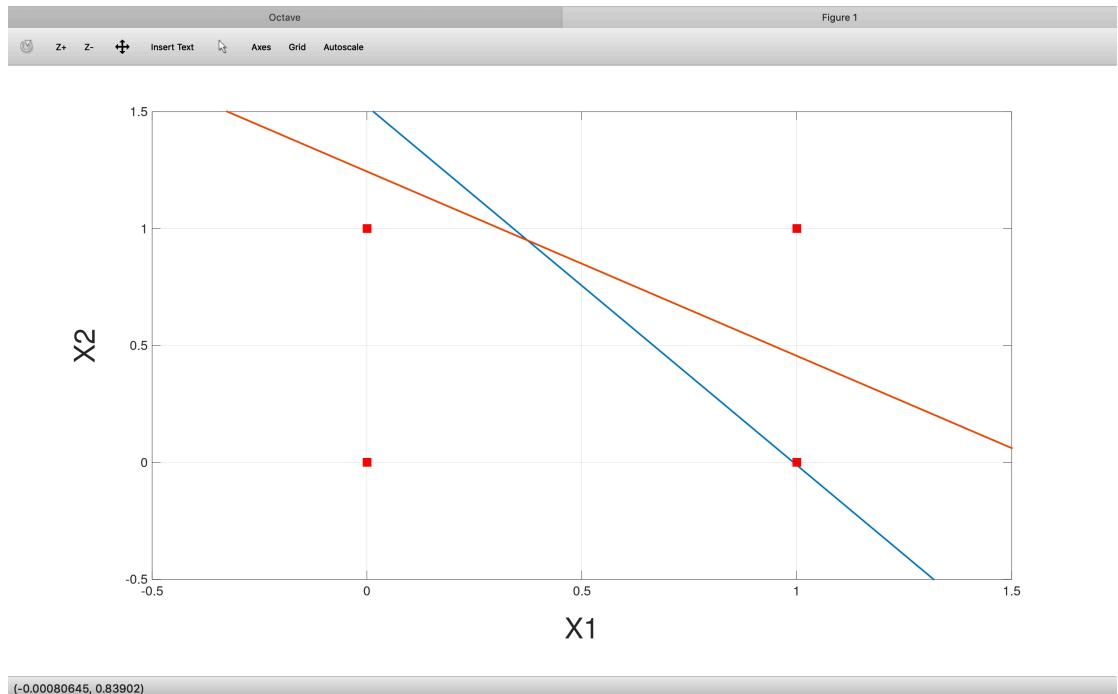
### i. 학습 전 초기값



### ii. 학습 중



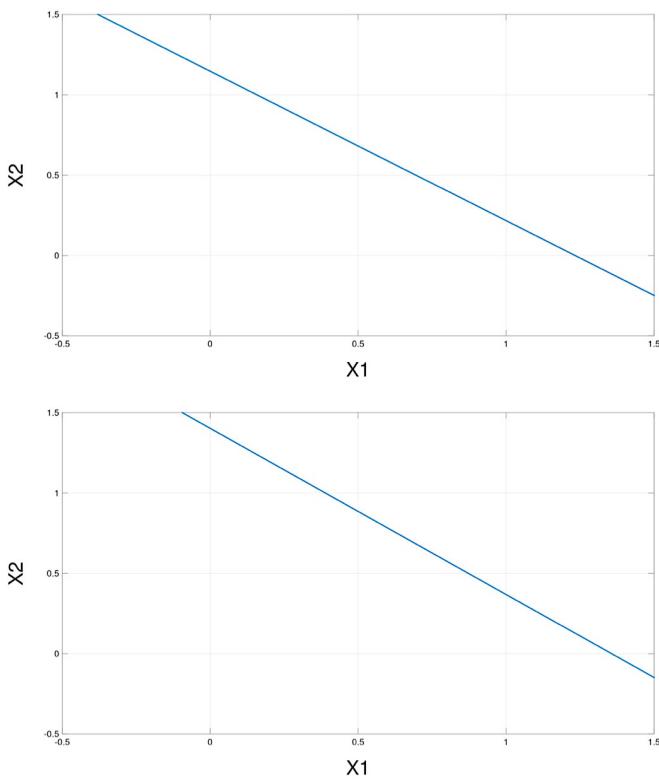
### iii. 학습 후



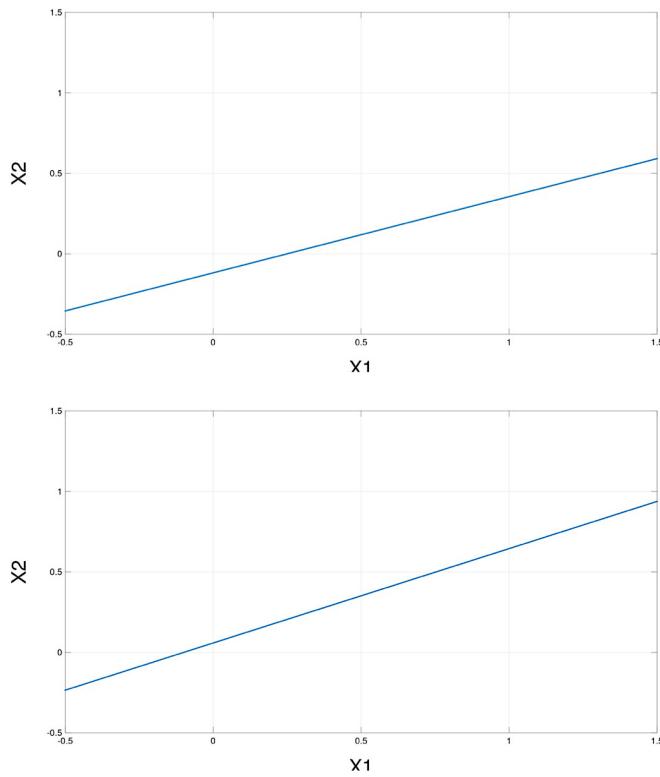
$(0,0), (0,1), (1,0)$ 점과  $(1,1)$ 점이 명확하게 구분된것을 볼수 있다.

다시 AND 게이트 구분을 보다 많은 레이어(은닉층 3 개, 각각의 노드 2 개)를 설정하고 프로그램을 실행하였을때 학습이 모두 끝난 후 각 노드마다의 직선은 다음과 같다.

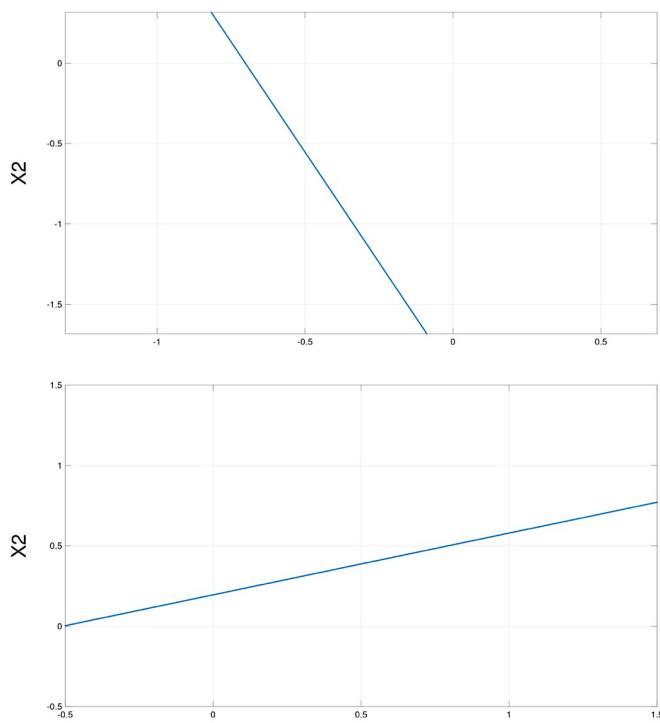
#### 1. 은닉층 첫번째 레이어의 노드별 직선



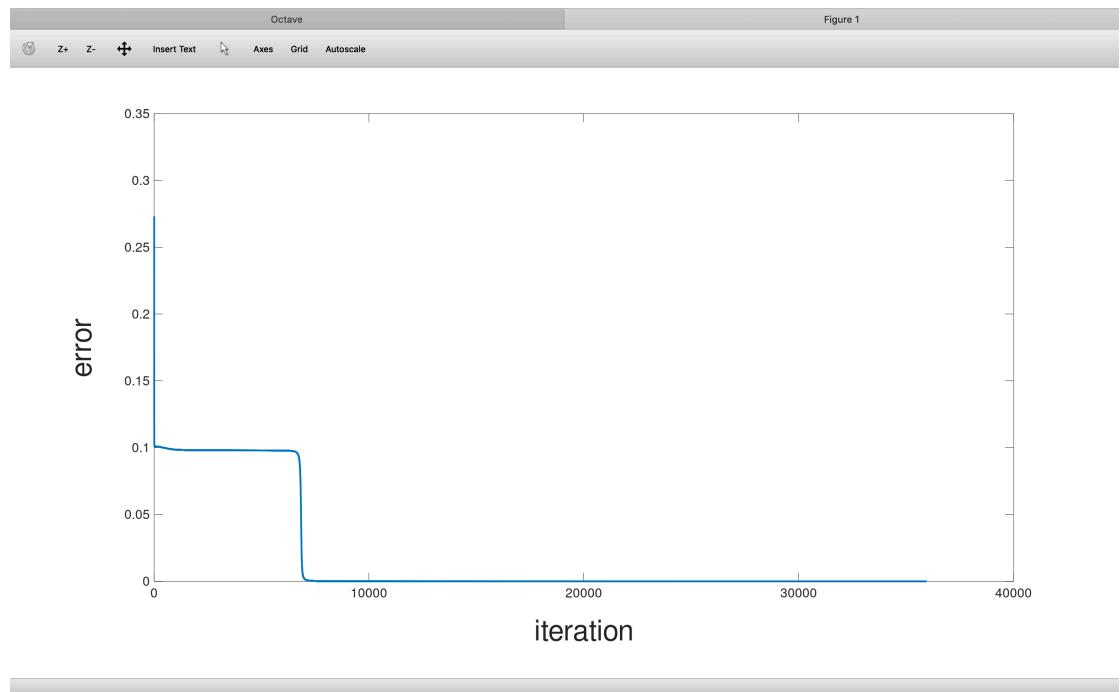
## 2. 은닉층 2 번째 레이어의 노드별 직선



## 3. 은닉층의 3 번째 레이어의 노드별 직선

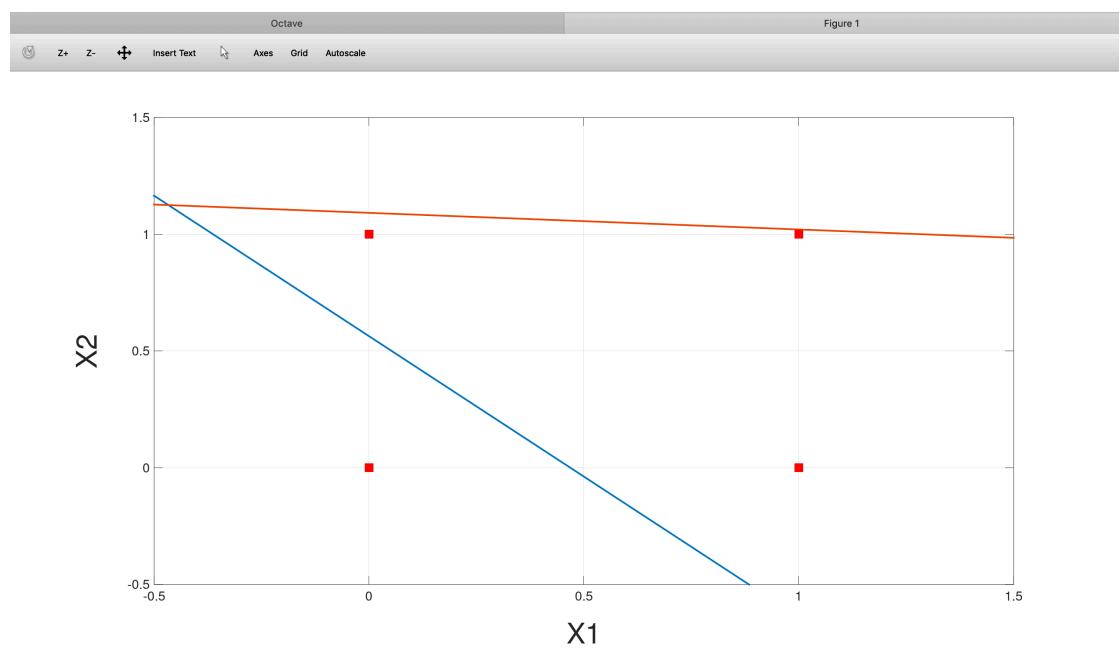


iv. 예제

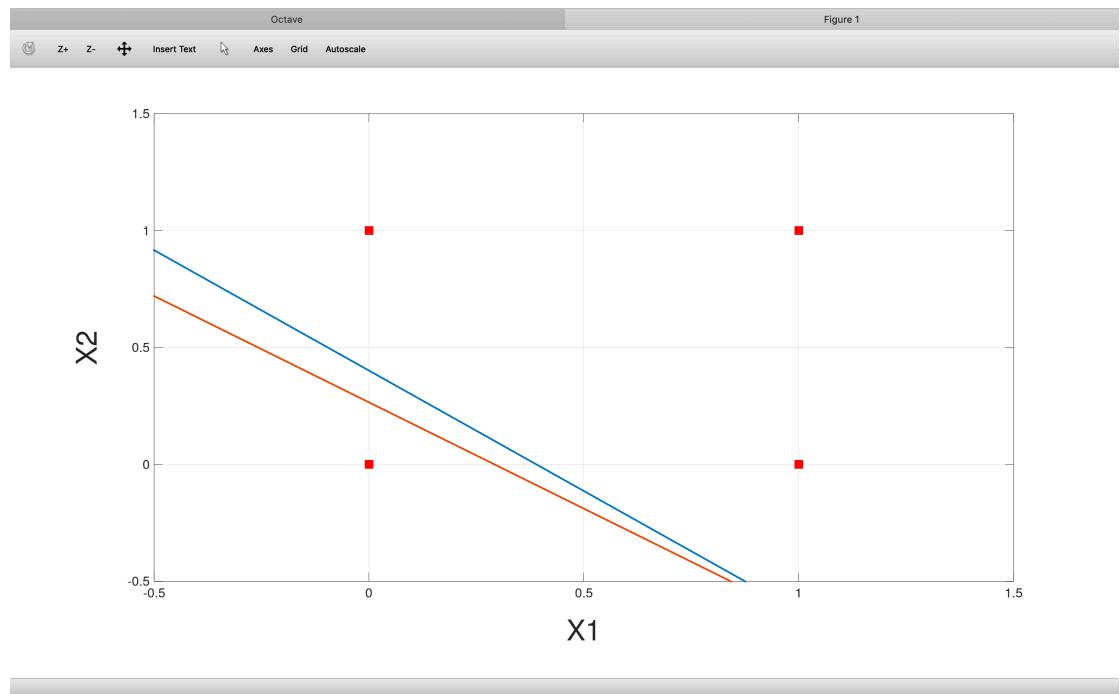


B. OR gate

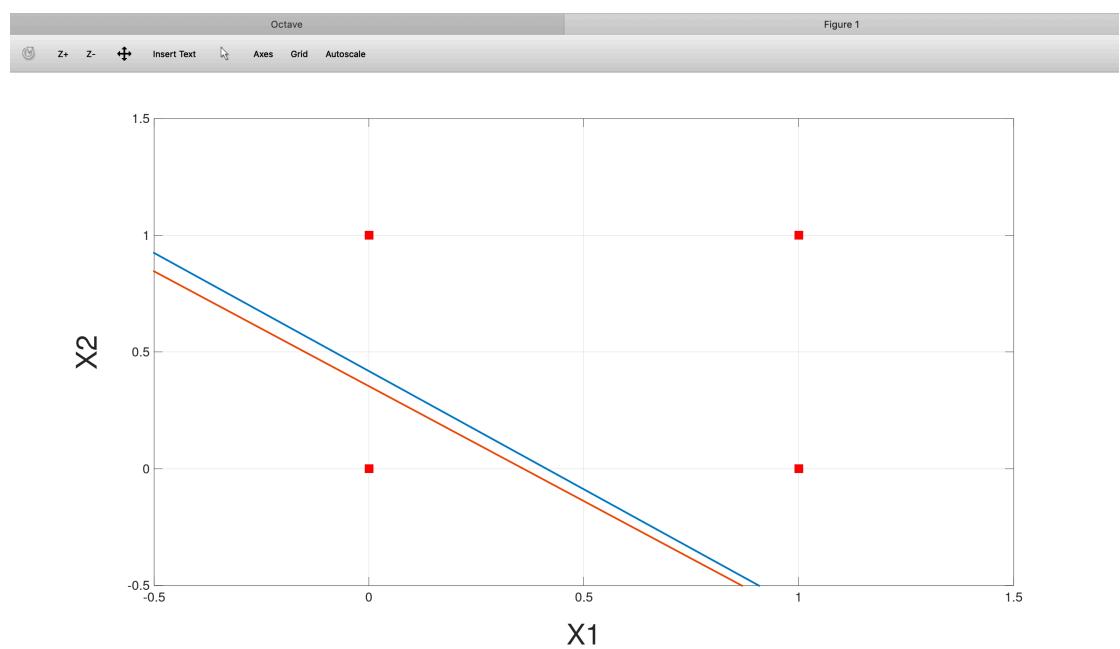
i. 학습 전



## ii. 학습 중

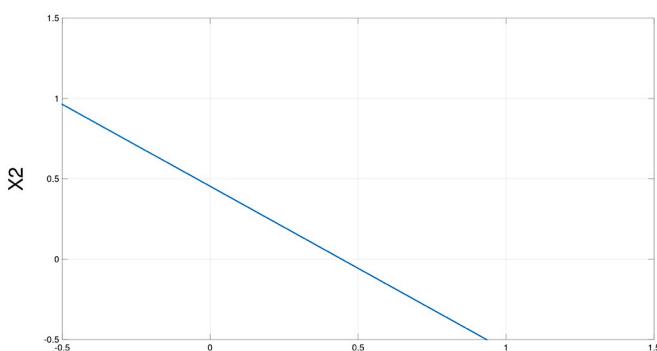
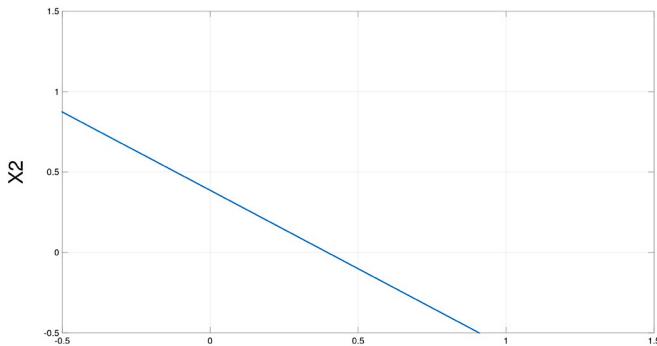


## iii. 학습 후

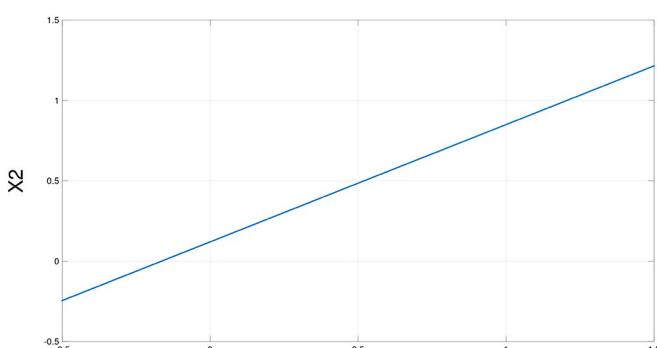
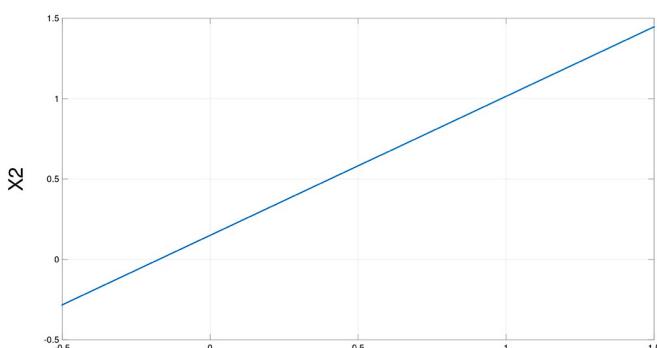


(0,0)과 (0,1), (1,0), (1,1)을 직선으로 명확하게 구분 성공하였다. OR 게이트  
역시 은닉층을 2 개 더 늘린 후 노드 별 직선을 그리면 다음과 같다.

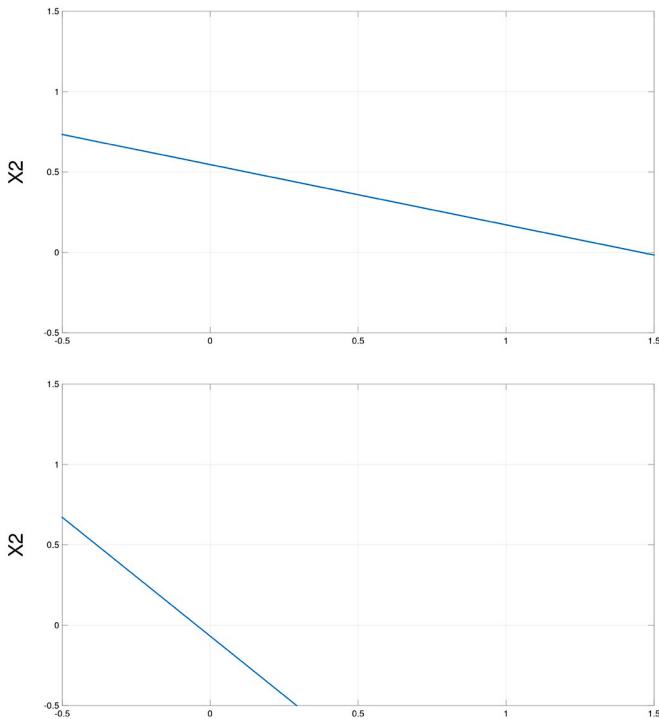
## 1. 은닉층의 첫번째 레이어의 노드별 직선



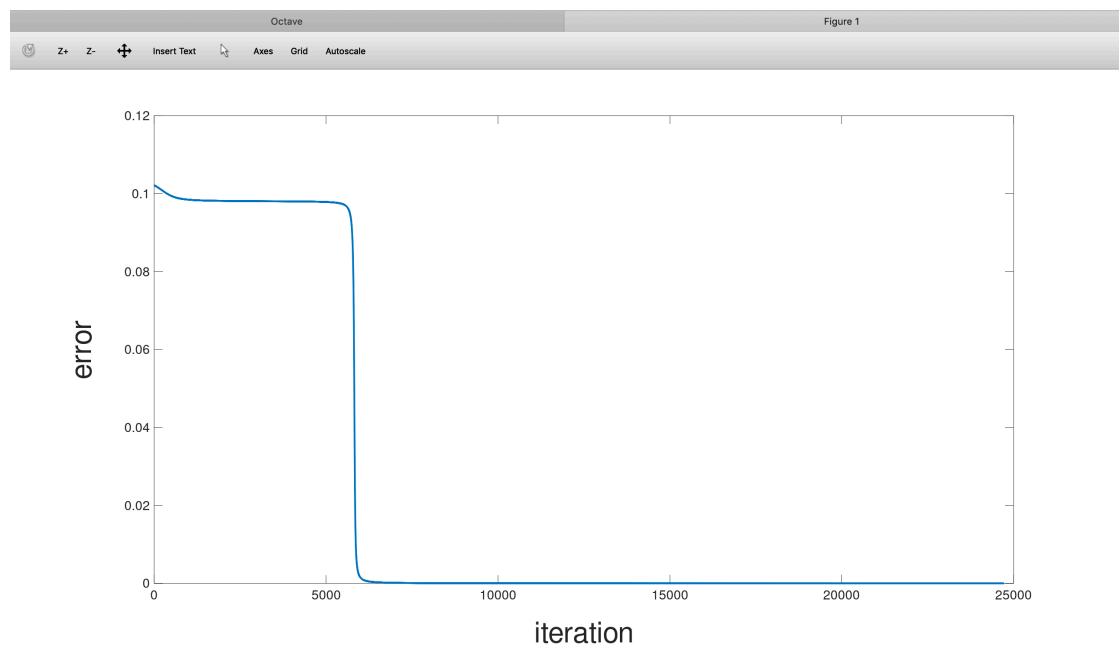
## 2. 은닉층의 두번째 레이어의 노드별 직선



### 3. 은닉층의 세 번째 레이어의 노드별 직선

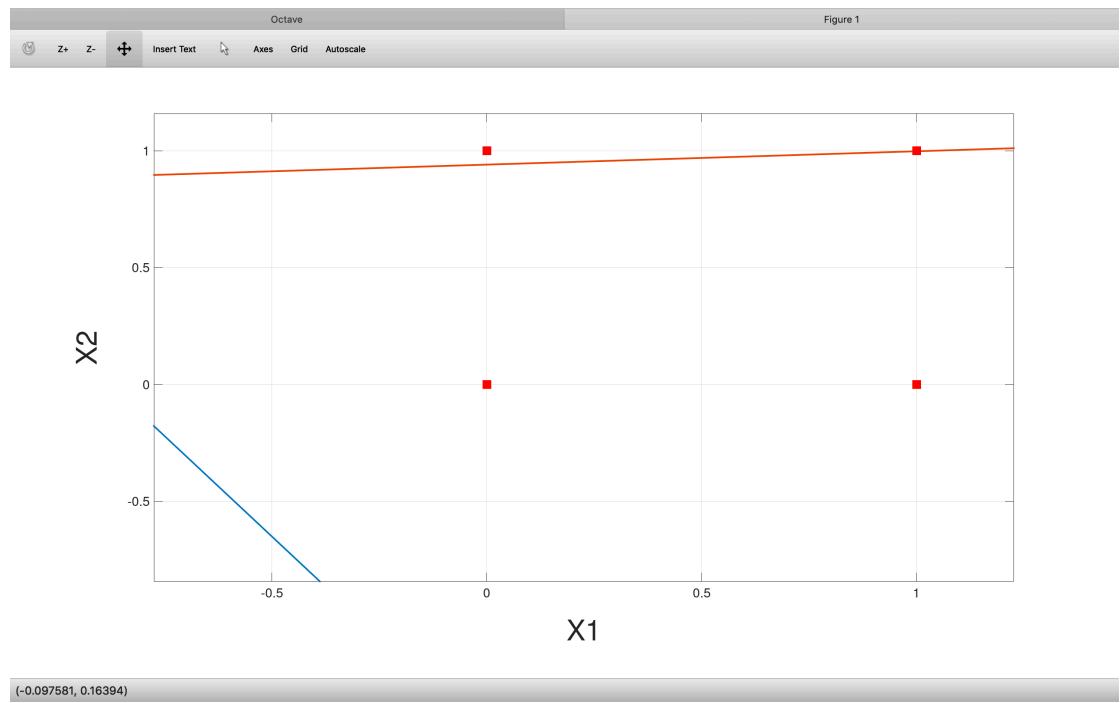


#### iv. 예제

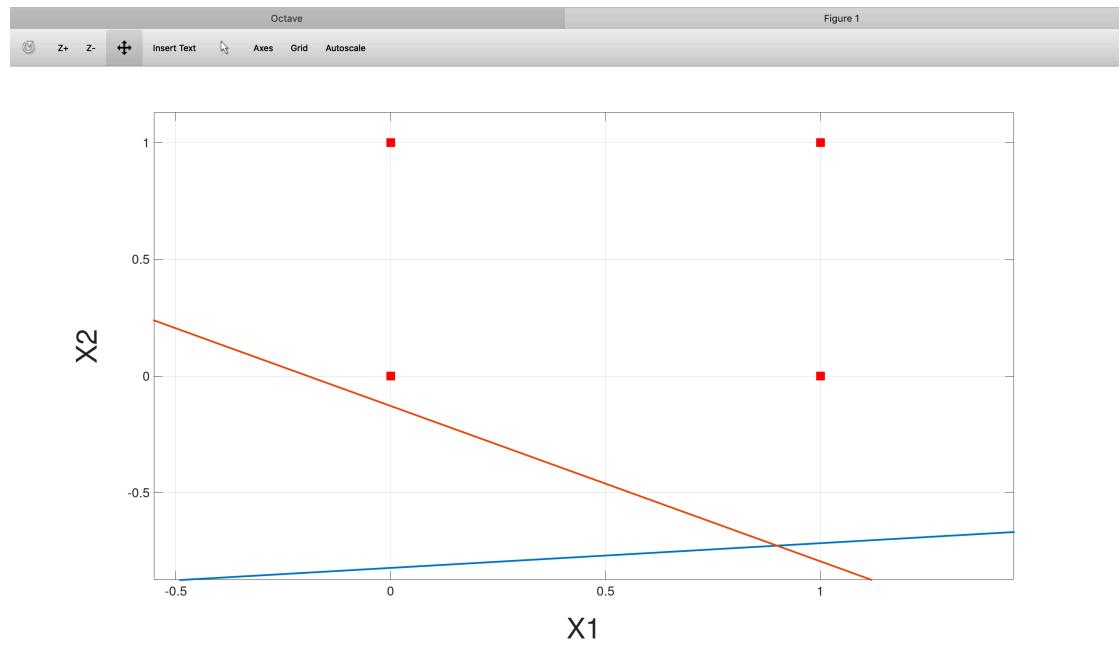


### C. XOR gate

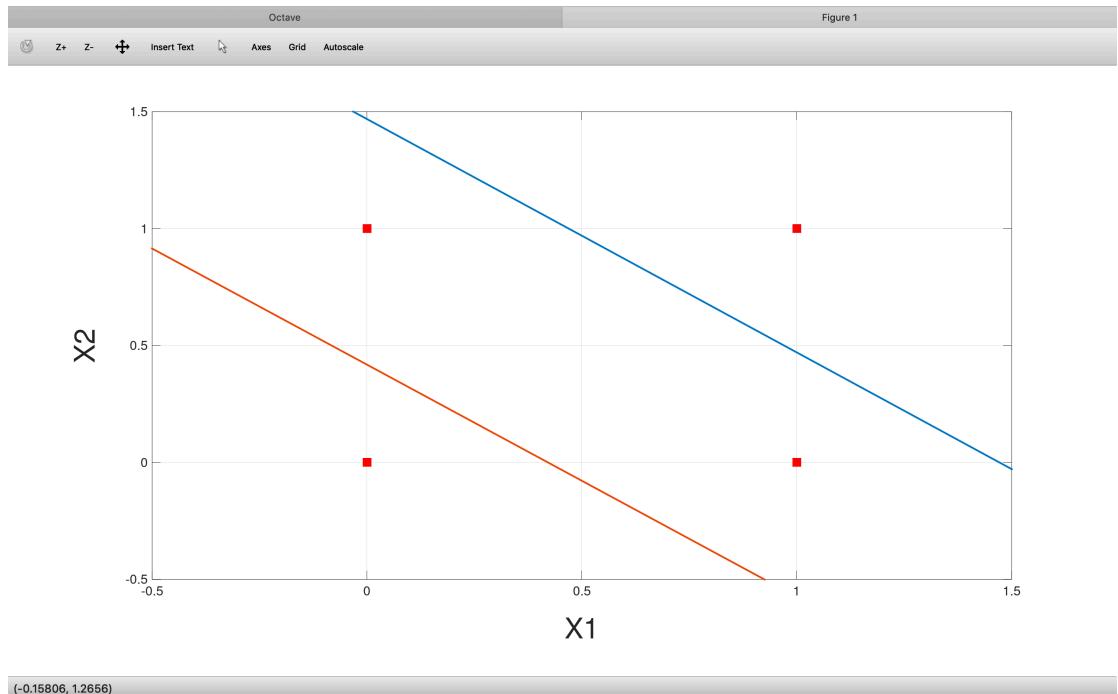
#### i. 학습 전



#### ii. 학습 중

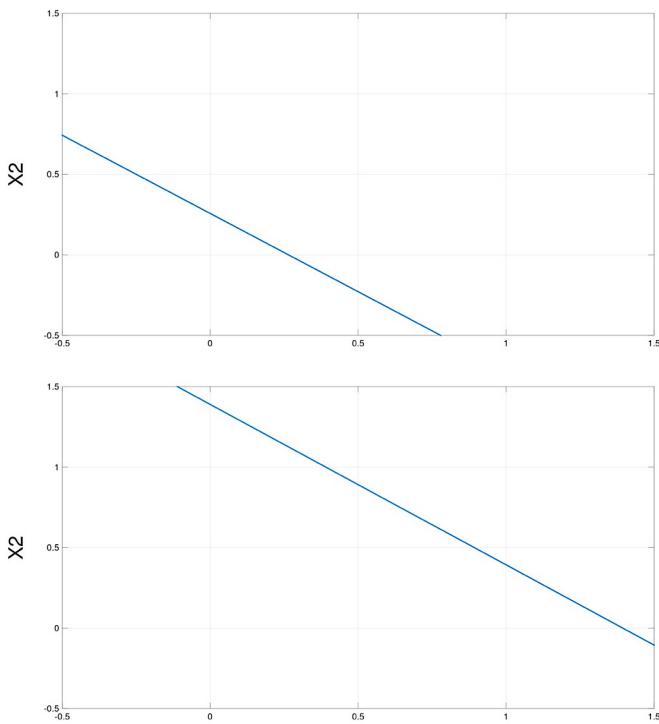


### iii. 학습 후

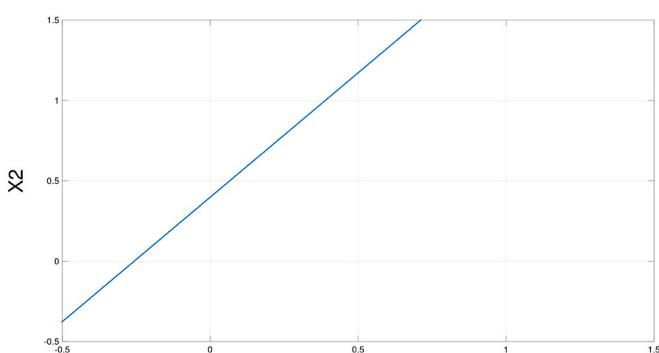
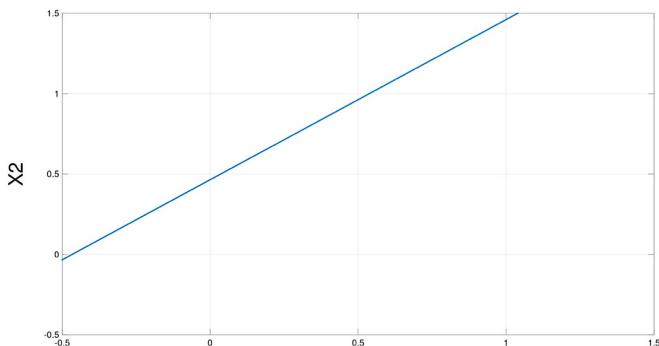


1- layer perceptron 으로는 해결 할 수 없었던 xor gate 구분도 multi-layer perceptron 으로 거뜬하게 구분할 수 있다. 학습 후 결과를 보면  $(0,0)$  과  $(0,1)$ ,  $(1,0)$  과  $(1,1)$ 을 직선 2 개로 구분하였다. 레이어를 2 개 더 늘려서 노드별 가중치의 직선을 그려보면 다음과 같다.

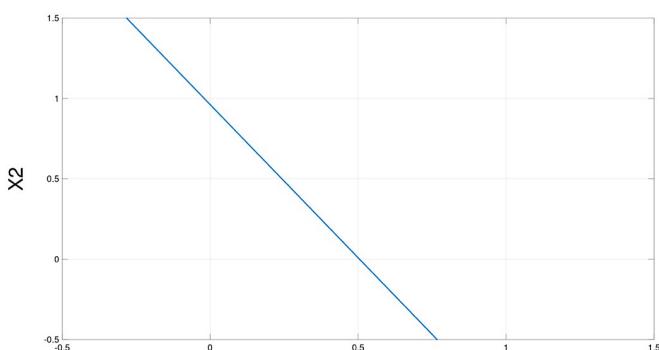
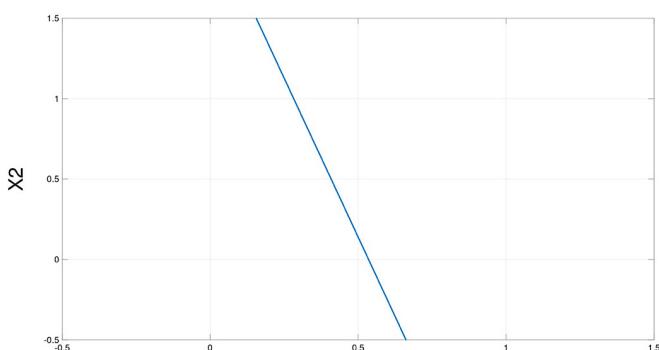
#### 1. 은닉층의 첫번째 레이어의 노드별 직선



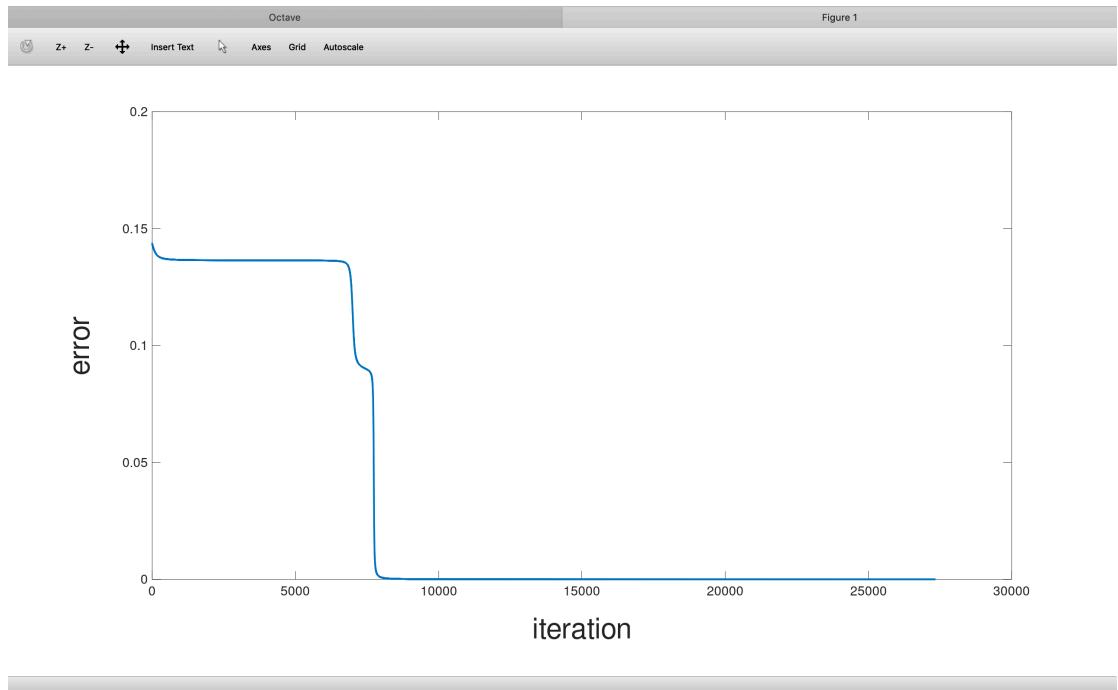
## 2. 은닉층의 2 번째 레이어의 노드별 직선



## 3. 은닉층의 세번째 레이어의 노드별 직선

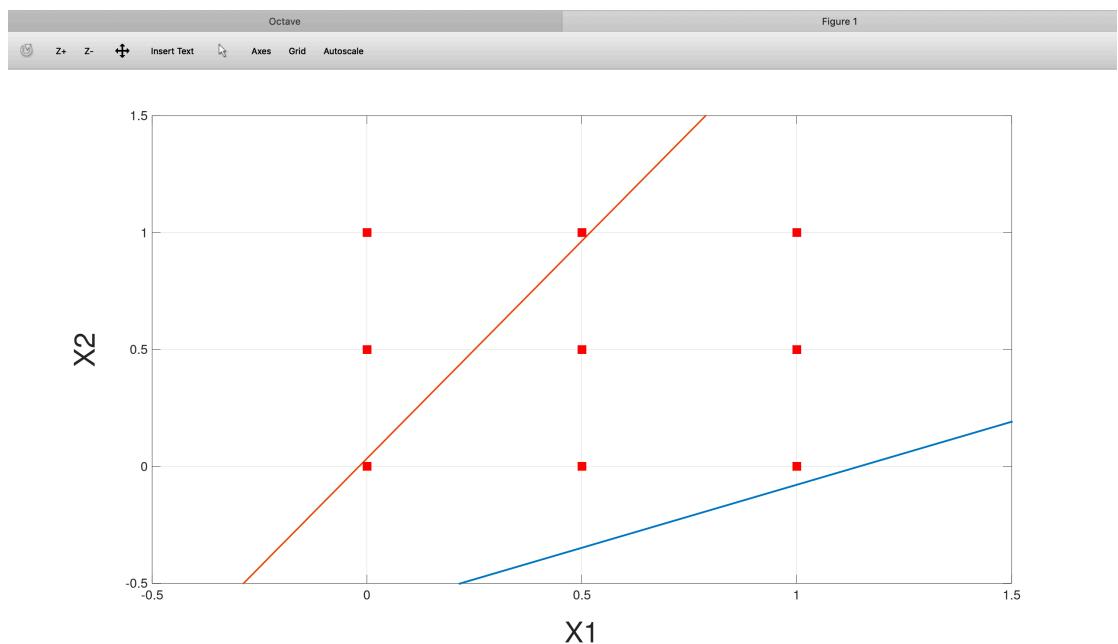


iv. 에러

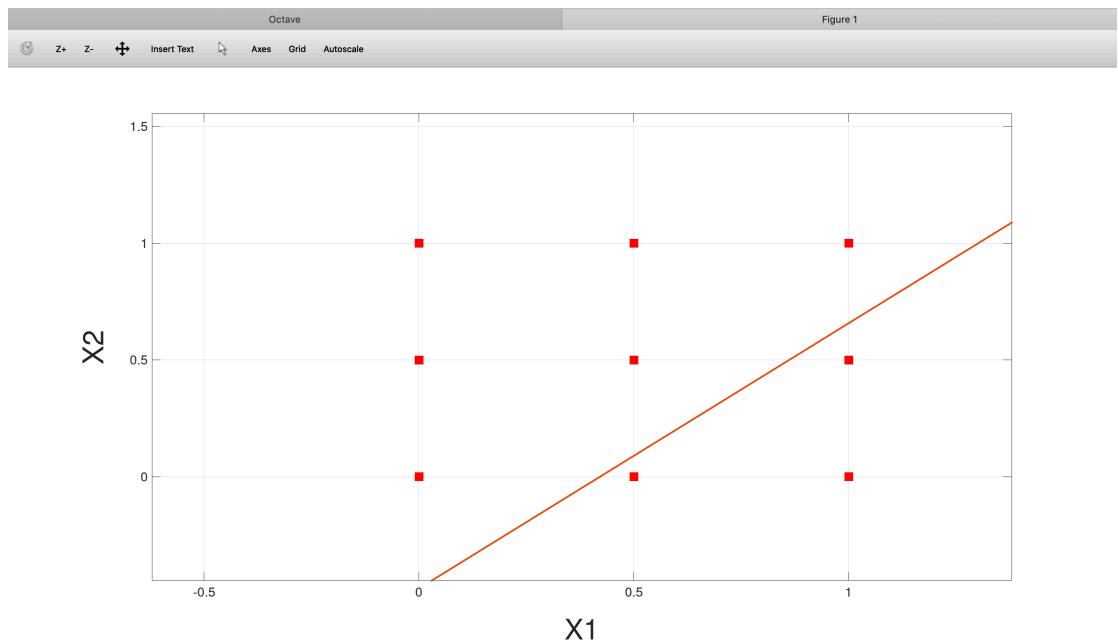


D. 도넛모양 구분 실험

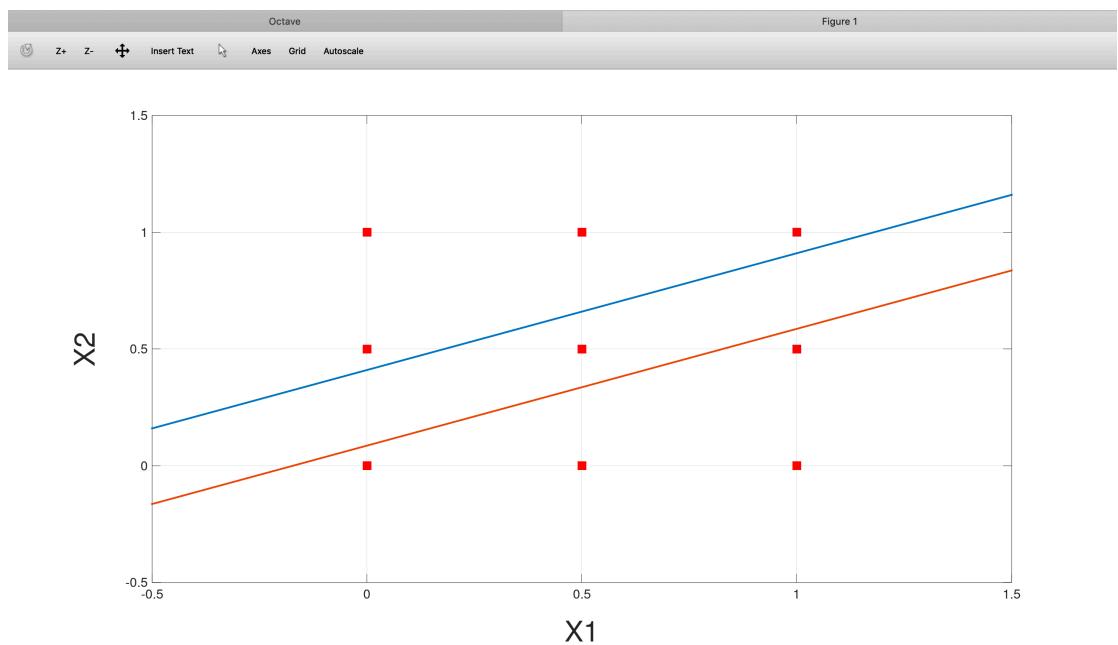
i. 학습 전



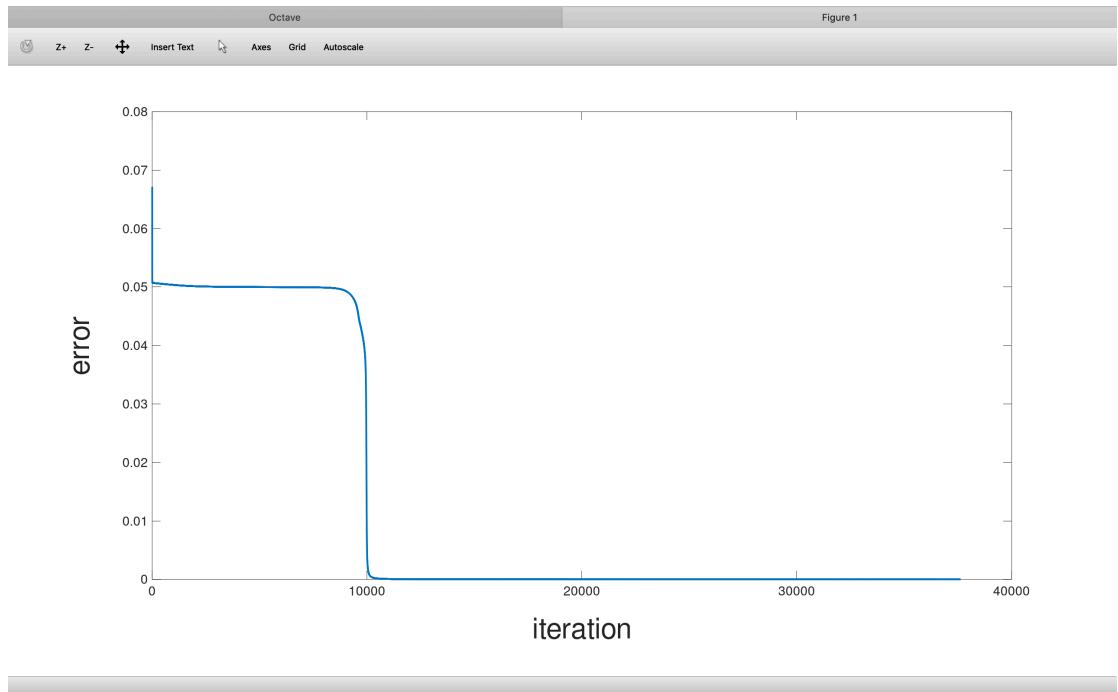
ii. 학습 중



iii. 학습 후



#### iv. 에러



## 10. 결론

각 게이트의 iteration 별 error 그래프를 보면 에러가 처음에 쭉 떨어지다가 한번의 정체기를 겪고 다시 내려가서 tolerance 에 도달하는 모습을 볼 수 있었다. 이를 보면 프로그램이 local minimum 에 갇히지 않고 tolerance 에 잘 도달하는 것을 볼 수 있다.

multi-layer perceptron 을 사용하면 기존에 1-layer perceptron 으로도 구현이 가능했던 and gate, or gate 뿐만 아니라 xor-gate, donut 모양 구분 역시 가능했다. 각 레이어별 노드 수, 레이어의 갯수 모두 변수로 지정해서 사용자가 변경 할 수 있다. 각 레이어 별, 각 노드 별 가중치들과 바이어스, 출력값을 모두 파일로 출력할 수 있게 구현하였다. 각 노드 별 가중치에 대한 직선의 그래프는 Octave 를 통해 그렸다. 또한 iteration 별 error 그래프 역시 프로그램에서 생성한 텍스트 파일을 통해 그렸다.