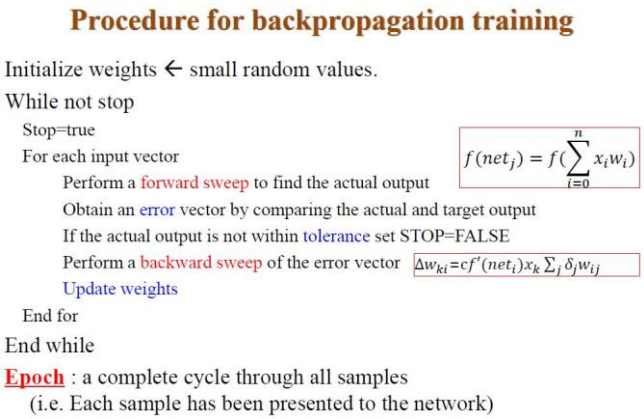
# Multi-Layer Perceptron 구현

컴퓨터과학부 2018920031 유승리 | 인공지능 | 과제 #3

## 1. 프로그램 설명

### > 개요




Procedure for Backpropagation Training | 인공지능 강의자료 #3

이 프로그램은 위의 Procedure for Backpropagation Training 알고리즘을 이용하여 사용자가 설정한 학습 데이터에 근접하도록 스스로 학습하는 multi-layer perceptron을 구현한 프로그램이다. 보고서에서는 input의 개수를 2로 설정하였으며, 학습 데이터로는 AND, OR, XOR gate의 값과 Doughnut 모양의 데이터를 설정하였다.

앞서 제출했던 과제 #2에서 설정했던 Neuron 클래스를 이용하여 multi-layer perceptron 전체를 생성하는 Network 클래스를 작성하였으며, 학습 데이터를 선택하는 기능, layer의 개수 및 각 layer마다의 neuron의 개수를 설정하는 기능, 그리고 각 layer마다의 weight 및 bias 값을 출력하는 기능 등을 추가하였다.

**> 소스코드 및 구조**

① 기본 설정

```
#define INPUT_SIZE 2    // input의 개수 = 2
#define LEARNING_RATE 0.2f      // 학습률
#define TOLERANCE 0.05f // 모든 (|t - o| <= TOLERANCE)일 때 학습 종료

using namespace std;
```

- input의 개수는 2로 고정했으며, LEARNING_RATE(학습률)은 0.2로 설정하였다.

- TOLERANCE는 모든 input 데이터(train_x) 조합에 대하여

$$|target - output| \leq \text{TOLERANCE}$$

  인 경우에 학습을 종료할 수 있도록 설정한 값이다. main 함수에서 학습을 종료하는 조건으로 사용된다.

```
// AND, OR, XOR gate 학습 데이터
vector<vector<float>> train_gate_x = {
        {0,0},{0,1},{1,0},{1,1}
};
vector<float> train_gate_AND_y = { 0,0,0,1 };
vector<float> train_gate_OR_y = { 0,1,1,1 };
vector<float> train_gate_XOR_y = { 0,1,1,0 };

// 도우넛 모양 데이터
vector<vector<float>> train_DOUGHNUT_x = {
        {0,0}, {0,1}, {1,0}, {1,1}, {0.5,1}, {1, 0.5}, {0, 0.5}, {0.5, 0}, {0.5, 0.5}
};
vector<float> train_DOUGHNUT_y = { 0,0,0,0,0,0,0,0,1 };
```

- 학습 데이터는 train_x = { {input 조합} }, train_y = { target } 형식으로 설정하였다.

```
// Sigmoid 함수
float Sigmoid(float x)
{
        return 1 / (1 + exp(-x));
}

// Sigmoid 함수의 도함수
float Sigmoid_D(float x)
{
        float y = Sigmoid(x);
        return y * (1 - y);
}
```

- Sigmoid 함수 연산 결과를 float형으로 반환하는 함수인 Sigmoid(float x)를 선언했다.

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

- Sigmoid 함수의 도함수인 Sigmoid_D(float x)를 선언했다. single-layer perceptron과 다르게 multi-layer perceptron에서는 δ 값을 구할 때 이 함수가 필요하다.

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

② Neuron 클래스

```
class Neuron
{
private:
        vector<float> input_this; // 해당 Neuron의 input
        vector<float> weight_this; // 해당 Neuron의 weight
        vector<float> weight_next; // Back Propagation 과정에서 필요한 변형된 next layer의 weight
        float bias;
        float net;
        float delta_this; // 해당 Neuron의 delta

        // Initial Value(초기값) 설정
        void setValue()
        {
                bias = -1;      // bias의 초기값은 -1로 설정

                random_device rd;
                mt19937 random(rd());   // 난수 생성
                uniform_real_distribution<float> distr(-1.0, 1.0);
                // 생성되는 난수는 -1.0 이상 1.0 미만의 실수

                for (size_t i = 0; i < weight_this.size(); i++)
                {
                        weight_this[i] = distr(random);// weight vector에 난수 저장
                }
        }
```

- private member로 input_this, weight_this, weight_next, bias, net, delta_this를 선언하였다.
- 뉴런의 초기 값을 설정하는 setValue()에서는 weight와 bias의 초기 값을 다음과 같이 설정하였다.

| weight | -1.0 ≤ weight < 1.0 범위의 난수 |
|---|---|
| bias | -1 |

```
public:
        // 생성자
        Neuron(size_t input_size)
        {
                weight_this.resize(input_size); // weight.size() == INPUT_SIZE
                input_this.resize(input_size);
                setValue();
        }

        // Sigmoid 함수의 연산 결과인 output을 반환하는 함수
        float Forward(const vector<float> & input)
        {
                float wx = 0.0;

                for (size_t i = 0; i < weight_this.size(); i++)
                {
                        wx += weight_this[i] * input[i];        // wx의 합
                }

                input_this = input;
                net = bias + wx;
                return Sigmoid(net); // = output
        }
```

- 생성자에서는 weight_this와 input_this vector의 크기를 input의 크기로 재설정하며, 초기 값을 설정한다.

- Sigmoid 함수의 연산 결과인 output을 float형으로 반환하는 Forward(const vector<float> & input)에서는 다음의 수식에 따라 output을 계산한다.

$$output = \sigma(\sum_{i=0}^{n} w_i x_i) = \sigma(bias + \sum_{i=1}^{n} w_i x_i) = \sigma(net)$$

```
// Back Propagation 과정에서의 연산을 나타낸 함수
void Backward(float c, float error, const vector<float> & input_this)
{
	delta_this = Sigmoid_D(net) * error; // delta의 연산

	for (size_t i = 0; i < input_this.size(); i++)
	{
		weight_this[i] += c * delta_this * input_this[i]; // weight 값 조정
	}
	bias += c * delta_this * 1; // bias 값 조정
}

// vector w를 인수로 받아 weight_next에 복사하는 함수
void setWeight_next(vector<float> w)
{
	weight_next.clear();
	for (size_t i = 0; i < w.size(); i++)
	{
		weight_next.push_back(w[i]);
	}
}
```

- neuron을 학습시키는 함수인 Backward(float c, float error, const vector<float> & input_this)은 Forward(const vector<float> & input)를 통해 output과 target을 비교하여 얻은 error 값 또한 인수로 받는다. single-layer perceptron에서는 다음의 수식에 의해 학습을 진행했다.

$$\Delta w_k = c(t_i - O_i)f'(net)x_k$$

$$w_k \leftarrow w_k + \Delta w_k$$

하지만, multi-layer perceptron에서는 다음의 수식에 의해 학습을 진행한다.

$$\Delta w_{ki} = cf'(net_i)x_k \sum_j \delta_j w_{ij} = c\delta_i x_k$$

| | |
|---|---|
| output layer의 δ 값 | $\delta_j = -\frac{\partial E}{\partial net_j} = -\frac{\partial\left(\frac{1}{2}\left(t - f\left(net_j\right)\right)^2\right)}{\partial net_j} = f'(net_j)(t - f\left(net_j\right))$ |
| 그 외 layer의 δ 값 | $\delta_i = f'(net_i)\sum_j \delta_j w_{ij}$ |

- private member인 weight_next의 setter인 setWeight_next(vector<float> w)를 선언하였다.

```
        vector<float> & getWeight_next()
        {
                return weight_next;
        }

        int getInputSize()
        {
                return weight_this.size();
        }

        float getNet()
        {
                return net;
        }

        vector<float> & getInput()
        {
                return input_this;
        }

        float getDelta()
        {
                return delta_this;
        }

        vector<float> & getWeight_this()
        {
                return weight_this;
        }

        float & getBias()
        {
                return bias;
        }
};
```

- private member들의 getter인 getWeight_next(), getInputSize(), getNet(), getInput(), getDelta(), getWeight_this(), getBias()를 선언하였다.

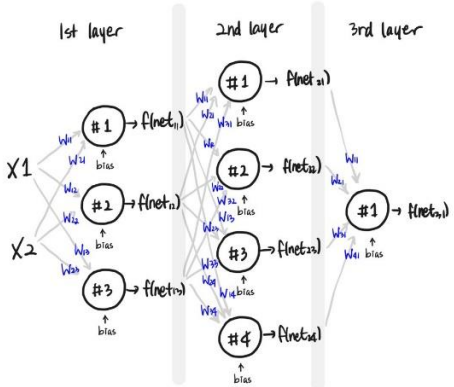
③ **Network** 클래스

```
class Network
{
private:
	vector<vector<Neuron>> layers; // 모든 layer를 모은 vector
	vector<float> output_this; // 한 layer의 output을 모은 vector
	vector<float> delta_next; // 한 layer의 다음 layer의 delta를 모은 vector
	vector<size_t> setting_all; // 생성자의 인수로 받은 vector
	size_t layer_num; // layer 개수
	float error_o, error_h; // error_o = (train_y) - (Forward 후 output)

public:
	// 생성자, 인수로 주어진 setting vector에 따라 신경망을 구성함
	Network(const vector<size_t> & setting)
	{
		layer_num = setting.size() - 1;
		for (size_t i = 0; i < setting.size(); i++)
		{
			setting_all.push_back(setting[i]);
		}

		for (size_t i = 1; i <= layer_num; i++)
		{
			vector<Neuron> layer;
			for (size_t j = 0; j < setting_all[i]; j++)
			{
				layer.push_back(Neuron(setting_all[i - 1]));
			}
			layers.push_back(layer);
		}
	}
```

- private member로 layers, output_this, delta_next, setting_all, layer_num, error_o, error_h를 설정하였다.
- 생성자에서는 인수로 주어진 setting vector에 따라 알맞게 multi-layer perceptron을 구현한다. 신경망 layers는 vector<vector<Neuron>>의 형태를 가진다. 예를 들어

```
Network net({2,3,4,1});
```

와 같이 Network 객체 net를 생성하면 setting vector = {2,3,4,1}이 되는데, 이때 첫번째 원소는 input의 개수를 설정한다. 현재 우리는 INPUT_SIZE를 2이라고 설정하였으므로 첫번째 원소는 2가 되어야한다. 두번째 원소부터는 각 layer마다의 neuron의 개수를 설정하는데, 학습 결과 output으로는 1개의 값이 나와야 하므로 마지막 원소는 1이어야 한다. 따라서 net 객체는 첫번째 layer에 3개의 neuron, 두번째 layer에 4개의 neuron, 세번째 layer에 1개의 neuron을 가지며, 각 neuron의 input의 개수는 2이다. 이를 그림으로 나타내면 다음과 같다.

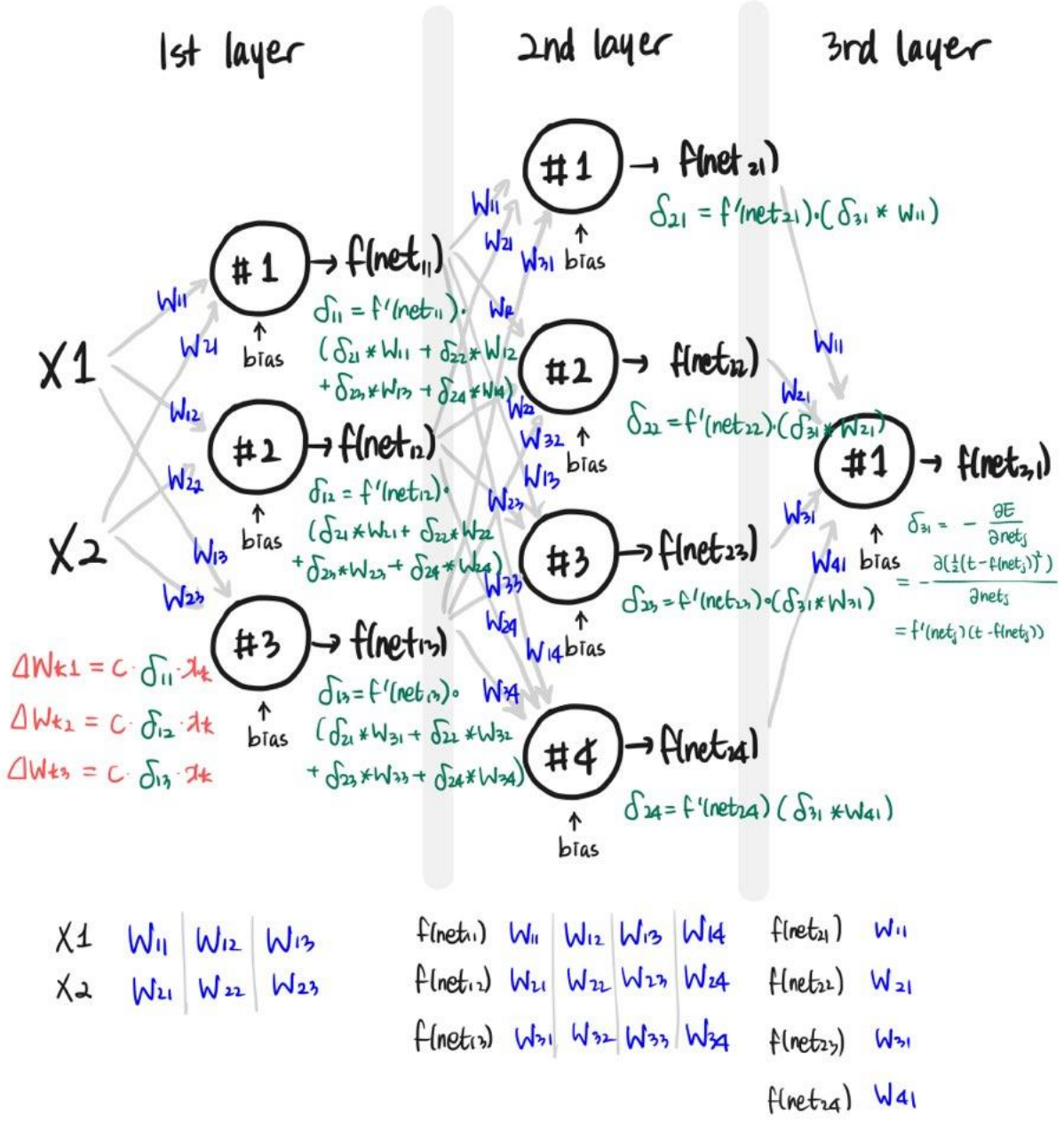


```
        // input layer ~ output layer 까지 한 번 연산하는 함수
        vector<float> Forward(const vector<float> & input_this)
        {
                vector<float> input_tmp;
                for (size_t i = 0; i < input_this.size(); i++)
                {
                        input_tmp.push_back(input_this[i]);
                }

                for (size_t i = 1; i <= layer_num; i++) /// layer 개수만큼 반복
                {
                        output_this.clear();
                        for (size_t j = 0; j < setting_all[i]; j++)
                        /// setting_all[i] = this layer의 neuron 개수
                        {
                                output_this.push_back(layers[i - 1][j].Forward(input_tmp));
                                /// layers[i - 1] = this layer
                        }

                        input_tmp.clear();
                        for (size_t j = 0; j < output_this.size(); j++)
                        {
                                input_tmp.push_back(output_this[j]);
                        }
                }

                // weight_next를 설정함
                for (size_t i = layer_num - 1; i >= 1; i--) /// 2번째 layer부터 (i=2)
                {
                        // next layer의 각 neuron 마다의 weight를 weight_next에 임시 저장
                        vector<vector<float>> weight_next;
                        weight_next.clear();
                        for (size_t j = 0; j < setting_all[i + 1]; j++)
                        /// setting_all[i + 1] = next layer의 neuron 개수
                        {
                                weight_next.push_back(layers[i][j].getWeight_this());
                                /// layers[i] = next layer
                        }

                        // weight_next를 transpose하여 this layer의 각 neuron의 weight_next로 설정
                        for (size_t k = 0; k < setting_all[i]; k++)
                        /// setting_all[i] = this layer의 neuron 개수
                        {
                                vector<float> weight_next_trans;
                                for (size_t j = 0; j < setting_all[i + 1]; j++)
                                /// setting_all[i + 1] = next layer의 neuron 개수
                                {
                                        weight_next_trans.push_back(weight_next[j][k]);
                                }
                                layers[i - 1][k].setWeight_next(weight_next_trans);
                                /// layers[i - 1] = this layer
                        }
                }

                return output_this;
        }
```

- 하나의 input 조합에 대해 input layer에서 output layer까지 1회 연산하는 함수인 Forward(const vector<float> & input_this)를 선언하였다. this layer의 neuron들의 Forward 연산 값인 output_this를 next layer의 input_this로 설정해야 하기 때문에 input_tmp vector에 인수로 받은 input_this를 복사하였다.

- Backpropagation 과정에서 next layer마다 각 neuron들의 weight_this vector를 모은 vector인 weight_next를 transpose한 형태의 weight_next_trans가 필요하므로 이를 설정한다. 이는 weight_next의 두 인덱스의 순서를 바꾸어 weight_next_trans에 저장하면 된다.

- `Network net({2,3,4,1});`




| | | | |
|---|---|---|---|
| X1 | W11 | W12 | W13 |
| X2 | W21 | W22 | W23 |

| | | | | |
|---|---|---|---|---|
| f(net11) | W11 | W12 | W13 | W14 |
| f(net12) | W21 | W22 | W23 | W24 |
| f(net13) | W31 | W32 | W33 | W34 |

| | |
|---|---|
| f(net21) | W11 |
| f(net22) | W21 |
| f(net23) | W31 |
| f(net24) | W41 |

```
// 전체 input 데이터에 대해서 한 번 Back Propagation 연산하는 함수
void Backward(float c, const vector<vector<float>> & train_x, const vector<float> &
train_y)
{
	// 전체 input 데이터에 대해 진행
	for (size_t n = 0; n < train_x.size(); n++)
	{
		// output layer의 연산
		float o = Forward(train_x[n])[0]; // Forward 연산
		error_o = train_y[n] - o; // output layer의 error
		layers[layer_num - 1][0].Backward(c, error_o, layers[layer_num - 1][0].
		getInput());
		/// layers[layer_num - 1] = this layer = output layer

		// hidden layer & input layer의 연산
		for (size_t i = layer_num - 1; i >= 1; i--)
		/// layers[layer_num - 1] = this layer = output layer
		{
			delta_next.clear();
			for (size_t j = 0; j < layers[i].size(); j++)
			/// layers[i] = next layer
			{
				delta_next.push_back(layers[i][j].getDelta());
				/// layers[i] = next layer
			}

			for (size_t j = 0; j < layers[i - 1].size(); j++)
			/// layers[i - 1] = this layer
			{
				error_h = 0.0f; // hidden layer & input layer의 error
				for (size_t k = 0; k < layers[i].size(); k++)
				/// layers[i] = next layer
				{
					error_h += delta_next[k] * layers[i - 1][j].
					getWeight_next()[k]; /// layers[i - 1] = this layer
				}
				layers[i - 1][j].Backward(c, error_h, layers[i - 1][j].
				getInput()); /// layers[i - 1] = this layer
			}
		}
	}
}
```

- 전체 input 데이터에 대해서 1회 Backpropagation 하는 함수인 `Backward(float c, const vector<vector<float>> & train_x, const vector<float> & train_y)`를 선언하였다.

- 모든 layer의 neuron들에 대하여 다음과 같은 식을 이용해서 `Backward(float c, float error, const vector<float> & input_this)` 연산을 수행한다.

| | |
|---|---|
| output layer의 δ 값 | $\delta_j = -\frac{\partial E}{\partial net_j} = -\frac{\partial\left(\frac{1}{2}\left(t - f(net_j)\right)^2\right)}{\partial net_j} = f'(net_j)\left(t - f(net_j)\right)$ $= f'(net_j) \times error\_o$ |
| 그 외 layer의 δ 값 | $\delta_i = f'(net_i)\sum_j \delta_j w_{ij} = f'(net_i) \times error\_h$ |

```
        vector<float> & getOutput_this()
        {
                return output_this;
        }

        size_t getLayer_num()
        {
                return layer_num;
        }

        vector <size_t> getSetting_all()
        {
                return setting_all;
        }

        vector<vector<Neuron>> getLayers()
        {
                return layers;
        }
};
```

- private member들의 getter인 getOuput_this(), getLayer_num(), getSetting_all(), getLayers()를 선언하였다.

④ **printResult** 함수

```
// Console 창에 결과를 출력하는 함수
// : Epoch 수, input & output 표, Total Error 값, 각 layer의 neuron 별 weight & bias 표
void printResult(Network net, vector<vector<float>> & train_x, vector<float> & train_y, int epoch,
float error_total)
{
	size_t input_size = train_x[0].size();
	size_t train_size = train_x.size();
	size_t layer_num = net.getLayer_num();
	vector<size_t> setting_all = net.getSetting_all();
	vector<vector<float>> input_output_layer; // output layer의 input을 저장함

	cout << " [Epoch " << epoch << "]" << "\n\n"; // 현재 Epoch 수 출력

	// input & output 표
	cout << " ";
	for (size_t i = 0; i < input_size; i++)
	{
		cout << "  X" << i + 1 << " |";
	}
	cout << "   OUTPUT   " << endl;
	cout << " ";
	for (size_t i = 0; i < input_size; i++)
	{
		cout << "-----|";
	}
	cout << "------------" << endl;
	for (size_t i = 0; i < train_size; i++)
	{
		cout << " ";
		for (size_t j = 0; j < input_size; j++)
		{
			cout.width(4);
			cout << train_x[i][j] << " |";
		}
		cout << " ";
		cout.width(9);
		cout << net.Forward(train_x[i])[0] << endl;
		input_output_layer.push_back((net.getLayers()[layer_num - 1][0]).getInput());
		// output layer의 input을 저장함
	}

	// Total Error 값 출력
	cout << "\n => Total Error = " << error_total << "\n\n";
```

- Console 창에 학습 결과를 출력하는 함수 `printResult(Network net, vector<vector<float>> & train_x, vector<float> & train_y, int epoch, float error_total)`를 선언하였다.
- Epoch 수, input과 output 표, Total Error 값, 각 layer의 neuron 별 weight와 bias 표를 출력한다.
- 학습 과정에서 변형된 train_x의 input 조합, 즉 output layer의 input_this를 모아 input_output_layer에 저장한다.

  ex) 모든 train_x 조합 {0,0}, {0,1}, {1,0}, {1,1} 각각에 대해 1회 Forward 연산을 진행하면

  **마지막 layer(output layer)**의 **input 조합**은

  **마지막에서 두번째 layer**의 **output을 모은 것**과 같다.

```
        // 각 layer의 neuron 별 weight & bias 표
        for (size_t i = 0; i < layer_num; i++)
        {
                cout << "   (Layer #" << i + 1 << ")" << endl;
                cout << "              ";
                for (size_t j = 0; j < setting_all[i]; j++)
                {
                        cout << "|    W" << j + 1 << "    ";
                }
                cout << "|   Bias   ";
                cout << "\n   -----------";
                for (size_t j = 0; j < setting_all[i] + 1; j++)
                {
                        cout << "|------------";
                }
                cout << endl;

                for (size_t j = 0; j < setting_all[i + 1]; j++)
                {
                        cout << "    Neuron #" << j + 1 << " |";
                        for (size_t k = 0; k < setting_all[i]; k++)
                        {
                                cout.width(11);
                                cout << net.getLayers()[i][j].getWeight_this()[k] << " |";
                        }
                        cout.width(11);
                        cout << net.getLayers()[i][j].getBias() << endl;
                }

                // output layer의 경우, train_x에 대한 각각의 변형된 input 조합 또한 출력 (그래프 그리기 위함)
                if (i == layer_num - 1)
                {
                        cout << "\n             |";
                        for (size_t j = 0; j < setting_all[i]; j++)
                        {
                                cout<< "    X" << j+1 << "    |";
                        }
                        cout << "\n             |";
                        for (size_t j = 0; j < setting_all[i]; j++)
                        {
                                cout << "------------|";
                        }
                        cout << "\n";
                        for (size_t j = 0; j < train_size; j++)
                        {
                                cout << "             |";
                                for (size_t k = 0; k < setting_all[i]; k++)
                                {
                                        cout.width(11);
                                        cout << input_output_layer[j][k] << " |";
                                }
                                cout << "\n";
                        }
                }
                cout << endl;
        }
        cout << "-----------------------------------------------------------------" << endl;
}
```

- output layer의 경우, output layer의 input_this를 모은 input_output_layer의 값을 출력한다.

  이는 학습 완료 후의 weight 그래프를 그리기 위함이다.

⑤ trainResult 함수

```
// 모든 (|t - o| <= TOLERANCE)일 때까지 학습을 반복하고, 지정된 Epoch마다 Console 창에 결과를 출력하는 함수
void trainResult(Network net, vector<vector<float>> & train_x, vector<float> & train_y, string
dataName)
{
	size_t input_size = train_x[0].size(); // == INPUT_SIZE
	size_t train_size = train_x.size(); // == train_y.size()
	int epoch = 0;
	vector<float> error; // |t - o| 값을 저장하는 vector
	float error_total;
	bool STOP;

	size_t layer_num = net.getLayer_num(); // layer 개수
	vector<size_t> setting_all = net.getSetting_all();
	// Network 생성자의 인수로 입력 받은 setting vector

	cout << "\n-----------------------------------------------------------------\n";
	cout << "============================ " << dataName << " ============================\n";
	cout << "-----------------------------------------------------------------\n";

	do
	{
		STOP = true;
		error_total = 0.0f;
		error.clear();
		net.Backward(LEARNING_RATE, train_x, train_y); // 학습 1회 진행

		for (size_t i = 0; i < train_size; i++)
		{
			error.push_back(abs(train_y[i] - net.Forward(train_x[i])[0]));
			// |t - o| 값을 error vector에 저장
			STOP = STOP && (error[i] <= TOLERANCE);
			// 모든 error 값(|train_y[i] - net.Forward(train_x[i])[0]|)이 TOLERANCE 보다
			작거나 같아야 학습 종료
			error_total += 0.5f * pow(error[i], 2); // MSE 값의 합, MSE = 0.5 * error[i]^2
		}
```

- 모든 (|target − output| <= TOLERANCE)일 때까지 학습을 반복하고, 지정된 Epoch마다 Console 창에 결과를 출력하는 함수인 trainResult(Network net, vector<vector<float>> & train_x, vector<float> & train_y, string dataName)를 선언하였다.

- error vector에는 abs 함수를 이용하여 (target − output)의 절댓값을 저장하였으며, bool 타입의 변수 STOP은 error vector의 모든 값이 TOLERANCE보다 작거나 같아야 true의 값을 가진다.

- error_total은 MSE 값의 합이므로 다음과 같다.

$$\sum_i MSE_i = \sum_i \frac{1}{2} \times error_i^2 = \sum_i \frac{1}{2} \times (target_i - output_i)^2$$

- do-while 문의 조건을 while(!STOP)으로 하여 STOP이 false, 즉 error vector의 값 중 하나라도 TOLERANCE보다 클 때 loop를 실행하도록 설정하였다.

```
                // Console 창 출력
                if (epoch <= 1000)
                {
                        if (epoch == 0 || epoch == 5 || epoch == 10 || epoch == 20 || epoch == 35 ||
                        epoch == 50 || epoch == 100 || (epoch % 200) == 0)
                        {
                                printResult(net, train_x, train_y, epoch, error_total);
                                cout << endl;
                        }
                }
                else if ((epoch < 10000) && (epoch % 2000) == 0)
                {
                        printResult(net, train_x, train_y, epoch, error_total);
                        cout << endl;
                }
                else if ((epoch % 10000) == 0)
                {
                        printResult(net, train_x, train_y, epoch, error_total);
                        cout << endl;
                }

                epoch++; // epoch 1 증가
        } while (!STOP);

        // 학습 완료 후 Console 창 출력
        printResult(net, train_x, train_y, epoch, error_total);
        cout << "*** 학습이 완료되었습니다. ***\n\n";
}
```

- Console 창에는 Epoch가 0, 5, 10, 20, 35, 50, 100, 200, 400, 600, 800, 1000, 2000, 4000, 6000, 8000, 10000, 20000, …일 때 학습 결과가 출력되도록 설정하였으며, 학습이 완료되어 do-while loop를 빠져나왔을 때의 학습 결과 또한 출력되도록 하였다.

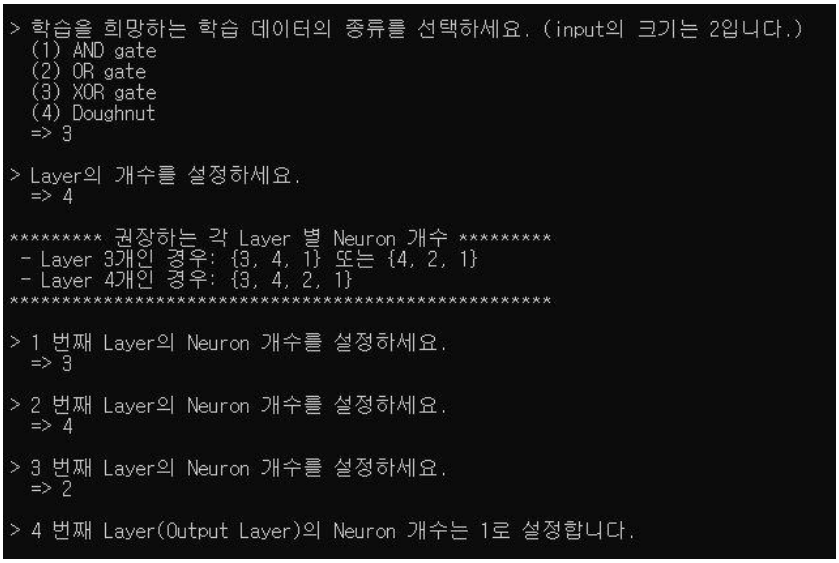
⑥ main 함수

```
int main(void)
{
	size_t data_num; // 선택된 데이터의 번호
	size_t layer_num; // 설정된 layer의 개수
	vector<size_t> setting_user; // 설정된 setting vector

	// 사용자로부터 학습을 희망하는 데이터의 종류, layer의 개수, 각 layer의 neuron 개수를 입력 받음
	cout << "\n > 학습을 희망하는 학습 데이터의 종류를 선택하세요. (input의 크기는 " << INPUT_SIZE <<
	"입니다.)" << endl;
	cout << "   (1) AND gate\n" << "   (2) OR gate\n" << "   (3) XOR gate\n" << "   (4)
	Doughnut\n" << "   => ";
	cin >> data_num;
	cout << "\n > Layer의 개수를 설정하세요.\n   => ";
	cin >> layer_num;
	setting_user.push_back(INPUT_SIZE);
	for (size_t i = 1; i < layer_num; i++)
	{
		size_t tmp;
		cout << "\n > " << i << " 번째 Layer의 Neuron 개수를 설정하세요.\n   => ";
		cin >> tmp;
		setting_user.push_back(tmp);
	}
	cout << "\n > " << layer_num << " 번째 Layer(Output Layer)의 Neuron 개수는 1로 설정합니다.\n\n";
	 // output이 1개여야 하기 때문임
	setting_user.push_back(1);
	Sleep(2000); // delay 부여

	// 사용자가 선택한 학습 데이터의 종류에 따른 학습 결과 출력
	if (data_num == 1)
	{
		Network net_AND(setting_user);
		trainResult(net_AND, train_gate_x, train_gate_AND_y, "AND gate");
	}
	else if (data_num == 2)
	{
		Network net_OR(setting_user);
		trainResult(net_OR, train_gate_x, train_gate_OR_y, "OR gate ");
	}
	else if (data_num == 3)
	{
		Network net_XOR(setting_user);
		trainResult(net_XOR, train_gate_x, train_gate_XOR_y, "XOR gate");
	}
	else if (data_num == 4)
	{
		Network net_DOUGHNUT(setting_user);
		trainResult(net_DOUGHNUT, train_DOUGHNUT_x, train_DOUGHNUT_y, "DOUGHNUT");
	}

	return 0;
}
```

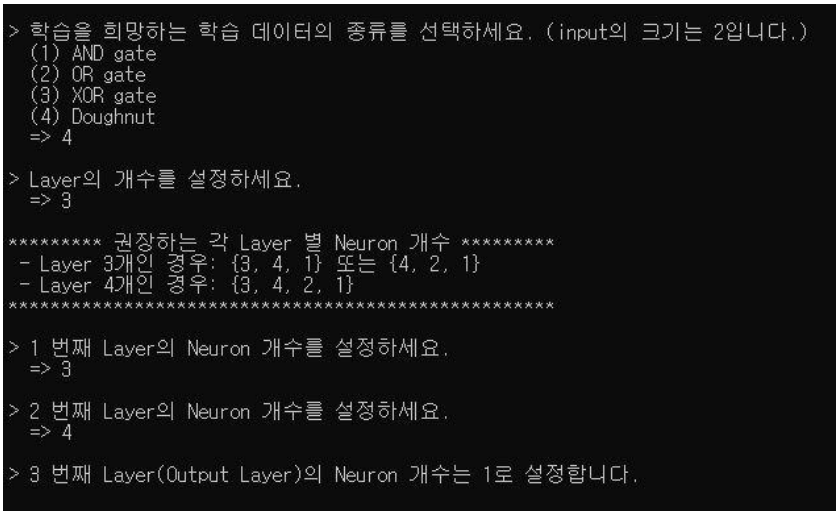
- main(void) 함수에서는 사용자로부터 학습을 희망하는 데이터의 종류, layer의 개수, 각 layer의 neuron 개수를 입력 받는다. 그리고 사용자가 선택한 학습 데이터의 종류에 따른 학습 결과를 출력한다.

**> 프로그램 실행 시 첫 화면 (학습을 희망하는 데이터의 종류, layer의 개수, 각 layer의 neuron 개수 입력 창)**

- layer의 개수 = 4인 경우




- layer의 개수 = 3인 경우



## 2. 프로그램을 통한 학습 결과

### > 설정

- layer의 개수가 각각 2, 3, 4, 5개인 경우로 여러 차례 실험한 결과, layer의 개수가 4개일 때 가장 multi-layer perceptron의 특징을 잘 보여주는 것 같아서 layer의 개수로 4를 입력했다. 또한, neuron 개수를 첫번째 layer의 경우 3개, 두번째 layer의 경우 4개, 세번째 layer의 경우 2개, 네번째layer의 경우 1개로 입력하여 setting vector를 `{2,3,4,2,1}` 로 설정하였는데, 이는 다음과 같은 이유에 따라서 결정된 값이다.

| | |
|---|---|
| 첫번째 원소 2 | train data의 input size 가 2이므로 |
| 마지막 원소 1 | train data의 output size 가 1이므로 |
| 마지막에서 두번째 원소 2 | 마지막 layer의 2차원 learning 과정 그래프를 그리기 위해서 |

- 이번 보고서에서는 각 train data에 대해 최소 3회 이상 실험한 결과, 최대 학습 시간과 최대 학습 횟수가 소요된 경우를 다룬다.

- 각 layer의 neuron마다의 weight & bias를 Console 창에서 행렬 형식으로 출력하였다. 보고서에는 이를 캡쳐하여 첨부하였다.

- learning 과정 그래프는 가로축이 $x_1$축이고, 세로축이 $x_2$축이다. 그리고 노란색 점 ●은 target output이 1인 점이며, 파란색 점 ●은 target output이 0인 점이다. 또한 그래프를 다음의 경우마다 작성하였다.

| | |
|---|---|
| 학습 시작 전의 | 첫번째 layer |
| | 마지막 layer |
| 학습 완료 후의 | 첫번째 layer |
| | 마지막 layer |

그래프에 나타낸 직선의 방정식은 다음의 식을 이용하여 구한다.

$$w_1x_1 + w_2x_2 + bias = 0$$

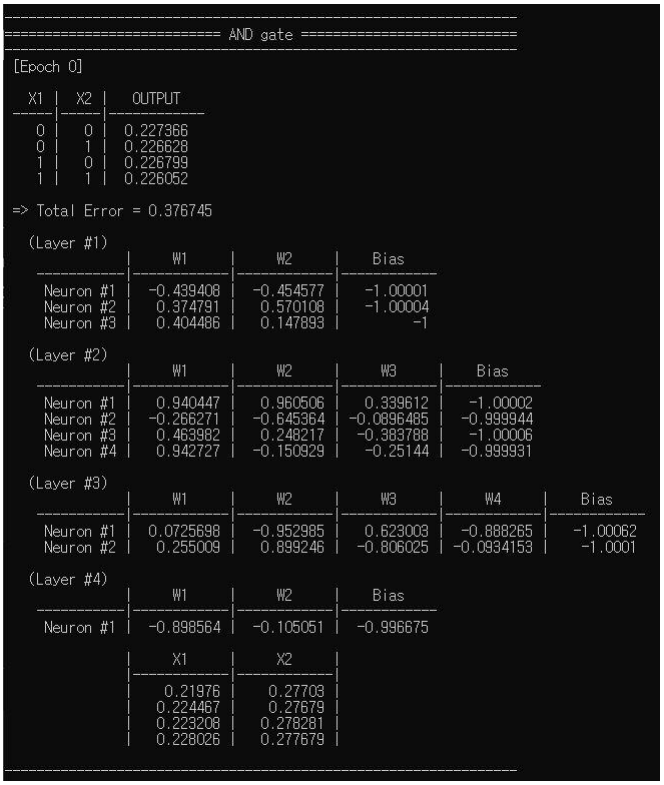
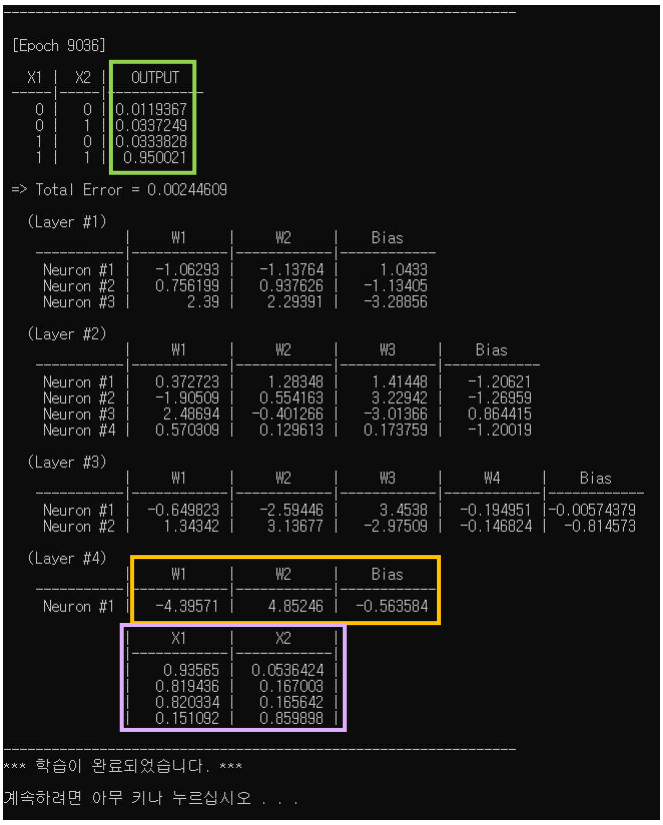
따라서 다음의 weight와 bias 값에 의하여 계산되는 직선의 방정식은 $7.21171x_1 - 5.14621x_2 - 1.44509 = 0$ 이다.

| | W1 | W2 | Bias |
|---|---|---|---|
| Neuron #1 | 7.21171 | -5.14621 | -1.44509 |

- iteration에 따른 error 그래프는 Epoch가 0, 5, 10, 20, 35, 50, 100, 200, 400, 600, 800, 1000, 2000, 4000, 6000, 8000, 10000, 20000, … , 학습 완료 후의 epoch일때의 total MSE 값으로 작성하였다.

**> AND gate**

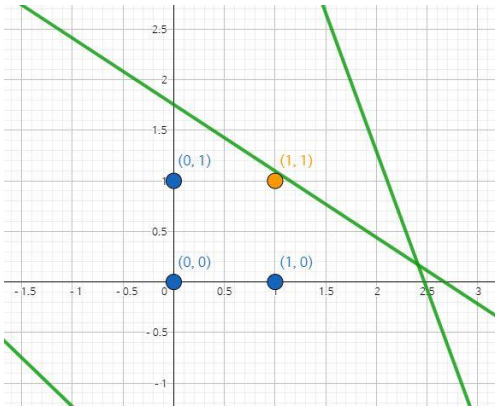
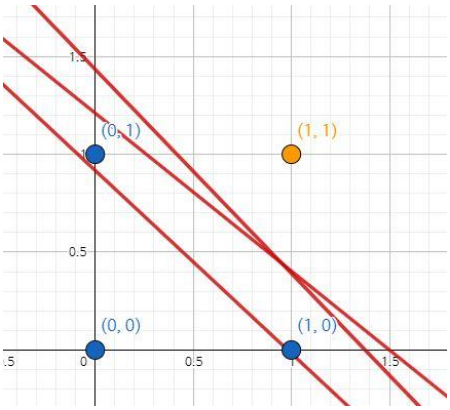
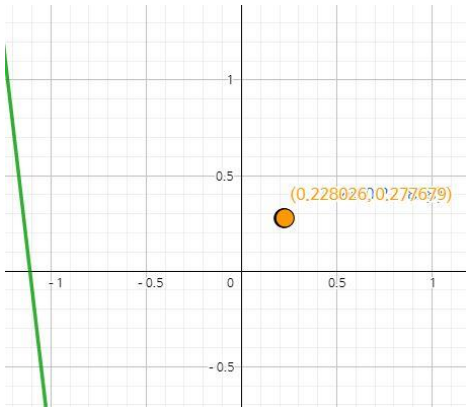
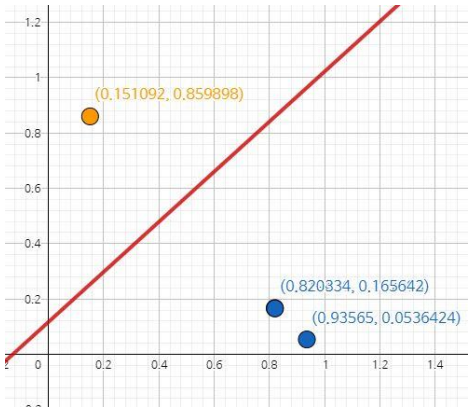
[각 layer의 neuron마다의 weight & bias]

| 학습 시작 전 (Epoch = 0) | 학습 완료 후 (Epoch = 9036) |
| --- | --- |
|  |  |

```
========================= AND gate =========================
[Epoch 0]

  X1 |  X2 |   OUTPUT
-----|-----|-----------
   0 |   0 |  0.227366
   0 |   1 |  0.226628
   1 |   0 |  0.226799
   1 |   1 |  0.226052

=> Total Error = 0.376745

  (Layer #1)
            |      W1      |      W2      |     Bias
 -----------|--------------|--------------|-------------
  Neuron #1 |  -0.439408   |  -0.454577   |   -1.00001
  Neuron #2 |   0.374791   |   0.570108   |   -1.00004
  Neuron #3 |   0.404486   |   0.147893   |         -1

  (Layer #2)
            |      W1      |      W2      |      W3      |     Bias
 -----------|--------------|--------------|--------------|-------------
  Neuron #1 |   0.940447   |   0.960506   |   0.339612   |   -1.00002
  Neuron #2 |  -0.266271   |  -0.645364   |  -0.0896485  |   -0.999944
  Neuron #3 |   0.463982   |   0.248217   |  -0.383788   |   -1.00006
  Neuron #4 |   0.942727   |  -0.150929   |   -0.25144   |   -0.999931

  (Layer #3)
            |      W1      |      W2      |      W3      |      W4      |     Bias
 -----------|--------------|--------------|--------------|--------------|-------------
  Neuron #1 |  0.0725698   |  -0.952985   |   0.623003   |   -0.888265  |   -1.00062
  Neuron #2 |   0.255009   |   0.899246   |  -0.806025   |  -0.0934153  |    -1.0001

  (Layer #4)
            |      W1      |      W2      |     Bias
 -----------|--------------|--------------|-------------
  Neuron #1 |  -0.898564   |  -0.105051   |  -0.996675

             |      X1      |      X2      |
             |--------------|--------------|
             |    0.21976   |    0.27703   |
             |   0.224467   |    0.27679   |
             |   0.223208   |   0.278281   |
             |   0.228026   |   0.277679   |
```

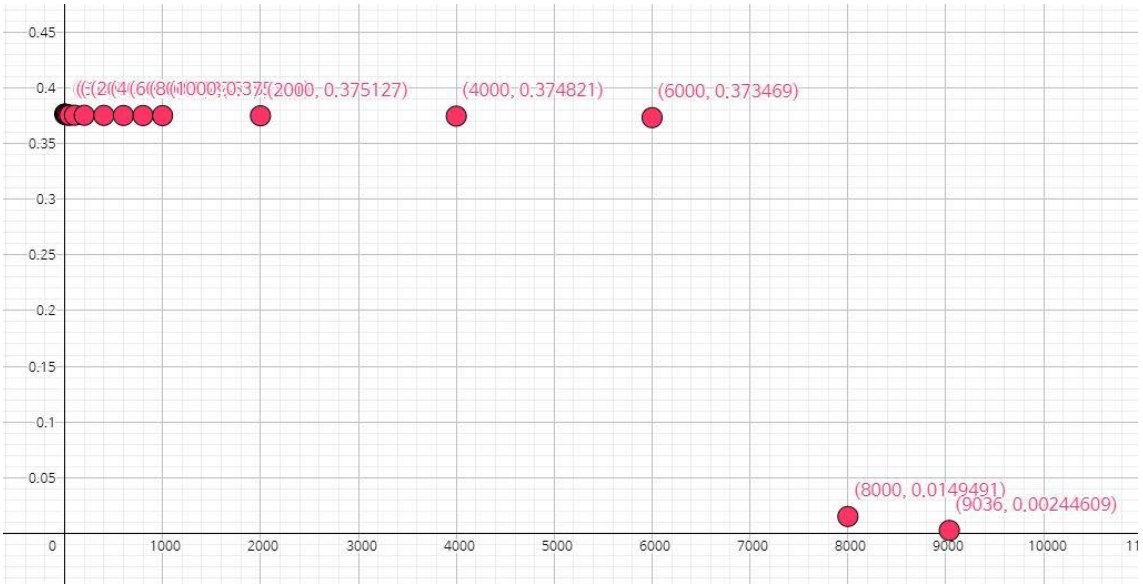
```
[Epoch 9036]

  X1 |  X2 |   OUTPUT
-----|-----|-----------
   0 |   0 |  0.0119367
   0 |   1 |  0.0337249
   1 |   0 |  0.0333828
   1 |   1 |   0.950021

=> Total Error = 0.00244609

  (Layer #1)
            |      W1      |      W2      |     Bias
 -----------|--------------|--------------|-------------
  Neuron #1 |   -1.06293   |   -1.13764   |    1.0433
  Neuron #2 |   0.756199   |   0.937626   |   -1.13405
  Neuron #3 |         2.39 |   2.29391    |   -3.28856

  (Layer #2)
            |      W1      |      W2      |      W3      |     Bias
 -----------|--------------|--------------|--------------|-------------
  Neuron #1 |   0.372723   |    1.28348   |    1.41448   |   -1.20621
  Neuron #2 |   -1.90509   |   0.554163   |    3.22942   |   -1.26959
  Neuron #3 |    2.48694   |  -0.401266   |   -3.01366   |   0.864415
  Neuron #4 |   0.570309   |   0.129613   |   0.173759   |   -1.20019

  (Layer #3)
            |      W1      |      W2      |      W3      |      W4      |     Bias
 -----------|--------------|--------------|--------------|--------------|-------------
  Neuron #1 |  -0.649823   |   -2.59446   |    3.4538    |   -0.194951  |  -0.00574379
  Neuron #2 |    1.34342   |    3.13677   |   -2.97509   |   -0.146824  |   -0.814573

  (Layer #4)
            |      W1      |      W2      |     Bias
 -----------|--------------|--------------|-------------
  Neuron #1 |   -4.39571   |    4.85246   |  -0.563584

             |      X1      |      X2      |
             |--------------|--------------|
             |    0.93565   |  0.0536424   |
             |   0.819436   |   0.167003   |
             |   0.820334   |   0.165642   |
             |   0.151092   |   0.859898   |

*** 학습이 완료되었습니다. ***
계속하려면 아무 키나 누르십시오 . . .
```

[learning 과정 그래프]

| | 학습 시작 전 (Epoch = 0) | 학습 완료 후 (Epoch = 9036) |
|---|---|---|
| 첫번째 layer |  |  |
| 마지막 layer |  |  |

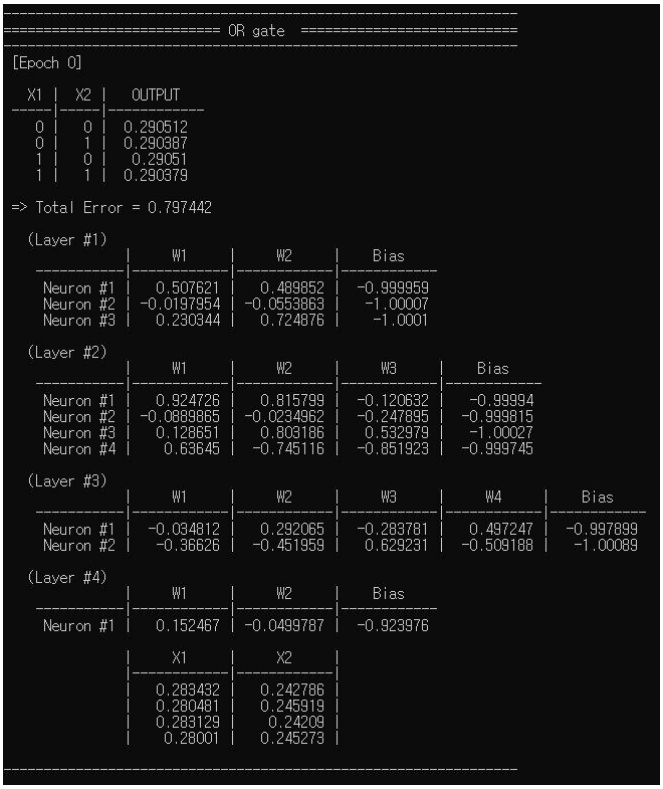
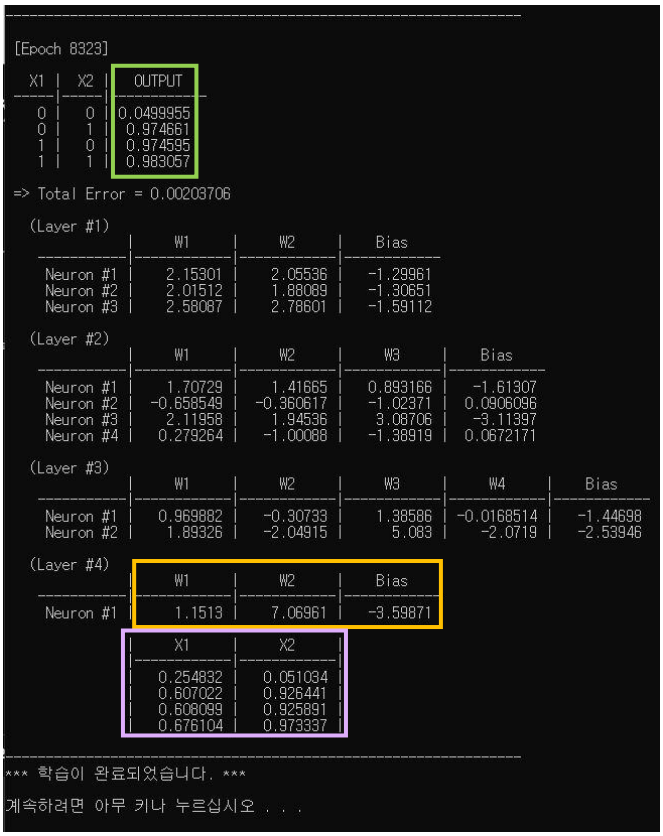
- 마지막 layer에서 input을 나타내는 점이 4개가 아닌 것은 그 개수가 감소한 것이 아니라 겹쳐 있기 때문이다.
- 학습 완료 후, 첫번째 layer와 마지막 layer에서 모두 linearly separable 하다.

[iteration에 따른 error 그래프]



**> OR gate**

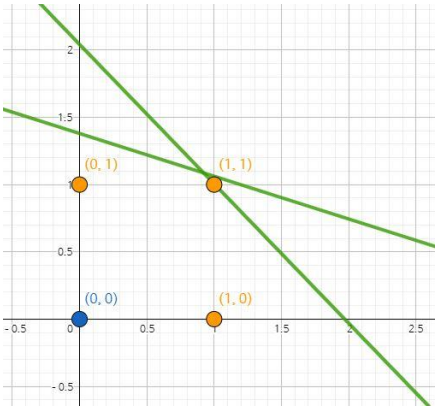
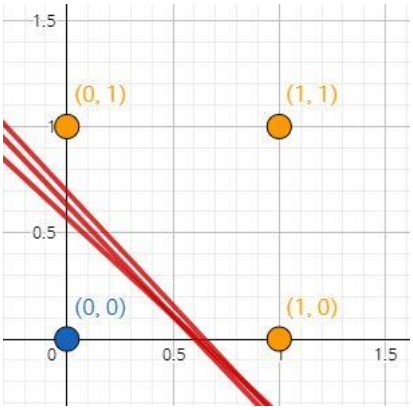
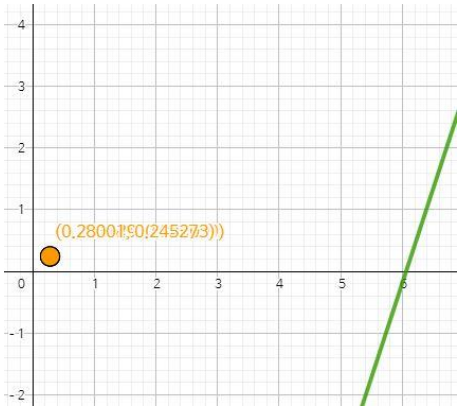
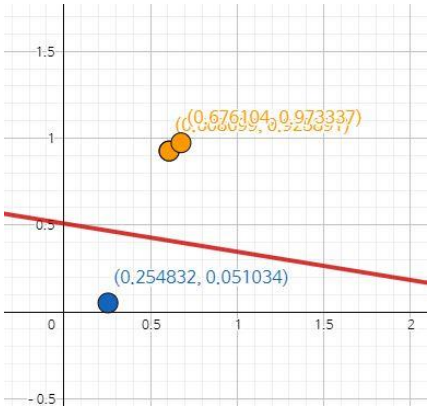
[각 layer의 neuron마다의 weight & bias]

| 학습 시작 전 (Epoch = 0) | 학습 완료 후 (Epoch = 8323) |
| --- | --- |
|  |  |

```
========================= OR gate  =========================

[Epoch 0]

  X1  |  X2  |   OUTPUT
------|------|-----------
   0  |   0  |  0.290512
   0  |   1  |  0.290387
   1  |   0  |   0.29051
   1  |   1  |  0.290379

=> Total Error = 0.797442

  (Layer #1)
             |      W1      |      W2      |     Bias
  -----------|--------------|--------------|-----------
   Neuron #1 |     0.507621 |     0.489852 |  -0.999959
   Neuron #2 |  -0.0197954  |  -0.0553863  |    -1.00007
   Neuron #3 |     0.230344 |     0.724876 |    -1.0001

  (Layer #2)
             |      W1      |      W2      |      W3      |     Bias
  -----------|--------------|--------------|--------------|-----------
   Neuron #1 |     0.924726 |     0.815799 |   -0.120632  |   -0.99994
   Neuron #2 |  -0.0889865  |  -0.0234962  |   -0.247895  |  -0.999815
   Neuron #3 |     0.128651 |     0.803186 |    0.532979  |    -1.00027
   Neuron #4 |      0.63645 |   -0.745116  |   -0.851923  |  -0.999745

  (Layer #3)
             |      W1      |      W2      |      W3      |      W4      |     Bias
  -----------|--------------|--------------|--------------|--------------|-----------
   Neuron #1 |   -0.034812  |     0.292065 |   -0.283781  |     0.497247 |  -0.997899
   Neuron #2 |    -0.36626  |   -0.451959  |    0.629231  |   -0.509188  |    -1.00089

  (Layer #4)
             |      W1      |      W2      |     Bias
  -----------|--------------|--------------|-----------
   Neuron #1 |     0.152467 |  -0.0499787  |  -0.923976

             |      X1      |      X2      |
             |--------------|--------------|
             |     0.283432 |     0.242786 |
             |     0.280481 |     0.245919 |
             |     0.283129 |      0.24209 |
             |      0.28001 |     0.245273 |
```

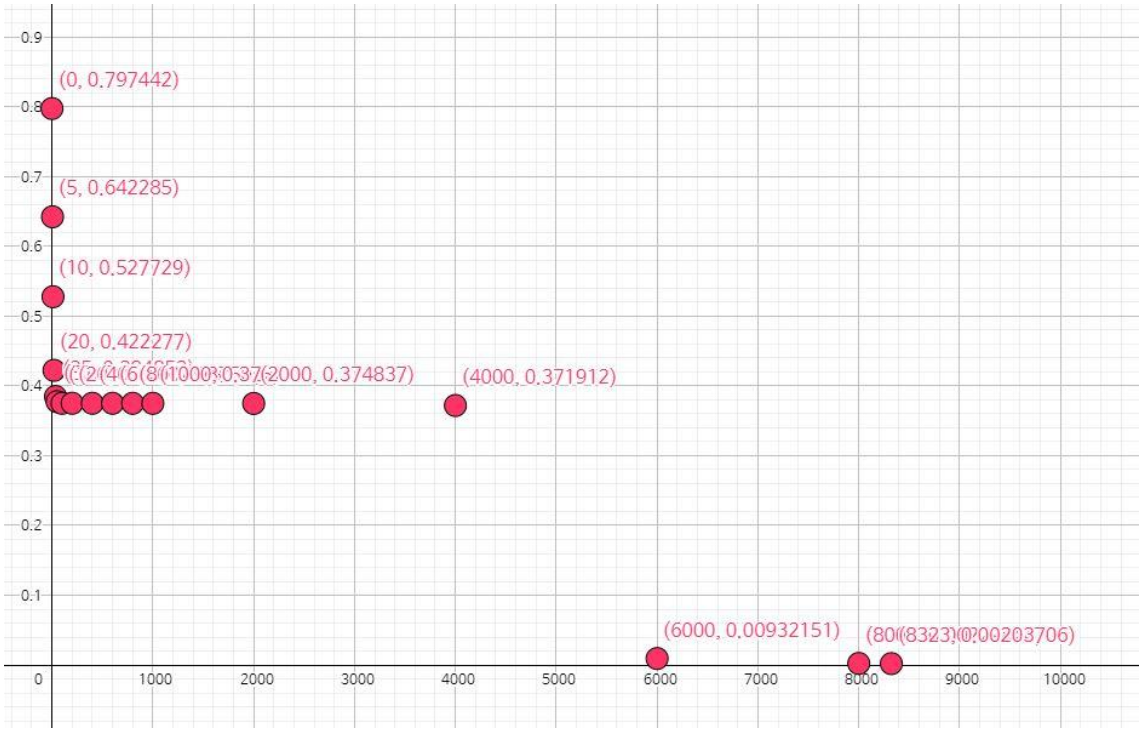
```
[Epoch 8323]

  X1  |  X2  |   OUTPUT
------|------|-----------
   0  |   0  |  0.0499955
   0  |   1  |   0.974661
   1  |   0  |   0.974595
   1  |   1  |   0.983057

=> Total Error = 0.00203706

  (Layer #1)
             |      W1      |      W2      |     Bias
  -----------|--------------|--------------|-----------
   Neuron #1 |      2.15301 |      2.05536 |   -1.29961
   Neuron #2 |      2.01512 |      1.88089 |   -1.30651
   Neuron #3 |      2.58087 |      2.78601 |   -1.59112

  (Layer #2)
             |      W1      |      W2      |      W3      |     Bias
  -----------|--------------|--------------|--------------|-----------
   Neuron #1 |      1.70729 |      1.41665 |    0.893166  |   -1.61307
   Neuron #2 |   -0.658549  |   -0.360617  |    -1.02371  |   0.0906096
   Neuron #3 |      2.11958 |      1.94536 |     3.08706  |    -3.11397
   Neuron #4 |     0.279264 |    -1.00088  |    -1.38919  |   0.0672171

  (Layer #3)
             |      W1      |      W2      |      W3      |      W4      |     Bias
  -----------|--------------|--------------|--------------|--------------|-----------
   Neuron #1 |     0.969882 |   -0.30733   |     1.38586  |  -0.0168514  |   -1.44698
   Neuron #2 |      1.89326 |    -2.04915  |       5.083  |    -2.0719   |   -2.53946

  (Layer #4)
             |      W1      |      W2      |     Bias
  -----------|--------------|--------------|-----------
   Neuron #1 |       1.1513 |      7.06961 |   -3.59871

             |      X1      |      X2      |
             |--------------|--------------|
             |     0.254832 |     0.051034 |
             |     0.607022 |     0.926441 |
             |     0.608099 |     0.925891 |
             |     0.676104 |     0.973337 |

*** 학습이 완료되었습니다. ***

계속하려면 아무 키나 누르십시오 . . .
```

[learning 과정 그래프]

| | 학습 시작 전 (Epoch = 0) | 학습 완료 후 (Epoch = 8323) |
|---|---|---|
| 첫번째 layer |  |  |
| 마지막 layer |  |  |

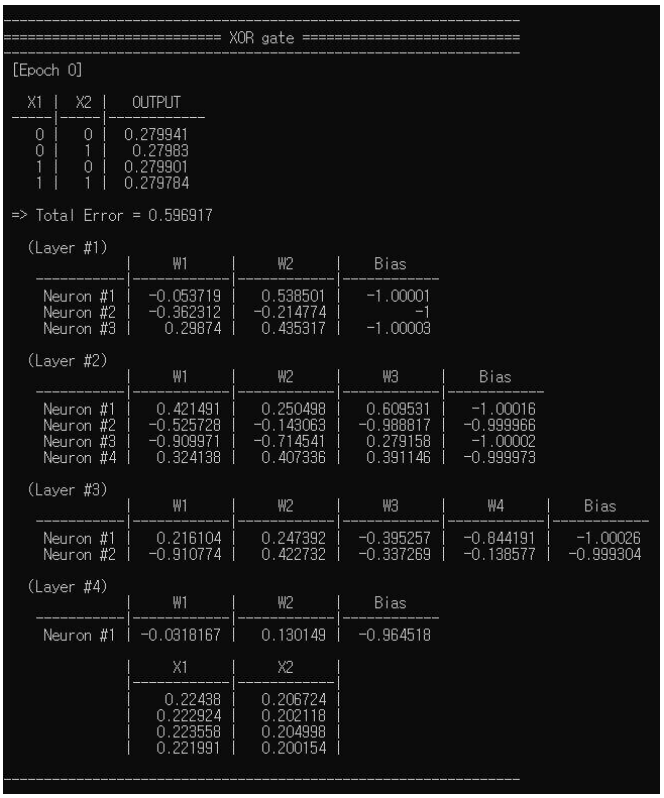
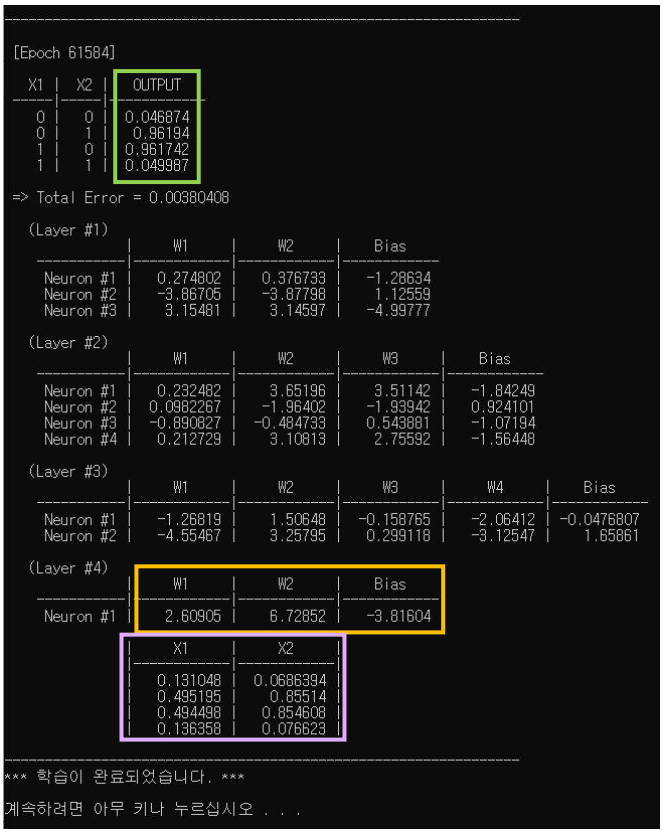
- 마지막 layer에서 input을 나타내는 점이 4개가 아닌 것은 그 개수가 감소한 것이 아니라 겹쳐 있기 때문이다.
- 학습 완료 후, 첫번째 layer와 마지막 layer에서 모두 linearly separable 하다.

[iteration에 따른 error 그래프]



**> XOR gate**

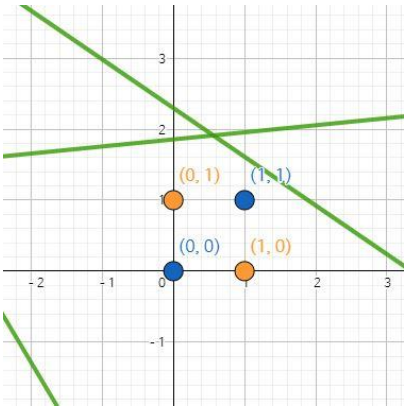
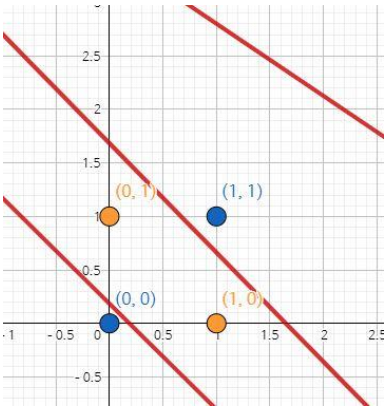
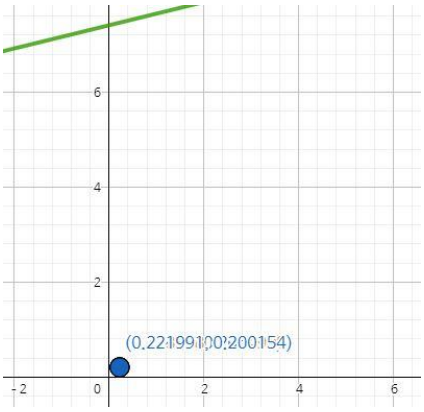
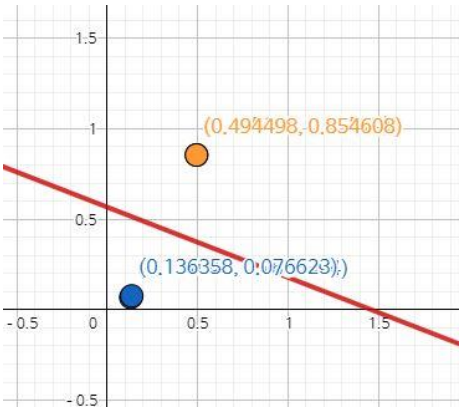
[각 layer의 neuron마다의 weight & bias]

| 학습 시작 전 (Epoch = 0) | 학습 완료 후 (Epoch = 61584) |
| --- | --- |
|  |  |

```
========================= XOR gate =========================

[Epoch 0]

  X1 |  X2 |   OUTPUT
-----|-----|-----------
   0 |   0 |  0.279941
   0 |   1 |   0.27983
   1 |   0 |  0.279901
   1 |   1 |  0.279784

=> Total Error = 0.596917

  (Layer #1)
             |      W1      |      W2      |     Bias
-------------|--------------|--------------|-------------
  Neuron #1  |  -0.053719   |   0.538501   |   -1.00001
  Neuron #2  |  -0.362312   |  -0.214774   |         -1
  Neuron #3  |    0.29874   |   0.435317   |   -1.00003

  (Layer #2)
             |      W1      |      W2      |      W3      |     Bias
-------------|--------------|--------------|--------------|-------------
  Neuron #1  |   0.421491   |   0.250498   |   0.609531   |   -1.00016
  Neuron #2  |  -0.525728   |  -0.143063   |  -0.988817   |  -0.999966
  Neuron #3  |  -0.909971   |  -0.714541   |   0.279158   |   -1.00002
  Neuron #4  |   0.324138   |   0.407336   |   0.391146   |  -0.999973

  (Layer #3)
             |      W1      |      W2      |      W3      |      W4      |     Bias
-------------|--------------|--------------|--------------|--------------|-------------
  Neuron #1  |   0.216104   |   0.247392   |  -0.395257   |  -0.844191   |   -1.00026
  Neuron #2  |  -0.910774   |   0.422732   |  -0.337269   |  -0.138577   |  -0.999304

  (Layer #4)
             |      W1      |      W2      |     Bias
-------------|--------------|--------------|-------------
  Neuron #1  |  -0.0318167  |   0.130149   |  -0.964518

             |      X1      |      X2      |
             |--------------|--------------|
             |    0.22438   |   0.206724   |
             |   0.222924   |   0.202118   |
             |   0.223558   |   0.204998   |
             |   0.221991   |   0.200154   |
```

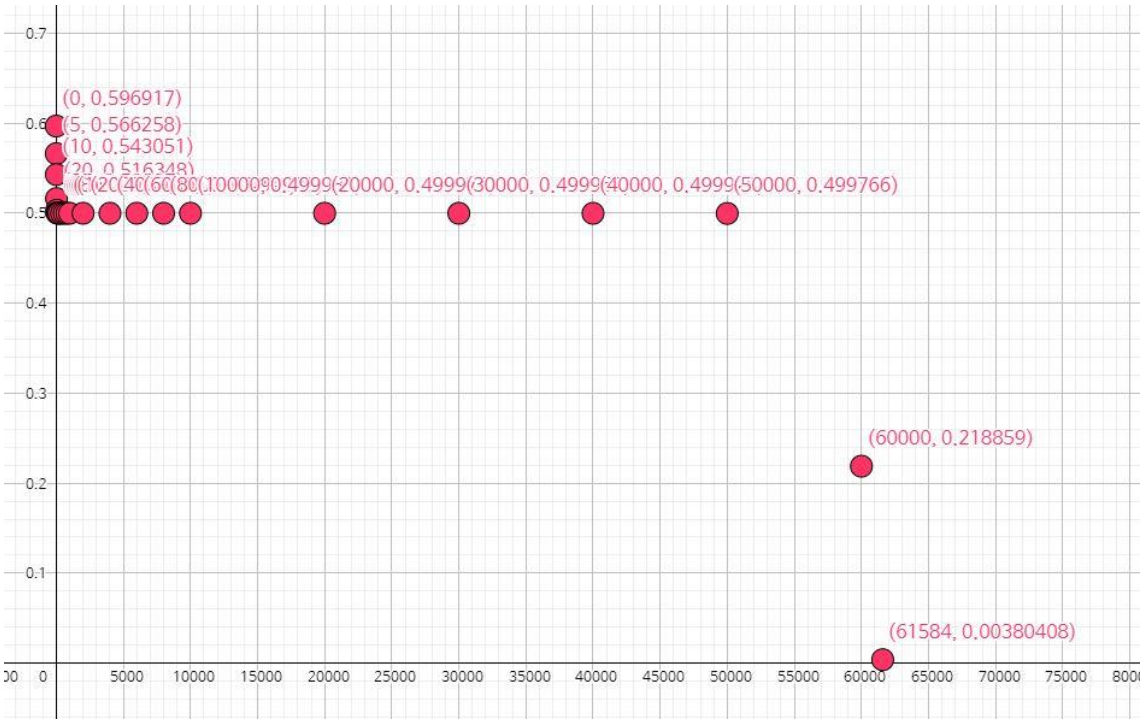
```
[Epoch 61584]

  X1 |  X2 |   OUTPUT
-----|-----|-----------
   0 |   0 |  0.046874
   0 |   1 |   0.96194
   1 |   0 |  0.961742
   1 |   1 |  0.049987

=> Total Error = 0.00380408

  (Layer #1)
             |      W1      |      W2      |     Bias
-------------|--------------|--------------|-------------
  Neuron #1  |   0.274802   |   0.376733   |   -1.28634
  Neuron #2  |   -3.86705   |   -3.87798   |    1.12559
  Neuron #3  |    3.15481   |    3.14597   |   -4.99777

  (Layer #2)
             |      W1      |      W2      |      W3      |     Bias
-------------|--------------|--------------|--------------|-------------
  Neuron #1  |   0.232482   |    3.65196   |    3.51142   |   -1.84249
  Neuron #2  |  0.0982267   |   -1.96402   |   -1.93942   |   0.924101
  Neuron #3  |  -0.890827   |  -0.484733   |   0.543881   |   -1.07194
  Neuron #4  |   0.212729   |    3.10813   |    2.75592   |   -1.56448

  (Layer #3)
             |      W1      |      W2      |      W3      |      W4      |     Bias
-------------|--------------|--------------|--------------|--------------|-------------
  Neuron #1  |   -1.26819   |    1.50648   |  -0.158765   |   -2.06412   |  -0.0476807
  Neuron #2  |   -4.55467   |    3.25795   |   0.299118   |   -3.12547   |    1.65861

  (Layer #4)
             |      W1      |      W2      |     Bias
-------------|--------------|--------------|-------------
  Neuron #1  |    2.60905   |    6.72852   |   -3.81604

             |      X1      |      X2      |
             |--------------|--------------|
             |   0.131048   |  0.0686394   |
             |   0.495195   |    0.85514   |
             |   0.494498   |   0.854608   |
             |   0.136358   |  0.076623    |

*** 학습이 완료되었습니다. ***
계속하려면 아무 키나 누르십시오 . . .
```

[learning 과정 그래프]

| | 학습 시작 전 (Epoch = 0) | 학습 완료 후 (Epoch = 61584) |
|---|---|---|
| 첫번째 layer |  |  |
| 마지막 layer |  |  |

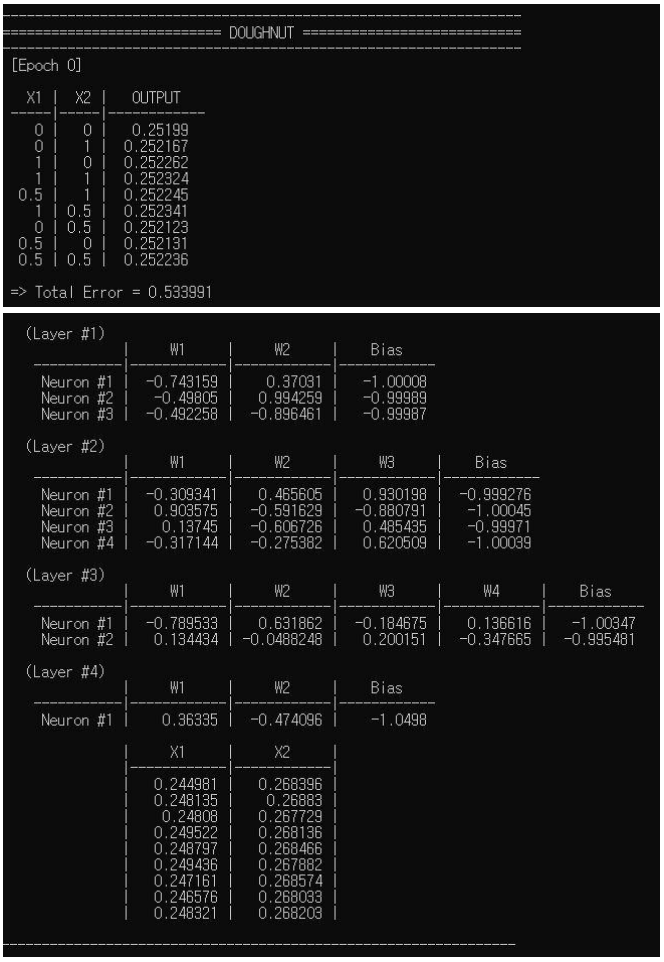
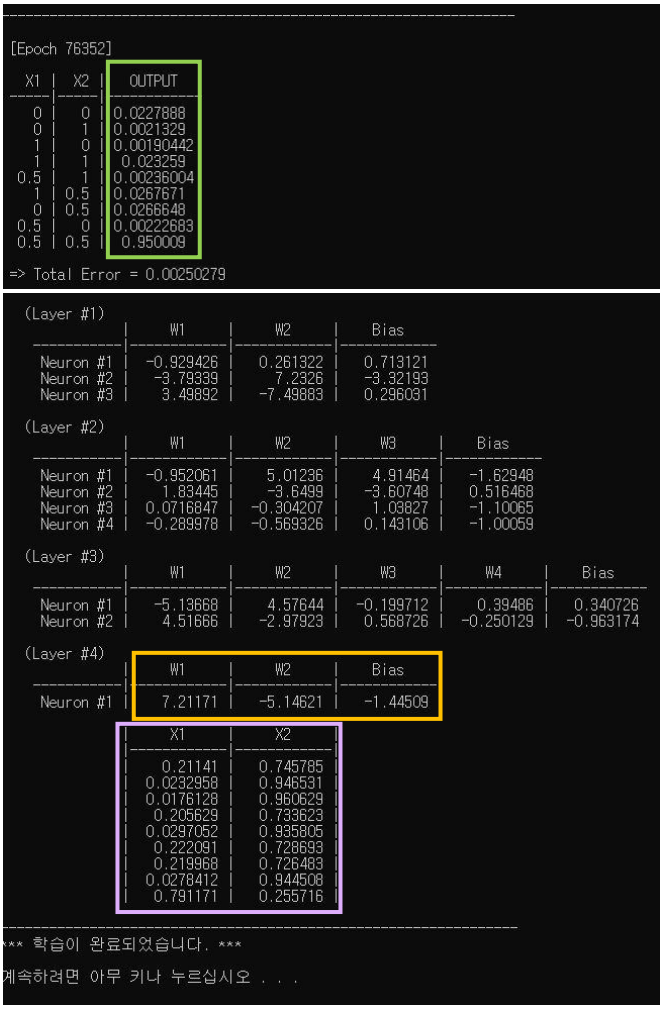
- 마지막 layer에서 input을 나타내는 점이 4개가 아닌 것은 그 개수가 감소한 것이 아니라 겹쳐 있기 때문이다.
- 학습 완료 후, 첫번째 layer에서는 not linearly separable 하지만, 마지막 layer에서는 input이 변형되어 linearly separable 하다.

[iteration에 따른 error 그래프]



**> Doughnut**

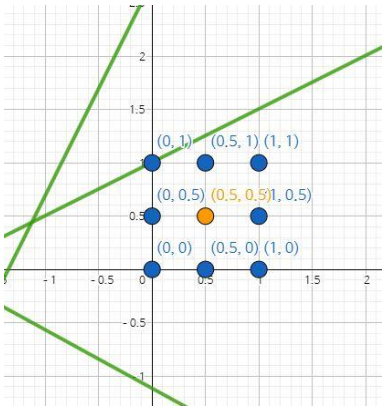
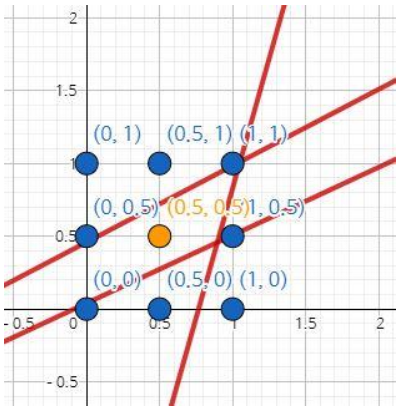
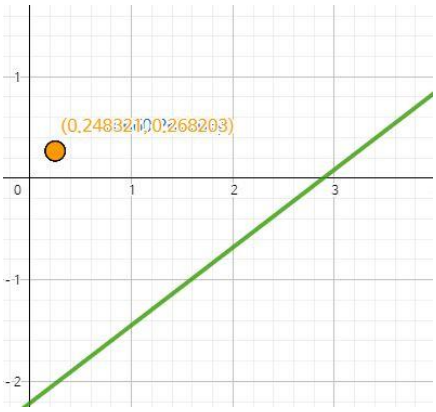
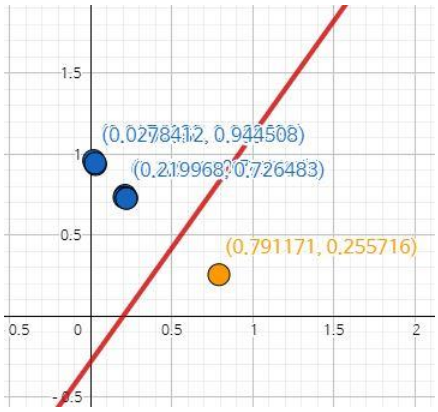
[각 layer의 neuron마다의 weight & bias]

| 학습 시작 전 (Epoch = 0) | 학습 완료 후 (Epoch = 76352) |
| --- | --- |
|  |  |

```
------------------------------------------------------------
========== DOUGHNUT ==========
------------------------------------------------------------
[Epoch 0]

  X1 |  X2 |    OUTPUT
-----|-----|-----------
   0 |   0 |   0.25199
   0 |   1 |  0.252167
   1 |   0 |  0.252262
   1 |   1 |  0.252324
 0.5 |   1 |  0.252245
   1 | 0.5 |  0.252341
   0 | 0.5 |  0.252123
 0.5 |   0 |  0.252131
 0.5 | 0.5 |  0.252236

=> Total Error = 0.533991

 (Layer #1)
            |     W1      |     W2      |     Bias
------------|-------------|-------------|-------------
 Neuron #1  |  -0.743159  |    0.37031  |    -1.00008
 Neuron #2  |   -0.49805  |   0.994259  |   -0.99989
 Neuron #3  |  -0.492258  |  -0.896461  |   -0.99987

 (Layer #2)
            |     W1      |     W2      |     W3      |     Bias
------------|-------------|-------------|-------------|-------------
 Neuron #1  |  -0.309341  |   0.465605  |   0.930198  |  -0.999276
 Neuron #2  |   0.903575  |  -0.591629  |  -0.880791  |   -1.00045
 Neuron #3  |    0.13745  |  -0.606726  |   0.485435  |   -0.99971
 Neuron #4  |  -0.317144  |  -0.275382  |   0.620509  |   -1.00039

 (Layer #3)
            |     W1      |     W2      |     W3      |     W4      |     Bias
------------|-------------|-------------|-------------|-------------|-------------
 Neuron #1  |  -0.789533  |   0.631862  |  -0.184675  |   0.136616  |   -1.00347
 Neuron #2  |   0.134434  |  -0.0488248 |   0.200151  |  -0.347665  |  -0.995481

 (Layer #4)
            |     W1      |     W2      |     Bias
------------|-------------|-------------|-------------
 Neuron #1  |    0.36335  |  -0.474096  |    -1.0498

            |     X1      |     X2      |
            |-------------|-------------|
            |   0.244981  |   0.268396  |
            |   0.248135  |    0.26883  |
            |    0.24808  |   0.267729  |
            |   0.249522  |   0.268136  |
            |   0.248797  |   0.268466  |
            |   0.249436  |   0.267882  |
            |   0.247161  |   0.268574  |
            |   0.246576  |   0.268093  |
            |   0.248321  |   0.268203  |
------------------------------------------------------------
```

```
------------------------------------------------------------
[Epoch 76352]

  X1 |  X2 |    OUTPUT
-----|-----|-----------
   0 |   0 |  0.0227888
   0 |   1 |  0.0021329
   1 |   0 |  0.00190442
   1 |   1 |   0.023259
 0.5 |   1 |  0.00236004
   1 | 0.5 |  0.0267671
   0 | 0.5 |  0.0266648
 0.5 |   0 |  0.00222683
 0.5 | 0.5 |    0.950009

=> Total Error = 0.00250279

 (Layer #1)
            |     W1      |     W2      |     Bias
------------|-------------|-------------|-------------
 Neuron #1  |  -0.929426  |   0.261322  |   0.713121
 Neuron #2  |   -3.79339  |    7.2326   |   -3.32193
 Neuron #3  |    3.49892  |   -7.49883  |   0.296091

 (Layer #2)
            |     W1      |     W2      |     W3      |     Bias
------------|-------------|-------------|-------------|-------------
 Neuron #1  |  -0.952061  |   5.01236   |   4.91464   |  -1.62948
 Neuron #2  |    1.83445  |   -3.6499   |  -3.60748   |   0.516468
 Neuron #3  |  0.0716847  |  -0.304207  |   1.03827   |  -1.10065
 Neuron #4  |  -0.289978  |  -0.569326  |   0.143106  |  -1.00059

 (Layer #3)
            |     W1      |     W2      |     W3      |     W4      |     Bias
------------|-------------|-------------|-------------|-------------|-------------
 Neuron #1  |   -5.13668  |   4.57644   |  -0.199712  |   0.39486   |   0.340726
 Neuron #2  |    4.51666  |   -2.97923  |   0.568726  |  -0.250129  |  -0.963174

 (Layer #4)
            |     W1      |     W2      |     Bias
------------|-------------|-------------|-------------
 Neuron #1  |    7.21171  |   -5.14621  |   -1.44509

            |     X1      |     X2      |
            |-------------|-------------|
            |   0.21141   |   0.745785  |
            |  0.0232958  |   0.946531  |
            |  0.0176128  |   0.960629  |
            |   0.205629  |   0.733623  |
            |  0.0297052  |   0.935805  |
            |   0.222091  |   0.728693  |
            |   0.219968  |   0.726483  |
            |  0.0278412  |   0.944508  |
            |   0.791171  |   0.255716  |
------------------------------------------------------------
*** 학습이 완료되었습니다. ***
계속하려면 아무 키나 누르십시오 . . .
```

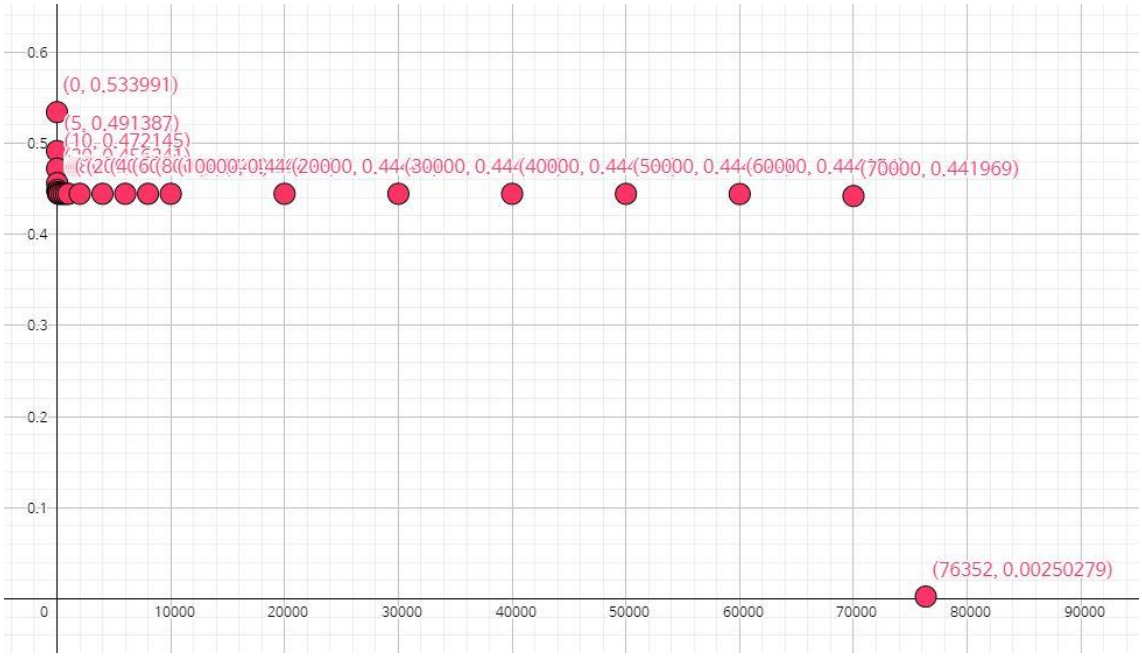
[learning 과정 그래프]

| | 학습 시작 전 (Epoch = 0) | 학습 완료 후 (Epoch = 76352) |
|---|---|---|
| 첫번째 layer |  |  |
| 마지막 layer |  |  |



- 마지막 layer에서 input을 나타내는 점이 9개가 아닌 것은 그 개수가 감소한 것이 아니라 겹쳐 있기 때문이다.
- 학습 완료 후, 첫번째 layer에서는 not linearly separable 하지만, 마지막 layer에서는 input이 변형되어 linearly separable 하다.

[iteration에 따른 error 그래프]



## 3. 결론 및 고찰

**> 결론**

- multi-layer perceptron은 single-layer perceptron에서도 성공적으로 학습되었던 AND gate, OR gate 데이터뿐만 아니라 XOR gate, Doughnut 모양 데이터 또한 성공적으로 학습시킬 수 있다.
- 이는 not linearly separable한 데이터 모델이 multi-layer를 거치며 forward / backward 연산을 통해 마지막 layer에서는 **linearly separable**한 데이터 모델로 변형되었기 때문이다.

**> 고찰**

- 무작정 layer의 개수 또는 각 layer마다의 neuron 개수를 늘린다고 해서 학습이 완료되기까지 소요되는 시간이 감소하는 것은 아니었다. 그리고 layer의 개수를 여러 개로 설정하여 수 차례 실험했을 때, 'layer의 개수가 3개이고 neuron의 개수가 각각 첫번째 layer에 3개, 두번째 layer에 4개, 세번째 layer 1개'인 경우에 모든 학습 데이터에서 최소의 학습 횟수로 학습이 완료되었다. 이 이유에 대해서 탐구해보면 학습을 위해 가장 최적인 multi-layer perceptron을 설계할 수 있을 것이다.
- Activation(Thresholding) function으로 Sigmoid 함수 대신 ReLU 함수, tanh 함수 등을 이용하면 학습 과정 및 횟수가 어떻게 달라질지, 또한 어떤 함수를 이용할 때 가장 효율적인 학습을 진행하는지, 그리고 각 함수에 최적인 데이터 모델의 특징은 무엇인지 추후 탐구해볼 예정이다.
- Doughnut 데이터의 경우, 어떤 실험에서는 20만 번 이상 학습을 진행했음에도 불구하고 학습이 완료되지 않았다. 이는 Sigmoid 함수로 인한 기울기 소실 문제에 기인한 것으로 판단되며, 이를 개선하기 위해서 Sigmoid 함수를 ReLU 함수 등으로 교체하거나 momentum을 부여해볼 수 있을 것이다.