C++ Assignment #1

215788 서유리

1. neuron01.cpp

```cpp
#include <iostream>
#include <ctime>
#include <random>
#include <iomanip>
#include <vector>

// Modify as needed
const int MIN = -1;
const int MAX = 1;

#define sigmoid(x) ( 1.0/(1.0+exp(-(x))) ) // sigmoid

using namespace std;
```

→ 가져다 쓸 라이브러리 파일들, MIN, MAX 는 가중치 생성 시 가중치 범위 설정을(-1~1)로 하기 위해 추가, sigmoid 는 학습할 때 사용할 활성화 함수

```cpp
class Neuron {
private:
    int num, cnt = 0; // cnt : work() 호출 시 pred_node 의 인덱스
    double lr, pred;
    vector <double> w;
    vector < vector <double> > w_node; // 2 차원 weight 벡터 (4x2)
    vector <double> pred_node;
    vector<double> loss;
```

→ Neuron class 의 private 변수
num : 입력 개수, cnt : work() 호출 시 pred_node 의 인덱스 num : 입력 개수, cnt : work() 호출 시 pred_node 의 인덱스, lr : learning rate, pred : learn() 호출시 저장하는 예측값, w : learn() 호출시 저장하는 가중치, w_node : 모든 w 을 담는 벡터, pred_node : 모든 pred 들을 담는 벡터, loss : 예측 정답 차이 -> error

```
public:
   Neuron(int num_of_input, double alpha) : num(num_of_input), lr(alpha) {}

   // learn overloading
   void learn(double(&x)[2], double(&label)) {
      for (int i = 0; i < num+1; i++)
         w.push_back(randomFunc());

      // 입력데이터(x)를 w 벡터에다가 push
      w.push_back(x[0]);
      w.push_back(x[1]);

      // 예측값 구하고
      pred = x[0] * w[0] + x[1] * w[1] + w[2];
      pred = sigmoid(pred);

      // pred_node 벡터에 담기
      pred_node.push_back(pred);
      loss.push_back(pred - label);

      // weight(w,b)들도 w_node 벡터에 담기
      w_node.push_back(w);

   }

   void learn(double(&x)[1], double(&label)) {
      for (int i = 0; i < num+1; i++)
         w.push_back(randomFunc());

      w.push_back(x[0]);
      pred = x[0] * w[0] + w[1];

      pred = sigmoid(pred);
      pred_node.push_back(pred);
      loss.push_back(pred - label);
      w_node.push_back(w);

   }
```

→ learn() 함수
AND/OR 연산과 NOT 연산은 입력의 개수가 다르기 때문에 오버로딩
입력의 개수가 다르니 가중치의 개수도 달라짐(개수 == 생성자 호출할 때 선언된 num)
For 문에서 num+1 을 한 이유는 b 라는 바이어스를 더해주기 위한 것
구조 : 랜덤으로 생성한 가중치와 입력값들로 선형식을 만들고 sigmoid 를 사용해서 비선형식으로 만든게 예측값, 그리고 예측값과 실제값의 차이로 loss(error)를 만듬

```
void fix() {

	int len = loss.size();
	// AND, OR
	if (num == 2) {
		for (int i = 0; i < len; i++) {
			// weight update
			w_node[i][0] = w_node[i][0] - lr * (loss[i] * w[3] * pred_node[i] * (1
- pred_node[i])); // w[3], w[4] = input
			w_node[i][1] = w_node[i][1] - lr * (loss[i] * w[4] * pred_node[i] * (1
- pred_node[i]));
			w_node[i][2] = w_node[i][2] - lr * (loss[i] * pred_node[i] * (1 -
pred_node[i]));

			// update된 weight들로 다시 예측값 구하기
			pred = w[3] * w_node[i][0] + w[4] * w_node[i][1] + w_node[i][2];
			pred_node[i] = sigmoid(pred);
		}
	}
	// NOT
	else {
		for (int i = 0; i < len; i++) {
			w_node[i][0] = w_node[i][0] - lr * (loss[i] * w[2] * pred_node[i] * (1
- pred_node[i])); // w[2] = input
			w_node[i][1] = w_node[i][1] - lr * (loss[i] * pred_node[i] * (1 -
pred_node[i]));

			pred = w[2] * w_node[i][0] + w_node[i][1];
			pred_node[i] = sigmoid(pred);

		}
	}

}
```

→ fix() 함수
Learn()에서 생성한 에러값(loss)과 예측값(pred)를 통해 가중치를 업데이트 해주는 함수
AND/OR 연산과 NOT 연산을 구별해주기 위해 if 문 사용

```
// work overloading
    double work(double(&x)[2]) {
        if (cnt == 4) cnt = 0;
        double out = pred_node[cnt];
        cnt += 1;
        return out;
    }

    double work(double(&x)[1]) {
        if (cnt == 2) cnt = 0;
        double out = pred_node[cnt];
        cnt += 1;
        return out;
    }

    // random 숫자 생성 함수
    double randomFunc() {
        std::random_device rd;
        std::default_random_engine eng(rd());
        std::uniform_real_distribution<double> distr(MIN, MAX);

        double n = distr(eng);

        return n;
    }
};
```

→ work() : 해당되는 예측값 반환 (오버로딩 사용)
→ randomFunc() : -1 ~ 1 사이의 값 반환

```
int main() {

    // 뉴런 클래스 생성자.
    // Neuron(int num_of_input, double alpha)
                //(입력의 수, learning rate)
    Neuron* neuron = new Neuron(2, 0.1);

    // AND
    double sample_input[4][2] = { {0,0},{0,1},{1,0},{1,1} };
    double sample_output[4] = { 0, 0, 0, 1 };

    for (int i = 0; i < 5000; i++) {
        for (int j = 0; j < 4; j++) {
            neuron->learn(sample_input[j], sample_output[j]); // 학습
        }
        neuron->fix(); // 가중치 수정

        if ((i + 1) % 100 == 0) {
```

```
        cout << "------ Learn " << i + 1 << " times -----" << endl;
        for (int j = 0; j < 4; j++) {
            cout << sample_input[j][0] << ' ' << sample_input[j][1] << " : "
                << neuron->work(sample_input[j]) << endl; // 결과값
        }
    }
}
delete neuron;

// NOT
Neuron* neuron2 = new Neuron(1, 0.1);

double sample_input2[2][1] = { {0}, {1} };
double sample_output2[2] = { 1, 0 };

for (int i = 0; i < 5000; i++) {
    for (int j = 0; j < 2; j++) {
        neuron2->learn(sample_input2[j], sample_output2[j]); // 학습
    }
    neuron2->fix(); // 가중치 수정

    if ((i + 1) % 100 == 0) {
        cout << "------ Learn " << i + 1 << " times -----" << endl;
        for (int j = 0; j < 2; j++) {
            cout << sample_input2[j][0] << " : "
                << neuron2->work(sample_input2[j]) << endl; // 결과값
        }
    }
}
delete neuron2;

// OR
Neuron* neuron3 = new Neuron(2, 0.1);

double sample_input3[4][2] = { {0,0},{0,1},{1,0},{1,1} };
double sample_output3[4] = { 0, 1, 1, 1 };

for (int i = 0; i < 5000; i++) {
    for (int j = 0; j < 4; j++) {
        neuron3->learn(sample_input3[j], sample_output3[j]); // 학습
    }
    neuron3->fix(); // 가중치 수정

    if ((i + 1) % 100 == 0) {
        cout << "------ Learn " << i + 1 << " times -----" << endl;
        for (int j = 0; j < 4; j++) {
            cout << sample_input3[j][0] << ' ' << sample_input3[j][1] << " : "
                << neuron3->work(sample_input3[j]) << endl; // 결과값
        }
    }
```

```
    }
    delete neuron3;

    return 0;
}
```

→ main()
neuron 객체 3 개 생성 (AND -> NOT -> OR)

2. neuron02.cpp

```
#include <iostream>
#include <ctime>
#include <random>
#include <iomanip>
#include <vector>


// Modify as needed
const int MIN = -1;
const int MAX = 1;

#define sigmoid(x) ( 1.0/(1.0+exp(-(x))) ) // sigmoid

using namespace std;

class Neuron {
private:
    int num, cnt = 0;
    double lr, pred;
    vector < vector <double> > w_node; // 2 차원 weight 벡터 (4x2)
    vector <double> pred_node;
    vector <double> w;
    vector<double> loss;
public:
    Neuron(int num_of_input, double alpha) : num(num_of_input), lr(alpha) {}

    // learn overloading
    void learn(double(&x)[2], double(&label)) {
        for (int i = 0; i < num+1; i++)
            w.push_back(randomFunc());

        // 입력데이터(x)를 w 벡터에다가 push
        w.push_back(x[0]);
        w.push_back(x[1]);

        // 예측값 구하고
        pred = x[0] * w[0] + x[1] * w[1] + w[2];
        pred = sigmoid(pred);
```

```
        // pred_node 벡터에 담기
        pred_node.push_back(pred);
        loss.push_back(pred - label);

        // weight(w,b)들도 w_node 벡터에 담기
        w_node.push_back(w);

    }

    void learn(double(&x)[1], double(&label)) {
        for (int i = 0; i < num+1; i++)
            w.push_back(randomFunc());

        w.push_back(x[0]);
        pred = x[0] * w[0] + w[1];

        pred = sigmoid(pred);
        pred_node.push_back(pred);
        loss.push_back(pred - label);
        w_node.push_back(w);

    }

    void fix() {

        int len = loss.size();
        // AND, OR
        if (num == 2) {
            for (int i = 0; i < len; i++) {
                // weight update
                w_node[i][0] = w_node[i][0] - lr * (loss[i] * w[3] * pred_node[i] * (1
- pred_node[i])); // w[3], w[4] = input
                w_node[i][1] = w_node[i][1] - lr * (loss[i] * w[4] * pred_node[i] * (1
- pred_node[i]));
                w_node[i][2] = w_node[i][2] - lr * (loss[i] * pred_node[i] * (1 -
pred_node[i]));

                // update된 weight들로 다시 예측값 구하기
                pred = w[3] * w_node[i][0] + w[4] * w_node[i][1] + w_node[i][2];
                pred_node[i] = sigmoid(pred);
            }
        }
        // NOT
        else {
            for (int i = 0; i < len; i++) {
                w_node[i][0] = w_node[i][0] - lr * (loss[i] * w[2] * pred_node[i] * (1
- pred_node[i])); // w[2] = input
                w_node[i][1] = w_node[i][1] - lr * (loss[i] * pred_node[i] * (1 -
pred_node[i]));
```

```
            pred = w[2] * w_node[i][0] + w_node[i][1];
            pred_node[i] = sigmoid(pred);

        }
    }

}

// work overloading
double work(double(&x)[2]) {
    if (cnt == 4) cnt = 0;
    double out = pred_node[cnt];
    cnt += 1;
    return out;
}

double work(double(&x)[1]) {
    if (cnt == 4) cnt = 0;
    double out = pred_node[cnt];
    cnt += 1;
    return out;
}

// pred_node(예측값) 반환 함수
vector<double> &retPred() {
    return pred_node;
}
// random 숫자 생성 함수
double randomFunc() {
    std::random_device rd;
    std::default_random_engine eng(rd());
    std::uniform_real_distribution<double> distr(MIN, MAX);

    double n = distr(eng);
    //cout << n << endl;

    return n;
}
};
```

→ Neuron class 는 1 번과 동일하게 사용
차이점은 randomFunc() 위에 retPred()는 학습한 객체들의 예측값 벡터(pred_node)를 반환해주는 함수가 존재함 (XOR 은 not -> and -> or 순으로 연산을 하기 때문에 그 사이사이에 결과값을 넘겨주기 위해 필요)

```
int main() {

   // step1. ~x1, ~x2 학습 (NOT 연산)
   Neuron* nx1 = new Neuron(1, 0.1);
   Neuron* nx2 = new Neuron(1, 0.1);

   double nx1_input[4][1] = { {0}, {0}, {1}, {1} };
   double nx1_output[4] = { 1, 1, 0, 0};

   double nx2_input[4][1] = { {0}, {1}, {0}, {1} };
   double nx2_output[4] = { 1, 0, 1, 0};

   for (int i = 0; i < 500; i++) {
      for (int j = 0; j < 4; j++) {
         nx1->learn(nx1_input[j], nx1_output[j]);
         nx2->learn(nx2_input[j], nx2_output[j]); // 학습
      }
      nx1->fix();
      nx2->fix(); // 가중치 수정

      if ((i + 1) % 100 == 0) {
         cout << "------ Learn " << i + 1 << " times -----" << endl;
         cout << "nx1 nx2    nx1 pred nx2 pred " << endl;
         for (int j = 0; j < 4; j++) {
            cout << nx1_input[j][0] << "    " << nx2_input[j][0] << "  :  "
               << nx1->work(nx1_input[j]) << ' ' << nx2->work(nx2_input[j]) <<
endl;


         }
      }
   }
```

→ NOT 연산 : ~x1, ~x2 구하기
(nx_input : 처음 x 값, nx_output : NOT 연산 실제 결과 값)

```
// step2. x1 && ~x2 , x2 && ~x1 학습 (AND 연산)
   Neuron* ax1 = new Neuron(2, 0.1);
   Neuron* ax2 = new Neuron(2, 0.1);

   double ax1_input[4][2] = { {0,}, {0,}, {1,}, {1,} };
   double ax1_output[4] = { 0, 0, 1, 0};

   double ax2_input[4][2] = { {0,}, {1,}, {0,}, {1,} };
   double ax2_output[4] = { 0, 1, 0, 0};

   // neuron 이 예측한 값 받아오기
   vector<double> p1, p2;
   p1 = nx1->retPred();
   p2 = nx2->retPred();
```

```
    // 출력값 가공 -> 0.5 기준으로 1, 0 구별
    for(int i = 0; i < 4; i++){
        if (p1[i] > 0.5) ax2_input[i][1] = 1;
        else ax2_input[i][1] = 0;
        if (p2[i] > 0.5) ax1_input[i][1] = 1;
        else ax1_input[i][1] = 0;
    }

    /*
    -- 그냥 출력값 그대로 쓰고 싶다면 이 코드 이용 --
    for(int i = 0; i < 4; i++){
        ax1_input[i][1] = p1[i];
        ax2_input[i][1] = p2[i];
    }
    */

    for (int i = 0; i < 500; i++) {
        for (int j = 0; j < 4; j++) {
            ax1->learn(ax1_input[j], ax1_output[j]);
            ax2->learn(ax2_input[j], ax2_output[j]); // 학습
        }
        ax1->fix();
        ax2->fix(); // 가중치 수정

        // Print result //
        if ((i + 1) % 100 == 0) {
            cout << "------ Learn " << i + 1 << " times -----" << endl;
            cout << "ax1           |  ax2 " << endl;
            for (int j = 0; j < 4; j++) {
                cout << ax1_input[j][0] << ' ' << ax1_input[j][1] << " : " << ax1-
>work(ax1_input[j]) << " | "
                    << ax2_input[j][0] << ' ' << ax2_input[j][1] << " : " << ax2-
>work(ax2_input[j]) << endl;


            }
        }
    }
```

→ AND 연산 : x1 && ~x2 와 x2 && ~x1 구하기
(각각의 output 은, ax1 -> 0 0 1 0 (0 0 1 1 && 1 0 1 0), ax2 -> 0 1 0 0 (0 1 0 1 && 1 1 0 0))
p1, p2 : 다음 연산을 위해 그 전에 학습한(NOT) 벡터들의 결과를 반환 받는 벡터들
편리하게 예측값 > 0.5 → 1, 예측값 < 0.5 → 0 으로 취급해서 입력데이터를 구성하고 학습

```
// step3. (x1 && ~x2) || (x2 && ~x1) 학습 (OR 연산) -> 최종 XOR
    Neuron* ox = new Neuron(2, 0.1);

    double ox_input[4][2] = {};
    double ox_output[4] = { 0, 1, 1, 0};

    p1 = ax1->retPred();
    p2 = ax2->retPred();

    for(int i = 0; i < 4; i++){
        if (p1[i] > 0.5) ox_input[i][0] = 1;
        else ox_input[i][0] = 0;
        if (p2[i] > 0.5) ox_input[i][1] = 1;
        else ox_input[i][1] = 0;
    }

    /*for (int i = 0; i < 4; i++){
        cout << ox_input[i][0] << ' ' << ox_input[i][1] << endl;
    }*/

    for (int i = 0; i < 500; i++) {
        for (int j = 0; j < 4; j++) {
            ox->learn(ax1_input[j], ox_output[j]);
        }
        ox->fix();

        if ((i + 1) % 100 == 0) {
            cout << "------ Learn " << i + 1 << " times -----" << endl;
            for (int j = 0; j < 4; j++) {
                cout << ox_input[j][0] << ' ' << ox_input[j][1] << " : " << ox-
>work(ox_input[j])<< endl;
            }
        }
    }
    return 0;
}
```

→ OR 연산
최종 XOR 결과값, 따라서 ox_output 은 XOR 연산의 출력값과 같음