E1.py

```python
from random import choice

N = 100


def sgn(x):
    if x < 0:
        return -1
    if x >= 0:
        return 1


def generate_patterns(p, n):
    # p is the number of patterns to generate
    # n is the number of bits
    result = list()
    for pattern in range(p):
        new_pattern = list()
        for bit in range(n):
            new_pattern.append(choice([-1, 1]))
        result.append(new_pattern)
    return result


def learn_patterns(patterns, n):
    w, h = n, n
    W = [[0 for x in range(w)] for y in range(h)]

    for p in patterns:
        for i in range(n):
            for j in range(i, n):
                if i != j:
                    W[i][j] += 1 / n * p[i] * p[j]
                else:
                    W[i][j] = 0
                W[j][i] = W[i][j]
    return W


def get_p_error_estimation(p):
    n_errors = 0

    for trial in range(100000):

        patterns = generate_patterns(p, N)
        # learn patterns
        W = learn_patterns(patterns, N)

        # choose one random pattern to feed
        v = choice(patterns)
        neuron = choice(range(N))

        previous_state = v[neuron]

        new_state = 0
        for j in range(N):
            new_state += W[neuron][j] * v[j]

        if sgn(new_state) != previous_state:
            n_errors += 1

    return n_errors / 100000


P_values = [12, 20, 40, 60, 80, 100]

for p in P_values:
    p_error = get_p_error_estimation(p)
    print(str(p) + " : " + str(p_error))
```

main.cpp

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

#include <algorithm>

#include "patterns.h"

using namespace std;


void writeVector(vector<int> *v) {

    cout << "[";
    int index = 0;
    for (auto &s : *v) {
        if (index % W == 0) {
            cout << "\n[";
        }

        cout << ((s == -1) ? 0 : 1) << "\t";

        if (index % W == (W - 1)) {
            cout << "],\n";
        } else {
            cout << ',';
        }

        index++;

    }

    cout << "]\n";
}


inline int sgn(double x) {
    if (x < 0)
        return -1;
    return 1;
}

class MatchStatus {
public:
    bool isFound;
    bool isReverse;
    char pattern;
};

class Hopfield {
public:
    double w[N][N] = {0};

    void train(vector<int> *p) {
        auto pattern = *p;
        for (int i = 0; i < N; i++) {
```

```cpp
                for (int j = 0; j < N; j++) {
                    if (i == j)
                        w[i][j] = 0;
                    else
                        w[i][j] += ((double) 1 / N) * pattern[i] * pattern[j];
                }
            }

        }


        vector<int> *classify(vector<int> *pattern) {
            auto *state = new vector<int>(*pattern);

            bool inSteadyState = false;


            while (!inSteadyState) {


                auto *previous_state = new vector<int>(*state);

                for (int neuron = 0; neuron < N; neuron++) {

                    double res = 0;
                    for (int j = 0; j < N; j++) {
                        res += w[neuron][j] * (*state)[j];
                    }

                    (*state)[neuron] = sgn(res);

                }


                //check if we reached steady state

                if (equal(previous_state->begin(), previous_state->end(), state->begin())) {
                    inSteadyState = true;
                } else {
                    delete previous_state;
                }
            }

            return state;
        }
    };

    vector<int> *matrix2vector(int matrix[H][W]) {
        auto v = new vector<int>();
        for (int i = 0; i < H; i++)
            for (int j = 0; j < W; j++)
                v->push_back(matrix[i][j]);

        return v;

    }


    bool match(vector<int> *first, vector<int> *second) {
```

```cpp
    for (int i = 0; i < N; i++) {
        if ((*first)[i] != (*second)[i])
            return false;
    }
    return true;
}


void reverseVector(vector<int> *v) {
    for (int &i : *v)
        i = i == -1 ? 1 : -1;
}

MatchStatus *findMatch(vector<int> *output, vector<vector<int> *> *patterns) {
    auto result = new MatchStatus();

    result->isFound = false;

    char patternIndex = '1';
    for (const auto &p : *patterns) {
        if (match(output, p)) {
            result->isFound = true;
            result->isReverse = false;
            result->pattern = patternIndex;
            break;
        }

        reverseVector(p);
        if (match(output, p)) {
            result->isReverse = true;
            result->isFound = true;
            result->pattern = patternIndex;
            break;
        }

        patternIndex++;
    }

    return result;
}


int test_1[H][W] = {{-1, 1,  1,  -1, -1, -1, -1, 1,  1,  -1},
                    {-1, 1,  1,  -1, -1, -1, -1, 1,  1,  -1},
                    {-1, 1,  1,  -1, -1, -1, -1, 1,  1,  -1},
                    {-1, 1,  1,  -1, -1, -1, -1, 1,  1,  -1},
                    {-1, 1,  1,  -1, -1, -1, -1, 1,  1,  -1},
                    {-1, 1,  1,  -1, -1, -1, -1, 1,  1,  -1},
                    {-1, 1,  1,  -1, -1, -1, -1, 1,  1,  -1},
                    {-1, 1,  1,  1,  1,  1,  1,  1,  1,  -1},
                    {-1, 1,  1,  1,  1,  1,  1,  1,  1,  -1},
                    {-1, -1, -1, -1, -1, -1, -1, 1,  1,  -1},
                    {-1, -1, -1, -1, -1, -1, -1, 1,  1,  -1},
                    {-1, -1, -1, -1, -1, -1, -1, 1,  1,  -1},
                    {-1, -1, -1, -1, -1, -1, -1, 1,  1,  -1},
                    {-1, -1, -1, -1, -1, -1, -1, 1,  1,  -1},
                    {1,  1,  1,  1,  1,  1,  1,  -1, -1, 1},
                    {1,  1,  1,  1,  1,  1,  1,  -1, -1, 1}};
```

```cpp
int test_2[H][W] = {{1, 1,  1,  1,  1,  1,  1,  1,  1,  1},
                    {1, 1,  1,  -1, -1, -1, -1, 1,  1,  1},
                    {1, 1,  -1, -1, -1, -1, -1, -1, 1,  1},
                    {1, -1, -1, -1, 1,  1,  -1, -1, -1, 1},
                    {1, -1, -1, -1, 1,  1,  -1, -1, -1, 1},
                    {1, -1, -1, -1, 1,  1,  -1, -1, -1, 1},
                    {1, -1, -1, -1, 1,  1,  -1, -1, -1, 1},
                    {1, -1, -1, -1, 1,  1,  -1, -1, -1, 1},
                    {1, -1, -1, -1, 1,  1,  -1, -1, -1, 1},
                    {1, -1, -1, -1, 1,  1,  -1, -1, -1, 1},
                    {1, -1, -1, -1, 1,  1,  -1, -1, -1, 1},
                    {1, -1, -1, -1, 1,  1,  -1, -1, -1, 1},
                    {1, -1, -1, -1, 1,  1,  -1, -1, -1, 1},
                    {1, 1,  -1, -1, -1, -1, -1, -1, 1,  1},
                    {1, 1,  1,  -1, -1, -1, -1, 1,  1,  1},
                    {1, 1,  1,  1,  1,  1,  1,  1,  1,  1}};


int test_3[H][W] = {{1,  -1, -1, 1,  -1, -1, -1, -1, 1,  1},
                    {-1, -1, -1, -1, -1, 1,  1,  1,  -1, -1},
                    {-1, -1, 1,  -1, -1, -1, -1, -1, -1, -1},
                    {1,  1,  -1, -1, 1,  -1, -1, -1, -1, 1},
                    {1,  -1, -1, 1,  -1, 1,  1,  -1, -1, -1},
                    {-1, 1,  -1, -1, 1,  -1, -1, -1, -1, -1},
                    {1,  1,  -1, -1, -1, -1, -1, 1,  -1, -1},
                    {1,  -1, 1,  -1, 1,  1,  1,  1,  -1, 1},
                    {-1, -1, -1, -1, 1,  -1, 1,  -1, -1, 1},
                    {1,  -1, 1,  -1, -1, -1, 1,  -1, -1, -1},
                    {-1, 1,  -1, -1, -1, -1, -1, 1,  -1, -1},
                    {-1, -1, -1, 1,  1,  1,  -1, -1, -1, 1},
                    {-1, 1,  -1, 1,  -1, 1,  1,  -1, -1, 1},
                    {-1, -1, -1, 1,  1,  -1, -1, 1,  1,  1},
                    {1,  1,  -1, 1,  1,  -1, -1, -1, 1,  1},
                    {-1, 1,  1,  -1, -1, -1, -1, -1, 1,  -1}};

int main() {
    auto hopfield = new Hopfield();

    auto patterns = new vector<vector<int> *>();


    patterns->push_back(matrix2vector(x1));
    patterns->push_back(matrix2vector(x2));
    patterns->push_back(matrix2vector(x3));
    patterns->push_back(matrix2vector(x4));
    patterns->push_back(matrix2vector(x5));

    for (const auto &pattern  : *patterns) {
        hopfield->train(pattern);
    }


    auto result = hopfield->classify(matrix2vector(test_3)); //test_1, test_2

    auto match = findMatch(result, patterns);

    writeVector(result);

    if (match->isFound) {
        cout << (match->isReverse ? '-' : ' ') << match->pattern << endl;
    } else {
        cout << "6\n";
    }


    return 0;
}
```

patterns.h

```
#ifndef PATTERNS_H
#define PATTERNS_H

#define N 160
#define W 10
#define H 16

int x1[H][W] = {
        {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {-1, -1, -1, 1,  1,  1,  1,  -1, -1, -1},
        {-1, -1, 1,  1,  1,  1,  1,  1,  -1, -1},
        {-1, 1,  1,  1,  -1, -1, 1,  1,  1,  -1},
        {-1, 1,  1,  1,  -1, -1, 1,  1,  1,  -1},
        {-1, 1,  1,  1,  -1, -1, 1,  1,  1,  -1},
        {-1, 1,  1,  1,  -1, -1, 1,  1,  1,  -1},
        {-1, 1,  1,  1,  -1, -1, 1,  1,  1,  -1},
        {-1, 1,  1,  1,  -1, -1, 1,  1,  1,  -1},
        {-1, 1,  1,  1,  -1, -1, 1,  1,  1,  -1},
        {-1, 1,  1,  1,  -1, -1, 1,  1,  1,  -1},
        {-1, 1,  1,  1,  -1, -1, 1,  1,  1,  -1},
        {-1, 1,  1,  1,  -1, -1, 1,  1,  1,  -1},
        {-1, -1, 1,  1,  1,  1,  1,  1,  -1, -1},
        {-1, -1, -1, 1,  1,  1,  1,  -1, -1, -1},
        {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}
};

int x2[H][W] = {{-1, -1, -1, 1, 1, 1, 1, -1, -1, -1},
                {-1, -1, -1, 1, 1, 1, 1, -1, -1, -1},
                {-1, -1, -1, 1, 1, 1, 1, -1, -1, -1},
                {-1, -1, -1, 1, 1, 1, 1, -1, -1, -1},
                {-1, -1, -1, 1, 1, 1, 1, -1, -1, -1},
                {-1, -1, -1, 1, 1, 1, 1, -1, -1, -1},
                {-1, -1, -1, 1, 1, 1, 1, -1, -1, -1},
                {-1, -1, -1, 1, 1, 1, 1, -1, -1, -1},
                {-1, -1, -1, 1, 1, 1, 1, -1, -1, -1},
                {-1, -1, -1, 1, 1, 1, 1, -1, -1, -1},
                {-1, -1, -1, 1, 1, 1, 1, -1, -1, -1},
                {-1, -1, -1, 1, 1, 1, 1, -1, -1, -1},
                {-1, -1, -1, 1, 1, 1, 1, -1, -1, -1},
                {-1, -1, -1, 1, 1, 1, 1, -1, -1, -1},
                {-1, -1, -1, 1, 1, 1, 1, -1, -1, -1},
                {-1, -1, -1, 1, 1, 1, 1, -1, -1, -1}
};

int x3[H][W] = {
        {1,  1,  1,  1,  1,  1,  1,  1,  -1, -1},
        {1,  1,  1,  1,  1,  1,  1,  1,  -1, -1},
        {-1, -1, -1, -1, -1, 1,  1,  1,  -1, -1},
        {-1, -1, -1, -1, -1, 1,  1,  1,  -1, -1},
        {-1, -1, -1, -1, -1, 1,  1,  1,  -1, -1},
        {-1, -1, -1, -1, -1, 1,  1,  1,  -1, -1},
        {-1, -1, -1, -1, -1, 1,  1,  1,  -1, -1},
        {1,  1,  1,  1,  1,  1,  1,  1,  -1, -1},
        {1,  1,  1,  1,  1,  1,  1,  1,  -1, -1},
        {1,  1,  1,  -1, -1, -1, -1, -1, -1, -1},
        {1,  1,  1,  -1, -1, -1, -1, -1, -1, -1},
        {1,  1,  1,  -1, -1, -1, -1, -1, -1, -1},
        {1,  1,  1,  -1, -1, -1, -1, -1, -1, -1},
        {1,  1,  1,  -1, -1, -1, -1, -1, -1, -1},
        {1,  1,  1,  1,  1,  1,  1,  1,  -1, -1},
        {1,  1,  1,  1,  1,  1,  1,  1,  -1, -1}};

int x4[H][W] = {{-1, -1, 1,  1,  1,  1,  1, 1, -1, -1},
                {-1, -1, 1,  1,  1,  1,  1, 1, 1,  -1},
                {-1, -1, -1, -1, -1, -1, 1, 1, 1,  -1},
                {-1, -1, -1, -1, -1, -1, 1, 1, 1,  -1},
                {-1, -1, -1, -1, -1, -1, 1, 1, 1,  -1},
                {-1, -1, -1, -1, -1, -1, 1, 1, 1,  -1},
                {-1, -1, -1, -1, -1, -1, 1, 1, 1,  -1},
                {-1, -1, 1,  1,  1,  1,  1, 1, -1, -1},
                {-1, -1, 1,  1,  1,  1,  1, 1, -1, -1},
                {-1, -1, -1, -1, -1, -1, 1, 1, 1,  -1},
                {-1, -1, -1, -1, -1, -1, 1, 1, 1,  -1},
                {-1, -1, -1, -1, -1, -1, 1, 1, 1,  -1},
                {-1, -1, -1, -1, -1, -1, 1, 1, 1,  -1},
                {-1, -1, -1, -1, -1, -1, 1, 1, 1,  -1},
                {-1, -1, 1,  1,  1,  1,  1, 1, 1,  -1},
                {-1, -1, 1,  1,  1,  1,  1, 1, -1, -1}};

int x5[H][W] = {{-1, 1,  1,  -1, -1, -1, -1, 1, 1, -1},
                {-1, 1,  1,  -1, -1, -1, -1, 1, 1, -1},
                {-1, 1,  1,  -1, -1, -1, -1, 1, 1, -1},
                {-1, 1,  1,  -1, -1, -1, -1, 1, 1, -1},
                {-1, 1,  1,  -1, -1, -1, -1, 1, 1, -1},
                {-1, 1,  1,  -1, -1, -1, -1, 1, 1, -1},
                {-1, 1,  1,  -1, -1, -1, -1, 1, 1, -1},
                {-1, 1,  1,  1,  1,  1,  1,  1, 1, -1},
                {-1, 1,  1,  1,  1,  1,  1,  1, 1, -1},
                {-1, -1, -1, -1, -1, -1, -1, 1, 1, -1},
                {-1, -1, -1, -1, -1, -1, -1, 1, 1, -1},
                {-1, -1, -1, -1, -1, -1, -1, 1, 1, -1},
                {-1, -1, -1, -1, -1, -1, -1, 1, 1, -1},
                {-1, -1, -1, -1, -1, -1, -1, 1, 1, -1},
                {-1, -1, -1, -1, -1, -1, -1, 1, 1, -1},
                {-1, -1, -1, -1, -1, -1, -1, 1, 1, -1}};

#endif //PATTERNS_H
```

main.cpp

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

#include <algorithm>


using namespace std;

#define N 200
#define BETA 2
#define T 100000


inline double g(double x) {
    return 1 / (1 + exp(-2 * x * BETA));
}

int stochastic(double x) {

    std::random_device rd;
    std::mt19937 gen(rd());
    std::bernoulli_distribution dis(g(x));

    return dis(gen) ? 1 : -1;
}


auto generatePatterns(int p) {

    vector<vector<int> *> patterns;

    std::random_device rd;
    std::mt19937 gen(rd());
    std::bernoulli_distribution dis(0.5);


    for (int i = 0; i < p; i++) {
        auto vec = new vector<int>;
        for (int j = 0; j < N; j++) {
            vec->push_back(dis(gen) ? 1 : -1);
        }

        patterns.push_back(vec);
    }


    return patterns;

}


class Hopfield {
public:
    double w[N][N] = {0};
```

```cpp
    void train(vector<int> *p) {
        auto pattern = *p;
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (i == j)
                    w[i][j] = 0;
                else
                    w[i][j] += ((double) 1 / N) * pattern[i] * pattern[j];
            }
        }

    }


    double calculateOrderParameter(vector<int> *pattern) {
        auto *state = new vector<int>(*pattern);


        double orderParameter = 0.0;

        for (int t = 0; t < T; t++) {

            for (int neuron = 0; neuron < N; neuron++) {


                double res = 0;
                for (int j = 0; j < N; j++) {
                    res += w[neuron][j] * (*state)[j];
                }

                (*state)[neuron] = stochastic(res);
                orderParameter += (*state)[neuron] * (*pattern)[neuron];
            }

            orderParameter /= N;


            if (!(t % 1000)) {
                cout << "Done with " << t << " trials\t" << orderParameter << "\n";
            }

        }

        return orderParameter;

    }
};


int main() {
    auto hopfield = new Hopfield();


    auto answer = 0.0;
    auto times = 0;
    while (times++ <= 100) {
        auto patterns = generatePatterns(40); // 5 (for the first task)

        for (const auto &pattern  : patterns) {
            hopfield->train(pattern);
        }

        auto temp = hopfield->calculateOrderParameter(patterns[0]);

        cout << "At the end of trial " << times << " we got" << temp;
        answer += temp;
        cout << "\nCurrent average = " << answer / times << endl;
        for (auto pattern: patterns) {
            delete (pattern);
        }
    }

    cout << "Final answer is\n" << (answer / 100) << endl;


    return 0;
}
```