Министерство образования Республики Беларусь

Учреждение образования

«Брестский государственный технический университет»

Кафедра ИИТ

**Лабораторная работа №4**

**Тема: «Нелинейные ИНС в задачах прогнозирования»**

**Выполнил:**

Студент 4 курса

Факультета ЭИС

Группы АС-55

Черноокий И.В.

**Проверил:**

Савицкий Ю.В.

Брест 2022

**Лабораторная работа №4**

**«Нелинейные ИНС в задачах прогнозирования»**

**Цель работы:** Изучить обучение и функционирование нелинейной ИНС при решении задач прогнозирования.

**Задание:** Написать на любом ЯВУ программу моделирования прогнозирующей нелинейной ИНС. Для тестирования использовать функцию

** .

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № варианта | a | b | с | d | Кол-во входов ИНС | Кол-во НЭ в скрытом слое |
| 3 | 0.3 | 0.3 | 0.07 | 0.3 | 10 | 4 |

**Текст программы:**

**Класс активационной функции:**

class ActivationFunction {

public:

virtual ~ActivationFunction() { }

virtual double active(double input) = 0;

virtual double derivative(double input, double output) = 0;

};

class AFSigmoid : public ActivationFunction {

public:

~AFSigmoid() { }

inline void setBeta(double value = 1.0) { beta = value; }

double active(double input) override;

double derivative(double input, double output) override;

private:

double beta = 1.0;

};

double AFSigmoid::active(double input) {

return 1.0 / (1.0 + exp(-beta \* input));

}

double AFSigmoid::derivative(double input, double output) {

return beta \* (1.0 - output) \* output;

}

**Класс Нейрона:**

class Neuron

{

public:

Neuron() { }

double input = 0;

double output = 0;

double propagationError = 0;

ActivationFunction\* function = nullptr;

void active();

double derivative();

};

void Neuron::active() {

output = function->active(input);

}

double Neuron::derivative() {

return function->derivative(input, output);

}

**Класс слоя и нейронной сети:**

class Layer

{

public:

Layer(int count);

Layer(int count, Layer& nextLayer);

std::vector<Neuron> neurons;

std::vector<double> weights;

double getWeight(int i, int j);

void setWeight(int i, int j, double value);

double getThreshold(int j);

void setThreshold(int j, double value);

};

class NeuralNet

{

public:

double alpha = 0.01;

NeuralNet(int inputSize, int hiddenCount, int hiddenSize, int outputSize);

std::vector<Layer> layers;

void input(std::vector<double>& inputData);

void feedForward();

std::vector<double> outputVector();

void trainNetwork(std::vector<double>& trainData);

void setActivationFunction(Layer& layer, std::shared\_ptr<ActivationFunction>& function);

double deviation(std::vector<double>& trainData);

};

**Прямое распространение по сети:**

void NeuralNet::feedForward() {

for (int l = 1; l < layers.size(); l++) {

for (int j = 0; j < layers[l].neurons.size(); j++) {

double S = 0.0;

for (int i = 0; i < layers[l-1].neurons.size(); i++) {

S += layers[l-1].neurons[i].output \* layers[l-1].getWeight(i, j);

}

S -= layers[l-1].getThreshold(j);

layers[l].neurons[j].input = S;

layers[l].neurons[j].active();

}

}

}

**Обратное распространение ошибки по сети:**

void NeuralNet::trainNetwork(std::vector<double>& trainData) {

feedForward();

int currentLayer = layers.size() - 1;

int nextLayer = layers.size() - 1;

alpha = 0.0;

for (int i = 0; i < layers[0].neurons.size(); i++)

alpha += layers[0].neurons[i].output \* layers[0].neurons[i].output;

alpha = 1.0 / (1.0 + alpha);

for (size\_t i = 0; i < layers[currentLayer].neurons.size(); i++) {

layers[currentLayer].neurons[i].propagationError = layers[currentLayer].neurons[i].output - trainData[i];

}

for (int nl = layers.size() - 2; nl > 0; nl--) {

currentLayer = nl;

nextLayer = nl + (size\_t)1;

for (int i = 0; i < layers[currentLayer].neurons.size(); i++){

double y = 0.0;

for (int j = 0; j < layers[nextLayer].neurons.size(); j++){

y += layers[nextLayer].neurons[j].propagationError \* layers[nextLayer].neurons[j].derivative() \* layers[currentLayer].getWeight(i, j);

}

layers[currentLayer].neurons[i].propagationError = y;

}

}

for (int i = 0; i < layers.size() - 1; i++) {

for (int k = 0; k < layers[i].neurons.size(); k++)

for (int j = 0; j < layers[i + (size\_t)1].neurons.size(); j++) {

layers[i].setWeight(k, j, layers[i].getWeight(k, j) - alpha \* layers[i + (size\_t)1].neurons[j].propagationError \* layers[i + (size\_t)1].neurons[j].derivative() \* layers[i].neurons[k].output);

}

for (int j = 0; j < layers[i + (size\_t)1].neurons.size(); j++) {

double T = layers[i].getThreshold(j) + alpha \* layers[i + (size\_t)1].neurons[j].propagationError \* layers[i + (size\_t)1].neurons[j].derivative();

layers[i].setThreshold(j, T);

}

}

}

**Среднеквадратическое отклонение одного обучающего образа:**

double NeuralNet::deviation(std::vector<double>& trainData) {

double dev = 0.0;

for (int i = 0; i < layers[layers.size() - 1].neurons.size(); i++){

dev += pow(layers[layers.size() - 1].neurons[i].output - trainData[i], 2);

}

return 0.5 \* dev;

}

double sinFunction(double x) {

double a = 0.3, b = 0.3, c = 0.07, d = 0.3;

return a \* cos(b \* x) + c \* sin(d \* x);

}

int main() {

std::shared\_ptr<ActivationFunction> sigmoid = std::shared\_ptr<ActivationFunction>(new AFSigmoid());

std::shared\_ptr<ActivationFunction> linear = std::shared\_ptr<ActivationFunction>(new AFLinear());

int hiddenCount = 1, inputCount = 5, outputCount = 1;

NeuralNet nn(inputCount, hiddenCount, 4, outputCount);

std::vector<std::vector<double>> inputData, trainData;

std::vector<double> outputData;

std::cout << "TRAIN" << std::endl;

double time = 0.0;

std::vector<double> set, set2;

int trainSize = 500;

for (int i = 0; i < inputCount; i++) {

set.push\_back(sinFunction(time));

time += 0.1;

}

for (int i = 0; i < outputCount; i++) {

set2.push\_back(sinFunction(time));

time += 0.1;

}

for (int k = 0; k < trainSize; k++) {

inputData.push\_back(set);

trainData.push\_back(set2);

set.erase(set.begin());

set.push\_back(sinFunction(time));

set2.erase(set2.begin());

set2.push\_back(sinFunction(time + 0.1));

time += 0.1;

}

double td = 200.0;

while (td >= 0.0001) {

td = 0;

for (int i = 0; i < inputData.size(); i++) {

nn.input(inputData[i]);

nn.trainNetwork(trainData[i]);

td += nn.deviation(trainData[i]);

}

}

std::cout << "Total deviation: " << td << std::endl;

std::cout << "\nTraining ends at " << time << " seconds." << std::endl;

std::cout << "Attempt to prediction sin function from " << time << " seconds..." << std::endl;

std::cout << "\tStandard\t\tOutput\t\t\tDeviation" << std::endl;

outputData = nn.outputVector();

time = double(rand() % 100);

double E = 0;

for (int k = 0; k < 50; k++) {

inputData.push\_back(set);

trainData.push\_back(set2);

set.erase(set.begin());

set.push\_back(sinFunction(time));

set2.erase(set2.begin());

set2.push\_back(sinFunction(time + 0.1));

time += 0.1;

}

for (int i = 0; i < inputData.size(); i++) {

nn.input(inputData[i]);

outputData = nn.outputVector();

E = 0.5 \* pow(outputData.at(0) - trainData[i][trainData[i].size() - 1], 2);

std::cout << "#" << i + 1 << "\t" << std::to\_string(trainData[i][trainData[i].size() - 1]).substr(0, 7) << "\t\t\t"

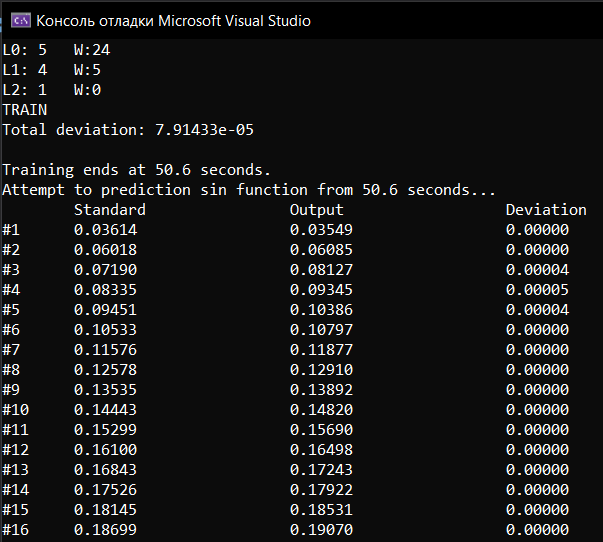
<< std::to\_string(outputData.at(0)).substr(0, 7) << "\t\t\t" << std::to\_string(E).substr(0, 7) << std::endl;

}

return 0;

}

**Результат работы:**

**

**Вывод:** Изучил алгоритм обучения и функционирования нелинейной ИНС при решении задач прогнозирования.