Реализовать гибридный алгоритм стохастической оптимизации, включающий:

* формирование обучающей выборки для вычисления математического ожидания функции посредством нейросети с помощью метода статистического моделирования;
* формирование нейросети и ее обучение для вычисления математического ожидания целевой функции;
* формирование генетического алгоритма для отыскания оптимального решения и отыскание решения, причем в качестве средства вычисления функции соответствия должна использоваться нейросеть.

Вариант 3



Тестирование программы

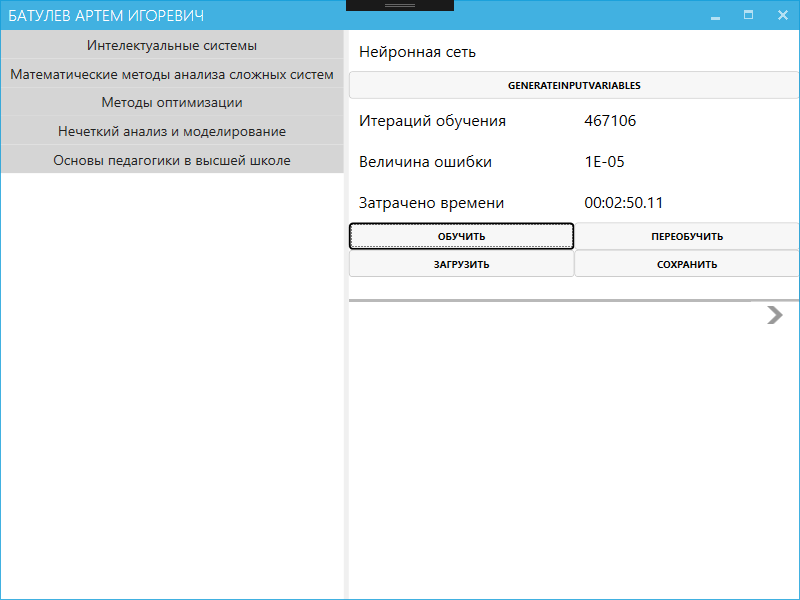
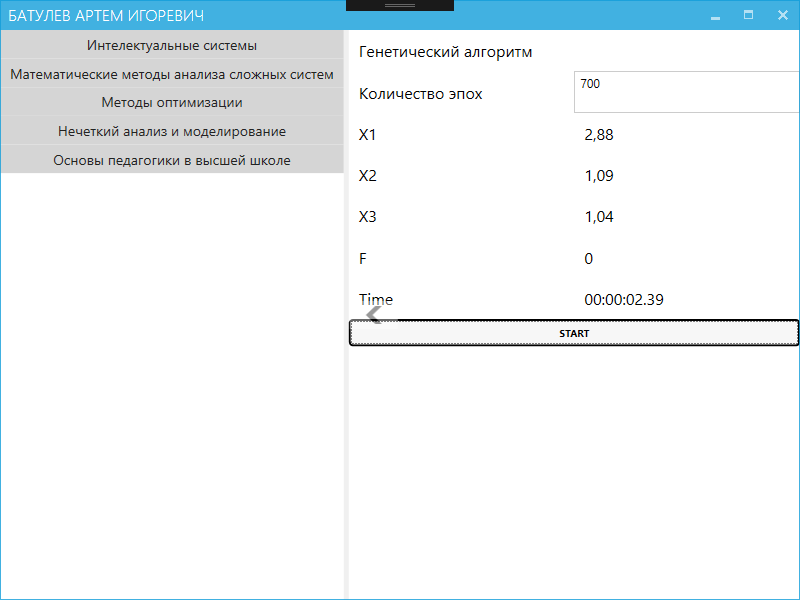


Рисунок 1 – Обучение нейронной сети

 Рисунок 2 – Генетический алгоритм

Листинг программы

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Windows.Input;

using Prism.Commands;

using Wolfram.NETLink;

namespace OptimizationMethods.ViewModels.Lab4

{

public class NeuralNetwork : BaseJob

{

Random randObj = new Random();

/\* Массивы весовых коэффициентов нейронной сети \*/

public double[] W01 = new double[50];

public double[,] W1 = new double[50, 3];

public double[] W2 = new double[50];

public double W02;

/\* Переменные для обучения нейронной сети \*/

public double max, min;

private string \_iteration;

private string \_errorValue;

private string \_trainingTime;

private bool \_isLearnung;

public NeuralNetwork(MathKernel mathKernel)

: base(mathKernel)

{

InputVariables = new double[100, 3];

Fsr = new double[100];

dataGridView1 = new DataGridView();

dataGridView1.ColumnCount = 6;

LoadNeuralNetworkCommand = new DelegateCommand(LoadTrainingNeuralNetwork);

SaveNeuralNetworkCommand = new DelegateCommand(SaveTrainingNeuralNetwork, () => IsLearnung);

GenerateInputVariablesCommand = new DelegateCommand(GenerateInputVariables);

RetrainNeuralNetworkCommand = new DelegateCommand(RetrainNeuralNetwork, () => IsLearnung);

LearnNeuralNetworkCommand = new DelegateCommand(StartLearn, () => Fsr.Any(x => x > 0) || IsLearnung);

}

public DelegateCommand LoadNeuralNetworkCommand { get; set; }

public DelegateCommand SaveNeuralNetworkCommand { get; set; }

public DelegateCommand GenerateInputVariablesCommand { get; set; }

public DelegateCommand RetrainNeuralNetworkCommand { get; set; }

public DelegateCommand LearnNeuralNetworkCommand { get; set; }

public DataGridView dataGridView1 { get; set; }

public double[,] InputVariables { get; set; }

public double[] Fsr { get; set; }

public string TrainingTime

{

get { return \_trainingTime; }

set { SetProperty(ref \_trainingTime , value); }

}

public string ErrorValue

{

get { return \_errorValue; }

set { SetProperty(ref \_errorValue, value); }

}

public string Iteration

{

get { return \_iteration; }

set { SetProperty(ref \_iteration, value); }

}

public bool IsLearnung

{

get { return \_isLearnung; }

set

{

\_isLearnung = value;

RetrainNeuralNetworkCommand.RaiseCanExecuteChanged();

SaveNeuralNetworkCommand.RaiseCanExecuteChanged();

LearnNeuralNetworkCommand.RaiseCanExecuteChanged();

}

}

public void StartLearn()

{

MeasureTime(() => Initialization(true));

IsLearnung = true;

}

public void RetrainNeuralNetwork()

{

MeasureTime(() => Initialization(false));

}

private void GenerateInputVariables()

{

Random r = new Random();

int k = 0;

while (k < 100)

{

double x1 = (double)(r.Next(600)) / 100;

double x2 = (double)(r.Next(600)) / 100;

double x3 = (double)(r.Next(600)) / 100;

if ((x1 + x2 + x3 < 41) && (2 \* x1 + x2 + 3 \* x3 < 41))

{

dataGridView1.Rows.Add(new DataGridViewRow());

dataGridView1.Rows[k].Cells[0].Value = (k + 1).ToString();

dataGridView1.Rows[k].Cells[1].Value = x1.ToString();

dataGridView1.Rows[k].Cells[2].Value = x2.ToString();

dataGridView1.Rows[k].Cells[3].Value = x3.ToString();

Compute(" a1=RandomVariate[UniformDistribution[{1, 2}],200];"

+ "a2=RandomVariate[ExponentialDistribution[0.35],200];"

+ "a3=RandomVariate[UniformDistribution[{3, 4}],200];");

Compute("f=(" + x1.ToString().Replace(",", ".") + "-a1^2)^2+3\*(" + x2.ToString().Replace(",", ".") + "-a2^2)^2+2\*(" + x3.ToString().Replace(",", ".") + "+a3^2)^2;");

dataGridView1.Rows[k].Cells[4].Value = Compute("Total[f]/200");

k++;

}

}

for (int i = 0; i < 100; i++)

{

InputVariables[i, 0] = Convert.ToDouble(dataGridView1.Rows[i].Cells[1].Value.ToString().Replace(".", ","));

InputVariables[i, 1] = Convert.ToDouble(dataGridView1.Rows[i].Cells[2].Value.ToString().Replace(".", ","));

InputVariables[i, 2] = Convert.ToDouble(dataGridView1.Rows[i].Cells[3].Value.ToString().Replace(".", ","));

Fsr[i] = Convert.ToDouble(dataGridView1.Rows[i].Cells[4].Value.ToString().Replace(".", ","));

}

LearnNeuralNetworkCommand.RaiseCanExecuteChanged();

}

private void LoadTrainingNeuralNetwork()

{

try

{

NeuralNetworkHelper.ReadArray(1, this);

NeuralNetworkHelper.ReadArray(2, this);

NeuralNetworkHelper.ReadArray(3, this);

NeuralNetworkHelper.ReadArray(4, this);

NeuralNetworkHelper.ReadArray(5, this);

NeuralNetworkHelper.ReadArray(6, this);

NeuralNetworkHelper.ReadArray(7, this);

NeuralNetworkHelper.ReadArray(8, this);

NeuralNetworkHelper.DisplayTrainingNeuralNetwork(dataGridView1, InputVariables, Fsr, this);

IsLearnung = true;

}

catch (Exception ex)

{

MessageBox.Show(ex.Message, "Ошибка!", MessageBoxButtons.OK);

return;

}

}

private void SaveTrainingNeuralNetwork()

{

try

{

NeuralNetworkHelper.WriteArray(1, this);

NeuralNetworkHelper.WriteArray(2, this);

NeuralNetworkHelper.WriteArray(3, this);

NeuralNetworkHelper.WriteArray(4, this);

NeuralNetworkHelper.WriteArray(5, this);

NeuralNetworkHelper.WriteArray(6, this);

NeuralNetworkHelper.WriteArray(7, this);

NeuralNetworkHelper.WriteArray(8, this);

MessageBox.Show("Запись завершена.", "Успешно!", MessageBoxButtons.OK);

}

catch (Exception e1)

{

MessageBox.Show(Convert.ToString(e1), "Ошибка!", MessageBoxButtons.OK);

return;

}

}

private void MeasureTime(Action a)

{

var myStopWatch = new Stopwatch();

myStopWatch.Start();

a();

myStopWatch.Stop();

TimeSpan ts = myStopWatch.Elapsed;

string elapsedTime = String.Format("{0:00}:{1:00}:{2:00}.{3:00}", ts.Hours, ts.Minutes, ts.Seconds, ts.Milliseconds / 10);

TrainingTime = elapsedTime;

}

/\* Инициализация. Создание и обучение нейронной сети \*/

private void Initialization(bool exp)

{

double sum, er\_sr;

double[] er = new double[100];

bool perem = true;

int z = 0;

if (exp)

for (int i = 0; i < 50; i++)

{

W01[i] = (double)randObj.Next(100) / (double)100000;

W2[i] = (double)randObj.Next(100) / (double)100000;

W02 = (double)randObj.Next(100) / (double)100000;

for (int j = 0; j < 3; j++)

W1[i, j] = (double)randObj.Next(100) / (double)100000;

}

min = 999999;

max = -999999;

for (int i = 0; i < 100; i++)

{

if (Fsr[i] > max)

{

max = Fsr[i];

}

if (Fsr[i] < min)

{

min = Fsr[i];

}

}

er\_sr = 99999;

while (perem)

{

sum = 0;

for (int i = 0; i < 100; i++)

{

er[i] = TrainingNeuralNetwork(InputVariables[i, 0], InputVariables[i, 1], InputVariables[i, 2], (Fsr[i] - min) / (2 \* (max - min)));

sum += Math.Pow(er[i], 2);

}

er\_sr = sum / 100;

z++;

if (er\_sr < 0.00001)

perem = false;

//Application.DoEvents();

ErrorValue = Convert.ToString(Math.Round(er\_sr, 6));

Iteration = z.ToString(); ;

}

NeuralNetworkHelper.DisplayTrainingNeuralNetwork(dataGridView1, InputVariables, Fsr, this);

}

/\* Обучение нейросети \*/

double TrainingNeuralNetwork(double x1, double x2, double x3, double f)

{

double[] layer1 = new double[50];

double etta, NetworkOutput, r, Delta1;

double[] Delta2 = new double[50];

double[] vh = new double[3];

etta = (double)0.2;

vh[0] = x1;

vh[1] = x2;

vh[2] = x3;

NetworkOutput = 0;

for (int i = 0; i < 50; i++)

{

layer1[i] = NeuralNetworkHelper.ActivationFunction(W01[i] + W1[i, 0] \* x1 + W1[i, 1] \* x2 + W1[i, 2] \* x3);

NetworkOutput += layer1[i] \* W2[i];

}

r = NeuralNetworkHelper.ActivationFunction(NetworkOutput + W02);

Delta1 = r \* (1 - r) \* (r - f);

for (int i = 1; i < 50; i++)

{

Delta2[i] = layer1[i] \* (1 - layer1[i]) \* Delta1 \* W2[i];

W2[i] = W2[i] - etta \* Delta1 \* layer1[i]; ;

for (int j = 0; j < 3; j++)

W1[i, j] = W1[i, j] - etta \* Delta2[i] \* vh[j];

}

for (int i = 1; i < 50; i++)

W01[i] = W01[i] - etta \* Delta2[i];

W02 = W02 - etta \* Delta1;

return Math.Abs(f - r);

}

}

}

using System;

using System.Diagnostics;

using System.Windows.Input;

using Prism.Commands;

using Prism.Mvvm;

namespace OptimizationMethods.ViewModels.Lab4

{

public class GeneticAlgorithm : BindableBase

{

private readonly NeuralNetwork \_neuralNetwork;

/\* Массивы генетического алгоритма \*/

double[,] Parents = new double[300, 4];

double[,] Children = new double[300, 3];

double[,] MutatedChildren = new double[300, 4];

/\* Вспомогательные переменные\*/

double max, min;

Random randObj = new Random();

int CountMutations;

private int \_numberOfPeriods;

public GeneticAlgorithm(NeuralNetwork neuralNetwork)

{

\_neuralNetwork = neuralNetwork;

NumberOfPeriods = 700;

StartGeneticAlgorithmCommand = new DelegateCommand(Start, () => neuralNetwork.IsLearnung);

}

public ICommand StartGeneticAlgorithmCommand { get; set; }

public int NumberOfPeriods

{

get { return \_numberOfPeriods; }

set { SetProperty(ref \_numberOfPeriods, value); }

}

public string Time { get; set; }

public string F { get; set; }

public string X3 { get; set; }

public string X2 { get; set; }

public string X1 { get; set; }

public void Start()

{

var myStopWatch = new Stopwatch();

myStopWatch.Start();

int m = NumberOfPeriods;

StartPopulation();

for (int i = 1; i <= m; i++)

{

Crossingover();

Mutation();

Sorting();

Selection();

}

myStopWatch.Stop();

TimeSpan ts = myStopWatch.Elapsed;

Time = String.Format("{0:00}:{1:00}:{2:00}.{3:00}", ts.Hours, ts.Minutes, ts.Seconds, ts.Milliseconds / 10);

X1 = Convert.ToString(Parents[0, 0]);

X2 = Convert.ToString(Parents[0, 1]);

X3 = Convert.ToString(Parents[0, 2]);

F = Convert.ToString(Math.Round(

NeuralNetworkHelper.NeuralNetworkEvaluate(Parents[0, 0], Parents[0, 1], Parents[0, 2], \_neuralNetwork) \* 2 \* (max - min) + min, 3));

}

void StartPopulation()

{

double x1, x2, x3;

for (int i = 0; i < 300; i++)

{

x1 = (double)randObj.Next(600) / (double)100;

x2 = (double)randObj.Next(600) / (double)100;

x3 = (double)randObj.Next(600) / (double)100;

if ((x1 + x2 + x3 < 17) && (2 \* x1 + x2 + 3 \* x3 < 51))

{

Parents[i, 0] = x1;

Parents[i, 1] = x2;

Parents[i, 2] = x3;

}

}

}

void Crossingover()

{

int nParent1, nParent2;

double[] Parents1 = new double[3];

double[] Parents2 = new double[3];

for (int i = 0; i < 300; i++)

{

nParent1 = randObj.Next(299) + 1;

nParent2 = randObj.Next(299) + 1;

while (nParent1 == nParent2)

nParent2 = randObj.Next(299) + 1;

for (int j = 0; j < 3; j++)

{

Parents1[j] = Parents[nParent1, j];

Parents2[j] = Parents[nParent2, j];

}

Children[i, 0] = Parents1[0];

Children[i, 1] = Parents1[1];

Children[i, 2] = Parents2[2];

}

}

void Mutation()

{

int k = 0;

for (int i = 0; i < 300; i++)

{

Children[i, randObj.Next(1) + 1] = Parents[randObj.Next(299) + 1, randObj.Next(1) + 1];

if ((Children[k, 0] + Children[k, 1] + Children[k, 2]) <= 40 && 2 \* Children[k, 0] + Children[k, 1] + 3 \* Children[k, 2] <= 40)

{

MutatedChildren[k, 0] = Children[i, 0];

MutatedChildren[k, 1] = Children[i, 1];

MutatedChildren[k, 2] = Children[i, 2];

k++;

}

}

CountMutations = k;

for (int i = 0; i < k; i++)

{

MutatedChildren[i, 3] =

NeuralNetworkHelper.NeuralNetworkEvaluate(MutatedChildren[i, 0], MutatedChildren[i, 1], MutatedChildren[i, 2], \_neuralNetwork);

}

}

void Selection()

{

int k, proc70;

double x1, x2, x3, buf;

for (int i = 0; i < 300; i++)

{

Parents[i, 3] = NeuralNetworkHelper.NeuralNetworkEvaluate(Parents[i, 0], Parents[i, 1], Parents[i, 2], \_neuralNetwork);

}

for (int i = 0; i < 300; i++)

for (int j = 0; j < 299; j++)

{

if (Parents[j, 3] < Parents[j + 1, 3])

{

buf = Parents[j, 3];

Parents[j, 3] = Parents[j + 1, 3];

Parents[j + 1, 3] = buf;

buf = Parents[j, 0];

Parents[j, 0] = Parents[j + 1, 0];

Parents[j + 1, 0] = buf;

buf = Parents[j, 1];

Parents[j, 1] = Parents[j + 1, 1];

Parents[j + 1, 1] = buf;

buf = Parents[j, 2];

Parents[j, 2] = Parents[j + 1, 2];

Parents[j + 1, 2] = buf;

}

}

proc70 = (int)Math.Round(CountMutations \* 0.75, 0);

for (int i = 0; i < proc70; i++)

{

for (int j = 0; j < 4; j++)

{

Parents[i, j] = MutatedChildren[i, j];

}

}

//Дополняем популяцию до 300

k = proc70 + 1;

while (k < 300)

{

x1 = (double)randObj.Next(600) / (double)100;

x2 = (double)randObj.Next(600) / (double)100;

x3 = (double)randObj.Next(600) / (double)100;

if ((x1 + x2 + x3 < 41) && (2 \* x1 + x2 + 3 \* x3 < 41))

{

Parents[k, 0] = x1;

Parents[k, 1] = x2;

Parents[k, 2] = x3;

k++;

}

}

}

void Sorting()

{

double buf;

for (int i = 0; i < CountMutations; i++)

for (int j = 0; j < CountMutations - 1; j++)

{

if (MutatedChildren[j, 3] < MutatedChildren[j + 1, 3])

{

buf = MutatedChildren[j, 3];

MutatedChildren[j, 3] = MutatedChildren[j + 1, 3];

MutatedChildren[j + 1, 3] = buf;

buf = MutatedChildren[j, 0];

MutatedChildren[j, 0] = MutatedChildren[j + 1, 0];

MutatedChildren[j + 1, 0] = buf;

buf = MutatedChildren[j, 1];

MutatedChildren[j, 1] = MutatedChildren[j + 1, 1];

MutatedChildren[j + 1, 1] = buf;

buf = MutatedChildren[j, 2];

MutatedChildren[j, 2] = MutatedChildren[j + 1, 2];

MutatedChildren[j + 1, 2] = buf;

}

}

}

}

}