**Advanced Computational Techniques**

SSE 635

Dr. MacNeil

Spring 2017

**Project III**

Brent Bitler

Matthew Robison

Table of Contents

[Table of Contents 2](#_Toc477011386)

[Table of Figures 3](#_Toc477011387)

[No table of figures entries found.Table of Tables 3](#_Toc477011388)

[No table of figures entries found. 3](#_Toc477011389)

[Introduction 4](#_Toc477011390)

[Background 4](#_Toc477011391)

[Purpose 4](#_Toc477011392)

[Back Propagation 5](#_Toc477011393)

[Overview 5](#_Toc477011394)

[Conclusion 5](#_Toc477011395)

[Overview 5](#_Toc477011396)

[Appendix 6](#_Toc477011397)

[Driver.cs 6](#_Toc477011398)

[Neural Network.cs 8](#_Toc477011399)

[NetworkTrainer.cs 22](#_Toc477011400)

[Bibliography 34](#_Toc477011401)

[Activity Log - Bitler 35](#_Toc477011402)

[Activity Log - Robison 37](#_Toc477011403)

Table of Figures

**No table of figures entries found.**Table of Tables

**No table of figures entries found.**

Introduction

## Background

SSE 635’s purpose is to educate us on the use of advanced computational techniques and their applications. Project III’s focus is on neural networks.

## Purpose

For Project III, we were directed to implement an application to demonstrate our mastery of Neural Networks since this is our first course in the realm of computational techniques. Our approach was to

Back Propagation

## Overview

Before

Conclusion

## Overview

In conclusion, much was learned about neural networks.

Appendix

## Driver.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using NeuralNetwork;

using System.Xml;

using System.IO;

namespace NeuralNetworkTutorialApp

{

class Program

{

static void Main(string[] args)

{

// Training Data

XmlDocument doc = new XmlDocument();

doc.Load(@"simpleData.xml");

DataSet ds = new DataSet();

ds.Load((XmlElement)doc.DocumentElement.ChildNodes[0]);

// Network to train

int[] layerSizes = new int[3] { 25, 35, 4 };

TransferFunction[] tFuncs = new TransferFunction[3] { TransferFunction.None,

TransferFunction.Sigmoid,

TransferFunction.Linear };

BackPropagationNetwork bpn = new BackPropagationNetwork(layerSizes, tFuncs);

// Network trainer

NetworkTrainer nt = new NetworkTrainer(bpn, ds);

nt.maxError = 0.001; nt.maxIterations = 100000;

nt.nudge\_window = 500;

// Train

Console.WriteLine("Training...");

nt.TrainDataSet();

Console.WriteLine("Done!");

// Save the network

nt.network.Save(@"Output.xml");

// Save the error history

double[] error = nt.GetErrorHistory();

string[] filedata = new string[error.Length];

for (int i = 0; i < error.Length; i++)

filedata[i] = i.ToString() + " " + error[i].ToString();

File.WriteAllLines(@"simple\_errors.txt", filedata);

// End of program

Console.WriteLine("\n\nPress Enter...");

Console.ReadLine();

}

}

}

## Neural Network.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Xml;

namespace NeuralNetwork

{

#region Transfer functions and their derivatives

public enum TransferFunction

{

None,

Sigmoid,

Linear,

Gaussian,

RationalSigmoid

}

static class TransferFunctions

{

public static double Evaluate(TransferFunction tFunc, double input)

{

switch (tFunc)

{

case TransferFunction.Sigmoid:

return sigmoid(input);

case TransferFunction.Linear:

return linear(input);

case TransferFunction.Gaussian:

return gaussian(input);

case TransferFunction.RationalSigmoid:

return rationalsigmoid(input);

case TransferFunction.None:

default:

return 0.0;

}

}

public static double EvaluateDerivative(TransferFunction tFunc, double input)

{

switch (tFunc)

{

case TransferFunction.Sigmoid:

return sigmoid\_derivative(input);

case TransferFunction.Linear:

return linear\_derivative(input);

case TransferFunction.Gaussian:

return gaussian\_derivative(input);

case TransferFunction.RationalSigmoid:

return rationalsigmoid\_derivative(input);

case TransferFunction.None:

default:

return 0.0;

}

}

/\* Transfer function definitions \*/

private static double sigmoid(double x)

{

return 1.0 / (1.0 + Math.Exp(-x));

}

private static double sigmoid\_derivative(double x)

{

return sigmoid(x) \* (1 - sigmoid(x));

}

private static double linear(double x)

{

return x;

}

private static double linear\_derivative(double x)

{

return 1.0;

}

private static double gaussian(double x)

{

return Math.Exp(-Math.Pow(x, 2));

}

private static double gaussian\_derivative(double x)

{

return -2.0 \* x \* gaussian(x);

}

private static double rationalsigmoid(double x)

{

return x / (1.0 + Math.Sqrt(1.0 + x \* x));

}

private static double rationalsigmoid\_derivative(double x)

{

double val = Math.Sqrt(1.0 + x \* x);

return 1.0 / (val \* (1 + val));

}

}

#endregion

public class BackPropagationNetwork

{

#region Constructors

public BackPropagationNetwork(int[] layerSizes, TransferFunction[] transferFunctions)

{

// Validate the input data

if (transferFunctions.Length != layerSizes.Length || transferFunctions[0] != TransferFunction.None)

throw new ArgumentException("Cannot construct a network with these parameters.");

// Initialize network layers

layerCount = layerSizes.Length - 1;

inputSize = layerSizes[0];

layerSize = new int[layerCount];

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

layerSize[i] = layerSizes[i + 1];

transferFunction = new TransferFunction[layerCount];

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

transferFunction[i] = transferFunctions[i + 1];

// Start dimensioning arrays

bias = new double[layerCount][];

previousBiasDelta = new double[layerCount][];

delta = new double[layerCount][];

layerOutput = new double[layerCount][];

layerInput = new double[layerCount][];

weight = new double[layerCount][][];

previousWeightDelta = new double[layerCount][][];

// Fill 2 dimensional arrays

for (int l = 0; l < layerCount; l++)

{

bias[l] = new double[layerSize[l]];

previousBiasDelta[l] = new double[layerSize[l]];

delta[l] = new double[layerSize[l]];

layerOutput[l] = new double[layerSize[l]];

layerInput[l] = new double[layerSize[l]];

weight[l] = new double[l == 0 ? inputSize : layerSize[l - 1]][];

previousWeightDelta[l] = new double[l == 0 ? inputSize : layerSize[l - 1]][];

for (int i = 0; i < (l == 0 ? inputSize : layerSize[l - 1]); i++)

{

weight[l][i] = new double[layerSize[l]];

previousWeightDelta[l][i] = new double[layerSize[l]];

}

}

// Initialize the weights

for (int l = 0; l < layerCount; l++)

{

for (int j = 0; j < layerSize[l]; j++)

{

bias[l][j] = Gaussian.GetRandomGaussian();

previousBiasDelta[l][j] = 0.0;

layerOutput[l][j] = 0.0;

layerInput[l][j] = 0.0;

delta[l][j] = 0.0;

}

for (int i = 0; i < (l == 0 ? inputSize : layerSize[l - 1]); i++)

{

for (int j = 0; j < layerSize[l]; j++)

{

weight[l][i][j] = Gaussian.GetRandomGaussian();

previousWeightDelta[l][i][j] = 0.0;

}

}

}

}

public BackPropagationNetwork(string FilePath)

{

loaded = false;

Load(FilePath);

loaded = true;

}

#endregion

#region Methods

// Public methods

public void Run(ref double[] input, out double[] output)

{

// Make sure we have enough data

if (input.Length != inputSize)

throw new ArgumentException("Input data is not of the correct dimension.");

// Dimension

output = new double[layerSize[layerCount - 1]];

/\* Run the network! \*/

for (int l = 0; l < layerCount; l++)

{

for (int j = 0; j < layerSize[l]; j++)

{

double sum = 0.0;

for (int i = 0; i < (l == 0 ? inputSize : layerSize[l - 1]); i++)

sum += weight[l][i][j] \* (l == 0 ? input[i] : layerOutput[l - 1][i]);

sum += bias[l][j];

layerInput[l][j] = sum;

layerOutput[l][j] = TransferFunctions.Evaluate(transferFunction[l], sum);

}

}

// Copy the output to the output array

for (int i = 0; i < layerSize[layerCount - 1]; i++)

output[i] = layerOutput[layerCount - 1][i];

}

public double Train(ref double[] input, ref double[] desired, double TrainingRate, double Momentum)

{

// Parameter Validation

if (input.Length != inputSize)

throw new ArgumentException("Invalid input parameter", "input");

if (desired.Length != layerSize[layerCount-1])

throw new ArgumentException("Invalid input parameter", "desired");

// Local variable

double error = 0.0, sum = 0.0, weightDelta = 0.0, biasDelta = 0.0;

double[] output = new double[layerSize[layerCount - 1]];

// Run the network

Run(ref input, out output);

// Back-propagate the error

for (int l = layerCount - 1; l >= 0; l--)

{

// Output layer

if (l == layerCount - 1)

{

for (int k = 0; k < layerSize[l]; k++)

{

delta[l][k] = output[k] - desired[k];

error += Math.Pow(delta[l][k], 2);

delta[l][k] \*= TransferFunctions.EvaluateDerivative(transferFunction[l],

layerInput[l][k]);

}

}

else // Hidden layer

{

for (int i = 0; i < layerSize[l]; i++)

{

sum = 0.0;

for (int j = 0; j < layerSize[l + 1]; j++)

{

sum += weight[l + 1][i][j] \* delta[l + 1][j];

}

sum \*= TransferFunctions.EvaluateDerivative(transferFunction[l], layerInput[l][i]);

delta[l][i] = sum;

}

}

}

// Update the weights and biases

for (int l = 0; l < layerCount; l++)

for(int i=0; i<(l==0 ? inputSize : layerSize[l-1]); i++)

for (int j = 0; j < layerSize[l]; j++)

{

weightDelta = TrainingRate \* delta[l][j] \* (l == 0 ? input[i] : layerOutput[l - 1][i])

+ Momentum \* previousWeightDelta[l][i][j];

weight[l][i][j] -= weightDelta;

previousWeightDelta[l][i][j] = weightDelta;

}

for(int l=0; l<layerCount; l++)

for (int i = 0; i < layerSize[l]; i++)

{

biasDelta = TrainingRate \* delta[l][i];

bias[l][i] -= biasDelta + Momentum \* previousBiasDelta[l][i];

previousBiasDelta[l][i] = biasDelta;

}

return error;

}

public void Save(string FilePath)

{

if (FilePath == null)

return;

XmlWriterSettings settings = new XmlWriterSettings();

settings.Indent = true;

settings.IndentChars = "\t";

XmlWriter writer = XmlWriter.Create(FilePath,settings);

// Begin document

writer.WriteStartElement("NeuralNetwork");

writer.WriteAttributeString("Type", "BackPropagation");

// Parameters element

writer.WriteStartElement("Parameters");

writer.WriteElementString("Name", Name);

writer.WriteElementString("inputSize", inputSize.ToString());

writer.WriteElementString("layerCount", layerCount.ToString());

// Layer sizes

writer.WriteStartElement("Layers");

for (int l = 0; l < layerCount; l++)

{

writer.WriteStartElement("Layer");

writer.WriteAttributeString("Index", l.ToString());

writer.WriteAttributeString("Size", layerSize[l].ToString());

writer.WriteAttributeString("Type", transferFunction[l].ToString());

writer.WriteEndElement(); // Layer

}

writer.WriteEndElement(); // Layers

writer.WriteEndElement(); // Parameters

// Weights and biases

writer.WriteStartElement("Weights");

for (int l = 0; l < layerCount; l++)

{

writer.WriteStartElement("Layer");

writer.WriteAttributeString("Index", l.ToString());

for (int j = 0; j < layerSize[l]; j++)

{

writer.WriteStartElement("Node");

writer.WriteAttributeString("Index", j.ToString());

writer.WriteAttributeString("Bias", bias[l][j].ToString());

for (int i = 0; i < (l == 0 ? inputSize : layerSize[l - 1]); i++)

{

writer.WriteStartElement("Axon");

writer.WriteAttributeString("Index", i.ToString());

writer.WriteString(weight[l][i][j].ToString());

writer.WriteEndElement(); // Axon

}

writer.WriteEndElement(); // Node

}

writer.WriteEndElement(); // Layer

}

writer.WriteEndElement(); // Weights

writer.WriteEndElement(); // NeuralNetwork

writer.Flush();

writer.Close();

}

public void Load(string FilePath)

{

if (FilePath == null)

return;

doc = new XmlDocument();

doc.Load(FilePath);

string BasePath = "", NodePath = "";

double value;

// Load from xml

if (xPathValue("NeuralNetwork/@Type") != "BackPropagation")

return;

BasePath = "NeuralNetwork/Parameters/";

Name = xPathValue(BasePath + "Name");

int.TryParse(xPathValue(BasePath + "inputSize"), out inputSize);

int.TryParse(xPathValue(BasePath + "layerCount"), out layerCount);

layerSize = new int[layerCount];

transferFunction = new TransferFunction[layerCount];

BasePath = "NeuralNetwork/Parameters/Layers/Layer";

for (int l = 0; l < layerCount; l++)

{

int.TryParse(xPathValue(BasePath + "[@Index='" + l.ToString() + "']/@Size"), out layerSize[l]);

Enum.TryParse<TransferFunction>(xPathValue(BasePath + "[@Index='" + l.ToString() + "']/@Type"), out transferFunction[l]);

}

// Parse the Weights element

// Start dimensioning arrays

bias = new double[layerCount][];

previousBiasDelta = new double[layerCount][];

delta = new double[layerCount][];

layerOutput = new double[layerCount][];

layerInput = new double[layerCount][];

weight = new double[layerCount][][];

previousWeightDelta = new double[layerCount][][];

// Fill 2 dimensional arrays

for (int l = 0; l < layerCount; l++)

{

bias[l] = new double[layerSize[l]];

previousBiasDelta[l] = new double[layerSize[l]];

delta[l] = new double[layerSize[l]];

layerOutput[l] = new double[layerSize[l]];

layerInput[l] = new double[layerSize[l]];

weight[l] = new double[l == 0 ? inputSize : layerSize[l - 1]][];

previousWeightDelta[l] = new double[l == 0 ? inputSize : layerSize[l - 1]][];

for (int i = 0; i < (l == 0 ? inputSize : layerSize[l - 1]); i++)

{

weight[l][i] = new double[layerSize[l]];

previousWeightDelta[l][i] = new double[layerSize[l]];

}

}

// Initialize the weights

for (int l = 0; l < layerCount; l++)

{

BasePath = "NeuralNetwork/Weights/Layer[@Index='" + l.ToString() + "']/";

for (int j = 0; j < layerSize[l]; j++)

{

NodePath = "Node[@Index='" + j.ToString() + "']/@Bias";

double.TryParse(xPathValue(BasePath + NodePath), out value);

bias[l][j] = value;

previousBiasDelta[l][j] = 0.0;

layerOutput[l][j] = 0.0;

layerInput[l][j] = 0.0;

delta[l][j] = 0.0;

}

for (int i = 0; i < (l == 0 ? inputSize : layerSize[l - 1]); i++)

{

for (int j = 0; j < layerSize[l]; j++)

{

NodePath = "Node[@Index='" + j.ToString() + "']/Axon[@Index='" + i.ToString() + "']";

double.TryParse(xPathValue(BasePath + NodePath), out value);

weight[l][i][j] = value;

previousWeightDelta[l][i][j] = 0.0;

}

}

}

// "release"

doc = null;

}

public void Nudge(double scalar)

{

// Go through all of the weights and biases and augment them

for (int l = 0; l < layerCount; l++)

{

for (int j = 0; j < layerSize[l]; j++)

{

// Nudge the weights

for (int i = 0; i < (l == 0 ? inputSize : layerSize[l - 1]); i++)

{

double w = weight[l][i][j];

double u = Gaussian.GetRandomGaussian(0f, w \* scalar);

weight[l][i][j] += u;

previousWeightDelta[l][i][j] = 0f;

}

// Nudge the bias

double b = bias[l][j];

double v = Gaussian.GetRandomGaussian(0f, b \* scalar);

bias[l][j] += v;

previousBiasDelta[l][j] = 0f;

}

}

}

// Private methods

private string xPathValue(string xPath)

{

XmlNode node = doc.SelectSingleNode(xPath);

if (node == null)

throw new ArgumentException("Cannot find specified node", xPath);

return node.InnerText;

}

#endregion

#region Public data

public string Name = "Default";

#endregion

#region Private data

private int layerCount;

private int inputSize;

private int[] layerSize;

private TransferFunction[] transferFunction;

private double[][] layerOutput;

private double[][] layerInput;

private double[][] bias;

private double[][] delta;

private double[][] previousBiasDelta;

private double[][][] weight;

private double[][][] previousWeightDelta;

private XmlDocument doc = null;

private bool loaded = true;

#endregion

}

public static class Gaussian

{

private static Random gen = new Random();

public static double GetRandomGaussian()

{

return GetRandomGaussian(0.0, 1.0);

}

public static double GetRandomGaussian(double mean, double stddev)

{

double rVal1, rVal2;

GetRandomGaussian(mean, stddev, out rVal1, out rVal2);

return rVal1;

}

public static void GetRandomGaussian(double mean, double stddev, out double val1, out double val2)

{

double u, v, s, t;

do

{

u = 2 \* gen.NextDouble() - 1;

v = 2 \* gen.NextDouble() - 1;

} while (u \* u + v \* v > 1 || (u == 0 && v == 0));

s = u \* u + v \* v;

t = Math.Sqrt((-2.0 \* Math.Log(s)) / s);

val1 = stddev \* u \* t + mean;

val2 = stddev \* v \* t + mean;

}

}

}

## NetworkTrainer.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Xml;

namespace NeuralNetwork

{

#region Support classes

public class DataPoint

{

public DataPoint() { }

public DataPoint(double[] Input, double[] Output) { Load(Input, Output); }

public DataPoint(XmlElement Elem) { Load(Elem); }

public void Load(double[] Input, double[] Output)

{

input = new double[Input.Length]; output = new double[Output.Length];

Array.Copy(Input, input, Input.Length);

Array.Copy(Output, output, Output.Length);

}

public void Load(XmlElement elem)

{

XmlNode nType;

int lIn, lOut, i;

nType = elem.SelectSingleNode("Input");

lIn = nType.ChildNodes.Count;

input = new double[lIn];

foreach (XmlNode node in nType.ChildNodes)

{

XmlElement Node = (XmlElement)node;

double val;

int.TryParse(Node.GetAttribute("Index"), out i);

double.TryParse(Node.InnerText, out val);

input[i] = val;

}

nType = elem.SelectSingleNode("Output");

lOut = nType.ChildNodes.Count;

output = new double[lOut];

foreach (XmlNode node in nType.ChildNodes)

{

XmlElement Node = (XmlElement)node;

double val;

int.TryParse(Node.GetAttribute("Index"), out i);

double.TryParse(Node.InnerText, out val);

output[i] = val;

}

}

public XmlElement ToXml(XmlDocument doc)

{

XmlElement nDataPoint, nType, node;

int lIn = input.Length, lOut = output.Length;

nDataPoint = doc.CreateElement("DataPoint");

nType = doc.CreateElement("Input");

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

{

node = doc.CreateElement("Data");

node.SetAttribute("Index", i.ToString());

node.AppendChild(doc.CreateTextNode(input[i].ToString()));

nType.AppendChild(node);

}

nDataPoint.AppendChild(nType);

nType = doc.CreateElement("Output");

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

{

node = doc.CreateElement("Data");

node.SetAttribute("Index", i.ToString());

node.AppendChild(doc.CreateTextNode(output[i].ToString()));

nType.AppendChild(node);

}

nDataPoint.AppendChild(nType);

return nDataPoint;

}

public double[] input, output;

public int inputSize { get { return input.Length; } }

public int outputSize { get { return output.Length; } }

}

public class DataSet

{

public DataSet() { Data = new List<DataPoint>(); }

public XmlElement ToXml(XmlDocument doc)

{

XmlElement nDataSet;

nDataSet = doc.CreateElement("DataSet");

foreach (DataPoint d in Data)

nDataSet.AppendChild(d.ToXml(doc));

return nDataSet;

}

public void Load(XmlElement nDataSet)

{

foreach (XmlNode node in nDataSet.ChildNodes)

{

DataPoint d = new DataPoint((XmlElement)node);

Data.Add(d);

}

}

public List<DataPoint> Data;

public int Size

{

get { return Data.Count; }

}

}

public class Permutator

{

public Permutator(int Size)

{

index = new int[Size];

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

index[i] = i;

Permute(Size);

}

public void Permute(int nTimes)

{

int i, j, t;

for (int n = 0; n < nTimes; n++)

{

i = gen.Next(index.Length);

j = gen.Next(index.Length);

if (i != j)

{

t = index[i];

index[i] = index[j];

index[j] = t;

}

}

}

public int this[int i]

{

get

{

return index[i];

}

}

private Random gen = new Random();

private int[] index;

}

#endregion

#region Network training classes

public class SimpleNetworkTrainer

{

// Constructors

public SimpleNetworkTrainer(BackPropagationNetwork BPN, DataSet DS)

{

network = BPN; dataSet = DS;

idx = new Permutator(dataSet.Size);

iterations = 0;

errorHistory = new List<double>();

}

// Training method

public void TrainDataSet()

{

do

{

// Prepare to train epoch

iterations++; error = 0.0;

idx.Permute(dataSet.Size);

// Train this epoch

for (int i = 0; i < dataSet.Size; i++)

{

error += network.Train( ref dataSet.Data[idx[i]].input,

ref dataSet.Data[idx[i]].output,

trainingRate, momentum);

}

// Track this error history

errorHistory.Add(error);

// Check whether to Nudge

if (iterations % nudge\_window == 0)

CheckNudge();

} while (error > maxError && iterations < maxIterations);

}

// Accessor method

public double[] GetErrorHistory()

{

return errorHistory.ToArray();

}

// Private method

private void CheckNudge()

{

double oldAvg = 0f, newAvg = 0f;

int l = errorHistory.Count;

// Do i enough data?

if (iterations < 2 \* nudge\_window) return;

// Compute our averages and compare

for (int i = 0; i < nudge\_window; i++)

{

oldAvg += errorHistory[l - 2 \* nudge\_window + i];

newAvg += errorHistory[l - nudge\_window + i];

}

oldAvg /= nudge\_window; newAvg /= nudge\_window;

Console.Write("Iter {0} oldAvg {1:0.0000} newAvg {2:0.0000}", iterations, oldAvg, newAvg);

if (((double)Math.Abs(newAvg - oldAvg)) / nudge\_window < nudge\_tolerance)

{

network.Nudge(nudge\_scale);

Console.Write(" Nudged.");

}

Console.Write("\n");

}

// Public fields

public double maxError = 0.1, maxIterations = 100000;

public double trainingRate = 0.25, momentum = 0.15;

public int nudge\_window = 50;

public double nudge\_scale = 0.25, nudge\_tolerance = 0.0001;

// Private fields

private double error;

private int iterations;

private Permutator idx;

private List<double> errorHistory;

// Training materials

public BackPropagationNetwork network;

public DataSet dataSet;

}

public class NetworkTrainer

{

// Constructor

public NetworkTrainer(BackPropagationNetwork BPN, DataSet DS)

{

network = BPN;

dataSet = DS;

Initialize();

}

public void Initialize()

{

iterations = 0;

if (idx == null)

idx = new Permutator(dataSet.Size);

else

idx.Permute(dataSet.Size);

if (errorHistory == null)

errorHistory = new List<double>();

else

errorHistory.Clear();

}

// Public train method

public bool TrainDataSet()

{

bool success = true;

if (success)

success = \_BeforeTrainDataSet();

if (success)

success = \_TrainDataSetAction();

if (success)

success = \_AfterTrainDataSet();

return success;

}

// Protected hook methods

protected virtual bool BeforeTrainDataSet() { return true; }

protected virtual bool AfterTrainDataSet() { return true; }

protected virtual bool BeforeTrainEpoch() { return true; }

protected virtual bool AfterTrainEpoch() { return true; }

protected virtual bool BeforeTrainDataPoint(ref double[] Input, ref double[] Output, int Index) { return true; }

protected virtual bool AfterTrainDataPoint(ref double[] Input, ref double[] Output, int Index) { return true; }

// Private training methods

private bool \_BeforeTrainDataSet()

{

Initialize();

return BeforeTrainDataSet();

}

private bool \_TrainDataSetAction()

{

bool success = true;

do

{

if (success)

success = \_BeforeTrainEpoch();

if (success)

success = \_TrainEpochAction();

if (success)

success = \_AfterTrainEpoch();

} while (error > maxError && iterations < maxIterations && success);

return success;

}

private bool \_AfterTrainDataSet()

{

return AfterTrainDataSet();

}

private bool \_BeforeTrainEpoch()

{

// Prepare to train epoch

iterations++;

error = 0.0;

idx.Permute(dataSet.Size);

return BeforeTrainEpoch();

}

private bool \_TrainEpochAction()

{

bool success = true;

// Train this epoch

for (int i = 0; i < dataSet.Size && success; i++)

{

// Make a local copy of the data point's data

double[] input = new double[dataSet.Data[idx[i]].inputSize];

double[] output = new double[dataSet.Data[idx[i]].outputSize];

Array.Copy(dataSet.Data[idx[i]].input, input, input.Length);

Array.Copy(dataSet.Data[idx[i]].output, output, output.Length);

if (success)

success = BeforeTrainDataPoint(ref input, ref output, idx[i]);

if (success)

error += network.Train(ref input, ref output, trainingRate, momentum);

if (success)

success = AfterTrainDataPoint(ref input, ref output, idx[i]);

}

return success;

}

private bool \_AfterTrainEpoch()

{

// Track this error history

errorHistory.Add(error);

// Check whether to Nudge

if (iterations % nudge\_window == 0 && nudge)

CheckNudge();

return AfterTrainEpoch();

}

// Accessor method

public double[] GetErrorHistory()

{

return errorHistory.ToArray();

}

// Private method

private void CheckNudge()

{

double oldAvg = 0f, newAvg = 0f;

int l = errorHistory.Count;

// Do i enough data?

if (iterations < 2 \* nudge\_window) return;

// Compute our averages and compare

for (int i = 0; i < nudge\_window; i++)

{

oldAvg += errorHistory[l - 2 \* nudge\_window + i];

newAvg += errorHistory[l - nudge\_window + i];

}

oldAvg /= nudge\_window; newAvg /= nudge\_window;

//Console.Write("Iter {0} oldAvg {1:0.0000} newAvg {2:0.0000}", iterations, oldAvg, newAvg);

if (((double)Math.Abs(newAvg - oldAvg)) / nudge\_window < nudge\_tolerance)

{

network.Nudge(nudge\_scale);

//Console.Write(" Nudged.");

}

//Console.Write("\n");

}

// Public fields

public double maxError = 0.1, maxIterations = 100000;

public double trainingRate = 0.25, momentum = 0.15;

public int nudge\_window = 50;

public bool nudge = true;

public double nudge\_scale = 0.25, nudge\_tolerance = 0.0001;

// Private fields

private double error;

private int iterations;

private Permutator idx;

private List<double> errorHistory;

// Training materials

public BackPropagationNetwork network;

public DataSet dataSet;

}

public class BinaryNoiseTrainer : NetworkTrainer

{

// Constructor

public BinaryNoiseTrainer(BackPropagationNetwork BPN, DataSet DS)

: base(BPN, DS)

{

// Additional initialization stuff here

rnd = new Random();

}

// Overrides

protected override bool BeforeTrainEpoch()

{

// Reset the NoisyData

NoisyData = new DataSet();

return true;

}

protected override bool BeforeTrainDataPoint(ref double[] Input, ref double[] Output, int Index)

{

// Add some noise to the input data

// Add noise to the input

for(int i=0; i<Input.Length; i++)

if(rnd.NextDouble() < \_noise\_density)

Input[i] = (Input[i] == 0.0 ? 1.0 : 0.0);

// Add this "dirty" data point to the data set

DataPoint dp = new DataPoint(Input, Output);

NoisyData.Data.Add(dp);

return true;

}

// Private data

private double \_noise\_density = 0.10;

public double noise\_density {

get { return \_noise\_density;}

set{

\_noise\_density = Math.Min(1.0, Math.Max(0.0, value));

}

}

public DataSet NoisyData;

private Random rnd;

}

#endregion

}

Bibliography

[1] Engelbrecht, Andries P. Computational Intelligence: An Introduction, (2nd ed.). Wiley, 2007.

Activity Log - Bitler

|  |  |  |
| --- | --- | --- |
| Date | Time (mins) | Description |
| *1/9/17* | *60* | *Looked through Naïve GA* |
| *1/10/17* | *60* | *Looked through and worked Naïve GA* |
| *1/11/17* |  |  |
| *1/12/17* | *120* | *Research GA’s online for topic ideas* |
| *1/13/17* | *180* | *Read Ch 9 - GA* |
| *1/14/17* | *90* | *Read Ch 9 - GA* |
| *1/15/17* |  |  |
| *1/16/17* | *120* | *Begin constructing class structure of string unscramble program* |
| *1/17/17* | *120* | *Begin chromosome class* |
| *1/18/17* | *120* | *Continue working chromosome class, begin developing population class* |
| *1/19/17* |  |  |
| *1/20/17* |  |  |
| *1/21/17* | *60* | *Finish chromosome class* |
| *1/22/17* | *240* | *Continue working population class* |
| *1/23/17* | *120* | *Connector class* |
| *1/24/17* | *30* | *Come up with initial variable settings* |
| *1/25/17* |  |  |
| *1/26/17* | *240* | *Begin generating report* |
| *1/27/17* | *180* | *Put together test data* |
| *1/28/17* | *360* | *Conclude paper, graphs, etc* |
| *1/29/17* | *180* | *Final review of paper* |
| *1/30/17* | *15* | *Submit paper* |
| 1/31/17 |  |  |
| 2/1/17 | 90 | Begin reading Ch 20 – Fuzzy sets |
| 2/2/17 | 120 | Continue reading Ch 20 – Fuzzy sets |
| 2/3/17 | 120 | Begin Ch 21 – Fuzzy Logic and Reasoning |
| 2/4/17 |  |  |
| 2/5/17 | 120 | Continue Ch 21 – Fuzzy Logic and Reasoning |
| 2/6/17 | 90 | Read Ch 22, 23 – Fuzzy controllers and Rough sets |
| 2/7/17 | 120 | Watch tutorials on fuzzy logic |
| 2/8/17 | 120 | Develop class structure for Heating/Cooling app |
| 2/9/17 | 120 | Work on Point class |
| 2/10/17 | 180 | Work on FuzzySet, MembershipFunction, and Region classes |
| 2/11/17 | 120 | Continue class development |
| 2/12/17 |  |  |
| 2/13/17 | 300 | Begin generating report |
| 2/14/17 |  |  |
| 2/15/17 | 360 | Continue report |
| 2/16/17 | 240 | Continue report |
| 2/17/17 | 240 | Conclude paper, graphs, etc |
| 2/18/17 | 45 | Final review of paper |
| 2/19/17 |  |  |
| 2/20/17 | 15 | Submit paper |
| *Project 1* | *2295* |  |
| *Project 2* | 2400 |  |
| Project 3 |  |  |
| TOTAL | 4695 |  |

Activity Log - Robison

|  |  |  |
| --- | --- | --- |
| Date | Time (mins) | Description |
| *1/9/17* |  |  |
| *1/10/17* |  |  |
| *1/11/17* | *120* | *Reading Chapter 8* |
| *1/12/17* | *120* | *Reading Chapter 9* |
| *1/13/17* | *90* | *Working on Project 0* |
| *1/14/17* |  |  |
| *1/15/17* | *60* | *Working on Project 0* |
| *1/16/17* |  |  |
| *1/17/17* | *120* | *Started Development of GA* |
| *1/18/17* |  |  |
| *1/19/17* |  |  |
| *1/20/17* | *240* | *Continued Development on GA* |
| *1/21/17* | *480* | *Continued Development on GA* |
| *1/22/17* | *120* | *Started Developing Report* |
| *1/23/17* |  |  |
| *1/24/17* |  |  |
| *1/25/17* |  |  |
| *1/26/17* |  |  |
| *1/27/17* | *120* | *Continue Developing Report* |
| *1/28/17* |  |  |
| *1/29/17* | *180* | *Continue Developing Report* |
| *1/30/17* | *90* | *Formatting Report*  *Upload Report* |
| *1/31/17* |  |  |
| *2/1/17* | *120* | *Reading Chapter 20* |
| *2/2/17* | *120* | *Reading Chapter 21* |
| *2/3/17* |  |  |
| *2/4/17* | *180* | *Reading Chapter 22* |
| *2/5/17* | *90* | *Reading Chapter 23* |
| *2/6/17* |  |  |
| *2/7/17* |  |  |
| *2/8/17* | *120* | *Develop class structure for Heating/Cooling app* |
| *2/9/17* | *240* | *Development of MembershipFunction Class* |
| *2/10/17* |  |  |
| *2/11/17* | *240* | *Development of FuzzySet Class* |
| *2/12/17* | *240* | *Testing and Debugging Heating/Cooling app* |
| *2/13/17* | *180* | *Begin generating report* |
| *2/14/17* |  |  |
| *2/15/17* | *240* | *Continue report* |
| *2/16/17* | *180* | *Continue report* |
| *2/17/17* | *120* | *Continue report* |
| *2/18/17* | *120* | *Final review of paper* |
| *2/19/17* |  |  |
| *2/20/17* | *15* | *Submit paper* |
| *Project 1* | *1740* |  |
| Project 2 | 2625 |  |
| TOTAL | 4365 |  |