CSCE 5262 - Neural Networks and Genetic Algorithms Assignment 1

Bishoy Boshra Labib
900132990

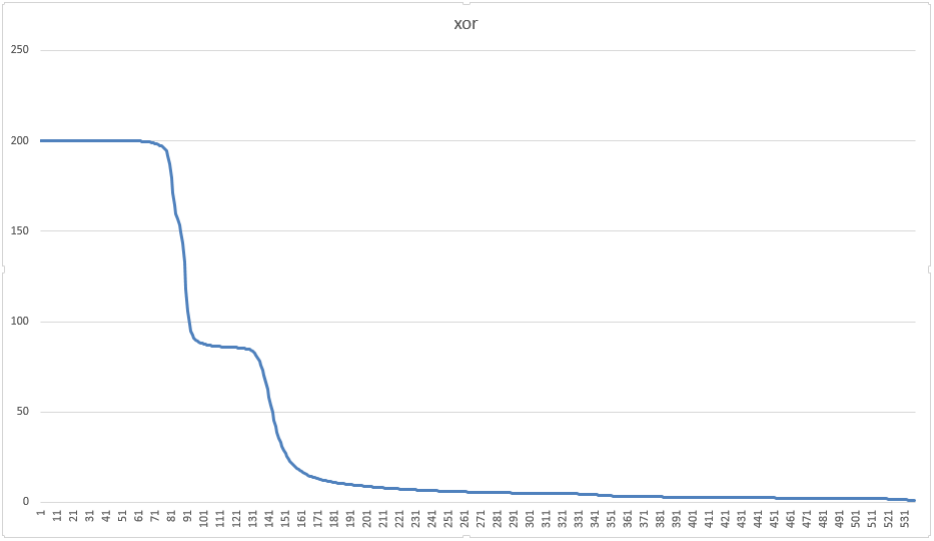
Oct. 24^{th}, 2016

1 Results

1.1 xor

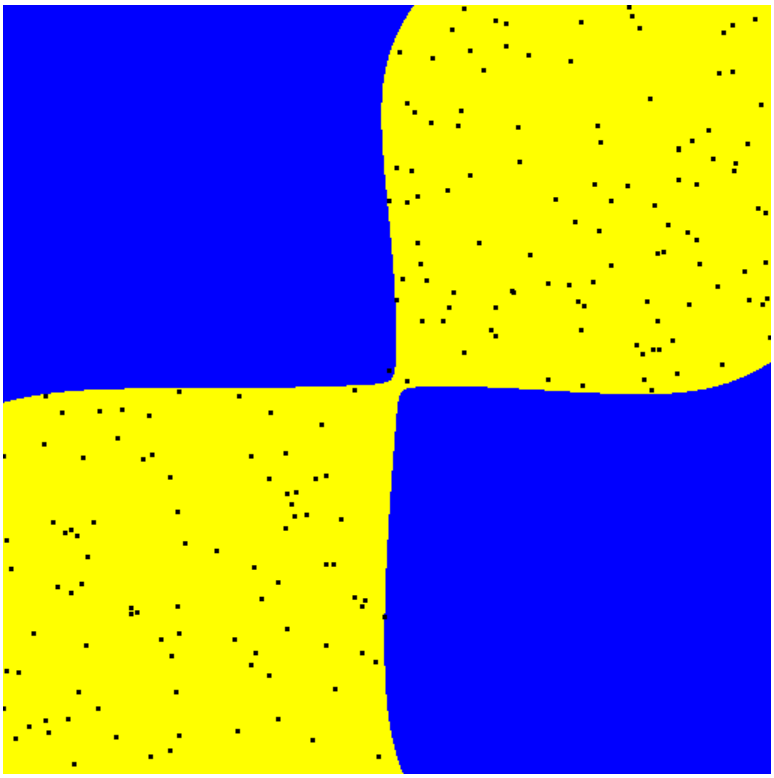
i. The network had 1 input layers of size 2 (representing the x and y coordinates of the points), 2 hidden layers of size 5 each, and an output layer containing a single neuron.

ii. The MSE developed as more epochs were fed into the network as shown in the following diagram:



Training stopped after reaching an accuracy of 100%

iii. The visualization of the decision boundary was as shown in the following graph:



MCCR = 1.0

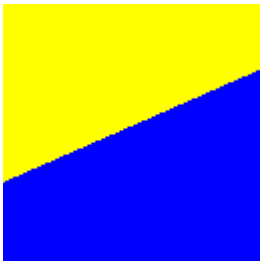
The progression of the *xor* boundary was as shown in the following figures:

1.

2. 

3. 

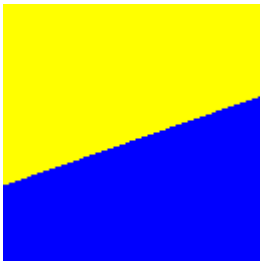
4. 

5. 

6. 

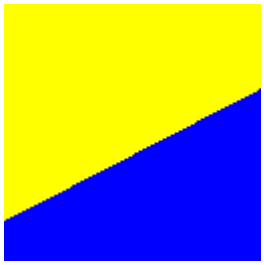
7. 

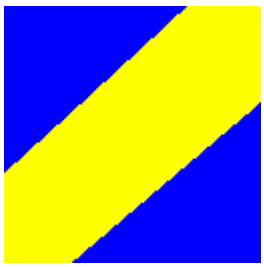
8. 

9. 

10. 

11. 

12. 

13. 

14. 

15. 

16. 

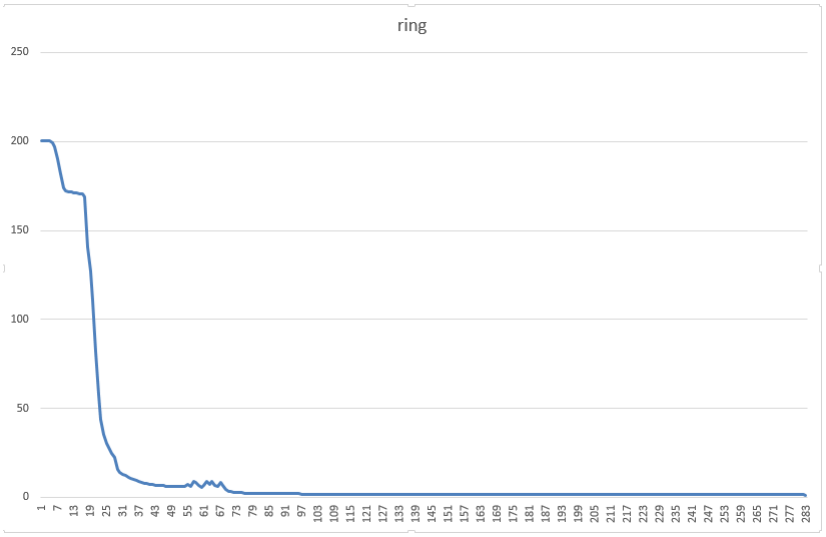
17. 

18. 

1.2 ring

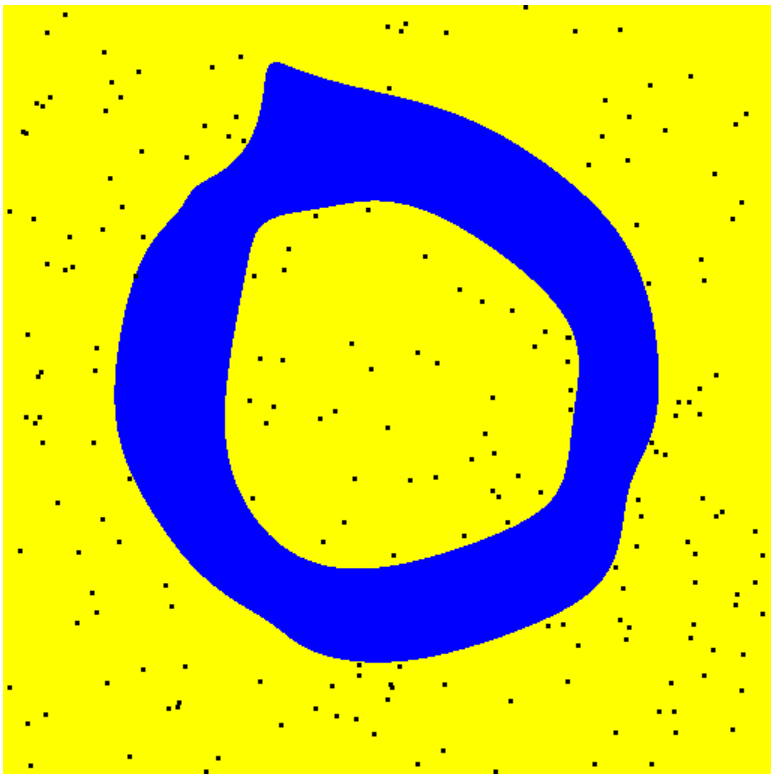
i. The network had 1 input layers of size 2 (representing the x and y coordinates of the points), 2 hidden layers of size 5 each, and an output layer containing a single neuron.

ii. The MSE developed as more epochs were fed into the network as shown in the following diagram:



Training stopped after reaching an accuracy of 100%

iii. The visualization of the decision boundary was as shown in the following graph:

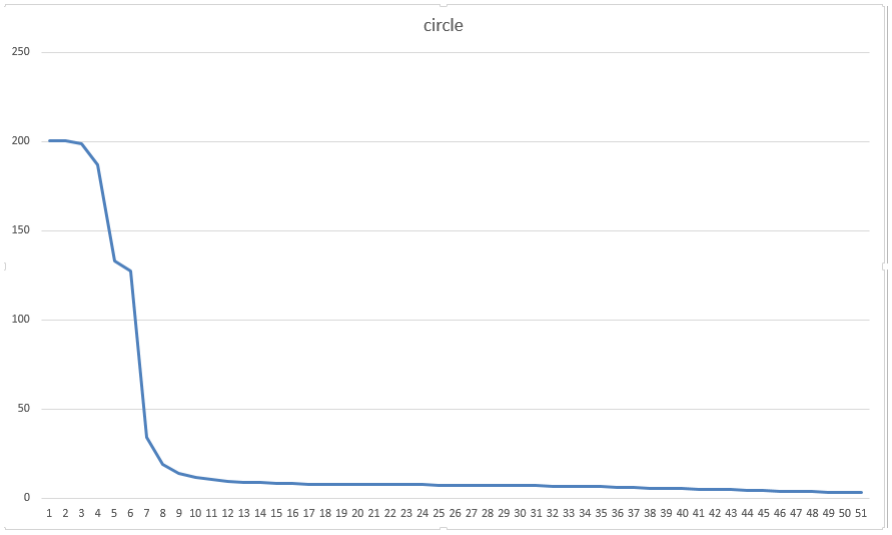


MCCR = 1.0

1.3 circle

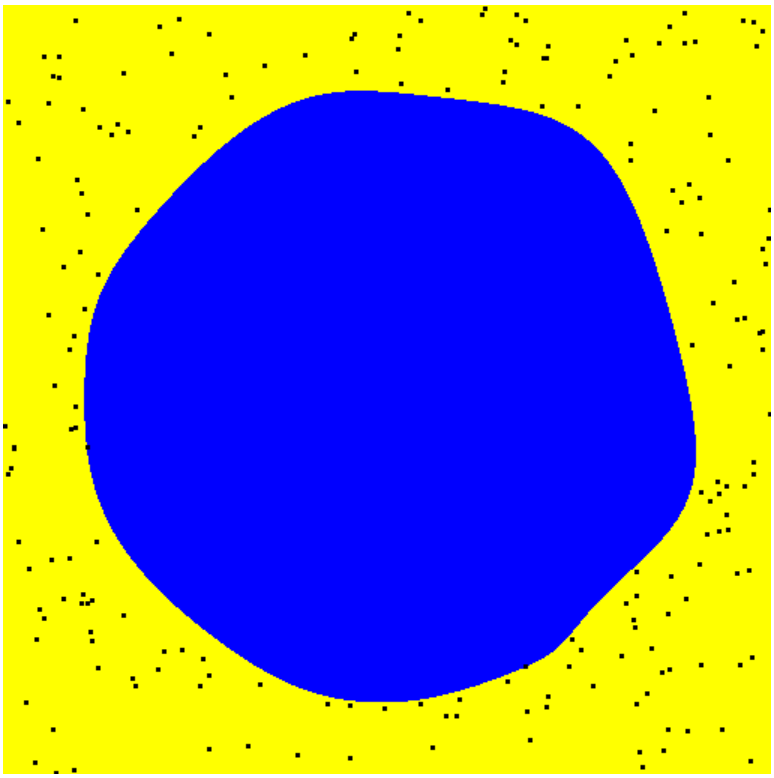
i. The network had 1 input layers of size 2 (representing the x and y coordinates of the points), 2 hidden layers of size 5 each, and an output layer containing a single neuron.

ii. The MSE developed as more epochs were fed into the network as shown in the following diagram:



Training stopped after reaching an accuracy of 100%

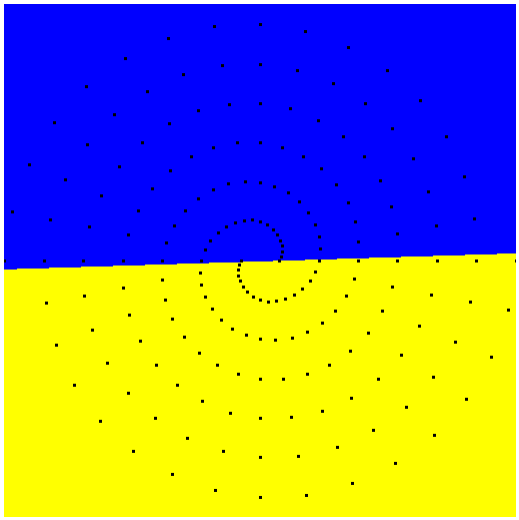
iii. The visualization of the decision boundary was as shown in the following graph:



MCCR = 1.0

1.4 spirals

I was not able to obtain reasonable results for the spirals data. The best I could achieve was an MCCR of 0.5 for each class, as shown in the following figure. The network progress freezed at the given boundary because of a bug in the code that I was unable to detect.



2 Source Code

2.1 Layer

```
namespace Classifier
{
    using System;

    /* Notice that in this implementation,
     * each node stores the weights on the edges
     * from the previous layer to herself
     * therefore, to set your output, you only need to be
     * passed the output of the previous layer, which is the input to yourself
     * you have the weights connecting that previous layer to you,
     * and you have everything you need
     */
    class Layer
    {
        public int InputSize;  // number of input neurons
        public int Size;  // number of output neurons

        public double[] YIn;
        // input to output neurons {YIn[j] = sum_i(0 <= i < n)x_i*w_ij}
        public double[] Output;
        // Y[j] = f(YIn[j])
        public double[] Delta;
        // Delta
        public double[,] W;
        // W[i, j] is the weight on the edge from x_i to y_j
        public double[] Bias;

        /* this constructor sets the sizes of the layer
         * as well as the size of the input to this layer
         */
        public Layer(int inputSize, int size)
        {
            InputSize = inputSize;
            Size = size;
            Bias = new double[Size];
            Output = new double[Size];
            YIn = new double[Size];
            Delta = new double[Size];
```

```
        W = new double[InputSize, Size];

        // initialize weights to random numbers
        Random R = new Random();
        for (int i = 0; i < InputSize; ++i)
            for (int j = 0; j < Size; ++j)
                W[i, j] = (R.NextDouble() - 0.5) / 5.0;
        for (int j = 0; j < Size; ++j)
                Bias[j] = (R.NextDouble() - 0.5) / 5.0;
    }

    // set the deltas of the neurons, knowing that this is the output layer
    public void SetOutputLayerDelta(double[] correctOutput)
    {
        for (int k = 0; k < this.Size; ++k)
        {
            this.Delta[k] =
                (correctOutput[k] - this.Output[k])
                * BipolarSigmoid.D(this.Output[k]);
        }
    }

    // set the deltas of the neurons, knowing that this is a hidden layer
    public void SetHiddenLayerDelta(Layer next)
    {
        for (int j = 0; j < this.Size; ++j)
        {
            this.Delta[j] = 0.0;
            for (int k = 0; k < next.Size; ++k)
            {
                this.Delta[j] += (next.W[j, k] * next.Delta[k]);
            }

            this.Delta[j] *= BipolarSigmoid.D(this.Output[j]);
        }
    }

    // passed the input to this layer, and a learning rate, adjust the weights
    // this method assumes that Delta has already been filled with proper values!
    public void adjustWeights(double[] previousLayerActivation, double alpha)
    {
        for (int i = 0; i < this.InputSize; ++i)
```

```
            {
                for (int j = 0; j < this.Size; ++j)
                {
                    this.W[i, j] += (alpha * this.Delta[j] * previousLayerActivation[i]);
                }
            }
            for (int j = 0; j < this.Size; ++j)
            {
                this.Bias[j] += (alpha * this.Delta[j]);
            }
        }

        /* this method accepts input to the layer,
         * and sets correct values to the output
         *
         * it assumes values of W have already been set
         * but it does not set delta
         */
        public void input(double[] previousLayerActivation)
        {
            // for each neuron in the layer
            for (int j = 0; j < this.Size; ++j)
            {
                // calculate input to the neuron
                this.YIn[j] = this.Bias[j];
                for (int i = 0; i < this.InputSize; ++i)
                {
                    this.YIn[j] += previousLayerActivation[i] * this.W[i, j];
                }
                // calculate output of the neuron
                this.Output[j] = BipolarSigmoid.F(this.YIn[j]);
            }
        }
    }
}
```

2.2 Network

```
namespace Classifier
{
    using System;
    using System.Collections.Generic;
```

```
using System.Linq;
using System.Drawing;
using System.Windows;

class Net
{
    List<Layer> Layers;             // network layers
    private int NumLayers;          // number of layers
    private int InputSize;          // size of input

    // net constructor
    public Net(int numLayers, int inputSize, int[] hiddenLayerSize)
    {
        InputSize = inputSize;
        NumLayers = numLayers;
        Layers = new List<Layer>();
        Layers.Add(new Layer(InputSize, hiddenLayerSize[0]));
        for (int i = 1; i < NumLayers; ++i)
            Layers.Add(new Layer(hiddenLayerSize[i - 1], hiddenLayerSize[i]));
    }

    public void input(double[] x)
    {
        Layers[0].input(x);
        for (int i = 1; i < NumLayers; ++i)
            Layers[i].input(Layers[i - 1].Output);
    }

    public double [] Output
    {
        set { }
        get
        {
            return Layers[Layers.Count - 1].Output;
        }
    }

    public void trainOnSample(double[] input, double[] correctOutput)
    {
        // give this sample to the network to compute
        this.input(input);
```

```
        // set the deltas
        Layers[Layers.Count - 1].SetOutputLayerDelta(correctOutput);
        for (int i = Layers.Count - 2; i >= 0; --i)
        {
            Layers[i].SetHiddenLayerDelta(Layers[i + 1]);
        }

        // now, adjust the weights
        Layers[0].adjustWeights(input, Program.Alpha);
        for (int i = 1; i < NumLayers; ++i)
            Layers[i].adjustWeights(Layers[i - 1].Output, Program.Alpha);
    }

    public Tuple<double, double> trainOnEpoch(
        List<Tuple<System.Windows.Point, double>> points)
    {
        /*
        Random R = new Random();
        for (int i = 0; i < points.Count; ++i)
        {
            int j = R.Next() % points.Count;
            var temp = points[i];
            points[i] = points[j];
            points[j] = temp;
        }
        */

        double Error = 0.0;
        int correctCount = 0;
        foreach(var point in points)
        {
            this.trainOnSample(new double[] {
                point.Item1.X, point.Item1.Y},
                new double[] {point.Item2});

            Error += (this.Output[0] - point.Item2) * (this.Output[0] - point.Item2);

            if ((this.Output[0] > 0.0) == (point.Item2 > 0.0))
            {
                ++correctCount;
            }
        }
```

```
        // precision and error!
        return Tuple.Create((double)correctCount / points.Count, Error * 0.5);
    }

    public void trainForPrecision(
        List<Tuple<System.Windows.Point, double>> points,
        double desiredPrecision,
        string precisionFileName,
        string errorFileName,
        int step)
    {
        System.IO.StreamWriter ErrorFile =
            new System.IO.StreamWriter(errorFileName);
        System.IO.StreamWriter PrecisionFile =
            new System.IO.StreamWriter(precisionFileName);

        double currentPrecision = 0.0, currentError = 1.0;
        int epochCounter = 0;
        for (; currentPrecision < desiredPrecision; ++epochCounter)
        {
            var currentState = trainOnEpoch(points);
            currentPrecision = currentState.Item1;
            currentError = currentState.Item2;
            if (epochCounter % step == 0)
            {
                Console.WriteLine("{0}\t: {1}", epochCounter, currentPrecision);
                ErrorFile.Write("{0}\t", currentError);
                PrecisionFile.Write("{0}\t", currentPrecision);
            }

            if (epochCounter % 10000 == 0)
            {
                PrintImage(
                    errorFileName.Substring(0, errorFileName.Length - 10) +
↪   epochCounter.ToString() + ".png",
                    128,
                    points.Select(x => Tuple.Create(x.Item1.X, x.Item1.Y)).ToList());
            }
        }

        ErrorFile.Write("{0}", currentError);
```

```
        PrecisionFile.Write("{0}", currentPrecision);
        ErrorFile.Close();
        PrecisionFile.Close();
    }
    /// <summary>
    /// this function prints the current state of the network as an image
    /// </summary>
    /// <param name="imageFilePath">name of file to print results in</param>
    /// <param name="dimension">number of pixels of the picture</param>
    /// <param name="specialPoints">a list of points representing the yellow class,
    /// to be highlighted in the diagram</param>
    public void PrintImage(
        string imageFilePath,
        int dimension,
        List<Tuple<double, double>> specialPoints = null)
    {
        int[] dx = new int[]{1, 1, 1, 0, 0, 0, -1, -1, -1};
        int[] dy = new int[]{1, 0, -1, 1, 0, -1, 1, 0, -1};
        int Origin = dimension >> 1;
        // origin would be (dimension, dimension)
        Bitmap image = new Bitmap(dimension, dimension);
        Random R = new Random();
        for (int i = 0; i < dimension; ++i)
        {
            for (int j = 0; j < dimension; ++j)
            {
                double[] coordinates = {
                    (double)(i - Origin) / Origin,
                    (double)(j - Origin) / Origin };
                this.input(coordinates);
                image.SetPixel(i, j, this.Output[0] > 0.0 ? Color.Yellow : Color.Blue);
            }
        }

        if (specialPoints != null)
        {
            foreach (Tuple<double, double> point in specialPoints)
            {
                int x = (int)(point.Item1 * Origin) + Origin,
                    y = (int)(point.Item2 * Origin) + Origin;
                for (int i = 0; i < 9; ++i)
                {
```

```
                        int cur_x = x + dx[i], cur_y = y + dy[i];
                        if (cur_x >= 0
                            && cur_x < dimension
                            && cur_y >= 0
                            && cur_y < dimension)
                            image.SetPixel(cur_x, cur_y, Color.Black);
                    }
                }
            }

            image.Save(imageFilePath);
        }
    }
}
```

2.3 Activation Function

```
namespace Classifier
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Threading.Tasks;
    class BipolarSigmoid
    {
        private const double Sigma = 1.0;

        // passed x, return f(x)
        public static double F(double value)
        {
            return (2.0 / (1.0 + Math.Exp(-Sigma * value))) - 1.0;
        }

        // passed the value of f(x), return f'(x)
        public static double D(double value)
        {
            return (1.0 + value) * (1.0 - value) * Sigma * 0.5;
        }
    }
}
```

2.4 Parser and Driver

```
namespace Classifier
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Windows;
    using System.IO;

    class Program
    {
        public static int NumLayers = 4;
        public static int InputSize = 2;
        public static int[] LayerSize = {5, 5, 5, 1};
        public const double Alpha = 0.001;

        public static double DesiredPrecision = 1.0;
        public static string FilePath = @"xor";
        public static int ImageDimension = 512;
        public static int Step = 100;

        static void Main(string[] args)
        {
            Net N = new Net(NumLayers, InputSize, LayerSize);
            var Points = ParseTrainingPoints(FilePath + ".txt");
            N.trainForPrecision(
                Points,
                DesiredPrecision,
                FilePath + "_precision.txt",
                FilePath + "_error.txt",
                Step);
            N.PrintImage(
            FilePath + "_boundary.png",
            512,
            Points.Where(x => x.Item2 > 0.0).Select(x => Tuple.Create(x.Item1.X,
↪   x.Item1.Y)).ToList());
        }

        public static List<Tuple<Point, double>> ParseTrainingPoints(string filePath)
        {
            List<Tuple<Point, double>> ans = new List<Tuple<Point, double>>();
            StreamReader reader = new StreamReader(filePath);
```

```
            for (string line = reader.ReadLine(); line != null; line = reader.ReadLine())
            {
                string[] arr = line.Split('\t');
                ans.Add(Tuple.Create(
                    new Point(double.Parse(arr[0]), double.Parse(arr[1])),
                    int.Parse(arr[2]) == 1 ? 1.0 : -1.0));
            }

            return ans;
        }
    }
}
```