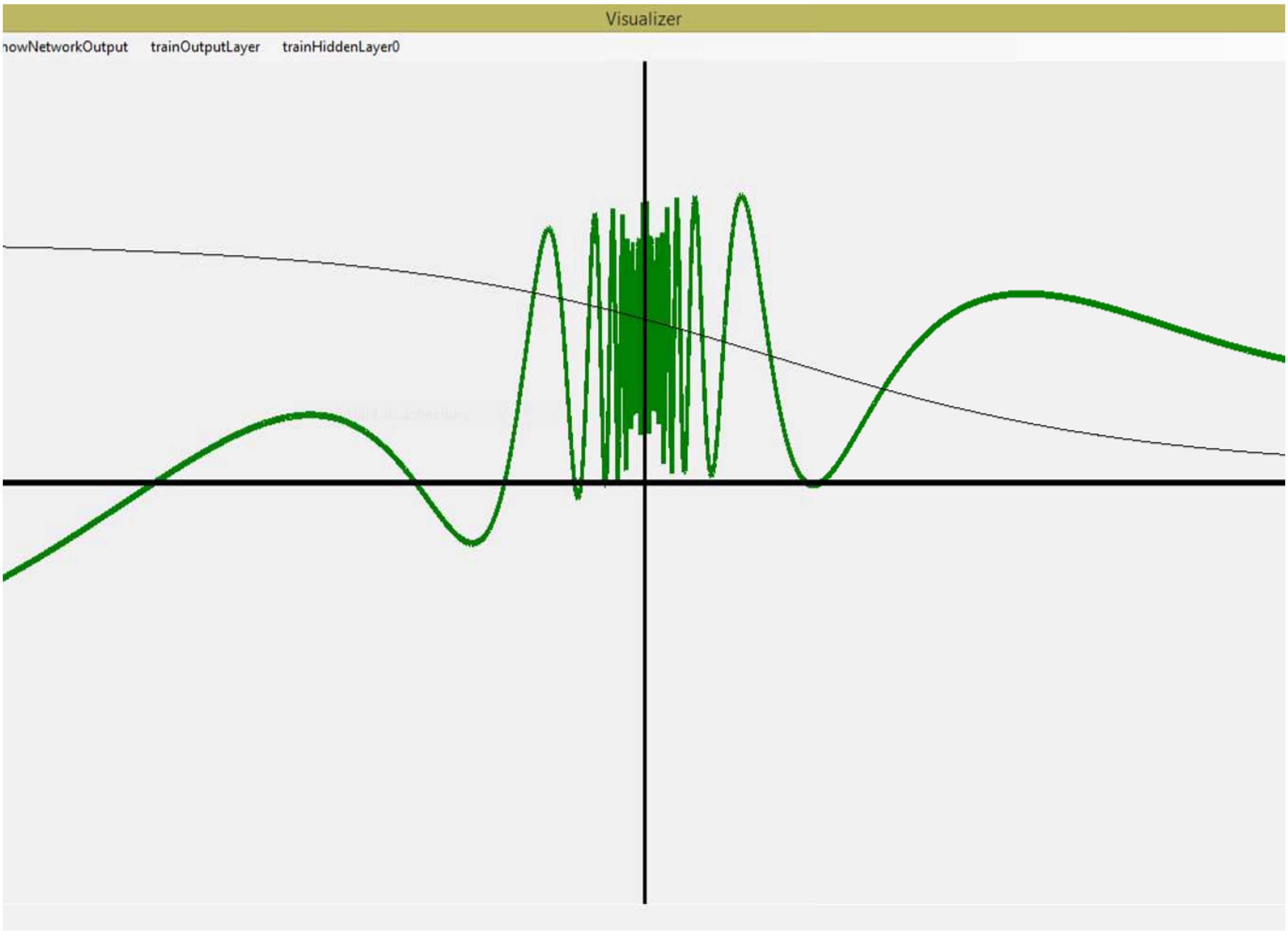
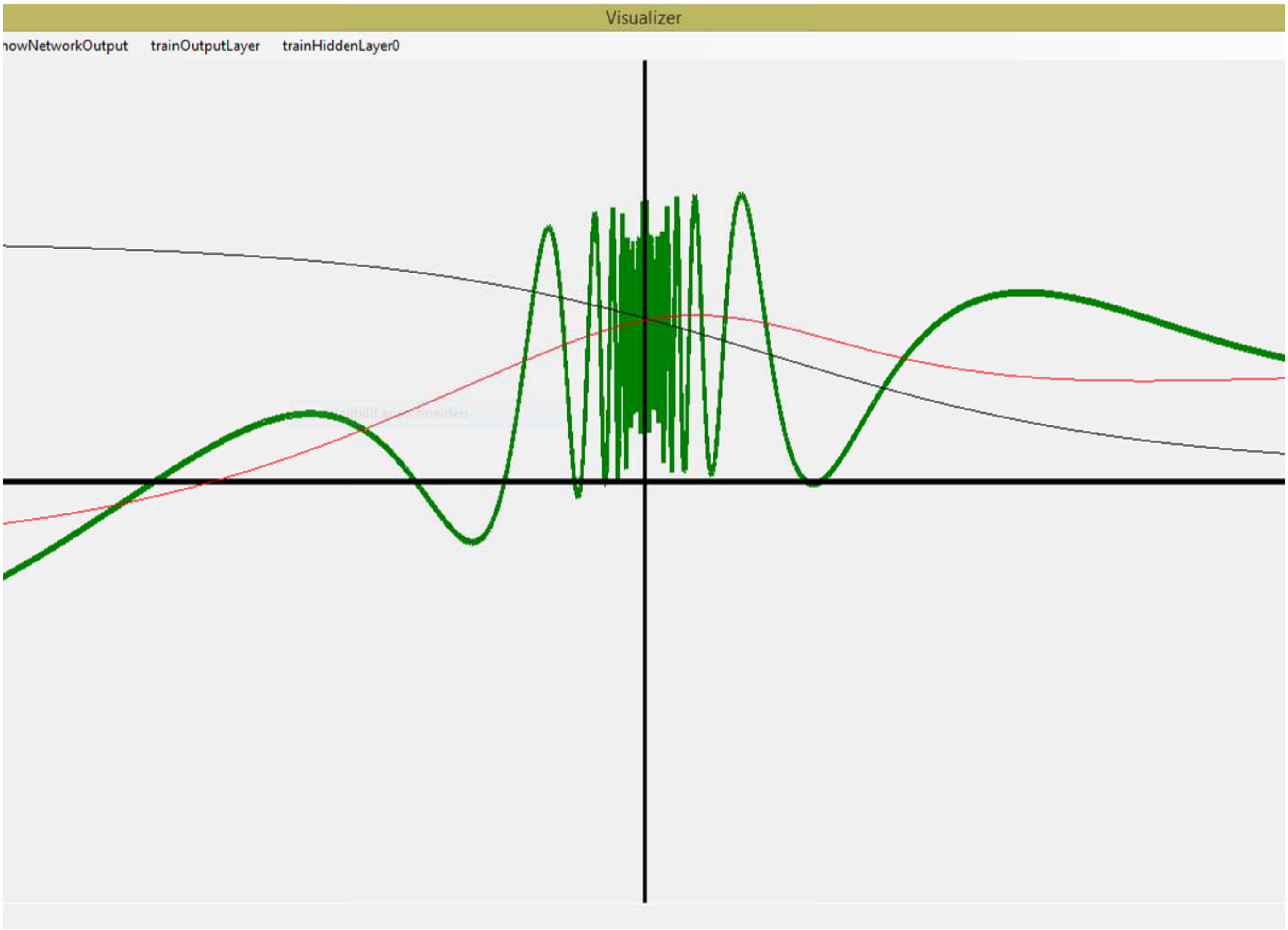
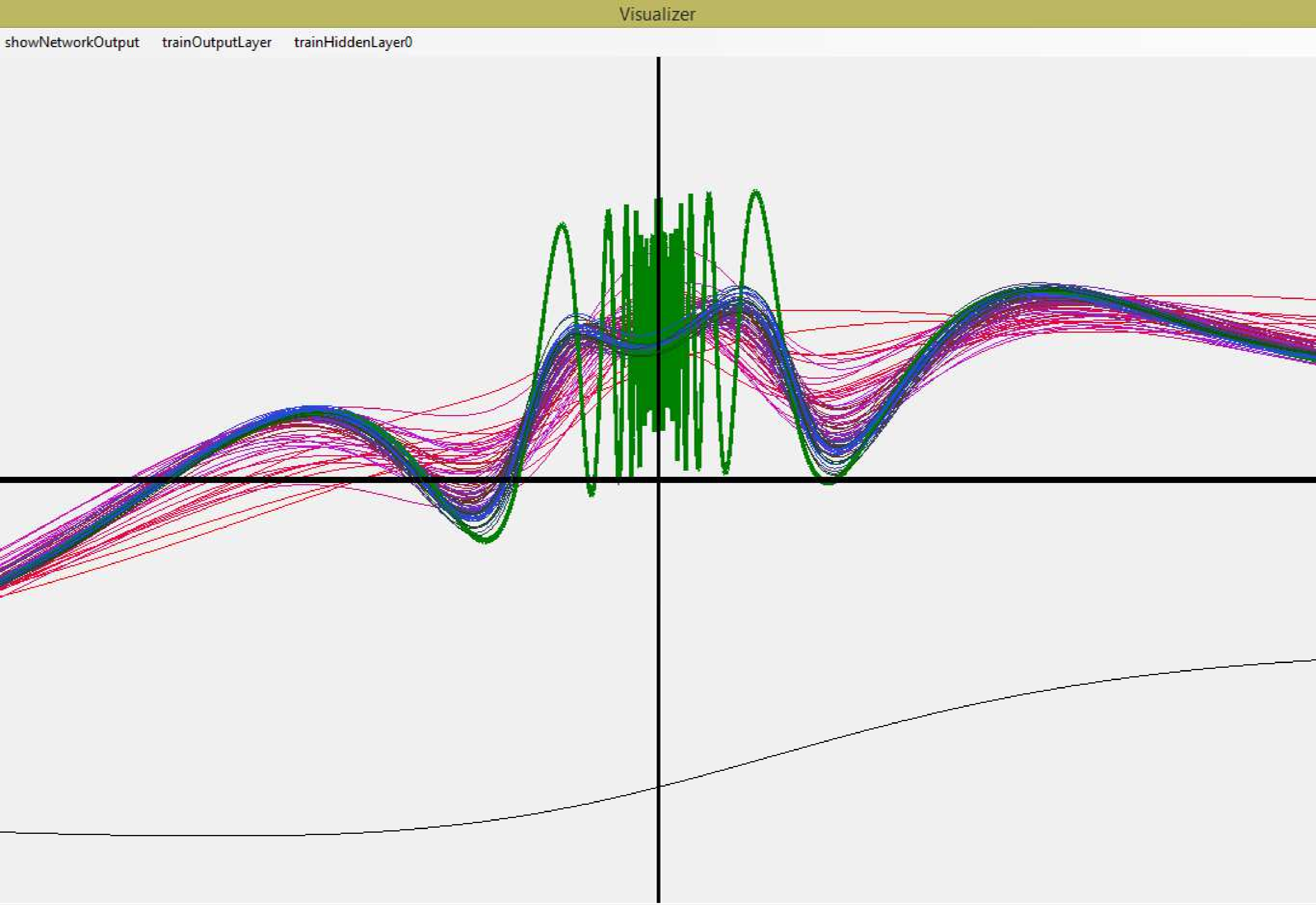
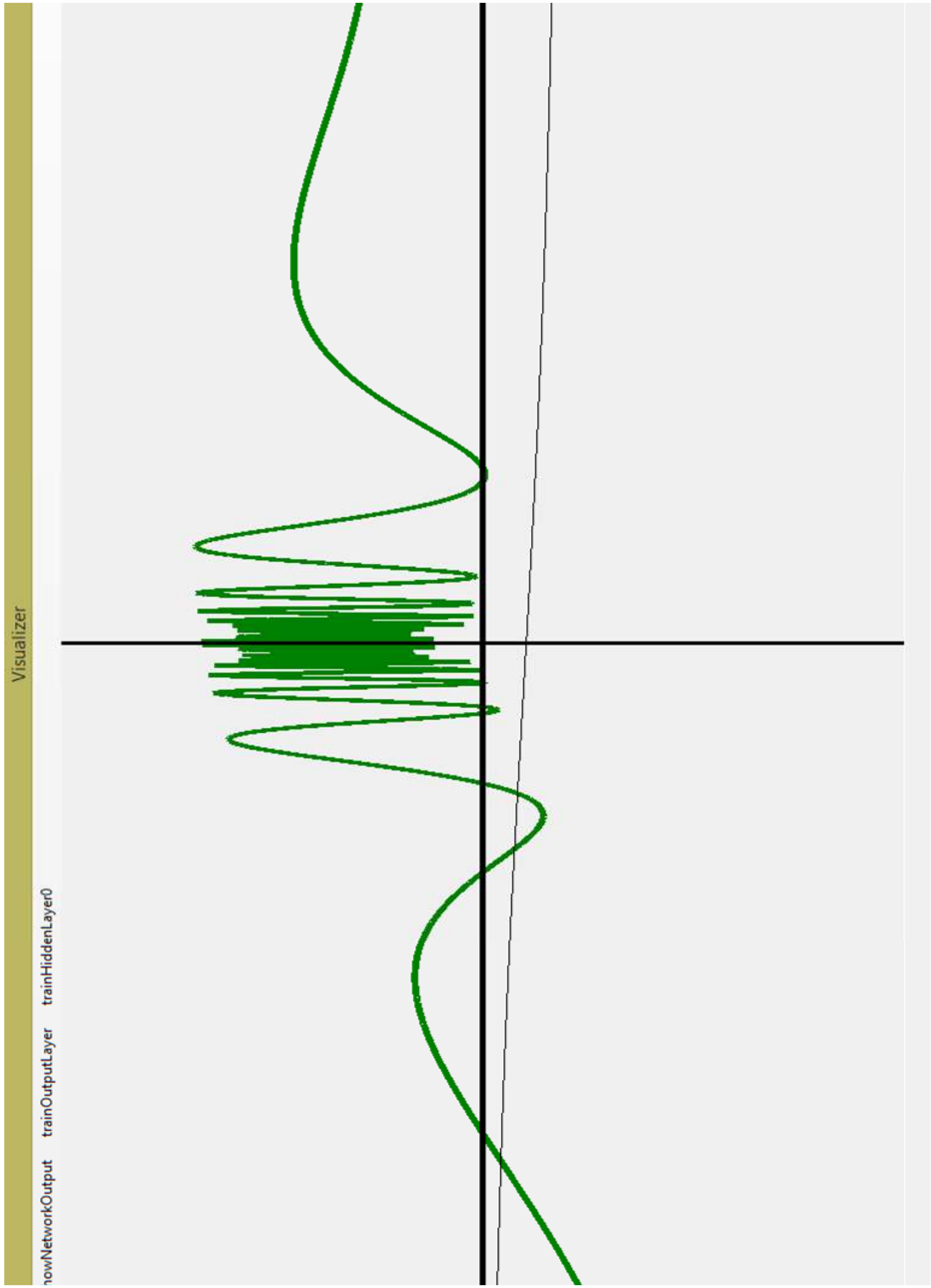
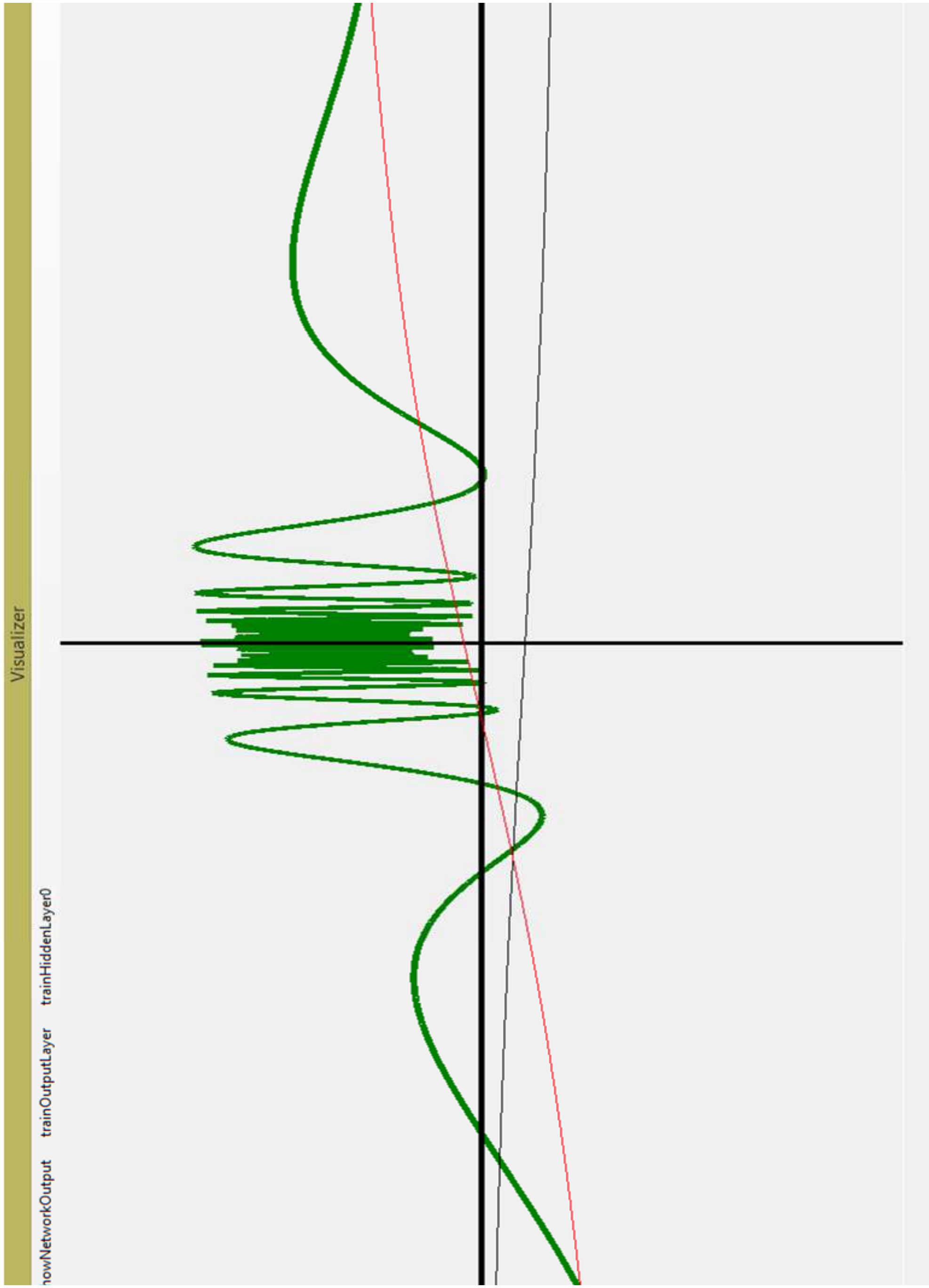
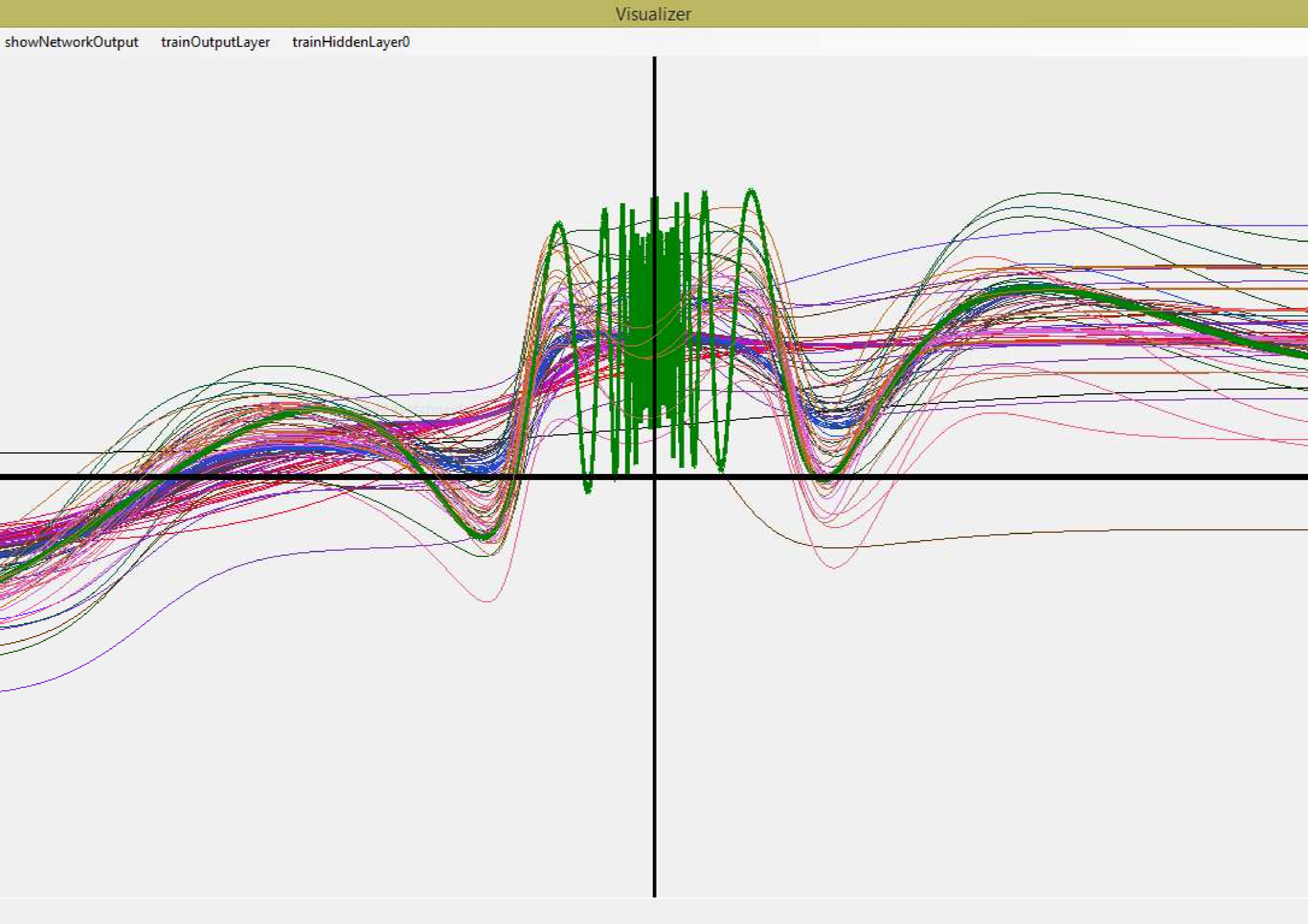

Visualizer
howNetworkOutput
trainOutputLayer
trainHiddenLayer0


Visualizer
howNetworkOutput
trainOutputLayer
trainHiddenLayer0


Visualizer
showNetworkOutput
trainOutputLayer
trainHiddenLayer0


Visualizer
nowNetworkOutput
trainOutputLayer
trainHiddenLayer0


Visualizer
nowNetworkOutput
trainOutputLayer
trainHiddenLayer0


Visualizer
showNetworkOutput
trainOutputLayer
trainHiddenLayer0

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Neural_Network {
    class BiasNeuron : Neuron {
        public override void learn(TrainingInstance t) {
            return;
        }
        public override void addIncomingSynapse(Synapse s, double initWeight) {
            throw new Exception("Bias Neuron cannot have incoming synapses");
        }

        public override void addOutgoingSynapse(Synapse s) {
            base.addOutgoingSynapse(s);
            s.voltage = 1;
        }

        public override void calc() {
            foreach (Synapse s in outgoingSynapses) {
                s.voltage = 1;
            }
            currentOutputVoltage = 1;
        }
        public override void setStaticOutput(double v) {
            throw new NotImplementedException();
        }
    }
}

```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Neural_Network {
    abstract class Layer {
        public List<Neuron> neurons = new List<Neuron>();
        public Layer(int capacity) {
            construct(capacity);
        }

        protected abstract void construct(int capacity);
    }
    class InputLayer : Layer {
        public InputLayer(int capacity)
            : base(capacity) {
        }
        protected override void construct(int capacity) {
            neurons.Add(new BiasNeuron());
            for (int i = 0; i < capacity; ++i) {
                neurons.Add(new PerceptronInputCell());
            }
        }
    }
    class HiddenLayer : Layer {
        public HiddenLayer(int capacity)
            : base(capacity) {
        }
        protected override void construct(int capacity) {
            neurons.Add(new BiasNeuron());
            for (int i = 0; i < capacity; ++i) {
                neurons.Add(new PerzeptronHiddenCell());
            }
        }
    }
    class OutputLayer : Layer {
        public OutputLayer(int capacity)
            : base(capacity) {
        }
        protected override void construct(int capacity) {
            for (int i = 0; i < capacity; ++i) {
                neurons.Add(new PerceptronOutputCell());
            }
        }
    }
}

```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Neural_Network
{
    abstract class Neuron{
        protected List<Synapse> incomingSynapses = new List<Synapse>();
        protected List<Synapse> outgoingSynapses = new List<Synapse>();

        protected double learningRate = 0.3;
        public double delta = 0.0;

        public abstract void learn(TrainingInstance t);
        public virtual void setDelta(TrainingInstance t) { }

        protected double currentOutputVoltage;
        public double getCurrentOutputValue() {
            return activate(excitation());
        }

        public virtual void setStaticOutput(double v){}

        public virtual void addIncomingSynapse(Synapse s, double initWeight) {
            incomingSynapses.Add(s);
            s.weight = initWeight;
        }

        public virtual void addOutgoingSynapse(Synapse s) {
            outgoingSynapses.Add(s);
        }

        protected double excitation() {
            double sum = 0.0;
            foreach (Synapse s in incomingSynapses) {
                sum += s.voltage * s.weight;
            }
            return sum;
        }

        protected virtual double activate(double sum) { return sum; }
        protected virtual double activateDifferentiated(double sum) { return 1; }

        public virtual void calc() {
            currentOutputVoltage = activate(excitation());
            foreach (Synapse s in outgoingSynapses) {
                s.voltage = currentOutputVoltage;
            }
        }
    }
}

```

```
using System;
using System.Collections.Generic;

namespace Neural_Network {
    class Perceptron {
        public Layer inputLayer;
        public Layer outputLayer;
        public List<Layer> hiddenLayers;
        private double initWeightMin;
        private double initWeightMax;

        public Perceptron(List<int> numberOfNeurons, double initWeightMin, double initWeightMax) {
            inputLayer = new InputLayer(numberOfNeurons[0]);
            hiddenLayers = new List<Layer>();
            for (int i = 1; i < numberOfNeurons.Count - 1; ++i) {
                hiddenLayers.Add(new HiddenLayer(numberOfNeurons[i]));
            }
            outputLayer = new OutputLayer(numberOfNeurons[numberOfNeurons.Count - 1]);

            this.initWeightMax = initWeightMax;
            this.initWeightMin = initWeightMin;

            connectFully();
        }

        private void connectFully() {
            Random rand = new Random();
            for (int i = 0; i < hiddenLayers.Count+1; ++i) {
                Layer current;
                current = i < hiddenLayers.Count ? hiddenLayers[i] : inputLayer;
                Layer next;
                if(i < hiddenLayers.Count - 1)
                    next=hiddenLayers[i + 1];
                else if(i==hiddenLayers.Count-1)
                    next=outputLayer;
                else
                    next=hiddenLayers[0];
                foreach (Neuron from in current.neurons) {
                    foreach (Neuron to in next.neurons) {
                        if (to.GetType() == typeof(BiasNeuron)) {
                            continue;
                        }
                        Synapse s = new Synapse(from, to);
                        from.addOutgoingSynapse(s);
                        to.addIncomingSynapse(s, rand.NextDouble()*(initWeightMax-initWeightMin)+initWeightMin);
                    }
                }
            }
        }

        public List<double> feedForward(TrainingInstance tr){
            if (tr.inputVector.Count != inputLayer.neurons.Count-1) { //-1 due to bias neuron
                throw new Exception("input vector size does not match input layer neuron count");
            }

            for (int i = 1; i < inputLayer.neurons.Count; ++i){
                inputLayer.neurons[i].setStaticOutput(tr.inputVector[i-1]);
            }
            for (int i = 0; i < hiddenLayers.Count; ++i) {
                for (int j = 0; j < hiddenLayers[i].neurons.Count; ++j){
                    hiddenLayers[i].neurons[j].calc();
                }
            }
            List<double> results = new List<double>();

            for (int j = 0; j < outputLayer.neurons.Count; ++j) {
                outputLayer.neurons[j].calc();
                results.Add(outputLayer.neurons[j].getCurrentOutputValue());
            }
            return results;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Neural_Network {
    class PerceptronInputCell : Neuron {
        public override void setStaticOutput(double value) {
            currentOutputVoltage = value;
            foreach (Synapse s in outgoingSynapses) {
                s.voltage = currentOutputVoltage;
            }
        }

        public override void learn(TrainingInstance t) {
            throw new InvalidOperationException("Input Cells cannot learn");
        }
    }
}

```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Neural_Network {
    class PerceptronOutputCell : Neuron {

        public override void learn(TrainingInstance t) {
            setDelta(t);

            foreach (Synapse s in incomingSynapses) {
                s.weight += -learningRate * s.voltage * delta;
            }
        }

        public override void setDelta(TrainingInstance t) {
            calc();
            delta = activateDifferentiated(excitation()) * (currentOutputVoltage - t.expectedOutput);
        }

        protected override double activate(double sum)
        {
            return sum;
        }

        protected override double activateDifferentiated(double sum)
        {
            return 1;
        }
    }
}

```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Neural_Network
{
    class PerzeptronHiddenCell : Neuron
    {
        public override void learn(TrainingInstance ti) {
            setDelta(ti);

            foreach (Synapse s in incomingSynapses){
                s.weight += -learningRate * s.voltage * delta;
            }
        }

        public override void setDelta(TrainingInstance t) {
            double sumout = 0.0;
            foreach (Synapse s in outgoingSynapses) {
                s.to.calc();
                s.to.setDelta(t);
                sumout += s.weight * s.to.delta;
            }
            delta = activateDifferentiated(excitation()) * sumout;
        }

        protected override double activate(double sum)
        {
            //fermi function
            return 1 / (1 + Math.Pow(Math.E, -sum));
        }

        protected override double activateDifferentiated(double sum)
        {
            //differentiated fermi function
            double a = activate(sum);
            return a*(1.0-a);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Neural_Network
{
    class Program
    {
        static void Main(string[] args)
        {
            Visualizer vs = new Visualizer();
            vs.ShowDialog();
        }
    }
}

```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Neural_Network
{
    class Synapse
    {
        public Neuron from;
        public Neuron to;
        public double voltage = 0.0;
        public double weight = 0.0; //only to be set by "to" neuron

        public Synapse(Neuron from, Neuron to) {
            this.from = from;
            this.to = to;
        }
    }
}

```

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Neural_Network
{
    class Trainer
    {
        private Perceptron perceptron;
        private Random r = new Random();
        public List<TrainingInstance> training;

        public Trainer() {
            // {1,10,1} -> 1 neuron in input layer, 10 in hidden layer, 1 in output layer
            List<int> numberOfNeurons = new List<int>(new int[] {1,10,1});
            // random init weights in -0.5 0.5
            perceptron = new Perceptron(numberOfNeurons, -0.5, 0.5);
            createTrainingSet();
        }

        private double f(double x) { // the function to be approximated
            return (Math.Cos(x / 3) + Math.Sin(10 / (Math.Abs(x) + 0.1)) + 0.1 * x);
        }

        public void createTrainingSet() {
            training = new List<TrainingInstance>();
            for (int i = 0; i < 1001; ++i) {
                training.Add(new TrainingInstance(new List<double>(new double[] { -10.0 + i * 20.0 / 1001.0 }), f(-10.0 + i * 20.0 / 1001.0)));
            }
        }

        public List<List<double>> trainingResults(){
            List<List<double>> results = new List<List<double>>();
            foreach (TrainingInstance ti in training) {
                results.Add(perceptron.feedForward(ti));
            }
            return results;
        }

        public void trainOutputLayer(){
            var permutated = training.OrderBy(item => r.Next());
            foreach (TrainingInstance ti in permutated) {
                perceptron.feedForward(ti);
                foreach (Neuron n in perceptron.outputLayer.neurons){
                    n.learn(ti);
                }
            }
        }

        public void trainHiddenLayer() {
            var permutated = training.OrderBy(item => r.Next());
            foreach (TrainingInstance ti in permutated) {
                perceptron.feedForward(ti);
                foreach(Neuron n in perceptron.hiddenLayers[0].neurons) {
                    n.learn(ti);
                }
            }
        }

        public double meanSquareError() {
            double d=0.0;
            foreach (TrainingInstance ti in training) {
                perceptron.feedForward(ti);
                d+=Math.Pow(perceptron.outputLayer.neurons[0].getCurrentOutputValue()-ti.expectedOutput, 2.0);
            }
            d /= (2*training.Count);
            return d;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Neural_Network
{
    class TrainingInstance
    {
        public TrainingInstance(List<double> inputVector, double expectedOutput)
        {
            this.expectedOutput = expectedOutput;
            this.inputVector = inputVector;
        }
        public double expectedOutput;
        public List<double> inputVector;
    }
}

```

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Neural_Network
{
    public partial class Visualizer : Form
    {
        public Visualizer()
        {
            InitializeComponent();
        }

        Trainer trainer = new Trainer();

        private void runToolStripMenuItem_Click(object sender, EventArgs e)
        {
            for (int i = 0; i < trainer.training.Count-1; ++i) {
                g1.DrawLine(p1,
                    Convert.ToSingle(trainer.training[i].inputVector[0]),
                    Convert.ToSingle(trainer.training[i].expectedOutput),
                    Convert.ToSingle(trainer.training[i + 1].inputVector[0]),
                    Convert.ToSingle(trainer.training[i + 1].expectedOutput)
                    );
                //g1.DrawRectangle(p,Convert.ToSingle(trainer.training[i].inputVector[0]), Convert.ToSingle(trainer.training[i].expectedOutput), 0.001f, 0.001f);
            }
        }

        Graphics g1;
        int w, h;
        Pen p1, p2;
        protected override void OnLoad(EventArgs e)
        {
            base.OnLoad(e);

            g1 = pictureBox1.CreateGraphics();

            w = pictureBox1.Width;
            h = pictureBox1.Height;


            g1.TranslateTransform(pictureBox1.Width / 2, pictureBox1.Height / 2);
            g1.ScaleTransform(pictureBox1.Width / 20.0F, -pictureBox1.Height / 6.0F);


            p1 = new Pen(Color.Green, 0.05F); // target function
            p2 = new Pen(Color.Black, 0.05F); // coord axis

        }

        int outputCounter = 0;

        private void showNetworkOutputToolStripMenuItem_Click(object sender, EventArgs e)
        {
            List<List<double>> tr = trainer.trainingResults();

            //vary output color
            Pen p = new Pen(Color.FromArgb((255*5-3*outputCounter)%255,(outputCounter)%255,(10*outputCounter++)%255), 0.001F);

            //axis
            g1.DrawLine(p2, -10f, 0f, 10f, 0f);
            g1.DrawLine(p2, 0f, -3f, 0f, 3f);

            for (int i = 0; i < tr.Count-1; ++i)
            {
              g1.DrawLine(p,
                    Convert.ToSingle(trainer.training[i].inputVector[0]),
                    Convert.ToSingle(tr[i][0]),
                    Convert.ToSingle(trainer.training[i + 1].inputVector[0]),
                    Convert.ToSingle(tr[i+1][0])
                    );
            }

            toolStripStatusLabel1.Text = trainer.meanSquareError().ToString();
        }

        private void trainOutputLayerToolStripMenuItem_Click(object sender, EventArgs e)
        {
            trainer.trainOutputLayer();
            showNetworkOutputToolStripMenuItem_Click(sender, e);
        }

        private void trainHiddenLayer0ToolStripMenuItem_Click(object sender, EventArgs e) {
            trainer.trainHiddenLayer();
            showNetworkOutputToolStripMenuItem_Click(sender, e);
        }
    }
}
```