# CPEN 502 Assignment-b: Reinforcement Learning (Look Up Table)

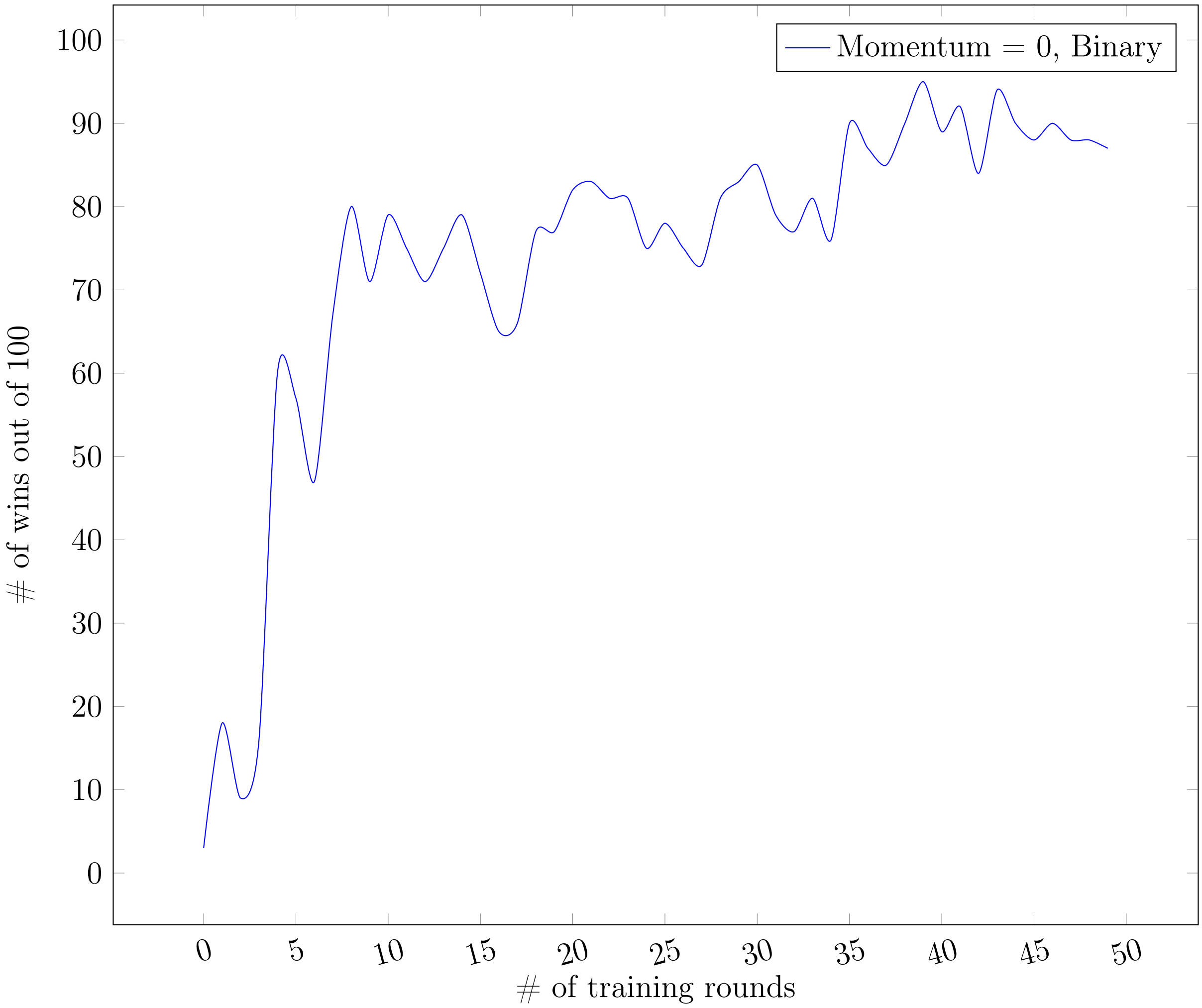
Ali Asgari Khoshouyeh (Student #24868739)

24. November 2021

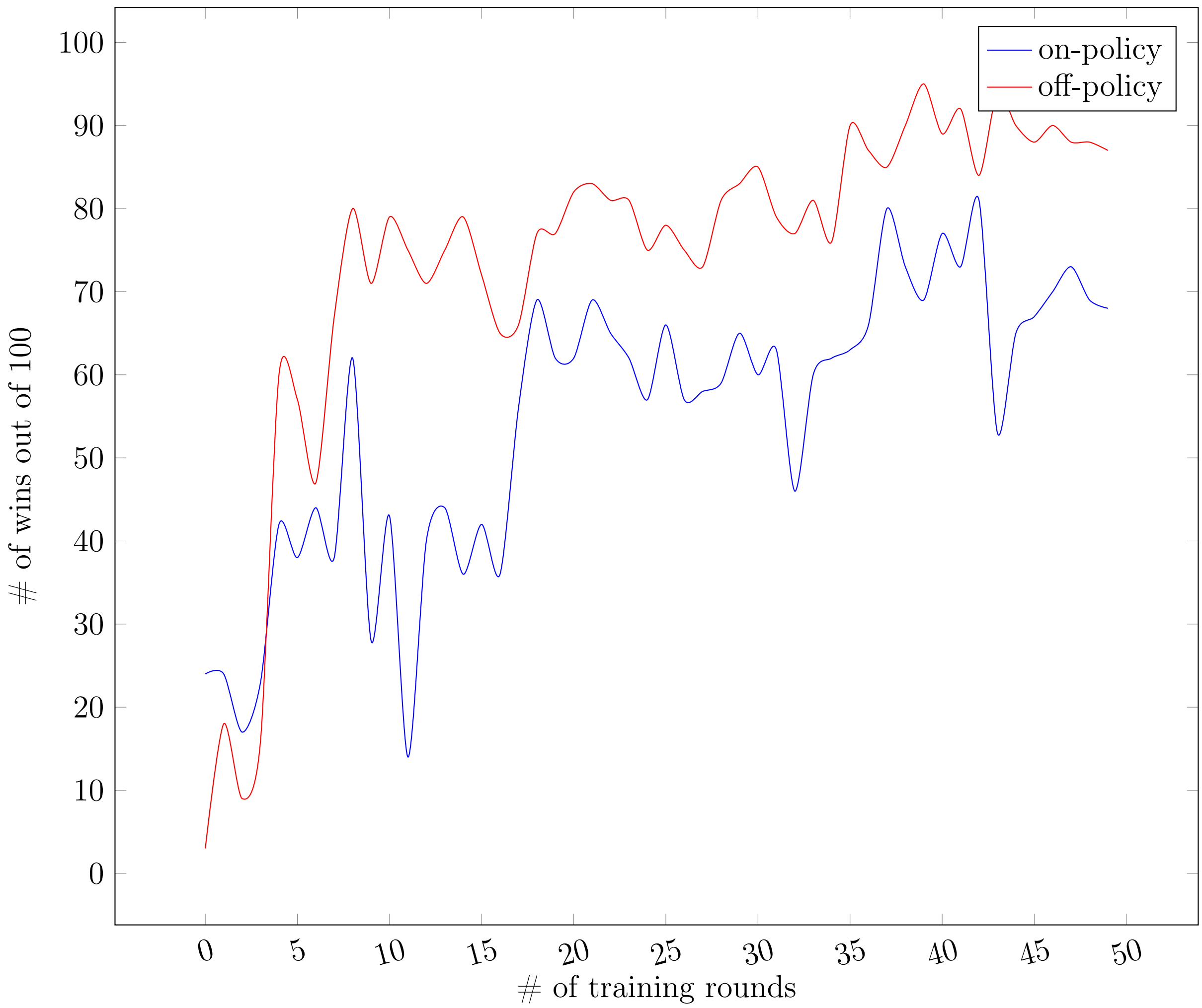
## 1 Q Learning Robot

(2) Once you have your robot working, measure its learning performance as follows:

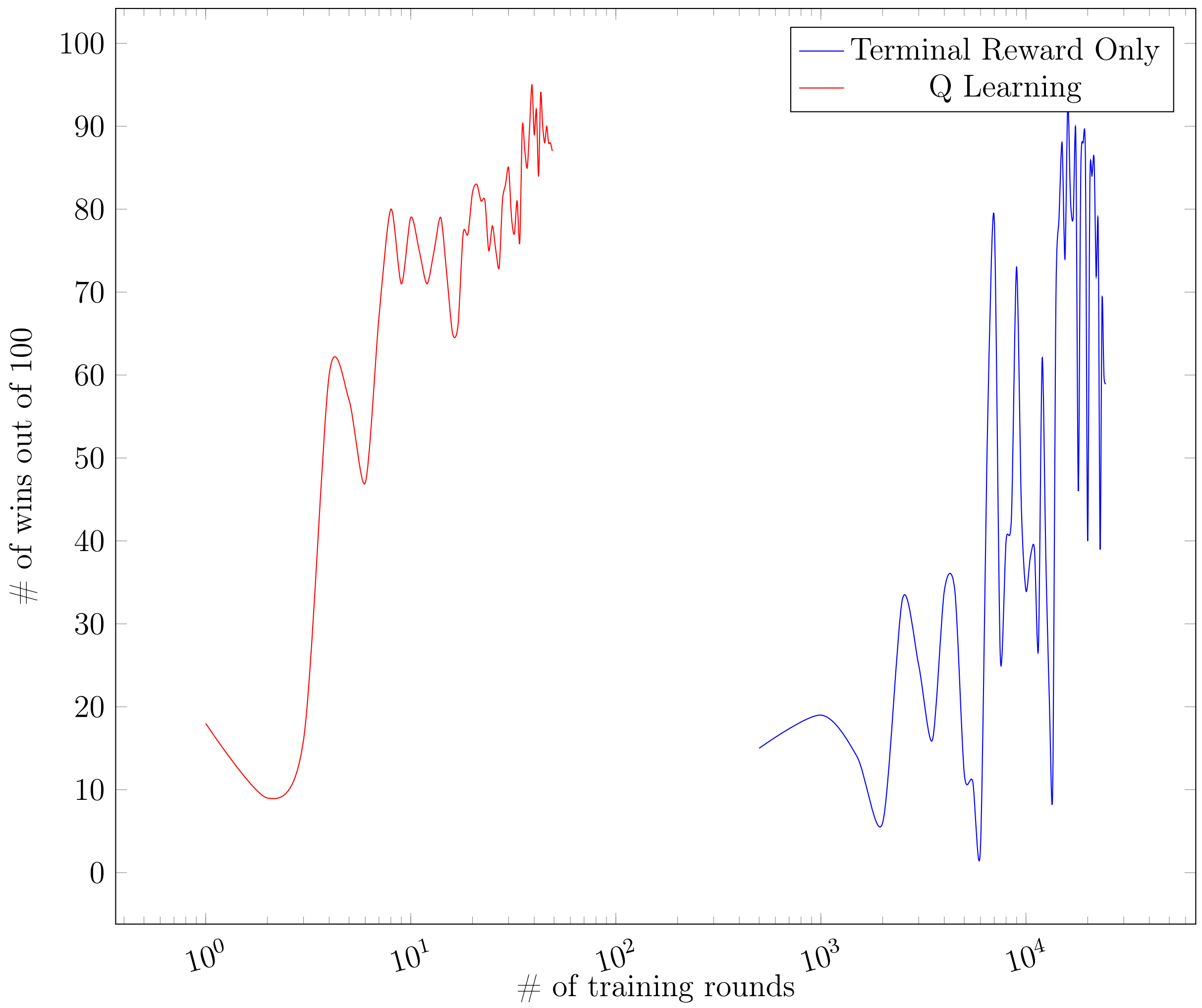
a) Draw a graph of a parameter that reflects a measure of progress of learning and comment on the convergence of learning of your robot.



b) Using your robot, show a graph comparing the performance of your robot using on-policy learning vs off-policy learning.



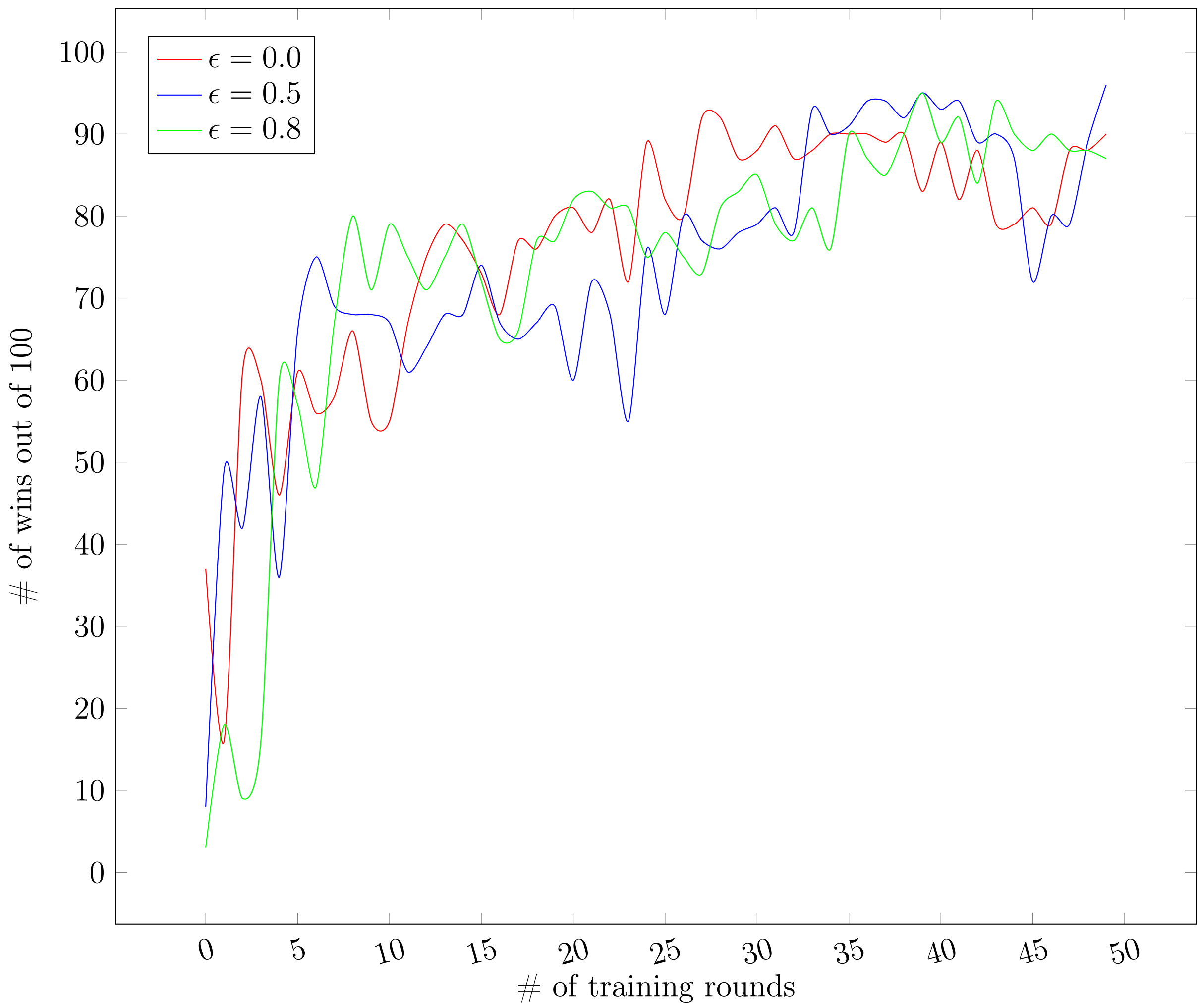
c) Implement a version of your robot that assumes only terminal rewards and show & compare its behaviour with one having intermediate rewards.

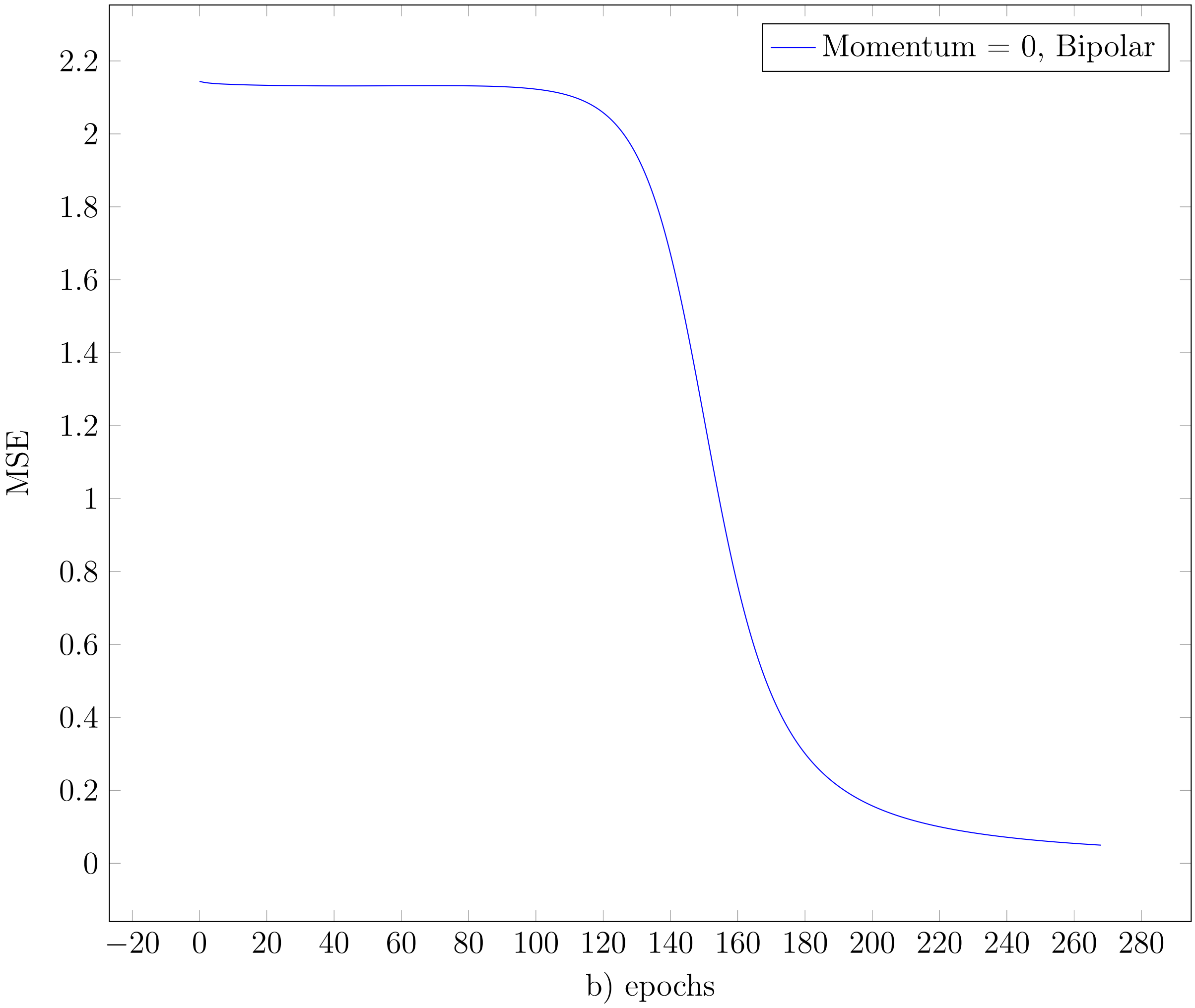


## 2 Role of $\epsilon$

(3) This part is about exploration. While training via RL, the next move is selected randomly with probability $\epsilon$ and greedily with probability $1 - \epsilon$

a) Compare training performance using different values of e including no exploration at all. Provide graphs of the measured performance of your tank vs $\epsilon$


$\epsilon = 0.0$
$\epsilon = 0.5$
$\epsilon = 0.8$
# of wins out of 100
0
10
20
30
40
50
60
70
80
90
100
0
5
10
15
20
25
30
35
40
45
50
# of training rounds



# Appendices

## A Source Codes

```java
package autograd;

import jdk.jshell.spi.ExecutionControl;

public class Addition extends Operator {
    @Override
    public double evaluate(IVariable[] operands) {
        double result = 0.;
        for (IVariable operand :
                operands) {
            result += operand.evaluate();
        }
        return result;
    }

    @Override
    public void backwards(IVariable[] operands, IVariable[] sources, double gradient) throws ExecutionControl.NotImplementedException {
        for (IVariable o :
                operands) {
            o.backward(sources, gradient);
        }
    }
}
```

Listing 1: autograd/Addition.java

```
package autograd;

import jdk.jshell.spi.ExecutionControl;

public class Exponentiation extends Operator {
    @Override
    public double evaluate(IVariable[] operands) {
        if (operands.length != 2) {
            throw new IllegalArgumentException("Exponentiation accepts 2 arguments.");
        }
        return Math.pow(operands[0].evaluate(), operands[1].evaluate());
    }

    @Override
    public void backwards(IVariable[] operands, IVariable[] sources, double gradient) throws ExecutionControl.NotImplementedException {
        IVariable baseVariable = operands[0];
        var baseValue = baseVariable.evaluate();
        IVariable exponentVariable = operands[1];
        var exponentValue = exponentVariable.evaluate();
        if (exponentVariable.getParameters().length > 1) {
            throw new ExecutionControl.NotImplementedException("Back propagation to the exponent is not implemented.");
        }
        var gradientToPropagate = Math.pow(gradient * baseValue * exponentValue, exponentValue - 1);
        baseVariable.backward(sources, gradientToPropagate);
    }
}
```

Listing 2: autograd/Exponentiation.java

```
package autograd;

public interface IInitializer {
    double next();
}
```

Listing 3: autograd/IInitializer.java

```
package autograd;


import jdk.jshell.spi.ExecutionControl;

public interface IOperator {
    IVariable apply(IVariable... operands);

    double evaluate(IVariable[] operands);

    void backwards(IVariable[] operands, IVariable[] sources, double
        gradient) throws ExecutionControl.NotImplementedException;
}
```

Listing 4: autograd/IOperator.java

```
package autograd;

import jdk.jshell.spi.ExecutionControl;

public interface IVariable {
    double evaluate();

    void backward(IVariable[] sources, double gradient) throws
        ExecutionControl.NotImplementedException;

    Parameter[] getParameters();
}
```

Listing 5: autograd/IVariable.java

```
package autograd;

import jdk.jshell.spi.ExecutionControl;

public class Multiplication extends Operator {

    @Override
    public double evaluate(IVariable[] operands) {
        double result = 1.;
        for (IVariable operand :
                operands) {
            result *= operand.evaluate();
        }
        return result;
    }

    @Override
    public void backwards(IVariable[] operands, IVariable[] sources, double
         gradient) throws ExecutionControl.NotImplementedException {
        validateOperands(operands);
        var multiplier = operands[0];
        var multiplicand = operands[1];
        var multiplierValue = multiplier.evaluate();
        var multiplicandValue = multiplicand.evaluate();
        multiplier.backward(sources, gradient * multiplicandValue);
        multiplicand.backward(sources, gradient * multiplierValue);
    }
}
```

Listing 6: autograd/Multiplication.java

```
package autograd;

```

```java
import jdk.jshell.spi.ExecutionControl;

public class Negation extends Operator {

    public Negation() {
        this.numberOfOperands = 1;
    }

    @Override
    public double evaluate(IVariable[] operands) {
        validateOperands(operands);
        return -operands[0].evaluate();
    }

    @Override
    public void backwards(IVariable[] operands, IVariable[] sources, double
         gradient) throws ExecutionControl.NotImplementedException {
        operands[0].backward(sources, -gradient);
    }
}
```

Listing 7: autograd/Negation.java

```java
package autograd;

import jdk.jshell.spi.ExecutionControl;

import java.util.Arrays;
import java.util.HashSet;

public class Operation implements IVariable {
    private final IOperator operator;
    private final IVariable[] operands;

    public Operation(IOperator operator, IVariable... operands) {
        this.operator = operator;
        this.operands = operands;
    }

    @Override
    public double evaluate() {
        return operator.evaluate(operands);
    }

    @Override
    public void backward(IVariable[] sources, double gradient) throws
        ExecutionControl.NotImplementedException {
        operator.backwards(operands, sources, gradient);
    }


    @Override
    public Parameter[] getParameters() {
        HashSet<Parameter> result = new HashSet<>();
        for (IVariable o :
                this.operands) {
            result.addAll(Arrays.asList(o.getParameters()));
        }
        return result.toArray(new Parameter[0]);
    }

    public IVariable[] getOperands() {
        return operands;
    }
}
```

Listing 8: autograd/Operation.java

```java
package autograd;

public abstract class Operator implements IOperator {
    protected Integer numberOfOperands;

    public Operator() {
        this.numberOfOperands = null;
    }

    @Override
    public IVariable apply(IVariable... operands) {
        return new Operation(this, operands);
    }

    protected void validateOperands(IVariable[] operands) {
        if (this.numberOfOperands == null) {
            return;
        }
        if (operands.length != this.numberOfOperands) {
            throw new IllegalArgumentException(String.format("%s accepts only one operand.", this.getClass().getName()));
        }
    }
}
```

Listing 9: autograd/Operator.java

```java
package autograd;

import java.util.Arrays;

public class Parameter implements IVariable {
    private double value;
    private double gradient;
    private boolean trainable;
    private int layer;

    public Parameter() {

    }

    public Parameter(double value) {
        this.value = value;
        trainable = true;
    }

    public Parameter(double value, boolean trainable) {
        this.value = value;
        this.trainable = trainable;
    }

    public static IVariable[] createTensor(double[] desired) {
        var result = new Parameter[desired.length];
        for (int i = 0; i < result.length; i++) {
            result[i] = new Parameter(desired[i]);
        }
        return result;
    }

    @Override
    public double evaluate() {
        return value;
    }

    @Override
    public void backward(IVariable[] sources, double gradient) {
        if (Arrays.stream(sources).anyMatch(x -> x == this)) {
            setGradient(gradient + getGradient());
        }
    }

    @Override
    public Parameter[] getParameters() {
        return new Parameter[]{this};
    }

    public double getValue() {
        return this.value;
    }

    public void setValue(double value) {
        this.value = value;
    }

    public double getGradient() {
        return gradient;
    }

    private void setGradient(double gradient) {
        this.gradient = gradient;
    }

    public boolean isTrainable() {
        return this.trainable;
    }

    public void zeroGradient() {
        this.setGradient(0);
    }

    public int getLayer() {
        return layer;
    }

    public void setLayer(int layer) {
        this.layer = layer;
    }
}
```

Listing 10: autograd/Parameter.java

```java
package autograd;

import jdk.jshell.spi.ExecutionControl;

public class Sigmoid extends Operator {

    public Sigmoid() {
        this.numberOfOperands = 1;
    }

    @Override
    public double evaluate(IVariable[] operands) {
        if (operands.length != 1) {
            throw new IllegalArgumentException("Sigmoid operator only accepts one operand");
        }
        return 1. / (1 + Math.exp(-operands[0].evaluate()));
    }

    @Override
    public void backwards(IVariable[] operands, IVariable[] sources, double gradient) throws ExecutionControl.NotImplementedException {
        validateOperands(operands);
        var x = operands[0];
        var y = evaluate(operands);
        x.backward(sources, gradient * y * (1 - y));
    }
}
```

Listing 11: autograd/Sigmoid.java

```java
package autograd;

import java.util.Random;

public class UniformInitializer implements IInitializer {

    double a;
    double b;
    Random random;

    public UniformInitializer(double a, double b) {
        this.a = a;
        this.b = b;
        this.random = new Random();
    }

    @Override
    public double next() {
        return random.nextDouble() * (b - a) + a;
    }
}
```

Listing 12: autograd/UniformInitializer.java

```java
package dataset;

public class BinaryToBipolarWrapper implements IDataSet {

    IDataSet binaryDataSet;

    public BinaryToBipolarWrapper(IDataSet binaryDataSet) {
        this.binaryDataSet = binaryDataSet;
    }

    @Override
    public DataPoint next() {
        DataPoint result = binaryDataSet.next();
        if (result == null) return null;
        double[] x = result.getX().clone();
        double[] y = result.getY().clone();
        for (int i = 0; i < x.length; i++) {
            x[i] = 2 * x[i] - 1;
        }
        for (int i = 0; i < y.length; i++) {
            y[i] = 2 * y[i] - 1;
        }
        return new DataPoint(x, y);
    }

    @Override
    public void reset() {
        binaryDataSet.reset();
    }
}
```

Listing 13: dataset/BinaryToBipolarWrapper.java

```java
package dataset;

public class DataPoint {
    private final double[] x;
    private final double[] y;

    public DataPoint(double[] x, double[] y) {
        this.x = x;
        this.y = y;
    }

    public double[] getY() {
        return y;
    }

    public double[] getX() {
        return x;
    }
}
```

Listing 14: dataset/DataPoint.java

```java
package dataset;

public interface IDataSet {
    DataPoint next();

    void reset();
}
```

Listing 15: dataset/IDataSet.java

```java
package dataset;

public class XORBinaryDataSet implements IDataSet {

    protected double[][] x;
    protected double[] y;
    private int index;

    public XORBinaryDataSet() {
        index = 0;
        x = new double[][]{
                {0., 0.},
                {0., 1.},
                {1., 0.},
                {1., 1.},
        };
        y = new double[]{
                0.,
                1.,
                1.,
                0.,
        };
    }

    @Override
    public DataPoint next() {
        if (index < x.length) {
            var result = new DataPoint(x[index], new double[]{y[index]});
            index++;
            return result;
        }
        return null;
    }

    @Override
    public void reset() {
        index = 0;
    }
}
```

Listing 16: dataset/XORBinaryDataSet.java

```java
package nn;

import autograd.IVariable;
import autograd.Parameter;

public class BipolarSigmoid implements ILayer {

    @Override
    public IVariable[] apply(IVariable[] input) {
        var sigmoid = new autograd.Sigmoid();
        var scalar = new Parameter(2, false);
        var constant = new Parameter(-1, false);
        var addition = new autograd.Addition();
        var multiplication = new autograd.Multiplication();
        var result = new IVariable[input.length];
        for (int i = 0; i < input.length; i++) {
            result[i] = addition.apply(
                    multiplication.apply(
                            scalar,
                            sigmoid.apply(input[i])),
                    constant
            );
        }
        return result;
    }
}
```

Listing 17: nn/BipolarSigmoid.java

```java
package nn;

import java.util.ArrayList;

public class ConvergenceCollector implements IFitCallback {
    ArrayList<Double> loss;

    public ConvergenceCollector() {
        this.loss = new ArrayList<>();
    }

    @Override
    public void collect(int epoch, double loss) {
        this.loss.add(loss);
    }

    public int getEpochs() {
        return loss.size();
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < loss.size(); i++) {
            sb.append(+i + " " + loss.get(i) + "\n");
        }
        return sb.toString();
    }
}
```

Listing 18: nn/ConvergenceCollector.java

```java
package nn;

import autograd.IInitializer;
import autograd.IVariable;
import autograd.Parameter;
import autograd.UniformInitializer;
```

```java

public class Factory {
    public static Model createNeuralNetwork(int[] sizes, ILayer activation, IInitializer initializer) {
        if (sizes.length < 2) {
            throw new IllegalArgumentException("Sizes must at least contain 2 integers for the first and the second layer.");
        }
        var inputs = new Parameter[sizes[0]];
        for (int i = 0; i < inputs.length; i++) {
            inputs[i] = new Parameter(initializer.next());
        }
        IVariable[] lastLayerOutput = inputs;
        for (int i = 1; i < sizes.length; i++) {
            lastLayerOutput = new Linear(sizes[i - 1], sizes[i], initializer).apply(lastLayerOutput);
            lastLayerOutput = activation.apply(lastLayerOutput);
        }
        return new Model(inputs, lastLayerOutput);
    }

    public static Model createNeuralNetwork(int[] sizes, ILayer activation) {
        return createNeuralNetwork(sizes, activation, new UniformInitializer(-0.5, 0.5));
    }
}
```

Listing 19: nn/Factory.java

```java
package nn;

public interface IFitCallback {
    void collect(int epoch, double loss);
}
```

Listing 20: nn/IFitCallback.java

```java
package nn;

import autograd.IVariable;

public interface ILayer {
    IVariable[] apply(IVariable[] input);
}
```

Listing 21: nn/ILayer.java

```java
package nn;

import autograd.*;

public class Linear implements ILayer {
    private final IVariable[][] weight;
    private final IVariable[] bias;

    public Linear(int inFeatures, int outFeatures, IInitializer initializer) {
        this.weight = new Parameter[outFeatures][inFeatures];
        this.bias = new Parameter[outFeatures];
        for (int i = 0; i < outFeatures; i++) {
            for (int j = 0; j < inFeatures; j++) {
                this.weight[i][j] = new Parameter(initializer.next());
            }
            this.bias[i] = new Parameter(initializer.next());
        }
    }

    @Override
    public IVariable[] apply(IVariable[] input) {
        var result = new IVariable[this.weight.length];
        for (int i = 0; i < this.weight.length; i++) {
            int inputSize = this.weight[i].length;
            IVariable[] muls = new IVariable[inputSize + 1];
            for (int j = 0; j < inputSize; j++) {
                muls[j] = new Multiplication().apply(this.weight[i][j],
                    input[j]);
            }
            muls[inputSize] = this.bias[i];
            result[i] = new Addition().apply(muls);
        }
        return result;
    }


    private int getWidth() {
        return this.weight.length;
    }
}
```

Listing 22: nn/Linear.java

```java
package nn;

import autograd.*;
import jdk.jshell.spi.ExecutionControl;
import optimization.ILoss;

public class MinimumSquaredError implements IVariable, ILoss {

    private final IVariable operation;
    private final Parameter[] desired;

    public MinimumSquaredError(IVariable[] output) {
        var negation = new Negation();
        var addition = new Addition();
        var multiplication = new Multiplication();
        var exponentiation = new Exponentiation();
        Parameter two = new Parameter(2, false);
        Parameter half = new Parameter(0.5, false);
        int length = output.length;
        desired = new Parameter[output.length];
        var summationTerms = new IVariable[length];
        for (int i = 0; i < length; i++) {
            desired[i] = new Parameter();
            summationTerms[i] = exponentiation.apply(
                    addition.apply(output[i], negation.apply(desired[i])),
                    two
            );
        }
        this.operation = multiplication.apply(addition.apply(summationTerms), half);
    }

    @Override
    public double evaluate() {
        return operation.evaluate();
    }

    @Override
    public void backward(IVariable[] sources, double gradient) throws ExecutionControl.NotImplementedException {
        operation.backward(sources, gradient);
    }

    @Override
    public Parameter[] getParameters() {
        return this.operation.getParameters();
    }

    @Override
    public void setDesired(double[] desired) {
        for (int i = 0; i < this.desired.length; i++) {
            this.desired[i].setValue(desired[i]);
        }
    }
}
```

Listing 23: nn/MinimumSquaredError.java

```
package nn;

import autograd.IVariable;
import autograd.Operation;
import autograd.Parameter;
import dataset.DataPoint;
import dataset.IDataSet;
import jdk.jshell.spi.ExecutionControl;
import optimization.ILoss;
import optimization.IOptimizer;

import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class Model {
    private final Parameter[] input;
    private final IVariable[] output;

    public Model(Parameter[] input, IVariable[] output) {
        this.input = input;
        this.output = output;
```

```java
    }


    public double[] evaluate(double[] input) {
        var result = new double[output.length];
        for (int i = 0; i < input.length; i++) {
            this.input[i].setValue(input[i]);
        }
        for (int i = 0; i < output.length; i++) {
            result[i] = output[i].evaluate();
        }
        return result;
    }

    public Parameter[] getParameters() {
        HashSet<Parameter> result = new HashSet<>();
        for (IVariable o :
                this.output) {
            result.addAll(Arrays.asList(o.getParameters()));
        }
        return result.toArray(new Parameter[0]);
    }

    public Parameter[] getTrainableParameters() {
        var results = new HashSet<Parameter>();
        for (Parameter p :
                getParameters()) {
            if (p.isTrainable()) {
                results.add(p);
            }
        }
        for (Parameter p : input) {
            results.remove(p);
        }

        return results.toArray(new Parameter[0]);
    }

    public IVariable[] getOutput() {
        return output;
    }

    public double fit(IDataSet dataSet, IOptimizer optimizer, ILoss loss, int epochs, double lossLimit) throws ExecutionControl.NotImplementedException {
        return fit(dataSet, optimizer, loss, epochs, lossLimit, (epoch, l) -> {
        });
    }

    public double fit(IDataSet dataSet, IOptimizer optimizer, ILoss loss, int epochs, double lossLimit, IFitCallback callback) throws ExecutionControl.NotImplementedException {
        var parameters = getTrainableParameters();
        Map<Integer, List<Parameter>> layeredParameters = layerParameters(parameters);
        if (epochs < 1) {
            throw new IllegalArgumentException("At least one epochs
                    required.");
        }
        double totalLoss = 0;
        for (int i = 0; i < epochs; i++) {
            totalLoss = 0;
            dataSet.reset();
            DataPoint dataPoint;
            while ((dataPoint = dataSet.next()) != null) {
                setInput(dataPoint.getX());
                loss.setDesired(dataPoint.getY());
                totalLoss += loss.evaluate();
                for (Integer j : layeredParameters.keySet().stream().sorted
                    ().collect(Collectors.toList())) {
                    Parameter[] layerParameters = layeredParameters.get(j).
                        toArray(new Parameter[0]);
                    loss.backward(layerParameters, 1.);
                    optimizer.update(layerParameters);
                }
            }
            callback.collect(i, totalLoss);
            if (totalLoss < lossLimit) {
                break;
            }
        }
        return totalLoss;
    }

    private Map<Integer, List<Parameter>> layerParameters(Parameter[]
        parameters) {
        setLayers(getOutput(), 0);
        return Arrays.stream(parameters).collect(Collectors.groupingBy(
            Parameter::getLayer));

    }

    private void setLayers(IVariable[] outputs, int layer) {
        if (outputs.length == 0) return;
        HashSet<IVariable> nextOutput = new HashSet<>();
        for (IVariable i : outputs) {
            if (i instanceof Parameter) {
                ((Parameter) i).setLayer(layer);
            }
            if (i instanceof Operation) {
                nextOutput.addAll(Arrays.asList(((Operation) i).getOperands
                    ()));
            }
        }
        setLayers(nextOutput.toArray(new IVariable[0]), layer + 1);
    }

    private void setInput(double[] x) {
        for (int i = 0; i < input.length; i++) {
            input[i].setValue(x[i]);
        }
    }
}
```

Listing 24: nn/Model.java

```java
package nn;

import autograd.IVariable;

public class Sigmoid implements ILayer {

    @Override
    public IVariable[] apply(IVariable[] input) {
        var operator = new autograd.Sigmoid();
        var result = new IVariable[input.length];
        for (int i = 0; i < input.length; i++) {
            result[i] = operator.apply(input[i]);
        }
        return result;
    }
}
```

Listing 25: nn/Sigmoid.java

```java
package optimization;

import autograd.Parameter;

import java.util.HashMap;

public class GradientDescent implements IOptimizer {

    private final HashMap<Parameter, Double> lastDelta;
    private final double learningRate;
    private final double momentum;

    public GradientDescent(double learningRate, double momentum) {
        this.lastDelta = new HashMap<>();
        this.learningRate = learningRate;
        this.momentum = momentum;
    }

    @Override
    public void update(Parameter[] parameters) {
        for (Parameter p :
                parameters) {
            double delta = -p.getGradient() * learningRate + momentum *
                lastDelta.getOrDefault(p, 0.);
            p.setValue(p.getValue() + delta);
            p.zeroGradient();
            lastDelta.put(p, delta);
        }
    }
}
```

Listing 26: optimization/GradientDescent.java

```java
package optimization;

import autograd.IVariable;

public interface ILoss extends IVariable {
    void setDesired(double[] desired);
}
```

Listing 27: optimization/ILoss.java

```
package optimization;

import autograd.Parameter;

public interface IOptimizer {
    void update(Parameter[] parameters);
}
```

Listing 28: optimization/IOptimizer.java

```
package autograd;


import org.junit.Assert;
import org.junit.Test;

public class VariableTest {

    @Test
    public void testAddition() {
        Assert.assertEquals(new Addition().apply(new Parameter(12), new Parameter(2.)).evaluate(), 14., 0.);
    }

    @Test
    public void testVariableEvaluation() {
        Assert.assertEquals(new Parameter(250).evaluate(), 250., 0);
    }

}
```

Listing 29: autograd/VariableTest.java

```
package nn;

import autograd.Parameter;
import jdk.jshell.spi.ExecutionControl;
import org.junit.Assert;
import org.junit.Test;

import java.util.Arrays;

public class NeuralNetworkTest {

    @Test
    public void testNeuralNetworkFactory() {
        var model = Factory.createNeuralNetwork(new int[]{2, 4, 1}, new Sigmoid());
        var result = model.evaluate(new double[]{0, 0});
        Assert.assertEquals(result.length, 1);
    }

    @Test
    public void testNeuralNetworkGradient() throws ExecutionControl.NotImplementedException {
```

```java
        var model = Factory.createNeuralNetwork(new int[]{2, 4, 1}, new Sigmoid());
        Parameter[] parameters = model.getTrainableParameters();
        for (Parameter parameter :
                parameters) {
            parameter.setValue(1);
        }
        var result = model.evaluate(new double[]{1, 0});
        double[] desired = new double[]{1};
        Assert.assertEquals(result.length, 1);
        double delta = 1e-5;
        double expected = 0.9892621636390686; // obtained by pytorch
        Assert.assertEquals(result[0], expected, delta);
        var loss = new MinimumSquaredError(model.getOutput());
        loss.setDesired(desired);
        loss.backward(parameters, 1);
        var gradients = new double[parameters.length];
        for (int i = 0; i < parameters.length; i++) {
            gradients[i] = parameters[i].getGradient();
        }

        Assert.assertArrayEquals(Arrays.stream(new double[]{ // calculated by pytorch
                -0.000114063048386015, -0.00010046639363281429,
                    -0.00010046639363281429, -0.00010046639363281429,
                    -0.00010046639363281429, -1.197589335788507e-05,
                    -1.197589335788507e-05, -1.197589335788507e-05,
                    -1.197589335788507e-05, -1.197589335788507e-05,
                    -1.197589335788507e-05, -1.197589335788507e-05,
                    -1.197589335788507e-05, -0.0, -0.0, -0.0, -0.0
        }).sorted().toArray(), Arrays.stream(gradients).sorted().toArray(), delta);
    }

    @Test
    public void testNeuralNetworkGradientBipolar() throws ExecutionControl.NotImplementedException {
        var model = Factory.createNeuralNetwork(new int[]{2, 4, 1}, new BipolarSigmoid());
        Parameter[] parameters = model.getTrainableParameters();
        for (Parameter parameter :
                parameters) {
            parameter.setValue(1);
        }
        var result = model.evaluate(new double[]{1, -1});
        double[] desired = new double[]{1};
        Assert.assertEquals(result.length, 1);
        double delta = 1e-5;
        double expected = 0.8904789686203003; // obtained by pytorch
        Assert.assertEquals(expected, result[0], delta);
        var loss = new MinimumSquaredError(model.getOutput());
        loss.setDesired(desired);
        loss.backward(parameters, 1);
        var gradients = new double[parameters.length];
        for (int i = 0; i < parameters.length; i++) {
            gradients[i] = parameters[i].getGradient();
        }

        Assert.assertArrayEquals(Arrays.stream(new double[]{ // calculated by pytorch
                -0.011338012292981148, -0.005239490419626236, -0.005239490419626236, -0.005239490419626236, -0.005239490419626236, -0.004458376672118902, -0.004458376672118902, -0.004458376672118902, -0.004458376672118902, -0.004458376672118902, -0.004458376672118902, -0.004458376672118902, -0.004458376672118902, 0.004458376672118902, 0.004458376672118902, 0.004458376672118902, 0.004458376672118902
        }).sorted().toArray(), Arrays.stream(gradients).sorted().toArray(), delta);
    }
}
```

Listing 30: nn/NeuralNetworkTest.java

```java
package optimization;

import autograd.UniformInitializer;
import dataset.BinaryToBipolarWrapper;
import dataset.XORBinaryDataSet;
import jdk.jshell.spi.ExecutionControl;
import nn.*;
import org.junit.Assert;
import org.junit.Ignore;
import org.junit.Test;

import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.Optional;

public class GradientDescentTest {

    private final static int trials = 300;

    @Ignore
    @Test
    public void TestSimpleGD() throws ExecutionControl.NotImplementedException {
        var model = Factory.createNeuralNetwork(
                new int[]{2, 4, 1},
                new Sigmoid(),
                new UniformInitializer(-0.5, 0.5)
        );
        var dataSet = new XORBinaryDataSet();
        var optimizer = new GradientDescent(0.2, 0.);
        var loss = new MinimumSquaredError(model.getOutput());
        double finalLoss = model.fit(dataSet, optimizer, loss, 40000, 0.05);
        Assert.assertTrue("Big loss " + finalLoss, finalLoss < 0.05);
    }

    @Ignore("Skipping slow convergence tests.")
    @Test
    public void TestConvergence() throws ExecutionControl.NotImplementedException, IOException {
        int diverged = 0;
        ArrayList<ConvergenceCollector> stats = new ArrayList<>();
        for (int i = 0; i < GradientDescentTest.trials; i++) {
            var model = Factory.createNeuralNetwork(
                    new int[]{2, 4, 1},
                    new Sigmoid(),
                    new UniformInitializer(-0.5, 0.5)
            );
            var dataSet = new XORBinaryDataSet();
            var optimizer = new GradientDescent(0.2, 0.);
            var loss = new MinimumSquaredError(model.getOutput());
            var collector = new ConvergenceCollector();
            double finalLoss = model.fit(dataSet, optimizer, loss, 40000, 0.05, collector);
            stats.add(collector);
            if (finalLoss > 0.05) {
                diverged += 1;
            }
        }
        outputGraphData("a", stats);
        Assert.assertTrue("Convergence with high probability busted!", diverged < 6);
    }

    private void outputGraphData(String assignmentPart, ArrayList<ConvergenceCollector> stats) throws IOException {
        FileWriter of = new FileWriter("doc/" + assignmentPart + "_avg.tex");
        double average = stats.stream().mapToInt(ConvergenceCollector::getEpochs).average().getAsDouble();
        of.write(String.valueOf(average));
        of.close();

        Optional<ConvergenceCollector> representative = stats.stream().min(Comparator.comparingDouble(c -> Math.abs(c.getEpochs() - average)));
        of = new FileWriter("doc/" + assignmentPart + ".tex");
        of.write(representative.get().toString());
        of.close();
    }

    @Ignore("Skipping slow convergence tests.")
    @Test
    public void TestBipolarGD() throws ExecutionControl.NotImplementedException, IOException {
        int diverged = 0;
        int trials = GradientDescentTest.trials;
        ArrayList<ConvergenceCollector> stats = new ArrayList<>();
        for (int i = 0; i < trials; i++) {
            var model = Factory.createNeuralNetwork(
                    new int[]{2, 4, 1},
                    new BipolarSigmoid(),
                    new UniformInitializer(-0.5, 0.5)
            );
            var dataSet = new BinaryToBipolarWrapper(new XORBinaryDataSet());
```

```java
            var optimizer = new GradientDescent(0.2, 0.);
            var loss = new MinimumSquaredError(model.getOutput());
            var collector = new ConvergenceCollector();
            double finalLoss = model.fit(dataSet, optimizer, loss, 3500, 0.05, collector);
            if (finalLoss > 0.05) {
                diverged += 1;
            }
            stats.add(collector);
        }
        outputGraphData("b", stats);
        Assert.assertTrue("Convergence with high probability busted! " + diverged + " failure out of " + trials, diverged < 6);
    }

    @Ignore
    @Test
    public void TestBipolarMomentumGD() throws ExecutionControl.NotImplementedException, IOException {

        int diverged = 0;
        int trials = GradientDescentTest.trials;
        ArrayList<ConvergenceCollector> stats = new ArrayList<>();
        for (int i = 0; i < trials; i++) {
            var model = Factory.createNeuralNetwork(
                    new int[]{2, 4, 1},
                    new BipolarSigmoid(),
                    new UniformInitializer(-0.5, 0.5)
            );
            var dataSet = new BinaryToBipolarWrapper(new XORBinaryDataSet());
            var optimizer = new GradientDescent(0.2, 0.9);
            var loss = new MinimumSquaredError(model.getOutput());
            var collector = new ConvergenceCollector();
            double finalLoss = model.fit(dataSet, optimizer, loss, 1000, 0.05, collector);
            if (finalLoss > 0.05) {
                diverged += 1;
            }
            stats.add(collector);
        }
        outputGraphData("c", stats);
        Assert.assertTrue("Convergence with high probability busted! " + diverged + " failure out of " + trials, diverged < 6);
    }
}
```

Listing 31: optimization/GradientDescentTest.java