

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

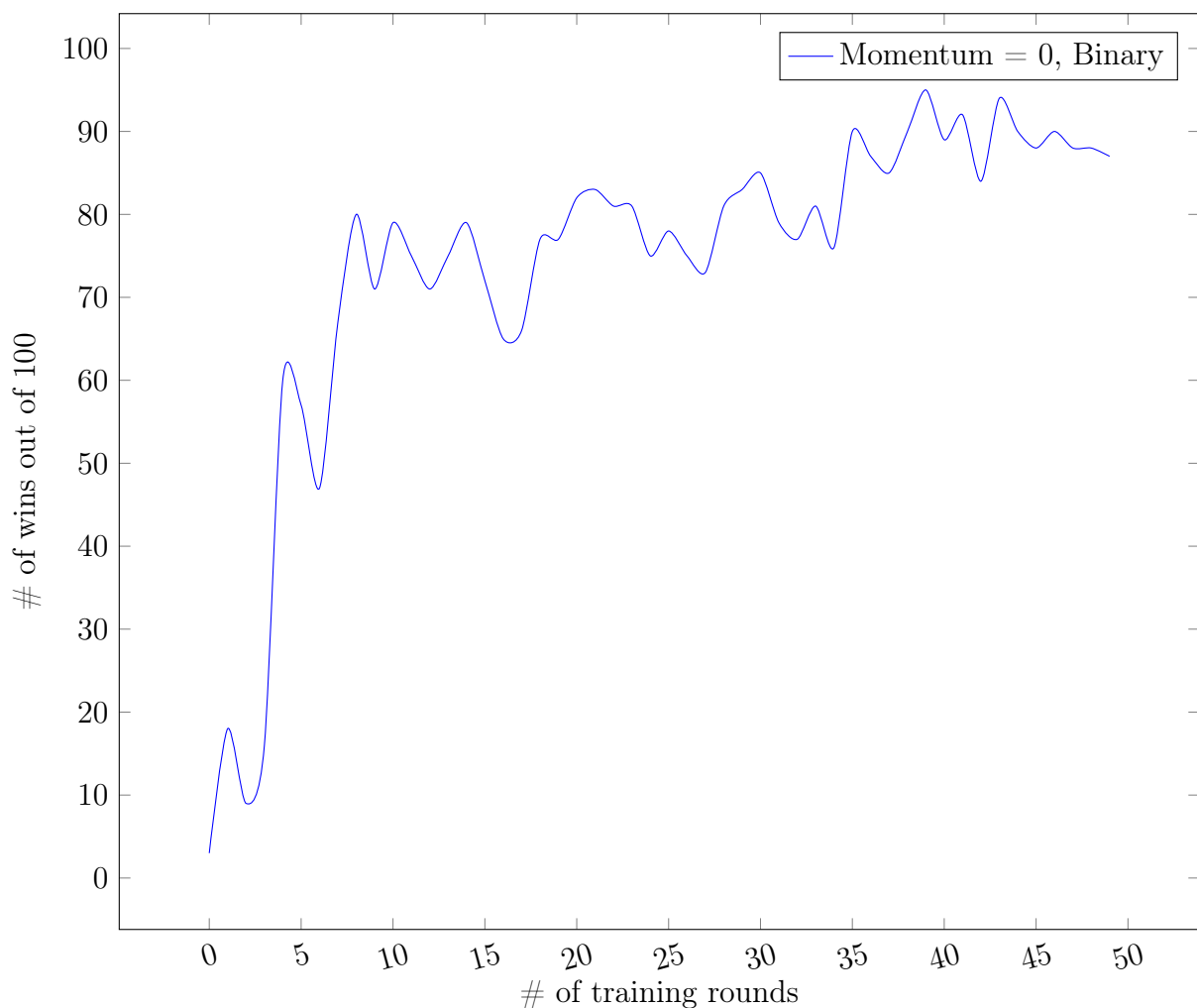
Ali Asgari Khoshouyeh (Student #24868739)

24. November 2021

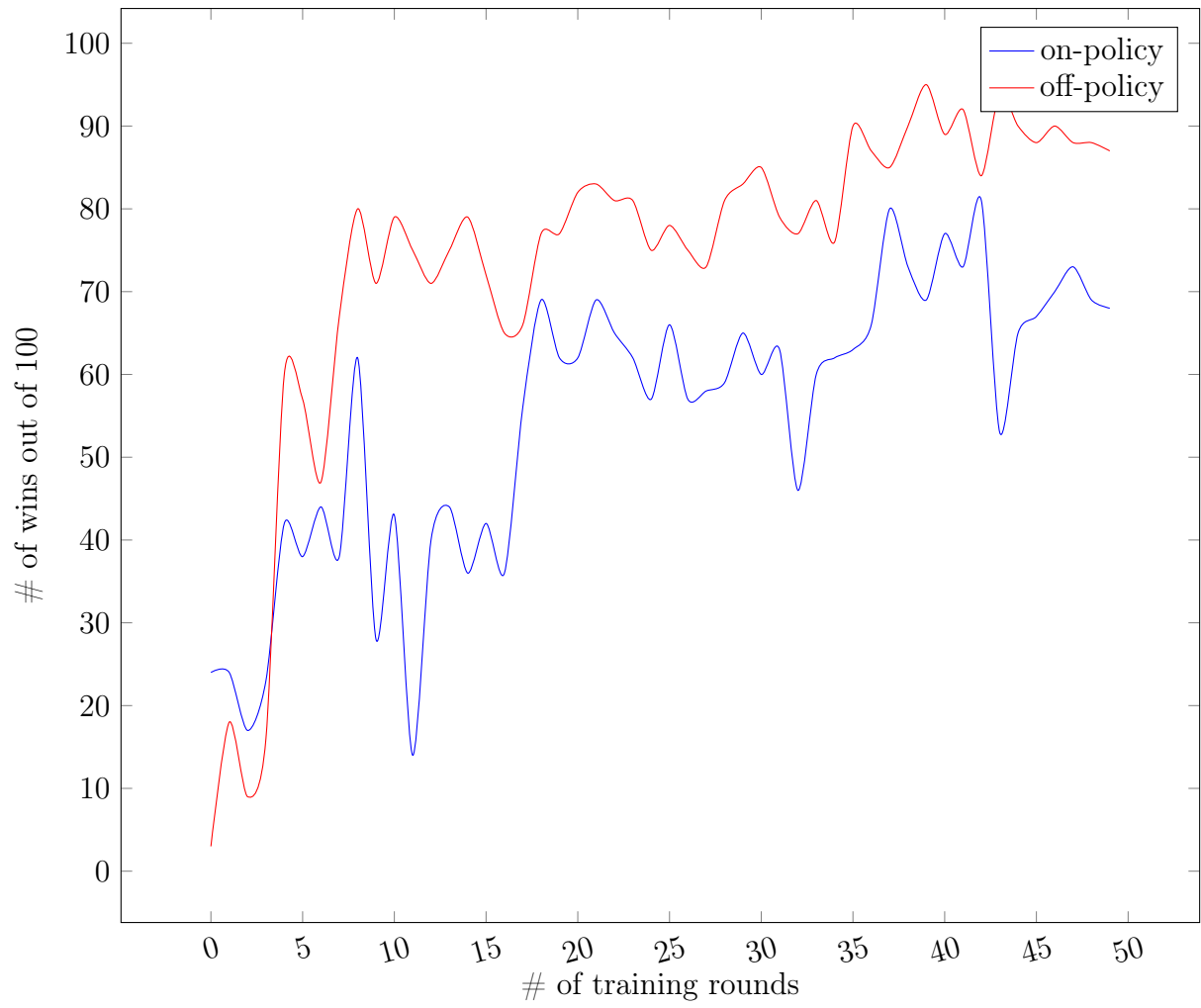
1 Q Learning Robot

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

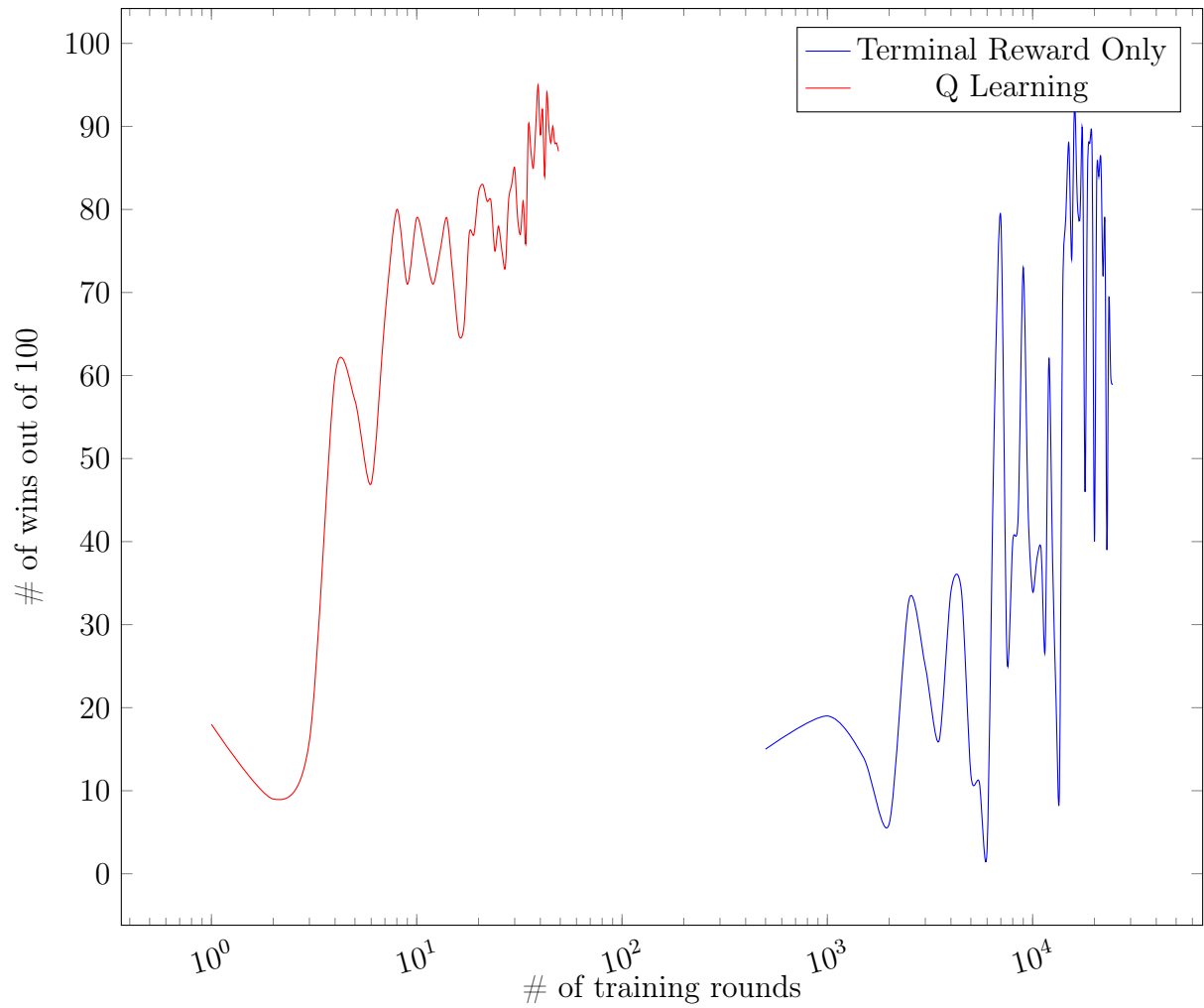
a) Draw a graph of a parameter that reflects a measure of progress of learning and comment on the convergence of learning of your robot.



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



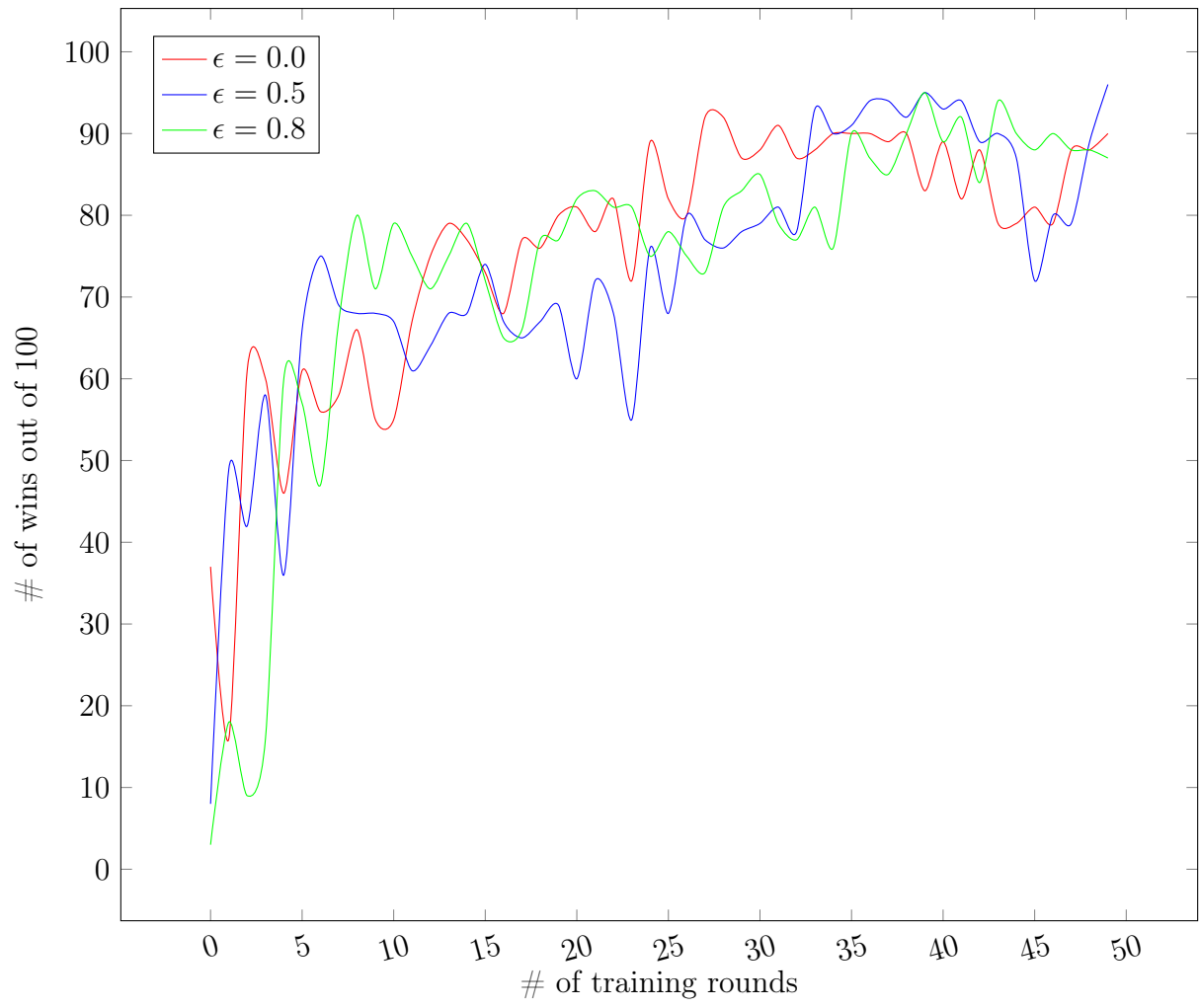
c) Implement a version of your robot that assumes only terminal rewards and show & compare its behaviour with one having intermediate rewards.

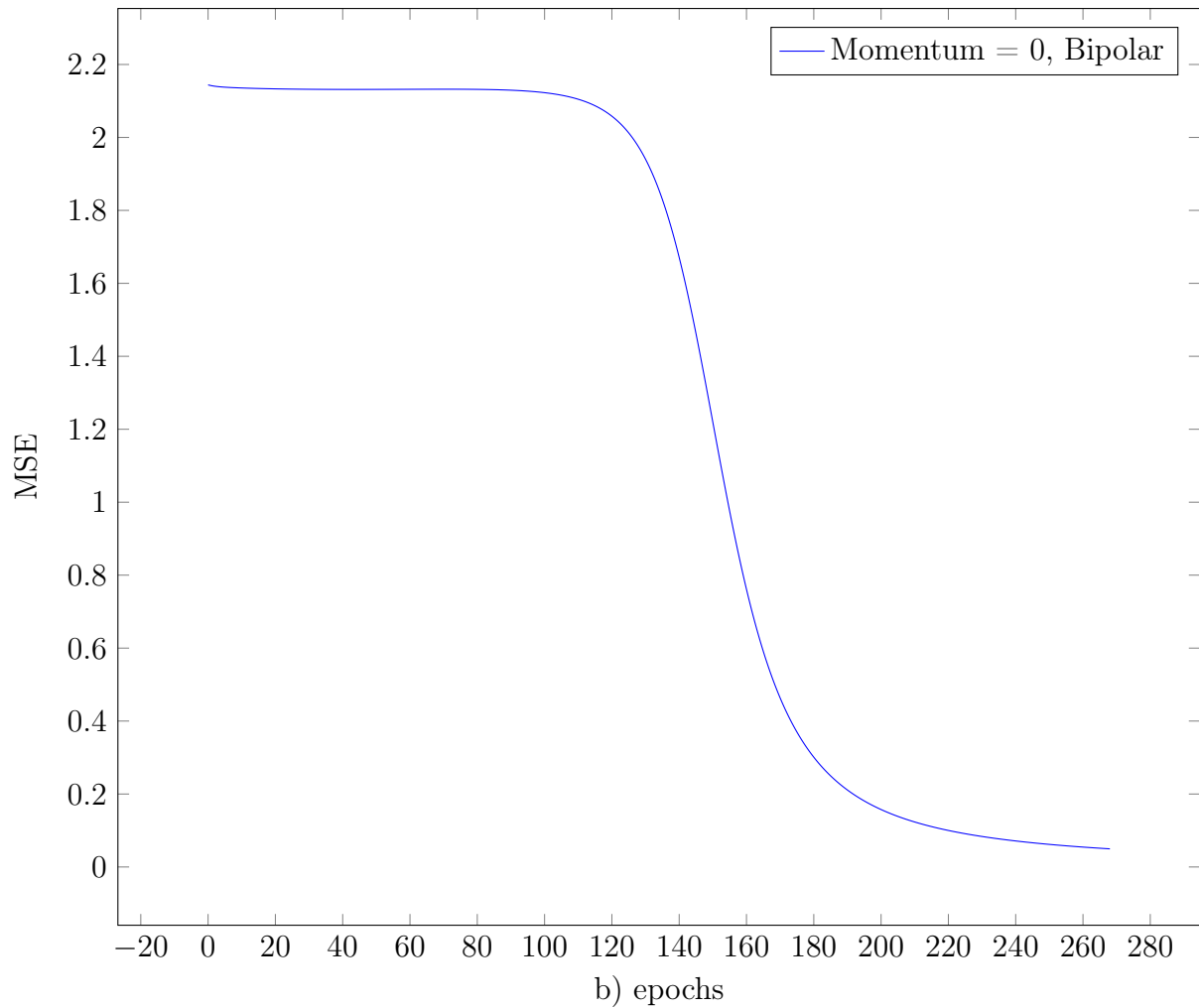


2 Role of ϵ

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

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





Appendices

A Source Codes

```

1 package autograd;
2
3 import jdk.jshell.spi.ExecutionControl;
4
5 public class Addition extends Operator {
6     @Override
7     public double evaluate(IVariable[] operands) {
8         double result = 0.;
9         for (IVariable operand :
10             operands) {
11             result += operand.evaluate();
12         }
13         return result;
14     }
15
16     @Override

```

```

17     public void backwards(IVariable[] operands, IVariable[] sources, double
18         gradient) throws ExecutionControl.NotImplementedException {
19         for (IVariable o :
20             operands) {
21             o.backward(sources, gradient);
22         }
23     }

```

Listing 1: autograd/Addition.java

```

1 package autograd;
2
3 import jdk.jshell.spi.ExecutionControl;
4
5 public class Exponentiation extends Operator {
6     @Override
7     public double evaluate(IVariable[] operands) {
8         if (operands.length != 2) {
9             throw new IllegalArgumentException("Exponentiation accepts 2
10                 arguments.");
11         }
12         return Math.pow(operands[0].evaluate(), operands[1].evaluate());
13     }
14
15     @Override
16     public void backwards(IVariable[] operands, IVariable[] sources, double
17         gradient) throws ExecutionControl.NotImplementedException {
18         IVariable baseVariable = operands[0];
19         var baseValue = baseVariable.evaluate();
20         IVariable exponentVariable = operands[1];
21         var exponentValue = exponentVariable.evaluate();
22         if (exponentVariable.getParameters().length > 1) {
23             throw new ExecutionControl.NotImplementedException("Back
24                 propagation to the exponent is not implemented.");
25         }
26         var gradientToPropagate = Math.pow(gradient * baseValue *
27             exponentValue, exponentValue - 1);
28         baseVariable.backward(sources, gradientToPropagate);
29     }
30 }

```

Listing 2: autograd/Exponentiation.java

```

1 package autograd;
2
3 public interface IInitializer {
4     double next();
5 }

```

Listing 3: autograd/IInitializer.java

```

1 package autograd;
2
3
4 import jdk.jshell.spi.ExecutionControl;
5
6 public interface IOperator {

```

```

7   IVariable apply(IVariable... operands);
8
9   double evaluate(IVariable[] operands);
10
11  void backwards(IVariable[] operands, IVariable[] sources, double
      gradient) throws ExecutionControl.NotImplementedException;
12 }

```

Listing 4: autograd/IOperator.java

```

1 package autograd;
2
3 import jdk.jshell.spi.ExecutionControl;
4
5 public interface IVariable {
6     double evaluate();
7
8     void backward(IVariable[] sources, double gradient) throws
        ExecutionControl.NotImplementedException;
9
10    Parameter[] getParameters();
11 }

```

Listing 5: autograd/IVariable.java

```

1 package autograd;
2
3 import jdk.jshell.spi.ExecutionControl;
4
5 public class Multiplication extends Operator {
6
7     @Override
8     public double evaluate(IVariable[] operands) {
9         double result = 1.;
10        for (IVariable operand :
11            operands) {
12            result *= operand.evaluate();
13        }
14        return result;
15    }
16
17    @Override
18    public void backwards(IVariable[] operands, IVariable[] sources, double
        gradient) throws ExecutionControl.NotImplementedException {
19        validateOperands(operands);
20        var multiplier = operands[0];
21        var multiplicand = operands[1];
22        var multiplierValue = multiplier.evaluate();
23        var multiplicandValue = multiplicand.evaluate();
24        multiplier.backward(sources, gradient * multiplicandValue);
25        multiplicand.backward(sources, gradient * multiplierValue);
26    }
27 }

```

Listing 6: autograd/Multiplication.java

```

1 package autograd;
2

```

```

3 import jdk.jshell.spi.ExecutionControl;
4
5 public class Negation extends Operator {
6
7     public Negation() {
8         this.numberOfOperands = 1;
9     }
10
11     @Override
12     public double evaluate(IVariable[] operands) {
13         validateOperands(operands);
14         return -operands[0].evaluate();
15     }
16
17     @Override
18     public void backwards(IVariable[] operands, IVariable[] sources, double
19         gradient) throws ExecutionControl.NotImplementedException {
20         operands[0].backward(sources, -gradient);
21     }
22 }

```

Listing 7: autograd/Negation.java

```

1 package autograd;
2
3 import jdk.jshell.spi.ExecutionControl;
4
5 import java.util.Arrays;
6 import java.util.HashSet;
7
8 public class Operation implements IVariable {
9     private final IOperator operator;
10    private final IVariable[] operands;
11
12    public Operation(IOperator operator, IVariable... operands) {
13        this.operator = operator;
14        this.operands = operands;
15    }
16
17    @Override
18    public double evaluate() {
19        return operator.evaluate(operands);
20    }
21
22    @Override
23    public void backward(IVariable[] sources, double gradient) throws
24        ExecutionControl.NotImplementedException {
25        operator.backwards(operands, sources, gradient);
26    }
27
28    @Override
29    public Parameter[] getParameters() {
30        HashSet<Parameter> result = new HashSet<>();
31        for (IVariable o :
32            this.operands) {
33            result.addAll(Arrays.asList(o.getParameters()));
34        }
35    }
36 }

```



```

35         return result.toArray(new Parameter[0]);
36     }
37
38     public IVariable[] getOperands() {
39         return operands;
40     }
41 }

```

Listing 8: autograd/Operation.java

```

1 package autograd;
2
3 public abstract class Operator implements IOperator {
4     protected Integer numberOfOperands;
5
6     public Operator() {
7         this.numberOfOperands = null;
8     }
9
10    @Override
11    public IVariable apply(IVariable... operands) {
12        return new Operation(this, operands);
13    }
14
15    protected void validateOperands(IVariable[] operands) {
16        if (this.numberOfOperands == null) {
17            return;
18        }
19        if (operands.length != this.numberOfOperands) {
20            throw new IllegalArgumentException(String.format("%s accepts
21                only one operand.", this.getClass().getName()));
22        }
23    }
24 }

```

Listing 9: autograd/Operator.java

```

1 package autograd;
2
3 import java.util.Arrays;
4
5 public class Parameter implements IVariable {
6     private double value;
7     private double gradient;
8     private boolean trainable;
9     private int layer;
10
11     public Parameter() {
12
13     }
14
15     public Parameter(double value) {
16         this.value = value;
17         trainable = true;
18     }
19
20     public Parameter(double value, boolean trainable) {
21         this.value = value;

```

```

22         this.trainable = trainable;
23     }
24
25     public static IVariable[] createTensor(double[] desired) {
26         var result = new Parameter[desired.length];
27         for (int i = 0; i < result.length; i++) {
28             result[i] = new Parameter(desired[i]);
29         }
30         return result;
31     }
32
33     @Override
34     public double evaluate() {
35         return value;
36     }
37
38     @Override
39     public void backward(IVariable[] sources, double gradient) {
40         if (Arrays.stream(sources).anyMatch(x -> x == this)) {
41             setGradient(gradient + getGradient());
42         }
43     }
44
45     @Override
46     public Parameter[] getParameters() {
47         return new Parameter[]{ this };
48     }
49
50     public double getValue() {
51         return this.value;
52     }
53
54     public void setValue(double value) {
55         this.value = value;
56     }
57
58     public double getGradient() {
59         return gradient;
60     }
61
62     private void setGradient(double gradient) {
63         this.gradient = gradient;
64     }
65
66     public boolean isTrainable() {
67         return this.trainable;
68     }
69
70     public void zeroGradient() {
71         this.setGradient(0);
72     }
73
74     public int getLayer() {
75         return layer;
76     }
77
78     public void setLayer(int layer) {
79         this.layer = layer;

```

```

80     }
81 }

```

Listing 10: autograd/Parameter.java

```

1 package autograd;
2
3 import jdk.jshell.spi.ExecutionControl;
4
5 public class Sigmoid extends Operator {
6
7     public Sigmoid() {
8         this.numberOfOperands = 1;
9     }
10
11     @Override
12     public double evaluate(IVariable[] operands) {
13         if (operands.length != 1) {
14             throw new IllegalArgumentException("Sigmoid operator only
15                 accepts one operand");
16         }
17         return 1. / (1 + Math.exp(-operands[0].evaluate()));
18     }
19
20     @Override
21     public void backwards(IVariable[] operands, IVariable[] sources, double
22         gradient) throws ExecutionControl.NotImplementedException {
23         validateOperands(operands);
24         var x = operands[0];
25         var y = evaluate(operands);
26         x.backward(sources, gradient * y * (1 - y));
27     }
28 }

```

Listing 11: autograd/Sigmoid.java

```

1 package autograd;
2
3 import java.util.Random;
4
5 public class UniformInitializer implements IInitializer {
6
7     double a;
8     double b;
9     Random random;
10
11     public UniformInitializer(double a, double b) {
12         this.a = a;
13         this.b = b;
14         this.random = new Random();
15     }
16
17     @Override
18     public double next() {
19         return random.nextDouble() * (b - a) + a;
20     }
21 }

```

Listing 12: autograd/UniformInitializer.java

```

1 package dataset;
2
3 public class BinaryToBipolarWrapper implements IDataset {
4
5     IDataset binaryDataSet;
6
7     public BinaryToBipolarWrapper(IDataset binaryDataSet) {
8         this.binaryDataSet = binaryDataSet;
9     }
10
11     @Override
12     public DataPoint next() {
13         DataPoint result = binaryDataSet.next();
14         if (result == null) return null;
15         double[] x = result.getX().clone();
16         double[] y = result.getY().clone();
17         for (int i = 0; i < x.length; i++) {
18             x[i] = 2 * x[i] - 1;
19         }
20         for (int i = 0; i < y.length; i++) {
21             y[i] = 2 * y[i] - 1;
22         }
23         return new DataPoint(x, y);
24     }
25
26     @Override
27     public void reset() {
28         binaryDataSet.reset();
29     }
30 }

```

Listing 13: dataset/BinaryToBipolarWrapper.java

```

1 package dataset;
2
3 public class DataPoint {
4     private final double[] x;
5     private final double[] y;
6
7     public DataPoint(double[] x, double[] y) {
8         this.x = x;
9         this.y = y;
10    }
11
12    public double[] getY() {
13        return y;
14    }
15
16    public double[] getX() {
17        return x;
18    }
19 }

```

Listing 14: dataset/DataPoint.java

```

1 package dataset;
2
3 public interface IDataset {

```

```

4     DataPoint next();
5
6     void reset();
7 }

```

Listing 15: dataset/IDataset.java

```

1 package dataset;
2
3 public class XORBinaryDataSet implements IDataset {
4
5     protected double[][] x;
6     protected double[] y;
7     private int index;
8
9     public XORBinaryDataSet() {
10         index = 0;
11         x = new double[][] {
12             {0., 0.},
13             {0., 1.},
14             {1., 0.},
15             {1., 1.},
16         };
17         y = new double[] {
18             0.,
19             1.,
20             1.,
21             0.,
22         };
23     }
24
25     @Override
26     public DataPoint next() {
27         if (index < x.length) {
28             var result = new DataPoint(x[index], new double[] {y[index]});
29             index++;
30             return result;
31         }
32         return null;
33     }
34
35     @Override
36     public void reset() {
37         index = 0;
38     }
39 }

```

Listing 16: dataset/XORBinaryDataSet.java

```

1 package nn;
2
3 import autograd.IVariable;
4 import autograd.Parameter;
5
6 public class BipolarSigmoid implements ILayer {
7
8     @Override
9     public IVariable[] apply(IVariable[] input) {

```

```

10     var sigmoid = new autograd.Sigmoid();
11     var scalar = new Parameter(2, false);
12     var constant = new Parameter(-1, false);
13     var addition = new autograd.Addition();
14     var multiplication = new autograd.Multiplication();
15     var result = new IVariable[input.length];
16     for (int i = 0; i < input.length; i++) {
17         result[i] = addition.apply(
18             multiplication.apply(
19                 scalar,
20                 sigmoid.apply(input[i])),
21             constant
22         );
23     }
24     return result;
25 }
26 }

```

Listing 17: nn/BipolarSigmoid.java

```

1 package nn;
2
3 import java.util.ArrayList;
4
5 public class ConvergenceCollector implements IFitCallback {
6     ArrayList<Double> loss;
7
8     public ConvergenceCollector() {
9         this.loss = new ArrayList<>();
10    }
11
12    @Override
13    public void collect(int epoch, double loss) {
14        this.loss.add(loss);
15    }
16
17    public int getEpochs() {
18        return loss.size();
19    }
20
21    @Override
22    public String toString() {
23        StringBuilder sb = new StringBuilder();
24        for (int i = 0; i < loss.size(); i++) {
25            sb.append(+i + " " + loss.get(i) + "\n");
26        }
27        return sb.toString();
28    }
29 }

```

Listing 18: nn/ConvergenceCollector.java

```

1 package nn;
2
3 import autograd.IInitializer;
4 import autograd.IVariable;
5 import autograd.Parameter;
6 import autograd.UniformInitializer;

```

```

7
8 public class Factory {
9     public static Model createNeuralNetwork(int[] sizes, ILayer activation,
10         IInitializer initializer) {
11         if (sizes.length < 2) {
12             throw new IllegalArgumentException("Sizes must at least contain
13                 2 integers for the first and the second layer.");
14         }
15         var inputs = new Parameter[sizes[0]];
16         for (int i = 0; i < inputs.length; i++) {
17             inputs[i] = new Parameter(initializer.next());
18         }
19         IVariable[] lastLayerOutput = inputs;
20         for (int i = 1; i < sizes.length; i++) {
21             lastLayerOutput = new Linear(sizes[i - 1], sizes[i],
22                 initializer).apply(lastLayerOutput);
23             lastLayerOutput = activation.apply(lastLayerOutput);
24         }
25         return new Model(inputs, lastLayerOutput);
26     }
27
28     public static Model createNeuralNetwork(int[] sizes, ILayer activation)
29     {
30         return createNeuralNetwork(sizes, activation, new
31             UniformInitializer(-0.5, 0.5));
32     }
33 }

```

Listing 19: nn/Factory.java

```

1 package nn;
2
3 public interface IFitCallback {
4     void collect(int epoch, double loss);
5 }

```

Listing 20: nn/IFitCallback.java

```

1 package nn;
2
3 import autograd.IVariable;
4
5 public interface ILayer {
6     IVariable[] apply(IVariable[] input);
7 }

```

Listing 21: nn/ILayer.java

```

1 package nn;
2
3 import autograd.*;
4
5 public class Linear implements ILayer {
6     private final IVariable[][] weight;
7     private final IVariable[] bias;
8
9     public Linear(int inFeatures, int outFeatures, IInitializer initializer
10         ) {

```

```

10     this.weight = new Parameter[outFeatures][inFeatures];
11     this.bias = new Parameter[outFeatures];
12     for (int i = 0; i < outFeatures; i++) {
13         for (int j = 0; j < inFeatures; j++) {
14             this.weight[i][j] = new Parameter(initializer.next());
15         }
16         this.bias[i] = new Parameter(initializer.next());
17     }
18 }
19
20 @Override
21 public IVariable[] apply(IVariable[] input) {
22     var result = new IVariable[this.weight.length];
23     for (int i = 0; i < this.weight.length; i++) {
24         int inputSize = this.weight[i].length;
25         IVariable[] muls = new IVariable[inputSize + 1];
26         for (int j = 0; j < inputSize; j++) {
27             muls[j] = new Multiplication().apply(this.weight[i][j],
28             input[j]);
29         }
30         muls[inputSize] = this.bias[i];
31         result[i] = new Addition().apply(muls);
32     }
33     return result;
34 }
35
36 private int getWidth() {
37     return this.weight.length;
38 }
39 }

```

Listing 22: nn/Linear.java

```

1 package nn;
2
3 import autograd.*;
4 import jdk.jshell.spi.ExecutionControl;
5 import optimization.ILoss;
6
7 public class MinimumSquaredError implements IVariable, ILoss {
8
9     private final IVariable operation;
10    private final Parameter[] desired;
11
12    public MinimumSquaredError(IVariable[] output) {
13        var negation = new Negation();
14        var addition = new Addition();
15        var multiplication = new Multiplication();
16        var exponentiation = new Exponentiation();
17        Parameter two = new Parameter(2, false);
18        Parameter half = new Parameter(0.5, false);
19        int length = output.length;
20        desired = new Parameter[length];
21        var summationTerms = new IVariable[length];
22        for (int i = 0; i < length; i++) {
23            desired[i] = new Parameter();
24            summationTerms[i] = exponentiation.apply(

```



```

25         addition.apply(output[i], negation.apply(desired[i])),
26         two
27     );
28 }
29 this.operation = multiplication.apply(addition.apply(summationTerms
30     ), half);
31 }
32 @Override
33 public double evaluate() {
34     return operation.evaluate();
35 }
36
37 @Override
38 public void backward(IVariable[] sources, double gradient) throws
39     ExecutionControl.NotImplementedException {
40     operation.backward(sources, gradient);
41 }
42
43 @Override
44 public Parameter[] getParameters() {
45     return this.operation.getParameters();
46 }
47
48 @Override
49 public void setDesired(double[] desired) {
50     for (int i = 0; i < this.desired.length; i++) {
51         this.desired[i].setValue(desired[i]);
52     }
53 }

```

Listing 23: nn/MinimumSquaredError.java

```

1 package nn;
2
3 import autograd.IVariable;
4 import autograd.Operation;
5 import autograd.Parameter;
6 import dataset.DataPoint;
7 import dataset.IDataset;
8 import jdk.jshell.spi.ExecutionControl;
9 import optimization.ILoss;
10 import optimization.IOptimizer;
11
12 import java.util.Arrays;
13 import java.util.HashSet;
14 import java.util.List;
15 import java.util.Map;
16 import java.util.stream.Collectors;
17
18 public class Model {
19     private final Parameter[] input;
20     private final IVariable[] output;
21
22     public Model(Parameter[] input, IVariable[] output) {
23         this.input = input;
24         this.output = output;

```

```

25     }
26
27
28     public double[] evaluate(double[] input) {
29         var result = new double[output.length];
30         for (int i = 0; i < input.length; i++) {
31             this.input[i].setValue(input[i]);
32         }
33         for (int i = 0; i < output.length; i++) {
34             result[i] = output[i].evaluate();
35         }
36         return result;
37     }
38
39     public Parameter[] getParameters() {
40         HashSet<Parameter> result = new HashSet<>();
41         for (IVariable o :
42             this.output) {
43             result.addAll(Arrays.asList(o.getParameters()));
44         }
45         return result.toArray(new Parameter[0]);
46     }
47
48     public Parameter[] getTrainableParameters() {
49         var results = new HashSet<Parameter>();
50         for (Parameter p :
51             getParameters()) {
52             if (p.isTrainable()) {
53                 results.add(p);
54             }
55         }
56         for (Parameter p : input) {
57             results.remove(p);
58         }
59
60         return results.toArray(new Parameter[0]);
61     }
62
63     public IVariable[] getOutput() {
64         return output;
65     }
66
67     public double fit(IDataset dataSet, IOptimizer optimizer, ILoss loss,
68         int epochs, double lossLimit) throws ExecutionControl.
69         NotImplementedException {
70         return fit(dataSet, optimizer, loss, epochs, lossLimit, (epoch, l)
71             => {
72             });
73     }
74
75     public double fit(IDataset dataSet, IOptimizer optimizer, ILoss loss,
76         int epochs, double lossLimit, IFitCallback callback) throws
77         ExecutionControl.NotImplementedException {
78         var parameters = getTrainableParameters();
79         Map<Integer, List<Parameter>> layeredParameters = layerParameters(
80             parameters);
81         if (epochs < 1) {
82             throw new IllegalArgumentException("At least one epochs

```

```

77         required.");
78     }
79     double totalLoss = 0;
80     for (int i = 0; i < epochs; i++) {
81         totalLoss = 0;
82         dataSet.reset();
83         DataPoint dataPoint;
84         while ((dataPoint = dataSet.next()) != null) {
85             setInput(dataPoint.getX());
86             loss.setDesired(dataPoint.getY());
87             totalLoss += loss.evaluate();
88             for (Integer j : layeredParameters.keySet().stream().sorted()
89                 .collect(Collectors.toList())) {
90                 Parameter[] layerParameters = layeredParameters.get(j).
91                     toArray(new Parameter[0]);
92                 loss.backward(layerParameters, 1.);
93                 optimizer.update(layerParameters);
94             }
95         }
96         callback.collect(i, totalLoss);
97         if (totalLoss < lossLimit) {
98             break;
99         }
100     }
101     return totalLoss;
102 }
103
104 private Map<Integer, List<Parameter>> layerParameters(Parameter[]
105     parameters) {
106     setLayers(getOutput(), 0);
107     return Arrays.stream(parameters).collect(Collectors.groupingBy(
108         Parameter::getLayer));
109 }
110
111 private void setLayers(IVariable[] outputs, int layer) {
112     if (outputs.length == 0) return;
113     HashSet<IVariable> nextOutput = new HashSet<>();
114     for (IVariable i : outputs) {
115         if (i instanceof Parameter) {
116             ((Parameter) i).setLayer(layer);
117         }
118         if (i instanceof Operation) {
119             nextOutput.addAll(Arrays.asList(((Operation) i).getOperands()
120                 ));
121         }
122     }
123     setLayers(nextOutput.toArray(new IVariable[0]), layer + 1);
124 }
125
126 private void setInput(double[] x) {
127     for (int i = 0; i < input.length; i++) {
128         input[i].setValue(x[i]);
129     }
130 }
131 }

```

Listing 24: nn/Model.java

```

1 package nn;
2
3 import autograd.IVariable;
4
5 public class Sigmoid implements ILayer {
6
7     @Override
8     public IVariable[] apply(IVariable[] input) {
9         var operator = new autograd.Sigmoid();
10        var result = new IVariable[input.length];
11        for (int i = 0; i < input.length; i++) {
12            result[i] = operator.apply(input[i]);
13        }
14        return result;
15    }
16 }

```

Listing 25: nn/Sigmoid.java

```

1 package optimization;
2
3 import autograd.Parameter;
4
5 import java.util.HashMap;
6
7 public class GradientDescent implements IOptimizer {
8
9     private final HashMap<Parameter, Double> lastDelta;
10    private final double learningRate;
11    private final double momentum;
12
13    public GradientDescent(double learningRate, double momentum) {
14        this.lastDelta = new HashMap<>();
15        this.learningRate = learningRate;
16        this.momentum = momentum;
17    }
18
19    @Override
20    public void update(Parameter[] parameters) {
21        for (Parameter p :
22            parameters) {
23            double delta = -p.getGradient() * learningRate + momentum *
24                lastDelta.getOrDefault(p, 0.);
25            p.setValue(p.getValue() + delta);
26            p.zeroGradient();
27            lastDelta.put(p, delta);
28        }
29    }
30 }

```

Listing 26: optimization/GradientDescent.java

```

1 package optimization;
2
3 import autograd.IVariable;
4
5 public interface ILoss extends IVariable {
6     void setDesired(double[] desired);
7 }

```

```
7 }
```

Listing 27: optimization/ILoss.java

```
1 package optimization;
2
3 import autograd.Parameter;
4
5 public interface IOptimizer {
6     void update(Parameter[] parameters);
7 }
```

Listing 28: optimization/IOptimizer.java

```
1 package autograd;
2
3
4 import org.junit.Assert;
5 import org.junit.Test;
6
7 public class VariableTest {
8
9     @Test
10     public void testAddition() {
11         Assert.assertEquals(new Addition().apply(new Parameter(12), new
12             Parameter(2)).evaluate(), 14., 0.);
13     }
14
15     @Test
16     public void testVariableEvaluation() {
17         Assert.assertEquals(new Parameter(250).evaluate(), 250., 0);
18     }
19 }
```

Listing 29: autograd/VariableTest.java

```
1 package nn;
2
3 import autograd.Parameter;
4 import jdk.jshell.spi.ExecutionControl;
5 import org.junit.Assert;
6 import org.junit.Test;
7
8 import java.util.Arrays;
9
10 public class NeuralNetworkTest {
11
12     @Test
13     public void testNeuralNetworkFactory() {
14         var model = Factory.createNeuralNetwork(new int[]{2, 4, 1}, new
15             Sigmoid());
16         var result = model.evaluate(new double[]{0, 0});
17         Assert.assertEquals(result.length, 1);
18     }
19
20     @Test
21     public void testNeuralNetworkGradient() throws ExecutionControl.
22         NotImplementedException {
```

```

21     var model = Factory.createNeuralNetwork(new int[]{2, 4, 1}, new
        Sigmoid());
22     Parameter[] parameters = model.getTrainableParameters();
23     for (Parameter parameter :
24         parameters) {
25         parameter.setValue(1);
26     }
27     var result = model.evaluate(new double[]{1, 0});
28     double[] desired = new double[]{1};
29     Assert.assertEquals(result.length, 1);
30     double delta = 1e-5;
31     double expected = 0.9892621636390686; // obtained by pytorch
32     Assert.assertEquals(result[0], expected, delta);
33     var loss = new MinimumSquaredError(model.getOutput());
34     loss.setDesired(desired);
35     loss.backward(parameters, 1);
36     var gradients = new double[parameters.length];
37     for (int i = 0; i < parameters.length; i++) {
38         gradients[i] = parameters[i].getGradient();
39     }
40
41     Assert.assertArrayEquals(Arrays.stream(new double[] { // calculated
        by pytorch
42         -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
43     }).sorted().toArray(), Arrays.stream(gradients).sorted().toArray(),
        delta);
44 }
45
46 @Test
47 public void testNeuralNetworkGradientBipolar() throws ExecutionControl.
    NotImplementedException {
48     var model = Factory.createNeuralNetwork(new int[]{2, 4, 1}, new
        BipolarSigmoid());
49     Parameter[] parameters = model.getTrainableParameters();
50     for (Parameter parameter :
51         parameters) {
52         parameter.setValue(1);
53     }
54     var result = model.evaluate(new double[]{1, -1});
55     double[] desired = new double[]{1};
56     Assert.assertEquals(result.length, 1);
57     double delta = 1e-5;
58     double expected = 0.8904789686203003; // obtained by pytorch
59     Assert.assertEquals(expected, result[0], delta);
60     var loss = new MinimumSquaredError(model.getOutput());
61     loss.setDesired(desired);
62     loss.backward(parameters, 1);
63     var gradients = new double[parameters.length];
64     for (int i = 0; i < parameters.length; i++) {
65         gradients[i] = parameters[i].getGradient();
66     }
67

```

```

68     Assert.assertEquals(Arrays.stream(new double[] { // calculated
69         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
        }).sorted().toArray(), Arrays.stream(gradients).sorted().toArray(),
70         delta);
71     }
72 }

```

Listing 30: nn/NeuralNetworkTest.java

```

1 package optimization;
2
3 import autograd.UniformInitializer;
4 import dataset.BinaryToBipolarWrapper;
5 import dataset.XORBinaryDataSet;
6 import jdk.jshell.spi.ExecutionControl;
7 import nn.*;
8 import org.junit.Assert;
9 import org.junit.Ignore;
10 import org.junit.Test;
11
12 import java.io.FileWriter;
13 import java.io.IOException;
14 import java.util.ArrayList;
15 import java.util.Comparator;
16 import java.util.Optional;
17
18 public class GradientDescentTest {
19
20     private final static int trials = 300;
21
22     @Ignore
23     @Test
24     public void TestSimpleGD() throws ExecutionControl.
25         NotImplementedException {
26         var model = Factory.createNeuralNetwork(
27             new int[] {2, 4, 1},
28             new Sigmoid(),
29             new UniformInitializer(-0.5, 0.5)
30         );
31         var dataSet = new XORBinaryDataSet();
32         var optimizer = new GradientDescent(0.2, 0.);
33         var loss = new MinimumSquaredError(model.getOutput());
34         double finalLoss = model.fit(dataSet, optimizer, loss, 40000, 0.05);
35         ;
36         Assert.assertTrue("Big loss " + finalLoss, finalLoss < 0.05);
37     }
38
39     @Ignore("Skipping slow convergence tests.")
40     @Test

```

```

39 public void TestConvergence() throws ExecutionControl.
    NotImplementedException, IOException {
40     int diverged = 0;
41     ArrayList<ConvergenceCollector> stats = new ArrayList<>();
42     for (int i = 0; i < GradientDescentTest.trials; i++) {
43         var model = Factory.createNeuralNetwork(
44             new int[] {2, 4, 1},
45             new Sigmoid(),
46             new UniformInitializer(-0.5, 0.5)
47         );
48         var dataSet = new XORBinaryDataSet();
49         var optimizer = new GradientDescent(0.2, 0.);
50         var loss = new MinimumSquaredError(model.getOutput());
51         var collector = new ConvergenceCollector();
52         double finalLoss = model.fit(dataSet, optimizer, loss, 40000,
53             0.05, collector);
54         stats.add(collector);
55         if (finalLoss > 0.05) {
56             diverged += 1;
57         }
58     }
59     outputGraphData("a", stats);
60     Assert.assertTrue("Convergence with high probability busted!",
61         diverged < 6);
62 }
63
64 private void outputGraphData(String assignmentPart, ArrayList<
65     ConvergenceCollector> stats) throws IOException {
66     FileWriter of = new FileWriter("doc/" + assignmentPart + "_avg.tex"
67         );
68     double average = stats.stream().mapToInt(ConvergenceCollector::
69         getEpochs).average().getAsDouble();
70     of.write(String.valueOf(average));
71     of.close();
72
73     Optional<ConvergenceCollector> representative = stats.stream().min(
74         Comparator.comparingDouble(c -> Math.abs(c.getEpochs() - average
75         )));
76     of = new FileWriter("doc/" + assignmentPart + ".tex");
77     of.write(representative.get().toString());
78     of.close();
79 }
80
81 @Ignore("Skipping slow convergence tests.")
82 @Test
83 public void TestBipolarGD() throws ExecutionControl.
84     NotImplementedException, IOException {
85     int diverged = 0;
86     int trials = GradientDescentTest.trials;
87     ArrayList<ConvergenceCollector> stats = new ArrayList<>();
88     for (int i = 0; i < trials; i++) {
89         var model = Factory.createNeuralNetwork(
90             new int[] {2, 4, 1},
91             new BipolarSigmoid(),
92             new UniformInitializer(-0.5, 0.5)
93         );
94         var dataSet = new BinaryToBipolarWrapper(new XORBinaryDataSet())
95     };

```



```

87         var optimizer = new GradientDescent(0.2, 0.);
88         var loss = new MinimumSquaredError(model.getOutput());
89         var collector = new ConvergenceCollector();
90         double finalLoss = model.fit(dataSet, optimizer, loss, 3500,
91             0.05, collector);
92         if (finalLoss > 0.05) {
93             diverged += 1;
94         }
95         stats.add(collector);
96     }
97     outputGraphData("b", stats);
98     Assert.assertTrue("Convergence with high probability busted! " +
99         diverged + " failure out of " + trials, diverged < 6);
100 }
101
102 @Ignore
103 @Test
104 public void TestBipolarMomentumGD() throws ExecutionControl.
105     NotImplementedException, IOException {
106
107     int diverged = 0;
108     int trials = GradientDescentTest.trials;
109     ArrayList<ConvergenceCollector> stats = new ArrayList<>();
110     for (int i = 0; i < trials; i++) {
111         var model = Factory.createNeuralNetwork(
112             new int[]{2, 4, 1},
113             new BipolarSigmoid(),
114             new UniformInitializer(-0.5, 0.5)
115         );
116         var dataSet = new BinaryToBipolarWrapper(new XORBinaryDataSet());
117         var optimizer = new GradientDescent(0.2, 0.9);
118         var loss = new MinimumSquaredError(model.getOutput());
119         var collector = new ConvergenceCollector();
120         double finalLoss = model.fit(dataSet, optimizer, loss, 1000,
121             0.05, collector);
122         if (finalLoss > 0.05) {
123             diverged += 1;
124         }
125         stats.add(collector);
126     }
127     outputGraphData("c", stats);
128     Assert.assertTrue("Convergence with high probability busted! " +
129         diverged + " failure out of " + trials, diverged < 6);
130 }
131 }

```

Listing 31: optimization/GradientDescentTest.java