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

Ali Asgari Khoshouyeh (Student #24868739)

December 15, 2021

Team Members

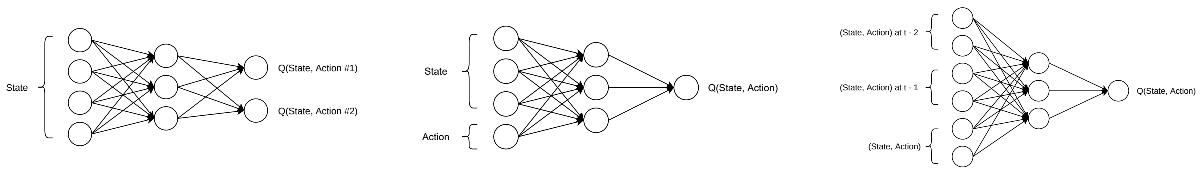
We are a team of three sharing the same code base.

- Christina Sun
- Husna Kalim
- Ali Asgari Khoushouyeh

It is noteworthy to mention that close to the extended deadline we realized that our code is orders of magnitude slower on my teammates' machines. So we sharing the plot data too.

(4) The use of a neural network to replace the look-up table and approximate the Q-function has some disadvantages and advantages.

a) There are 3 options for the architecture of your neural network. Describe and draw all three options and state which you selected and why. (3pts)



(a) Action as output (b) Action as input (c) Adding time dimention

Figure 1: Different architectures considered for the neural network as the function approximator for the Q values of different game actions.

FIg. 1 shows the different architectures we considered for our neural network to be plugged in instead of the lookup table. The first two architectures shown in Fig. 1a and Fig. 1b are the ones discussed in the class. The first one codes the Q value for different possible actions at each step as different neurons of the output. In the second architecture the action is coded as some input neurons and the Q value corresponding the the encoded action as the input at a certain state is coded as the single output neuron.

The novel architecture that we used is the one shown in Fig. 1c which is essentially same as the second architecture, but, it also receives the state-actions of the two past time-steps as input. This is following our intuition used in the second part of the assignment where we included the past two state-actions in the lookup table keys to capture the delay of rewards as it may take some time for a bullet to hit the enemy and get reflected in the reward policy.

b) Show (as a graph) the results of training your neural network using the contents of the LUT from Part 2. Your answer should describe how you found the hyper-parameters which worked best for you (i.e. momentum, learning rate, number of hidden neurons). Provide graphs to backup your selection process. Compute the RMS error for your best results. (5 pts)

Learning Rate	Momentum	Hidden Neurons	RMSE
1.00E-04	0	5	9.666091389
1.00E-04	0	15	9.660078042
1.00E-04	0	30	9.628909289
0.001	0	5	7.632740399
0.001	0	15	6.943443362
0.001	0	30	6.711401519
0.01	0	5	7.985192317
0.01	0	15	7.018144655
0.01	0	30	6.530952131
1.00E-04	0.5	5	9.148578254
1.00E-04	0.5	15	8.993952348
1.00E-04	0.5	30	9.051175005
0.001	0.5	5	7.297405961
0.001	0.5	15	**6.335407802**
0.001	0.5	30	**6.289893926**
0.01	0.5	5	7.919711744
0.01	0.5	15	7.75168296
0.01	0.5	30	8.222388904
1.00E-04	0.9	5	7.713839065
1.00E-04	0.9	15	6.790881731
1.00E-04	0.9	30	6.754064308
0.001	0.9	5	7.772037228
0.001	0.9	15	7.099569645
0.001	0.9	30	6.728784713
0.01	0.9	5	11.50926595
0.01	0.9	15	13.85358874
0.01	0.9	30	18.89859967

Table 1: RMSE after 100 epochs under different hyper-parameter values for training the neural network on the static lookup table data.

We performed a grid search on different possible values of number of hidden neurons, momentum and learning rate. As the Q values in our lookup table ranged from -1.2 to several hundreds, we removed the activation of the last layer, allowing the last fully connected layer to act as a regression solver. Please note that we will keep a bipolar activation when we start training with live data, as the limited range of the output prevents the Q values to go beyond the range.

The results are shown in Table 1. As the difference between 15 hidden neurons and 30 hidden neurons (shown bold in the table) is negligible, we keep 15 neurons as it is much faster.

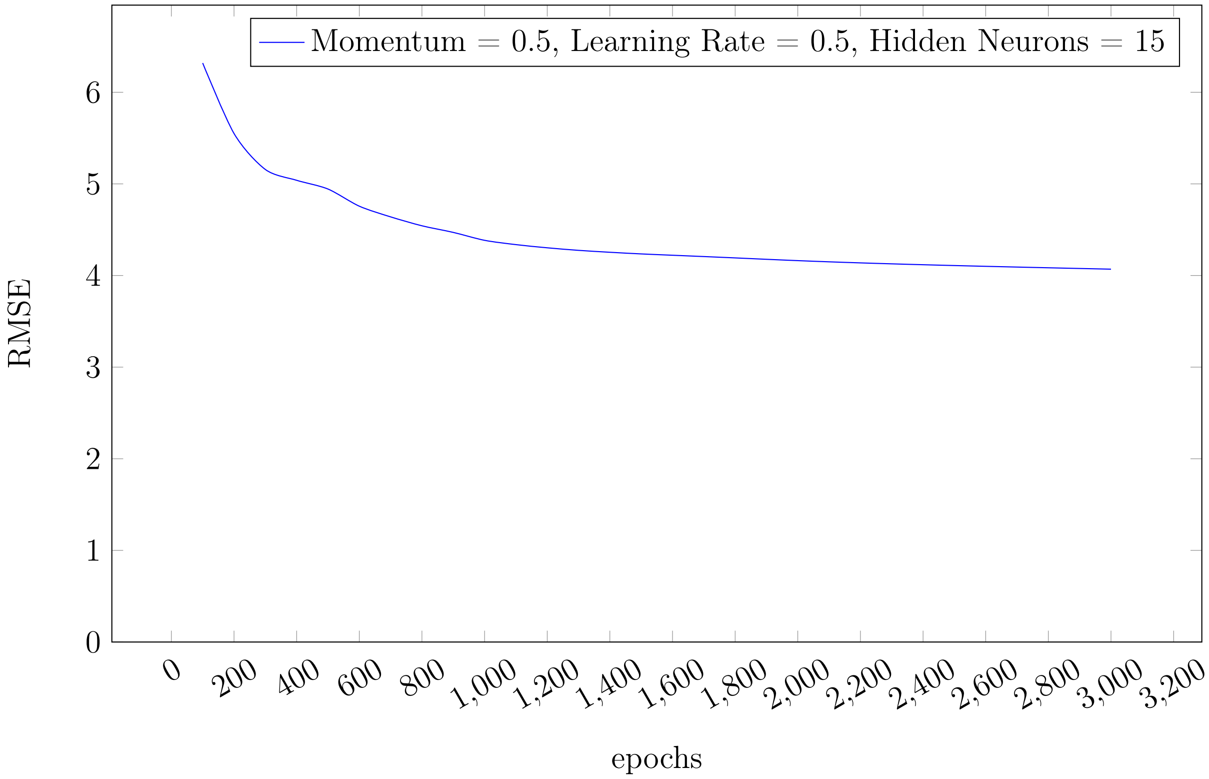


Figure 2: The convergence of the select hyper-parameters over static lookup table data.

We show the convergance of the selected settings in Fig. 2. It shows that our neural network is able to reduce the error fitting on the lookup table static data by decreasing the error from more than 6 to around **4**.

c) Comment on why theoretically a neural network (or any other approach to Q-function approximation) would not necessarily need the same level of state space reduction as a look up table. (2 pts)

As an example, we are using the distance to the enemy as one of the dimensions of the states. When using a lookup table, it matters to decrease the number of possible values to few, so the number of entries in the lookup table do not blow up and the get revisited often. But as the neural network treats the input as a real number and does not do exact-match like the lookup table, close values also let the states be somehow recalled by the neural network. Thus, using the large number of possible values for the distant to enemy is still tractable by the neural network. That said, it is still important to limit the range of that value to something close to the other inputs so the weight initialization would fit this dimension like the other dimensions.

(5) Hopefully you were able to train your robot to find at least one movement pattern that results in defeat of your chosen enemy tank, most of the time.

a) Identify two metrics and use them to measure the performance of your robot with online training. I.e. during battle. Describe how the results were obtained, particularly with regard to exploration? Your answer should provide graphs to support your results. (5 pts)

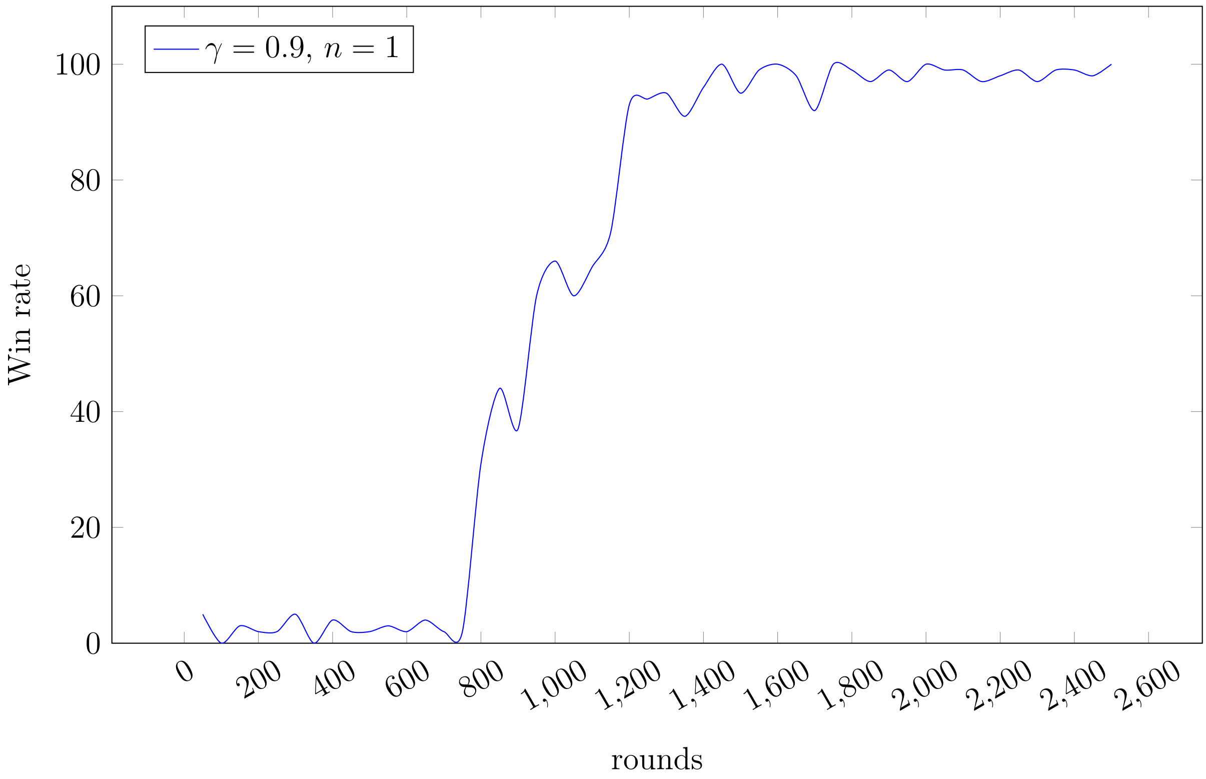


Figure 3: Win rate with regard to number of training rounds.

We measure the following metrics:

1. **Win rate**: The number of rounds won among 100 rounds.
2. **Average enemy energy**: The average enemy energy at the end of rounds for the course of 100 test rounds. Lower values show a better agent in causing damage to the enemy. This also will reflect our policy which only rewards on decrease in enemy energy.

We interleave battles of training with a robot with $\epsilon = 0.8$ and test battles of 100 rounds with a robot with ϵ set to 0.05. We only report the metrics for the test robot (i.e. ϵ set to 0.05).

We realized that the learning process is a lot quicker if we remove the enemy distance from the input dimensions. This makes sense about our opponent which is the Corners robot. The Corners robot navigates to one corner and then tries to hit us. The best

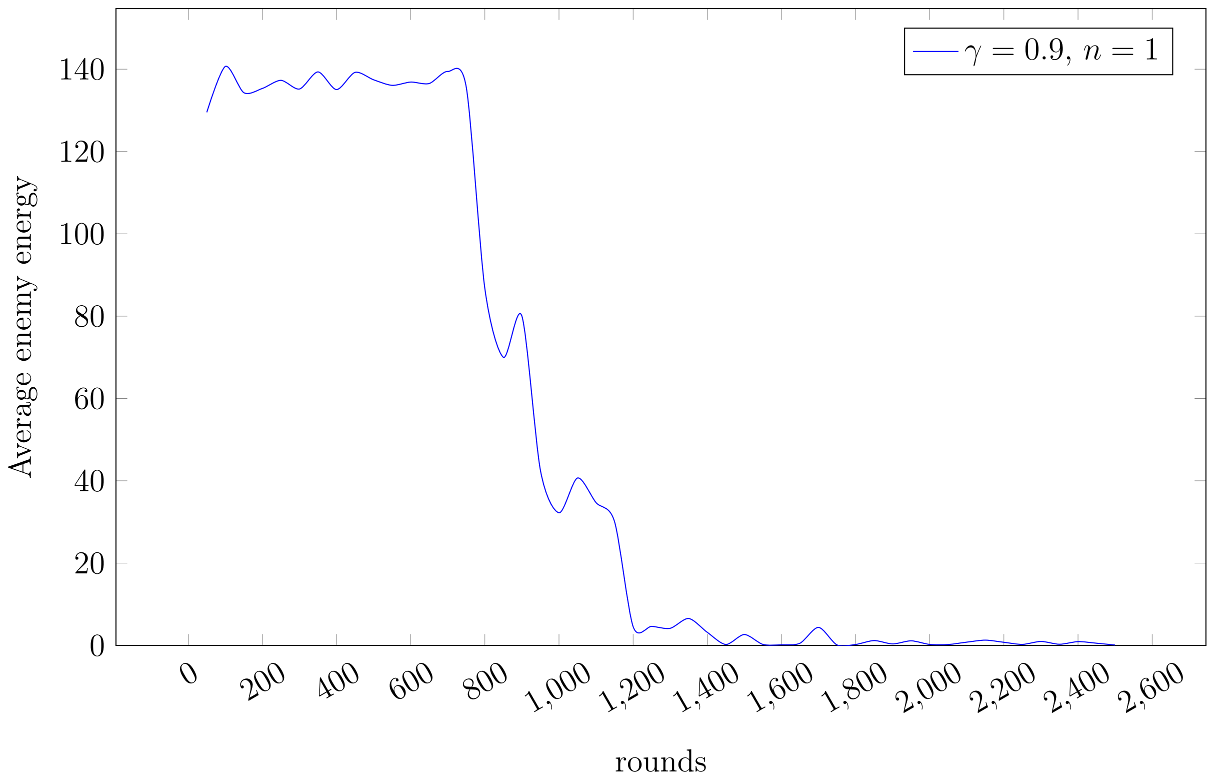


Figure 4: Average enemy energy with regard to number of training rounds.

strategy that our robot comes up with is to fire so often, and this strategy is independent of the distance to enemy. Thus, we are reporting the results with that simplification hereafter.

Fig. 3 shows that how the win rate for our robot increases after rounds of training. We observe a somehow abrupt increase in the win rate. My hypothesis explaining the reason this happens is that with the neural network the Q values change gradually towards the optimal values. The optimal actions are competing the other actions with such moving Q values. Probably, because of the fact that weights are shared for different state-actions, the optimal actions exceed the Q value of the other actions and win the competition in a short number of rounds. That said, they were shifting towards the competing actions since the beginning and exceed the Q value of other actions only after 1200 rounds.

As for the second metric, Fig. 4 shows that the average energy of the enemy decays as our agent learns more.

b) The discount factor γ can be used to modify influence of future reward. Measure the performance of your robot for different values of γ and plot your results. Would you expect higher or lower values to be better and why? (3 pts)

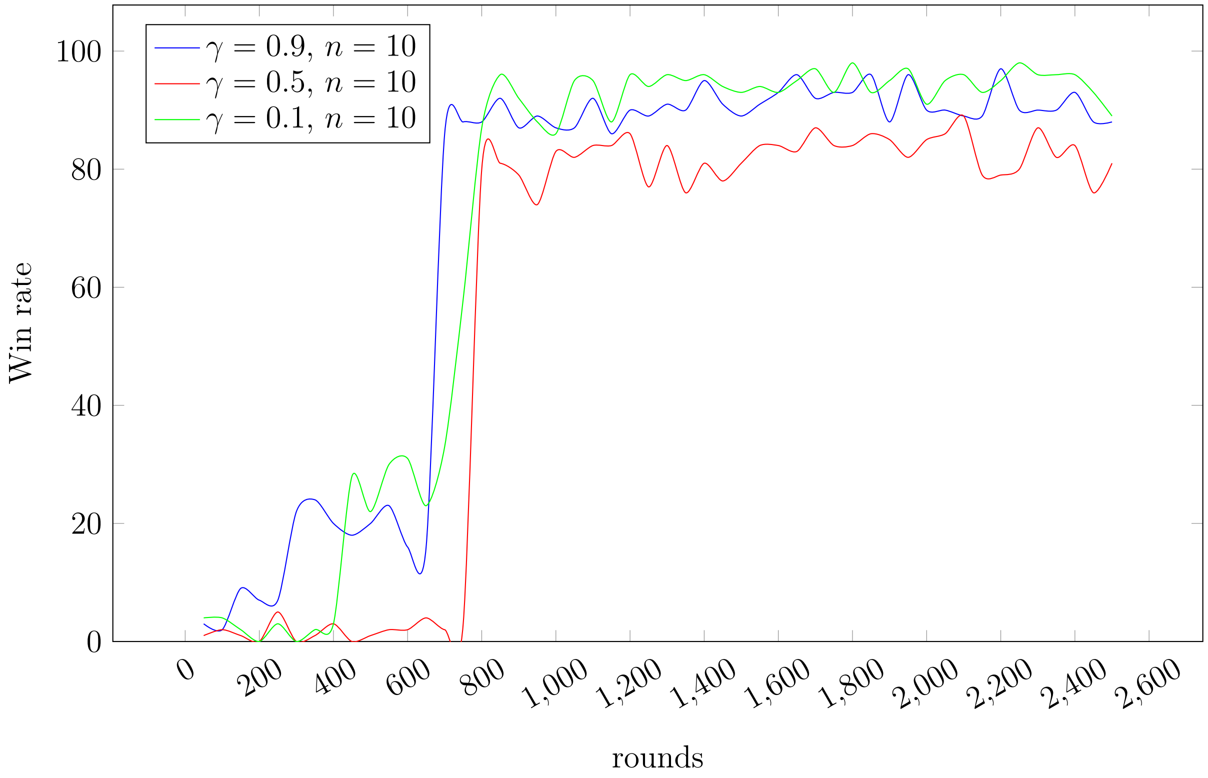


Figure 5: Win rate with regard to number of training rounds, under different γ.

Fig. 5 shows the convergence of win rate for different gamma values. As can be observed the trend is similar for $\gamma = 0.1$ and $\gamma = 0.9$, but, the training seems to be more complex and converging at a less win rate for $\gamma = 0.5$. My hypothesis to explain this is that high gamma values will lead to quicker convergence in the Q-learning side, meaning rewards propagate quicker to the earlier stages. With the low gamma values the Q values used to update the neural network change slower, so the neural network has more time to adapt. But the intermediate gamma value (0.5) has none of the advantages. It is neither slow while providing updates to the neural network, nor needing less steps in the convergence of Q values, thus it leads to a less ultimate effectiveness of training.

Intuitively, there are two dynamical systems interacting with each other and trying to converge here, one is the neural network, the other is the Q-learning values. These two are chained together. High gamma values help the Q-learning and low gamma values help the neural network. But the intermediate value does not provide any of the advantages.

c) Theory question: With a look-up table, the TD learning algorithm is proven to converge – i.e. will arrive at a stable set of Q-values for all visited states. This is not so when the Q-function is approximated. Explain this convergence in terms of the Bellman equation and also why when using approximation, convergence is no longer guaranteed. (3 pts)

If you unroll the Q-value based on the Q-learning formula, assuming the reward is some stochastic random variable, we can define the Q-value as the following expected:

$$V(s) = E[r_0 + \gamma V(s_1) | s_0 = s]$$

so $r_0 + \gamma V(s_1)$ is an unbiased estimate for the random variable $V(s)$ [1]. This means initializing V with arbitrary values for states and then sampling the consequent states of the game (s and s'), and updating the values based on the following rule with a positive learning rate (α) will converge to the unbiased estimates:

$$V(s) \leftarrow V(s) + \alpha(r + \gamma V(s') - V(s))$$

Unfortunately, with using a function approximation this update rule is not done precisely, instead the left-hand-side shifts closer to the right-hand-side, while implicitly corrupting values for other states, thus this update rule which is guaranteed to converge on the unbiased estimates is not performed and the guarantee does not hold.

[1] `https://en.wikipedia.org/wiki/Temporal_difference_learning#Mathematical_formulation`

d) When using a neural network for supervised learning, performance of training is typically measured by computing a total error over the training set. When using the NN for online learning of the Q-function in robocode this is not possible since there is no a-priori training set to work with. Suggest how you might monitor learning performance of the neural net now. (3 pts)

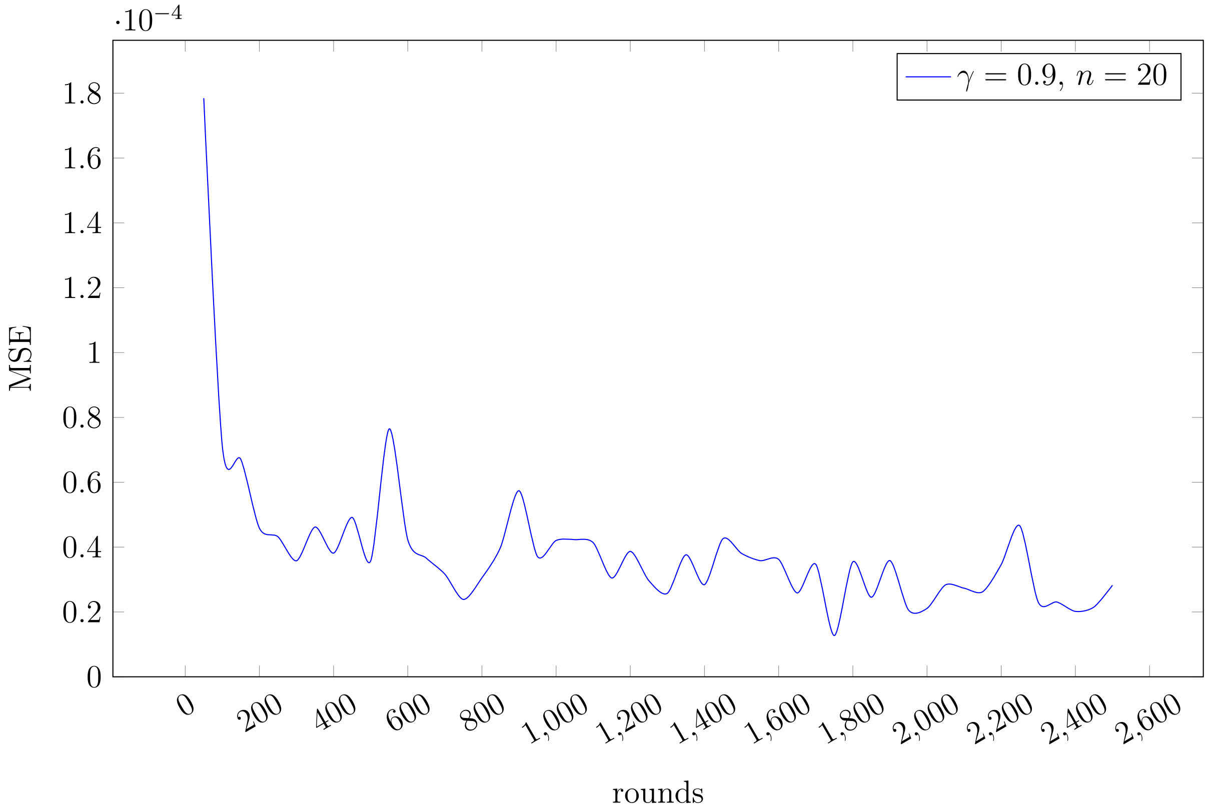


Figure 6: MSE over 20 subsequent states, with regard to rounds of training.

Inspired by the idea of replay memory, we use the loss over a local window of data points. Fig. 6 shows how the MSE as a metric for the performance of the neural network decreases after rounds of training. With the use of replay memory we are able to calculate this loss over 20 subsequent data points.

e) At each time step, the neural net in your robot performs a back propagation using a single training vector provided by the RL agent. Modify your code so that it keeps an array of the last say n training vectors and at each time step performs n back propagations. Using graphs compare the performance of your robot for different values of n. (4 pts)

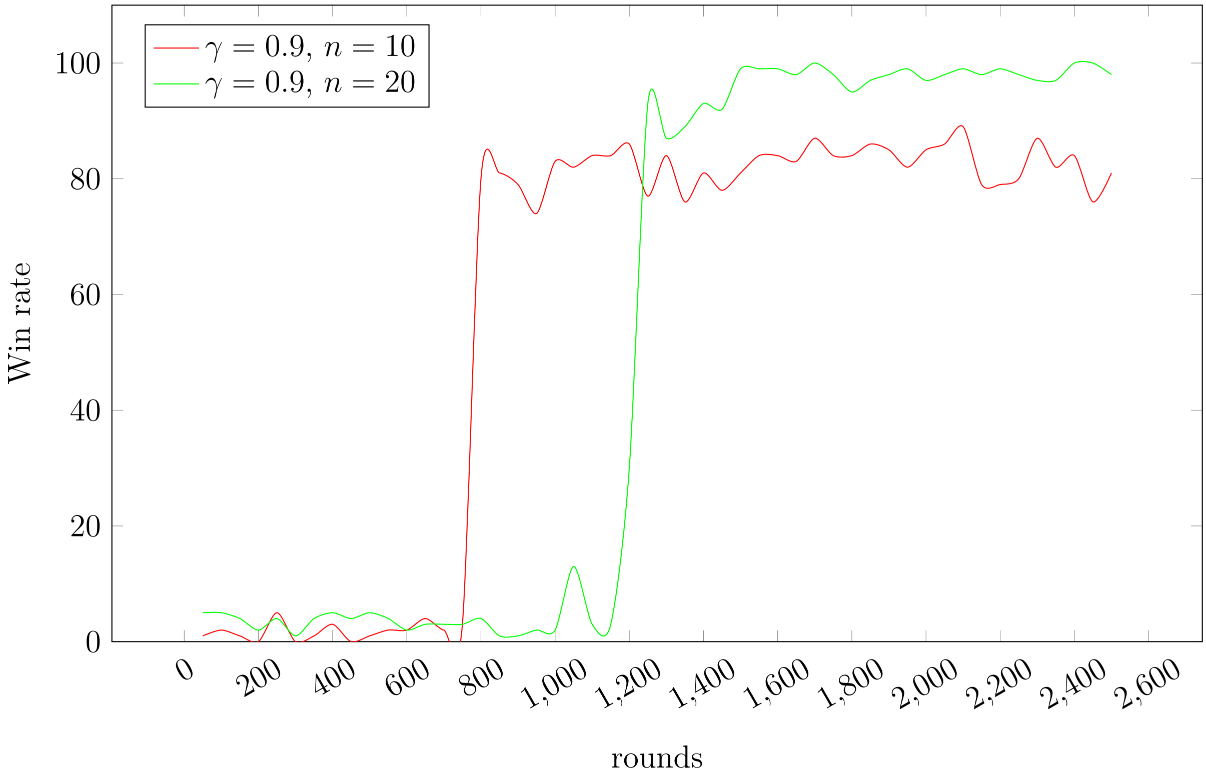


Figure 7: Win rate with regard to number of training rounds, under different γ.

Fig. 7 shows the different convergence behaviour with regard to different values for the window size of the replay memory. We observe that high value for the window size leads to a better win rate eventually. My hypothesis to explain this is that a larger local window allows the neural network to skip local optima and hopefully take better strategies.

(6) Overall Conclusions

a) This question is open-ended and offers you an opportunity to reflect on what you have learned overall through this project. For example, what insights are you able to offer with regard to the practical issues surrounding the application of RL & BP to your problem? E.g. What could you do to improve the performance of your robot? How would you suggest convergence problems be addressed? What advice would you give when applying RL with neural network based function approximation to other practical applications? (4 pts)

Here are some insights we came up with:

1. More hidden neurons make the neural network to be able to learn more complex patterns. We were able to achieve less numerical error over our static lookup table data with more number of hidden neurons.

2. In a game where there are delays for the rewards to get effective (e.g. for firing to result in change in enemy energy) one trick to turn the environment back to a Markov environment is to treat series of subsequent states as some macro states which are path independent and actually form a Markov chain.

3. A lot of the time it is possible that large learning rates result in lack of convergence of the neural network. The search for a working and yet efficient learning rate has a rule of thumb of searching with different orders of magnitude and it is not required to search with linear steps.

4. To approach a complex implementation such as Q-learning with a deep neural network, it is always better to first break it into sub-problems. We have done this several times during the project. To test our Q-learning code we used a simpler game with the goal of navigating the robot the top right corner of the game only. To debug our neural network we started with a single layer wide neural network which was quite imitating a lookup table. To check if our neural network is capable of learning the game patterns we were instructed to first train the neural network over the static data from a lookup table.

b) Theory question: Imagine a closed-loop control system for automatically delivering anesthetic to a patient under going surgery. You intend to train the controller using the approach used in this project. Discuss any concerns with this and identify one potential variation that could alleviate those concerns. (3 pts)

The essential problem here is that there is no grant for killing the patient because of the RL algorithm performing trial and errors.

To fix this, I would suggest training the control system on a pseudo patient which only measures the dosage of anaesthetic over time.

To further make things safe, after the training is done we can try applying formal verification to ensure no harmful state is met while the execution of the trained agent.

We can also put safeguards in the controller, such as maximum limits of the dosage so that it does not pass the safe limits.

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

```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) {
```

```java
            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

```java
package autograd;

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.io.Serializable;
import java.util.Arrays;

public class Parameter implements IVariable, Serializable {
    private double value;
    private double gradient;
    private boolean trainable;
    private int layer;
    private final int parameterId;
    static transient int parameterCounter = 0;

    public Parameter() {
        parameterId = parameterCounter++;
    }

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

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

    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;
    }

    public int getParameterId() {
        return parameterId;
    }
}
```

Listing 10: autograd/Parameter.java

```java
package autograd;

import jdk.jshell.spi.ExecutionControl;

public class ReLU extends Operator {

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

    @Override
    public double evaluate(IVariable[] operands) {
        if (operands.length != 1) {
            throw new IllegalArgumentException("Sigmoid operator only accepts one operand");
        }
        double result = Math.max(0., operands[0].evaluate());
//          System.out.println("ReLU " + result);
        return result;
    }

    @Override
    public void backwards(IVariable[] operands, IVariable[] sources, double gradient) throws ExecutionControl.NotImplementedException {
        validateOperands(operands);
        var x = operands[0];
        var y = evaluate(operands);
//        System.out.println(gradient);
        if (y > 0) {
//            System.out.println("Gradient " + y + " " + gradient);
            x.backward(sources, gradient);
        } else {
//            System.out.println("Gradient 0");
            x.backward(sources, 0.);
        }
    }
}
```

Listing 11: autograd/ReLU.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 12: 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 13: 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();
    }

    @Override
    public DataPoint onlyReadNext() {
        return binaryDataSet.onlyReadNext();
    }
}
```

Listing 14: dataset/BinaryToBipolarWrapper.java

```java
package dataset;

public class DataPointDataSet implements IDataSet {
    private final DataPoint dataPoint;
    private boolean endOfDataSet = false;

    public DataPointDataSet(DataPoint dataPoint) {
        this.dataPoint = dataPoint;
    }


    @Override
    public DataPoint next() {
        if (endOfDataSet) {
            return null;
        } else {
            endOfDataSet = true;
            return dataPoint;
        }
    }

    @Override
    public void reset() {
        endOfDataSet = false;
    }

    @Override
    public DataPoint onlyReadNext() {
        if (endOfDataSet) {
            return null;
        } else {
            return dataPoint;
        }
    }
}
```

Listing 15: dataset/DataPointDataSet.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 16: dataset/DataPoint.java

```java
package dataset;

public interface IDataSet {
    DataPoint next();

    void reset();

    DataPoint onlyReadNext();
}
```

Listing 17: dataset/IDataSet.java

```java
package dataset;

import fa.LUT;
import representation.IRepresentable;

import javax.naming.InsufficientResourcesException;
import javax.xml.crypto.Data;
import java.util.ArrayList;
import java.util.HashMap;

// TODO Christina and Husna
public class LookupTableDataSet implements IDataSet {

    ArrayList<DataPoint> points = new ArrayList<>();
    int index = 0;

    public LookupTableDataSet(LUT lut) {
        HashMap map = lut.getHashMap();
        for (Object key :
                map.keySet()) {
            IRepresentable representable = (IRepresentable) key;
            points.add(new DataPoint(
                    representable.toVector(),
                    lut.eval(representable)));

        }
    }


    @Override
    public DataPoint next() {
        if (index < points.size())
            return points.get(index++);
        return null;
    }

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

    @Override
    public DataPoint onlyReadNext() {
        if (index < points.size())
            return points.get(index);
        return null;
    }
}
```

Listing 18: dataset/LookupTableDataSet.java

```java
package dataset;

import representation.IRepresentable;

import java.awt.geom.Line2D;
import java.io.Serializable;
import java.lang.reflect.Array;
import java.util.ArrayList;
import java.util.LinkedList;

public class RobotDataSet implements IDataSet, Serializable {
    LinkedList<double[]> x = new LinkedList<>();
    LinkedList<double[]> y = new LinkedList<>();
    int index = 0;
    int offset = 0;
    int windowSize;
    public RobotDataSet(int windowSize) {
        this.windowSize = windowSize;
    }

    public void addPattern(double[] input, double[] output) {
        x.add(input);
        y.add(output);
        if (x.size() > windowSize) {
            offset = 0;
            x.removeFirst();
            y.removeFirst();
        }
    }


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

    }

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

    @Override
    public DataPoint onlyReadNext() {
        if (index < x.size()) {
            return new DataPoint(x.get(index), y.get(index));
        }
        return null;
    }

    public int getSize() {
        if (x.size() < windowSize) return x.size();
        return windowSize;
    }

    public LinkedList<double[]> getX() {
        return x;
    }

    public LinkedList<double[]> getY() {
        return y;
    }
}
```

Listing 19: dataset/RobotDataSet.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;
    }

    @Override
    public DataPoint onlyReadNext() {
        if (index < x.length) {
            return new DataPoint(x[index], new double[]{y[index]});
        }
        return null;     }
}
```

Listing 20: dataset/XORBinaryDataSet.java

```java
package fa;

import representation.IRepresentable;

import java.io.FileNotFoundException;
import java.io.IOException;

public interface IFunctionApproximation {
    void train(IRepresentable input, double[] output);
    double[] eval(IRepresentable input);

    void save() throws IOException;

    void load() throws IOException, ClassNotFoundException;

    int getSize();
}
```

Listing 21: fa/IFunctionApproximation.java

```java
package fa;

import representation.IRepresentable;

import java.io.*;
import java.util.HashMap;

public class LUT implements IFunctionApproximation {

    private final String filePath;
//    int distance_level = 3;
//    int robot_energy_level = 3;
//    int enemy_energy_level = 3;
//    int position_level = 48;
    private HashMap StateMap = new HashMap();
    boolean readOnly;


    public LUT(String filePath, boolean readOnly) {
        this.filePath = filePath;
        this.readOnly = readOnly;
    }

    public void save(File argFile) {

    }


    public void load(String argFileName) throws IOException {

    }

    //LEFT HERE: finish the two methods below for implementation tomorrow morning.
    // for each unique state vector, generate its key and match it with a state object that contains 5 actions

//    public void initialiseLUT() {
//        for (int distance = 0; distance < distance_level; distance++) {
//            for (int robot_energy = 0; robot_energy < robot_energy_level; robot_energy++) {
//                for (int enemy_energy = 0; enemy_energy < enemy_energy_level; enemy_energy++) {
//                    for (int position = 0; position < position_level; position++) {
                        //double[] state_vector = {distance, robot_energy, enemy_energy, position};
                        //double key = indexFor(state_vector);
//                        State newState = new State(distance, robot_energy, enemy_energy, position);
                        // Q values are automatically set to 0 by default
//                        newState.addAll(); // add all actions for each state?

//                        StateMap.put(newState, newState);
//
//                    }
//                }
//            }
//        }
//    }



    public double train(double[] X, double argValue) {




        return 0;
    }

    @Override
    public void train(IRepresentable input, double[] output) {
        if (readOnly) {return;}
        double[] repr = input.toVector();
        System.out.print("train " + input + " to " + output[0]);
        System.out.println();
        this.StateMap.put(input, output);
    }

    @Override
    public double[] eval(IRepresentable input) {
        return (double[]) StateMap.getOrDefault(input, new double[]{ 0 });
    }

    @Override
    public void save() throws IOException {
        if (readOnly) return;
        new ObjectOutputStream(new FileOutputStream(this.filePath)).writeObject(StateMap);
    }
```

```java

    @Override
    public void load() throws IOException, ClassNotFoundException {
        this.StateMap = (HashMap) new ObjectInputStream(new FileInputStream(this.filePath)).readObject();
    }

    public int getSize() {
        return this.StateMap.size();
    }

    public HashMap getHashMap() {
        return this.StateMap;
    }
}
```

Listing 22: fa/LUT.java

```java
package fa;

import autograd.IInitializer;
import autograd.Parameter;
import autograd.UniformInitializer;
import dataset.DataPoint;
import dataset.IDataSet;
import dataset.RobotDataSet;
import jdk.jshell.spi.ExecutionControl;
import nn.*;
import optimization.GradientDescent;
import optimization.ILoss;
import optimization.IOptimizer;
import representation.IRepresentable;

import java.io.*;

// TODO Christina and Husna
public class NN implements IFunctionApproximation {
    Model model = Factory.createNeuralNetwork(
            new int[]{12, 15, 1},
            new BipolarSigmoid(),
            new UniformInitializer(-1, 1),
//            new UniformInitializer(0, 0),
            true,
            true
    );
    IOptimizer optimizer = new GradientDescent(0.0001, 0.9);
    ILayer activation;
    IInitializer initializer;
    int windowSize;
    RobotDataSet dataSet;
    ILoss loss = new MeanSquaredError(model.getOutput());
    int epochs = 1;
    double lossLimit = 0.000001;
    IFitCallback collector = new ConvergenceCollector();

    private final String filePath;
    boolean readOnly;

    public NN(String filePath, boolean readOnly) {
        // construct the neural network
        this.filePath = filePath;
        this.readOnly = readOnly;
        this.windowSize = 1;
    }

    public NN(String filePath, boolean readOnly, int windowSize) {
        // construct the neural network
        this.filePath = filePath;
        this.readOnly = readOnly;
        this.windowSize = windowSize;
    }
    @Override
    public void train(IRepresentable input, double[] output) {
        if (readOnly) {return;}
        System.out.print("train " + input + " to " + output[0]);
        System.out.println();
        // construct a single datapoint dataset out of the data point
        dataSet.addPattern(toInternalRepresentation(input.toVector()),
            output);
        System.out.println("SIZE: " + dataSet.getSize());
        // call fit on the neural network
        try {
            dataSet.reset();
            DataPoint next = dataSet.next();
            System.out.println("Desired: " + next.getY()[0]);
            System.out.println("Before fit: " + model.evaluate(next.getX())
                [0]);
            var totalLoss = model.fit(dataSet, optimizer, loss, epochs,
                lossLimit);
            System.out.println("After fit: " + model.evaluate(next.getX())
                [0]);
            System.out.println("LOSS: " + Math.sqrt(totalLoss / dataSet.
                getSize()));
        } catch (ExecutionControl.NotImplementedException e) {
            e.printStackTrace();
        }
        System.out.println("So many parameters " + model.
            getTrainableParameters().length);
        for (var p :
                model.getTrainableParameters()) {
            System.out.print(p.getValue() + " ");
        }
        System.out.println();
    }

    @Override
    public double[] eval(IRepresentable input) {
        // feed the input to the neural network and return the outcome as
            the q value
        double[] input_vector = input.toVector();
        return model.evaluate(toInternalRepresentation(input_vector));
    }

    private double[] toInternalRepresentation(double[] rawPattern) {
        return rawPattern;
//        double[] sixtyFourPattern = new double[64];
```

```java
//        int sixtyFourIndex = 0;
//        for (int i = 0; i < 3; i++) {
//            for (int j = 0; j < 4; j++) {
//                sixtyFourIndex += Math.pow(4, i) * j * rawPattern[i * 4 + j];
//            }
//        }
//        for (int i = 0; i < 64; i++) {
//            if (sixtyFourIndex == i) {
//                sixtyFourPattern[i] = 1;
//            } else {
//                sixtyFourPattern[i] = 0;
//            }
//        }
//        return sixtyFourPattern;
    }

    @Override
    public void save() throws IOException {
        // save all the parameters of the neural network, the weights
        if (readOnly) return;
        ObjectOutputStream stream = new ObjectOutputStream(new FileOutputStream(this.filePath));
        stream.writeObject(dataSet);
        for (Parameter param :
                model.getTrainableParameters()) {
            stream.writeObject(param);
        }
        // save the entire model, or just the weights?
        // need to record the different versions of weight along the way?
        stream.close();
    }

    @Override
    public void load() throws IOException, ClassNotFoundException {
        // load the weights of the neural network from the filePath

//        ObjectInputStream modelStream = new ObjectInputStream(new FileInputStream(this.getClass().getClassLoader().getResource("TrainedNN.obj").getPath()));
//        Parameter[] parameters = (Parameter[])(modelStream.readObject());
//        Parameter[] trainables = model.getTrainableParameters();
//        for (int i = 0; i < parameters.length; i++) {
//            trainables[i].setValue(parameters[i].getValue());
//        }
//        modelStream.close();

        dataSet = new RobotDataSet(windowSize);
        ObjectInputStream stream = new ObjectInputStream(new FileInputStream(this.filePath));
        dataSet = (RobotDataSet) stream.readObject();
        for (var param: model.getTrainableParameters()) {
            param.setValue(((Parameter) stream.readObject()).getValue());
        }
        stream.close();
    }

    @Override
    public int getSize() {
        return dataSet.getSize();
    }

    public double getLoss() throws ExecutionControl.NotImplementedException
        {
        IDataSet wrappedDataset = new IDataSet() {
            int index = 0;

            @Override
            public DataPoint next() {
                if (NN.this.dataSet.getX().size() <= index) {
                    return null;
                }
                DataPoint dataPoint = new DataPoint(NN.this.dataSet.getX().
                    get(index), NN.this.dataSet.getY().get(index));
                index++;
                return dataPoint;
            }

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

            @Override
            public DataPoint onlyReadNext() {
                return null;
            }
        };
        return model.fit(dataSet, new GradientDescent(0, 0), loss, 1, 0) /
            this.dataSet.getSize();
    }

    public RobotDataSet getDataSet() {
        return dataSet;
    }
}
```

Listing 23: fa/NN.java

```java
package fa;

import representation.IRepresentable;

import java.io.IOException;

public class NNLUT implements IFunctionApproximation {

    NN nn = new NN("NNTNinetyRobot.NN", false);
    LUT lut = new LUT("NNTNinetyRobot.LUT", false);
    public NNLUT() {

    }

    @Override
    public void train(IRepresentable input, double[] output) {
        nn.train(input, output);
        lut.train(input, output);
    }

    @Override
    public double[] eval(IRepresentable input) {
        double first = nn.eval(input)[0];
        double second = lut.eval(input)[0];
        System.out.println("NN " +first + " LUT " + second);
        return new double[] {second};
    }

    @Override
    public void save() throws IOException {
        nn.save();
        lut.save();
    }

    @Override
    public void load() throws IOException, ClassNotFoundException {
        nn.load();
        lut.load();
    }

    @Override
    public int getSize() {
        return 0;
    }
}
```

Listing 24: fa/NNLUT.java

```java
package nn;

import autograd.IVariable;
import autograd.Parameter;

public class BipolarSigmoid implements ILayer {

    Parameter scale;

    public BipolarSigmoid(double scale) {
        this.scale = new Parameter(scale, false);
    }

    public BipolarSigmoid() {
        this.scale = new Parameter(1., false);;
    }

    @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] = multiplication.apply(
                    addition.apply(
                            multiplication.apply(
                                scalar,
                                sigmoid.apply(input[i])),
                        constant
                    ),
                    this.scale
            );
        }
        return result;
    }
}
```

Listing 25: 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 26: nn/ConvergenceCollector.java

```java
package nn;

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

public class Factory {
    public static Model createNeuralNetwork(int[] sizes, ILayer activation, IInitializer initializer) {
        return createNeuralNetwork(sizes, activation, initializer, true, true);
    }
    public static Model createNeuralNetwork(int[] sizes, ILayer activation, IInitializer initializer, boolean lastActivation) {
        return createNeuralNetwork(sizes, activation, initializer, lastActivation, true);
    }

    public static Model createNeuralNetwork(int[] sizes, ILayer activation, IInitializer initializer, boolean lastActivation, boolean useBiases) {
        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, useBiases).apply(lastLayerOutput);
            if (i < sizes.length - 1 || lastActivation) {
                lastLayerOutput = activation.apply(lastLayerOutput);
            }
        }
        return new Model(inputs, lastLayerOutput);
    }

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

    public static Model createNeuralNetwork(int[] sizes, ILayer activation) {
        return createNeuralNetwork(sizes, activation, true);
    }
}
```

Listing 27: nn/Factory.java

```java
package nn;

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

Listing 28: nn/IFitCallback.java

```java
package nn;

import autograd.IVariable;

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

Listing 29: 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, boolean useBiases) {
        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());
            }
            if (useBiases)
                this.bias[i] = new Parameter(initializer.next());
            else
                this.bias[i] = new Parameter(0., false);
        }
    }

    @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 30: nn/Linear.java

```java
package nn;

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

public class MeanSquaredError implements IVariable, ILoss {

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

    public MeanSquaredError(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 31: nn/MeanSquaredError.java

```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 javax.swing.*;
import javax.xml.crypto.Data;
import java.io.Serializable;
import java.util.*;
import java.util.function.IntFunction;
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;
    }


    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.stream().sorted(Comparator.comparing(Parameter::
            getParameterId)).toArray(Parameter[]::new);
    }

    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 {
        return fit(dataSet, optimizer, loss, epochs, lossLimit, callback, true);
    }
    public double fit(IDataSet dataSet, IOptimizer optimizer, ILoss loss, int epochs, double lossLimit, IFitCallback callback, boolean online) 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();
                if (online) {
                    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);
                    }
                } else {
                    loss.backward(parameters, 1.);
                }
            }
            callback.collect(i, totalLoss);
            if (totalLoss < lossLimit) {
                break;
            }
            if (!online) {
                optimizer.update(parameters);
            }
        }
        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 32: nn/Model.java

```java
package nn;

import autograd.IVariable;

public class ReLU implements ILayer {

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

Listing 33: nn/ReLU.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 34: 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.);
//             System.out.println(p.getValue() + " " + delta);
            p.setValue(p.getValue() + delta);
            p.zeroGradient();
            lastDelta.put(p, delta);
        }
    }
}
```

Listing 35: optimization/GradientDescent.java

```java
package optimization;

import autograd.IVariable;

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

Listing 36: optimization/ILoss.java

```java
package optimization;

```

```java
import autograd.Parameter;

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

Listing 37: optimization/IOptimizer.java

```java
package policy;

import representation.IState;
import representation.States;

public class EnergyReward implements IPolicy {
    @Override
    public double getReward(IState run, IState last) {
        // should we give rewards on one state or on the change of last two states?
        States states = (States) run;
        States lastStates = (States) last;
//          return states.getMyEnergy() - states.getEnemyEnergy();

        return (lastStates.getEnemyEnergy() - states.getEnemyEnergy()) / 100.;
    }
}
```

Listing 38: policy/EnergyReward.java

```java
package policy;

import representation.IState;
import representation.States;

public class EnergyRewardTerminal implements IPolicy {
    @Override
    public double getReward(IState run, IState last) {
        // should we give rewards on one state or on the change of last two states?
        States states = (States) run;
        States lastStates = (States) last;
//          return states.getMyEnergy() - states.getEnemyEnergy();

        if (states.getEnemyEnergy() == 0)
            return 1;
        if (states.getMyEnergy() == 0)
            return -1;
        return 0;
    }
}
```

Listing 39: policy/EnergyRewardTerminal.java

```java
package policy;

import representation.Coordinates;
import representation.IState;

```

```java
public class GoTopRight implements IPolicy {
    @Override
    public double getReward(IState run, IState dummy) {
        Coordinates coordinates = (Coordinates) run;
        var x = coordinates.getX();
        var y = coordinates.getY();
        if (x == 7 && y == 5) {
            return 1;
        }
        return 0;
    }
}
```

Listing 40: policy/GoTopRight.java

```java
package policy;

import representation.IState;

public interface IPolicy {
    double getReward(IState currentState, IState lastState);
}
```

Listing 41: policy/IPolicy.java

```java
package representation;

public class Action {

    enum ActionName {FIRE,RIGHT,LEFT, AHEAD, BACK};
    // want an instance variable "action"
    ActionName action;
    double QValue = 0;

    public Action (String name) {

        switch (name) {
            case "fire":
                action = ActionName.FIRE;
                break;
            case "right":
                action = ActionName.RIGHT;
                break;
            case "left":
                action = ActionName.LEFT;
                break;
            case "ahead":
                action = ActionName.AHEAD;
                break;
            case "back":
                action = ActionName.BACK;
                break;
        }
    }
    public static void main(String[] args) {
        Action action = new Action("right");
        System.out.println(action.action);
    }


}
```

Listing 42: representation/Action.java

```java
package representation;

import java.io.Serializable;

public class Concatenation implements IRepresentable, Serializable {
    private IRepresentable first;
    private IRepresentable second;

    public Concatenation(IRepresentable state, IRepresentable action) {
        this.first = state;
        this.second = action;
    }
    @Override
    public double[] toVector() {
        double[] stateVector = first.toVector();
        double[] actionVector = second.toVector();
        double[] result = new double[stateVector.length + actionVector.length];
        System.arraycopy(stateVector, 0, result, 0, stateVector.length);
        System.arraycopy(actionVector, 0, result, stateVector.length, actionVector.length);
        return result;
    }

    @Override
    public int hashCode() {
        return first.hashCode() + second.hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof IRepresentable)) return false;
        var testRepr = ((IRepresentable) obj).toVector();
        double[] result = toVector();
        if (testRepr.length != result.length) return false;
        for (int i = 0; i < result.length; i++) {
            if (result[i] != testRepr[i]) return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return "Concatenation{" +
                "first=" + first +
                ", second=" + second +
                '}';
    }
}
```

Listing 43: representation/Concatenation.java

```java
package representation;

public class ConcatenationRepresentation implements IStateActionRepresentation {



    @Override
    public IRepresentable represent(IState state, IAction action) {
        return new Concatenation(state, action);
    }
}
```

Listing 44: representation/ConcatenationRepresentation.java

```java
package representation;

import java.io.Serializable;
import java.util.Objects;

public class Coordinates implements IState, Serializable {
    private int x;
    private int y;
    private int heading;

    public Coordinates(int x, int y, int heading) {
        setX(x);
        setY(y);
        setHeading(heading);
    }

    public void setX(int x) {
        this.x = x;
    }

    public void setY(int y) {
        this.y = y;
    }

    public void setHeading(int bearing) {
        this.heading = bearing;
    }

    @Override
    public IState clone() {
        return new Coordinates(this.x, this.y, this.heading);
    }

    public int getX() {
        return this.x;
    }

    @Override
    public double[] toVector() {
        return new double[] {x, y, heading};
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Coordinates that = (Coordinates) o;
        return x == that.x && y == that.y && heading == that.heading;
    }

    @Override
    public int hashCode() {
        return Objects.hash(x, y, heading);
    }

    public int getY() {
        return y;
    }

    @Override
    public String toString() {
        return "Coordinates{" +
                "x=" + x +
                ", y=" + y +
                ", heading=" + heading +
                '}';
    }
}
```

Listing 45: representation/Coordinates.java

```java
package representation;

import robocode.Event;
import robocode.ScannedRobotEvent;
import robocode.StatusEvent;

public class CoordinatesRepresentation implements IStateRepresentation {
    @Override
    public IState represent(IState state, Event event) {
        if (state == null) {
            state = new Coordinates(0, 0, 0);
        }
        Coordinates coordinates = (Coordinates) state.clone();
        if (event instanceof StatusEvent) {
            StatusEvent statusEvent = (StatusEvent) event;
            coordinates.setX((int) (statusEvent.getStatus().getX() / 100));
            coordinates.setY((int) (statusEvent.getStatus().getY() / 100));
            coordinates.setHeading((int) ((statusEvent.getStatus().
                getHeading() + 45) / 90));
        }
        return coordinates;
    }
}
```

Listing 46: representation/CoordinatesRepresentation.java

```java
package representation;

public interface IAction extends IRepresentable {

}
```

Listing 47: representation/IAction.java

```java
package representation;

import robocode.Robot;

public interface IActionRepresentation extends IRepresentation {
    void takeAction(Robot robot, IAction action);

    IAction[] getActions();
}
```

Listing 48: representation/IActionRepresentation.java

```java
package representation;

import java.io.Serializable;

public interface IRepresentable extends Serializable {
    double[] toVector();
}
```

Listing 49: representation/IRepresentable.java

```java
package representation;

import robocode.Robot;
import robocode.Event;

public interface IRepresentation {
}
```

Listing 50: representation/IRepresentation.java

```java
package representation;

public interface IStateActionRepresentation {
    IRepresentable represent(IState state, IAction action);
}
```

Listing 51: representation/IStateActionRepresentation.java

```java
package representation;

public interface IState extends IRepresentable {
    public IState clone();
}
```

Listing 52: representation/IState.java

```java
package representation;

import robocode.Event;

public interface IStateRepresentation extends IRepresentation {
    /**
```

```java
     * Evolves the robot state given the last state and the event
     * @param state the previous state of the robot
     * @param event the robot event containing changes to the state
     * @return returns a new state expressing the changed state
     */
    IState represent(IState state, Event event);
}
```

Listing 53: representation/IStateRepresentation.java

```java
package representation;

import java.io.Serializable;
import java.util.Objects;

public class Move implements IAction, Serializable {
    @Override
    public double[] toVector() {
        double value = 0;
        switch (actionType) {
            case AHEAD:
                value = 1;
                break;
            case TURN_LEFT:
                value = 2;
                break;
            case TURN_RIGHT:
                value = 3;
                break;
            default:
                assert false;
        }
        return new double[] { value };
    }

    @Override
    public String toString() {
        return "Move{" +
                "actionType=" + actionType +
                '}';
    }

    public enum ActionType {
        TURN_RIGHT,
        TURN_LEFT,
        AHEAD,
    }

    ActionType actionType;

    public Move(ActionType actionType) {
        this.actionType = actionType;
    }

    public ActionType getActionType() {return actionType;}

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Move move = (Move) o;
        return actionType == move.actionType;
    }

    @Override
    public int hashCode() {
        return Objects.hash(actionType);
    }
}
```

Listing 54: representation/Move.java

```java
package representation;

import robocode.Robot;

public class MoveRepresentation implements IActionRepresentation {
    @Override
    public void takeAction(Robot qLearningRobot, IAction action) {
        if (action == null) return;
        if (!(action instanceof Move)) {
            throw new IllegalArgumentException("Move representation can only take move actions.");
        }
        Move move = (Move) action;
        System.out.println("CASTED");
        if (move.getActionType() == Move.ActionType.TURN_LEFT) {
            qLearningRobot.turnLeft(90);
        } else if (move.getActionType() == Move.ActionType.TURN_RIGHT) {
            qLearningRobot.turnRight(90);
        } else if (move.getActionType() == Move.ActionType.AHEAD) {
            qLearningRobot.ahead(100);
        }
    }

    @Override
    public IAction[] getActions() {
        return new IAction[] {
            new Move(Move.ActionType.AHEAD),
            new Move(Move.ActionType.TURN_LEFT),
            new Move(Move.ActionType.TURN_RIGHT),
        };
    }
}
```

Listing 55: representation/MoveRepresentation.java

```java
package representation;

public class Representation {

}
```

Listing 56: representation/Representation.java

```java
package representation;
```

```java

import java.util.ArrayList;
import java.util.List;
import java.util.Objects;

public class State {

    // The states

    // the relative distance to the enemy (<200, <300, >=300)
    // our energy (<30, >=30, >100)
    // the enemy's energy (<30, >=30, >100)
    // x, y position of our own (step by 100)

    // The actions

    // fire(1)
    // turn right (90)
    // turn left (90)
    // go ahead (100)
    // go back (100)
    // do nothing

    private int distance;
    private int energy;
    private int enemyEnergy;
    private int x;
    private int y;
    private int enemyBearing;

    List<Action> actions = new ArrayList<Action>();

    public void add(Action a) {
        actions.add(a);
    }

    public void addAll() {
        actions.add(new Action("fire"));
        actions.add(new Action("right"));
        actions.add(new Action("left"));
        actions.add(new Action("ahead"));
        actions.add(new Action("back"));
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        State state = (State) o;
        return distance == state.distance && energy == state.energy &&
            enemyEnergy == state.enemyEnergy && x == state.x;
    }

    @Override
    public int hashCode() {
        return Objects.hash(distance, energy, enemyEnergy, x);
    }




}
```

Listing 57: representation/State.java

```java
package representation;
import robocode.*;

public class StateRep implements IStateRepresentation {
    public StateRep()  {}
    @Override
    // represent method will be called under two circumstances: either from onStatus or from onScannedRobot.
    // the event passed in can be of either StatusEvent or ScannedRobotEvent
    public IState represent(IState state, Event event) {
        //passed states are the last states, all null at first turn
        if (state == null) {
            state = new States(0, 0, 0, 0, 0 , 0, 0);
        }
        States states = (States) state.clone();//cast State to States

        if (event instanceof ScannedRobotEvent) {
            ScannedRobotEvent scannedEvent = (ScannedRobotEvent) event;
            states.setDistance((int) (scannedEvent.getDistance()));
            states.setEnemyEnergy((int) scannedEvent.getEnergy());
            states.setBearing((int)scannedEvent.getBearing());
        }
        if (event instanceof StatusEvent) {
            StatusEvent statusEvent = (StatusEvent) event;
            states.setX((int) statusEvent.getStatus().getX());
            states.setY((int) statusEvent.getStatus().getY());
            states.setHeading((int) statusEvent.getStatus().getHeading());
            states.setMyEnergy((int) statusEvent.getStatus().getEnergy());
        }
        if (event instanceof WinEvent) {
            states.setEnemyEnergy(0);
        }
        if (event instanceof DeathEvent) {
            states.setMyEnergy(0);
        }
        return states;
    }

}
```

Listing 58: representation/StateRep.java

```java
package representation;

import java.util.Arrays;

public class States implements IState{

    private int distance;
    private int x;
    private int y;
    private int heading;
    private int bearing;

    public void setBearing(int bearing) {
        this.bearing = bearing;
    }

    public int getBearing() {
        return bearing;
    }

    public enum energy {LOW, MEDIUM, HIGH};
    private int myEnergy;
    private int enemyEnergy;

    public States(int distance, int x, int y, int heading, int myEnergy, int enemyEnergy, int bearing) {
        setDistance(distance);
        setX(x);
        setY(y);
        setHeading(heading);
        setMyEnergy(myEnergy);
        setEnemyEnergy(enemyEnergy);
        setBearing(bearing);
    }

    public void setDistance(int distance) {
        this.distance = distance;
    }
    public void setX(int x) {
        this.x = x;
    }
    public void setY(int y) {this.y = y;}
    public void setHeading(int heading) {
        this.heading = heading;
    }
    public void setMyEnergy(int myEnergy) {
        this.myEnergy = myEnergy;
    }
    public void setEnemyEnergy(int enemyEnergy) {
        this.enemyEnergy = enemyEnergy;
    }

    @Override
    public IState clone() {
        return new States(this.distance, this.x, this.y, this.heading, this.myEnergy, this.enemyEnergy, this.bearing);
    }

    @Override
    public double[] toVector() {
        return new double[]{
//                this.x / 200,
//                this.y / 200,
//                (this.heading + 45) / 90,
//                (this.bearing + 45) / 90,
//                this.myEnergy / 40,
//                this.enemyEnergy / 40,
//                this.distance / 200,
        };
    }

    public int getDistance() {
        return distance;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public int getHeading() {
        return heading;
    }

    public int getMyEnergy() {
        return myEnergy;
    }

    public int getEnemyEnergy() {
        return enemyEnergy;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        States states = (States) o;
        double[] mine = toVector();
        double[] theirs = states.toVector();
        for (int i = 0; i < mine.length; i++) {
            if (mine[i] != theirs[i]) {
                return false;
            }
        }
        return true;
    }

    @Override
    public int hashCode() {
        return Arrays.hashCode(toVector());
    }

    @Override
    public String toString() {
        return "States{" +
                "distance=" + distance +
                ", x=" + x +
                ", y=" + y +
                ", heading=" + heading +
                ", bearing=" + bearing +
                ", myEnergy=" + myEnergy +
                ", enemyEnergy=" + enemyEnergy +
                '}';
    }
}
```

Listing 59: representation/States.java

```
package representation;

import java.io.Serializable;
import java.util.Objects;

public class TNinetyAction implements IAction, Serializable {
    @Override
    public double[] toVector() {
        var value = new double[] {0, 0, 0, 0};
        switch (actionType) {
            case AHEAD:
                value = new double[] {1, 0, 0, 0};
                break;
            case TURN_LEFT:
                value = new double[] {0, 1, 0, 0};
                break;
            case TURN_RIGHT:
                value = new double[] {0, 0, 1, 0};
                break;
            case FIRE:
                value = new double[] {0, 0, 0, 1};
                break;
            default:
                assert false;
        }
        return value;
    }

    @Override
    public String toString() {
        return "Move{" +
                "actionType=" + actionType +
                '}';
    }

    public enum ActionType {
        TURN_RIGHT,
        TURN_LEFT,
        AHEAD,
        FIRE,
        RANDOMLY_MOVE,
    }

    ActionType actionType;

    public TNinetyAction(ActionType actionType) {
        this.actionType = actionType;
    }

    public ActionType getActionType() {return actionType;}

```

```java
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        TNinetyAction that = (TNinetyAction) o;
        return actionType == that.actionType;
    }

    @Override
    public int hashCode() {
        return Objects.hash(actionType);
    }
}
```

Listing 60: representation/TNinetyAction.java

```java
package representation;

import robocode.Robot;

import java.util.Random;

public class TNinetyActionRepresentation implements IActionRepresentation {
    @Override
    public void takeAction(Robot qLearningRobot, IAction action) {
        if (action == null) {
            System.out.println("Action is null");
            return;
        }
        if (!(action instanceof TNinetyAction)) {
            throw new IllegalArgumentException("TNinety representation can only take TNinety actions.");
        }
        TNinetyAction move = (TNinetyAction) action;
        if (move.getActionType() == TNinetyAction.ActionType.TURN_LEFT) {
            qLearningRobot.turnLeft(90);
        } else if (move.getActionType() == TNinetyAction.ActionType.TURN_RIGHT) {
            qLearningRobot.turnRight(90);
        } else if (move.getActionType() == TNinetyAction.ActionType.AHEAD) {
            qLearningRobot.ahead(100);
        } else if (move.getActionType() == TNinetyAction.ActionType.FIRE) {
            qLearningRobot.fire(18);
        } else if (move.getActionType() == TNinetyAction.ActionType.RANDOMLY_MOVE) {
            MoveRepresentation moveRepresentation = new MoveRepresentation();
            System.out.println("TAKING RANDOM");
            moveRepresentation.takeAction(qLearningRobot, moveRepresentation.getActions()[new Random().nextInt(3)]);
        }
    }

    @Override
    public IAction[] getActions() {
        return new IAction[] {
            new TNinetyAction(TNinetyAction.ActionType.AHEAD),
            new TNinetyAction(TNinetyAction.ActionType.TURN_LEFT),
            new TNinetyAction(TNinetyAction.ActionType.TURN_RIGHT),
            new TNinetyAction(TNinetyAction.ActionType.FIRE),
//              new TNinetyAction(TNinetyAction.ActionType.RANDOMLY_MOVE),
        };
    }
}
```

Listing 61: representation/TNinetyActionRepresentation.java

```
package rl;

import fa.IFunctionApproximation;
import fa.LUT;
import policy.IPolicy;
import representation.*;

public interface ILearning {
    IAction takeStep(IState lastState, IAction lastAction, IState currentState);

    IStateRepresentation getStateRepresentation();
    IActionRepresentation getActionRepresentation();

    IPolicy getPolicy();

    IFunctionApproximation getFunctionApproximation();
}
```

Listing 62: rl/ILearning.java

```
package rl;

import fa.IFunctionApproximation;
import org.jetbrains.annotations.NotNull;
import policy.IPolicy;
import representation.*;

import java.util.ArrayList;
import java.util.Random;

public class QLearning implements ILearning {

    private double epsilon;
    private final double alpha;
    private final double gamma;
    private final Random random;
    private IStateRepresentation stateRepresentation;
    private IActionRepresentation actionRepresentation;
    private IPolicy policy;
    private IFunctionApproximation functionApproximation;
    private IStateActionRepresentation stateActionRepresentation;
    private int depth;
    private boolean onlineLearning = false;
    private ArrayList<IRepresentable> history = new ArrayList<>();
    private ConcatenationRepresentation concatenation = new ConcatenationRepresentation();

```

```java
    public QLearning(IStateRepresentation stateRepresentation,
                     IActionRepresentation actionRepresentation,
                     IStateActionRepresentation stateActionRepresentation,
                     IPolicy policy, IFunctionApproximation functionApproximation, double epsilon, double alpha, double gamma) {
        this.stateActionRepresentation = stateActionRepresentation;
        this.stateRepresentation = stateRepresentation;
        this.actionRepresentation = actionRepresentation;
        this.policy = policy;
        this.epsilon = epsilon;
        this.alpha = alpha;
        this.gamma = gamma;
        this.random = new Random();
        this.functionApproximation = functionApproximation;
        this.depth = 1;
        this.onlineLearning = false;
    }


    public QLearning(IStateRepresentation stateRepresentation,
                     IActionRepresentation actionRepresentation,
                     IStateActionRepresentation stateActionRepresentation,
                     IPolicy policy, IFunctionApproximation functionApproximation,
                     double epsilon, double alpha, double gamma, int depth, boolean onlineLearning) {        this.epsilon = epsilon;
        this.alpha = alpha;
        this.gamma = gamma;
        this.stateRepresentation = stateRepresentation;
        this.actionRepresentation = actionRepresentation;
        this.policy = policy;
        this.functionApproximation = functionApproximation;
        this.stateActionRepresentation = stateActionRepresentation;
        this.depth = depth;
        this.random = new Random();
        this.onlineLearning = onlineLearning;
    }

    @Override
    public IAction takeStep(IState lastState, IAction lastAction, IState currentState) {
        if (lastState == null || lastAction == null || currentState == null) {
            return explore();
        }
        System.out.println("Current state " + currentState);
        IRepresentable oldSA = stateActionRepresentation.represent(
                lastState,
                lastAction
        );
        history.add(oldSA);
        while (history.size() > depth) {
            history.remove(0);
        }
        if (history.size() < depth) return explore();
        IAction bestAction = exploit(currentState);
        IAction backupAction;
        IAction toTakeAction;
        double randomDouble = this.random.nextDouble();
        boolean explored = false;
        if (randomDouble < this.epsilon) {
            IAction action = explore();
            backupAction = action;
            toTakeAction = action;
            explored = true;
        } else {
            backupAction = bestAction;
            toTakeAction = bestAction;
        }
        if (!onlineLearning) {
            backupAction = bestAction;
        }

        for (int i = 0; i < history.size() - 1; i++) {
            oldSA = new Concatenation(oldSA, history.get(i));
        }
        IRepresentable currentAction = stateActionRepresentation.represent(currentState, backupAction);
        for (int i = 1; i < history.size(); i++) {
            currentAction = new Concatenation(currentAction, history.get(i));
        }

        if (explored) {
            logAction(currentAction, "explored");
        } else {
            logAction(currentAction, "exploited");
        }

        double oldQ = functionApproximation.eval(oldSA)[0];
        double r = policy.getReward(currentState, lastState); // why would evaluate Rewards for last state?


        double currentQ = functionApproximation.eval(currentAction)[0];
        System.out.println("train " + oldQ + " = " + oldQ + " + " + alpha + " (" + r + " + " + gamma + " * " + currentQ  + " - " + oldQ + ")");
        System.out.println("train " + oldSA + " " + backupAction);
        functionApproximation.train(
                oldSA,
                new double[]{
                        oldQ + alpha * (r + gamma * currentQ - oldQ)
                }
        );
        return toTakeAction;
    }

    private IAction exploit(IState currentState) {
        IAction bestAction = actionRepresentation.getActions()[0];
        IRepresentable stateAction = stateActionRepresentation.represent(currentState, bestAction);
        for (int i = 1; i < history.size(); i++) {
            stateAction = new Concatenation(stateAction, history.get(i));
        }
        double bestQ = functionApproximation.eval(stateAction)[0];

        for (IAction action: actionRepresentation.getActions()) {
            stateAction = stateActionRepresentation.represent(
                    currentState, action
            );
            for (int i = 1; i < history.size(); i++) {
                stateAction = new Concatenation(stateAction, history.get(i));
            }
            double q = functionApproximation.eval(
                    stateAction)[0];
            System.out.print("    " + q + " " + action);
            System.out.println();
            if (q > bestQ) {
                bestAction = action;
                bestQ = q;
            }
        }
        return bestAction;
    }

    private void logAction(IRepresentable bestAction, String hint) {
        if (bestAction == null) return;
        System.out.print(bestAction);
        System.out.print(" " + hint + " ");
        System.out.println(functionApproximation.eval(bestAction)[0]);
    }

    private IAction explore() {
        IAction[] actions = actionRepresentation.getActions();
        IAction action = actions[getRandom().nextInt(actions.length)];
        return action;
    }

    @NotNull
    private Random getRandom() {
        return this.random;
    }

    @Override
    public IStateRepresentation getStateRepresentation() {
        return stateRepresentation;
    }

    @Override
    public IActionRepresentation getActionRepresentation() {
        return actionRepresentation;
    }

    @Override
    public IPolicy getPolicy() {
        return policy;
    }

    @Override
    public IFunctionApproximation getFunctionApproximation() {
        return functionApproximation;
    }

    public void setEpsilon(double epsilon) {
        this.epsilon = epsilon;
    }

    public double getEpsilon() {
        return this.epsilon;
    }
}
```

Listing 63: rl/QLearning.java

```java
package rl;

import fa.IFunctionApproximation;
import org.jetbrains.annotations.NotNull;
import policy.IPolicy;
import representation.*;

import java.util.Random;

public class SARSALearning implements ILearning {

    private double epsilon;
    private final double alpha;
    private final double gamma;
    private final Random random;
    private IStateRepresentation stateRepresentation;
    private IActionRepresentation actionRepresentation;
    private IPolicy policy;
    private IFunctionApproximation functionApproximation;
    private IStateActionRepresentation stateActionRepresentation;

    public SARSALearning(IStateRepresentation stateRepresentation,
                         IActionRepresentation actionRepresentation,
                         IStateActionRepresentation stateActionRepresentation,
                         IPolicy policy, IFunctionApproximation functionApproximation, double epsilon, double alpha, double gamma) {
        this.stateActionRepresentation = stateActionRepresentation;
        this.stateRepresentation = stateRepresentation;
        this.actionRepresentation = actionRepresentation;
        this.policy = policy;
        this.epsilon = epsilon;
        this.alpha = alpha;
        this.gamma = gamma;
        this.random = new Random();
        this.functionApproximation = functionApproximation;
    }

    @Override
    public IAction takeStep(IState lastState, IAction lastAction, IState currentState) {
        if (lastState == null || lastAction == null || currentState == null) {
            return explore();}
        System.out.println("Current state " + currentState);
        IRepresentable oldSA = stateActionRepresentation.represent(
                lastState,
                lastAction);
        double oldQ = functionApproximation.eval(oldSA)[0];
        double r = policy.getReward(currentState, lastState); // why would evaluate Rewards for last state?
        double newQ;

        if (this.random.nextDouble() < this.epsilon) {
            IAction action = explore();
            IRepresentable exploreSA = stateActionRepresentation.represent(
                    currentState,
                    action);
            newQ = functionApproximation.eval(exploreSA)[0];
            System.out.println("train " + oldQ + " = " + oldQ + " + " + alpha + " (" + r + " + " + gamma + " * " + newQ  + " - " + oldQ + ")");
            System.out.println("train " + oldSA + " " + action);
            functionApproximation.train(
                    oldSA,
                    new double[]{
                            oldQ + alpha * (r + gamma * newQ - oldQ)
                    });
            logAction(currentState, action, "explored");
            return action;
        } else {
            IAction bestAction = exploit(currentState);
            IRepresentable currentBest = stateActionRepresentation.represent(
                    currentState,
                    bestAction);
            newQ = functionApproximation.eval(currentBest)[0];
            System.out.println("train " + oldQ + " = " + oldQ + " + " + alpha + " (" + r + " + " + gamma + " * " + newQ  + " - " + oldQ + ")");
            System.out.println("train " + oldSA + " " + bestAction);
            functionApproximation.train(
                    oldSA,
                    new double[]{
                            oldQ + alpha * (r + gamma * newQ - oldQ)
                    });
            logAction(currentState, bestAction, "exploited");
            return bestAction;
        }
    }

    private IAction exploit(IState currentState) {
        double bestQ = 0;
        IAction bestAction = actionRepresentation.getActions()[0];
        for (IAction action: actionRepresentation.getActions()) {
            double q = functionApproximation.eval(
                    stateActionRepresentation.represent(
                            currentState, action
                    ))[0];
            System.out.print("    " + q + " " + action);
            System.out.println();
            if (q > bestQ) {
                bestAction = action;
                bestQ = q;
            }
        }
        return bestAction;
    }

    private void logAction(IState currentState, IAction bestAction, String hint) {
        if (bestAction == null) return;
        System.out.print(bestAction);
        System.out.print(" " + hint + " ");
        System.out.println(functionApproximation.eval(
                stateActionRepresentation.represent(
                        currentState, bestAction
                ))[0]);
    }

    private IAction explore() {
        IAction[] actions = actionRepresentation.getActions();
        IAction action = actions[getRandom().nextInt(actions.length)];
        return action;
    }

    @NotNull
    private Random getRandom() {
        return this.random;
    }

    @Override
    public IStateRepresentation getStateRepresentation() {
        return stateRepresentation;
    }

    @Override
    public IActionRepresentation getActionRepresentation() {
        return actionRepresentation;
    }

    @Override
    public IPolicy getPolicy() {
        return policy;
    }

    @Override
    public IFunctionApproximation getFunctionApproximation() {
        return functionApproximation;
    }

    public void setEpsilon(double epsilon) {
        this.epsilon = epsilon;
    }

    public double getEpsilon() {
        return this.epsilon;
    }
}
```

Listing 64: rl/SARSALearning.java

```java
/*
 * Copyright (c) 2001-2021 Mathew A. Nelson and Robocode contributors
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v1.0
 * which accompanies this distribution, and is available at
 * https://robocode.sourceforge.io/license/epl-v10.html
 */
package robot;


import robocode.DeathEvent;
import robocode.Robot;
import robocode.ScannedRobotEvent;
import static robocode.util.Utils.normalRelativeAngleDegrees;

import java.awt.*;


/**
 * Corners - a sample robot by Mathew Nelson.
 * <p>
 * This robot moves to a corner, then swings the gun back and forth.
 * If it dies, it tries a new corner in the next round.
 *
 * @author Mathew A. Nelson (original)
 * @author Flemming N. Larsen (contributor)
 */
public class Corners extends Robot {
    int others; // Number of other robots in the game
    static int corner = 0; // Which corner we are currently using
    // static so that it keeps it between rounds.
    boolean stopWhenSeeRobot = false; // See goCorner()

    /**
     * run:  Corners' main run function.
     */
    public void run() {
        // Set colors
        setBodyColor(Color.red);
        setGunColor(Color.black);
        setRadarColor(Color.yellow);
        setBulletColor(Color.green);
        setScanColor(Color.green);

        // Save # of other bots
        others = getOthers();

        // Move to a corner
        goCorner();

        // Initialize gun turn speed to 3
        int gunIncrement = 3;

        // Spin gun back and forth
        while (true) {
            for (int i = 0; i < 30; i++) {
                turnGunLeft(gunIncrement);
            }
            gunIncrement *= -1;
        }
    }

    /**
     * goCorner:  A very inefficient way to get to a corner.  Can you do better?
     */
    public void goCorner() {
        // We don't want to stop when we're just turning...
        stopWhenSeeRobot = false;
        // turn to face the wall to the "right" of our desired corner.
        turnRight(normalRelativeAngleDegrees(corner - getHeading()));
        // Ok, now we don't want to crash into any robot in our way...
        stopWhenSeeRobot = true;
        // Move to that wall
        ahead(5000);
        // Turn to face the corner
        turnLeft(90);
        // Move to the corner
        ahead(5000);
        // Turn gun to starting point
        turnGunLeft(90);
    }

    /**
     * onScannedRobot:  Stop and fire!
     */
    public void onScannedRobot(ScannedRobotEvent e) {
        // Should we stop, or just fire?
        if (stopWhenSeeRobot) {
            // Stop everything!  You can safely call stop multiple times.
            stop();
            // Call our custom firing method
            smartFire(e.getDistance());
            // Look for another robot.
            // NOTE:  If you call scan() inside onScannedRobot, and it sees a robot,
            // the game will interrupt the event handler and start it over
            scan();
            // We won't get here if we saw another robot.
            // Okay, we didn't see another robot... start moving or turning again.
            resume();
        } else {
            smartFire(e.getDistance());
        }
    }

    /**
     * smartFire:  Custom fire method that determines firepower based on distance.
     *
     * @param robotDistance the distance to the robot to fire at
     */
    public void smartFire(double robotDistance) {
        if (robotDistance > 200 || getEnergy() < 15) {
            fire(1);
        } else if (robotDistance > 50) {
            fire(2);
        } else {
            fire(3);
        }
    }

    /**
     * onDeath:  We died.  Decide whether to try a different corner next game.
     */
    public void onDeath(DeathEvent e) {
        // Well, others should never be 0, but better safe than sorry.
        if (others == 0) {
            return;
        }

        // If 75% of the robots are still alive when we die, we'll switch corners.
        if ((others - getOthers()) / (double) others < .75) {
            corner += 90;
            if (corner == 270) {
                corner = -90;
            }
            out.println("I died and did poorly... switching corner to " + corner);
        } else {
            out.println("I died but did well.  I will still use corner " + corner);
        }
    }
}
```

Listing 65: robot/Corners.java

```java
package robot;

import fa.IFunctionApproximation;
import fa.LUT;
import policy.EnergyReward;
import policy.IPolicy;
import representation.*;
import rl.ILearning;
import rl.QLearning;
import robocode.ScannedRobotEvent;

public class LUTTNinetyRobot05 extends QLearningRobot {
    public LUTTNinetyRobot05() {
        super(createLearning());
    }


    public static ILearning createLearning() {
        IActionRepresentation actionRepresentation = new TNinetyActionRepresentation();
        IStateRepresentation stateRepresentation = new StateRep();
        IFunctionApproximation functionApproximation = new LUT("LUTTNinetyRobot.obj", false);
        IPolicy policy = new EnergyReward();
        return new QLearning(
                stateRepresentation,
                actionRepresentation,
                new ConcatenationRepresentation(),
                policy,
                functionApproximation,
                0.5, 0.1, 0.9, 3, false);
    }

    @Override
    public void onScannedRobot(ScannedRobotEvent event) {
        super.onScannedRobot(event);
        System.out.println("HELLLLLOOOOOOO");
    }
}
```

Listing 66: robot/LUTTNinetyRobot05.java

```java
package robot;

import fa.IFunctionApproximation;
import fa.LUT;
import policy.EnergyReward;
import policy.IPolicy;
import representation.*;
import rl.ILearning;
import rl.QLearning;
import robocode.ScannedRobotEvent;

public class LUTTNinetyRobot0 extends QLearningRobot {
    public LUTTNinetyRobot0() {
        super(createLearning());
    }


    public static ILearning createLearning() {
        IActionRepresentation actionRepresentation = new TNinetyActionRepresentation();
        IStateRepresentation stateRepresentation = new StateRep();
        IFunctionApproximation functionApproximation = new LUT("LUTTNinetyRobot.obj", false);
        IPolicy policy = new EnergyReward();
        return new QLearning(
                stateRepresentation,
                actionRepresentation,
                new ConcatenationRepresentation(),
                policy,
                functionApproximation,
                0.0, 0.1, 0.9, 3, false);
    }

    @Override
    public void onScannedRobot(ScannedRobotEvent event) {
        super.onScannedRobot(event);
        System.out.println("HELLLLLOOOOOOO");
    }
}
```

Listing 67: robot/LUTTNinetyRobot0.java

```java
package robot;

import fa.IFunctionApproximation;
import fa.LUT;
import policy.EnergyReward;
import policy.IPolicy;
import representation.*;
import rl.ILearning;
import rl.QLearning;
import robocode.ScannedRobotEvent;

public class LUTTNinetyRobotConfident extends QLearningRobot {
    public LUTTNinetyRobotConfident() {
        super(createLearning());
    }


    public static ILearning createLearning() {
        IActionRepresentation actionRepresentation = new
            TNinetyActionRepresentation();
        IStateRepresentation stateRepresentation = new StateRep();
        IFunctionApproximation functionApproximation = new LUT("
            LUTTNinetyRobot.obj", true);
        IPolicy policy = new EnergyReward();
        return new QLearning(
                stateRepresentation,
                actionRepresentation,
                new ConcatenationRepresentation(),
                policy,
                functionApproximation,
                0.05, 0.1, 0.8, 3, false);
    }

    @Override
    public void onScannedRobot(ScannedRobotEvent event) {
        super.onScannedRobot(event);
        System.out.println("HELLLLLOOOOOOO");
    }
}
```

Listing 68: robot/LUTTNinetyRobotConfident.java

```java
package robot;

import fa.IFunctionApproximation;
import fa.LUT;
import policy.EnergyReward;
import policy.GoTopRight;
import policy.IPolicy;
import representation.*;
import rl.ILearning;
import rl.QLearning;
import robocode.ScannedRobotEvent;

public class LUTTNinetyRobot extends QLearningRobot {
    public LUTTNinetyRobot() {
        super(createLearning());
    }


    public static ILearning createLearning() {
        IActionRepresentation actionRepresentation = new
            TNinetyActionRepresentation();
        IStateRepresentation stateRepresentation = new StateRep();
        IFunctionApproximation functionApproximation = new LUT("
            LUTTNinetyRobot.obj", false);
        IPolicy policy = new EnergyReward();
        return new QLearning(
                stateRepresentation,
                actionRepresentation,
                new ConcatenationRepresentation(),
                policy,
                functionApproximation,
                0.8, 0.1, 0.9, 3, false);
    }

    @Override
    public void onScannedRobot(ScannedRobotEvent event) {
        super.onScannedRobot(event);
        System.out.println("HELLLLLOOOOOOO");
    }
}
```

Listing 69: robot/LUTTNinetyRobot.java

```java
package robot;

import fa.IFunctionApproximation;
import fa.LUT;
import policy.EnergyReward;
import policy.IPolicy;
import representation.*;
import rl.ILearning;
import rl.QLearning;
import robocode.ScannedRobotEvent;

public class LUTTNinetyRobotOnline extends QLearningRobot {
    public LUTTNinetyRobotOnline() {
        super(createLearning());
    }


    public static ILearning createLearning() {
        IActionRepresentation actionRepresentation = new
            TNinetyActionRepresentation();
        IStateRepresentation stateRepresentation = new StateRep();
        IFunctionApproximation functionApproximation = new LUT("
            LUTTNinetyRobot.obj", false);
        IPolicy policy = new EnergyReward();
        return new QLearning(
                stateRepresentation,
                actionRepresentation,
                new ConcatenationRepresentation(),
                policy,
                functionApproximation,
                0.8, 0.1, 0.9, 3, true);
    }

    @Override
    public void onScannedRobot(ScannedRobotEvent event) {
        super.onScannedRobot(event);
        System.out.println("HELLLLLOOOOOOO");
    }
}
```

Listing 70: robot/LUTTNinetyRobotOnline.java

```
package robot;

import fa.IFunctionApproximation;
import fa.LUT;
import policy.EnergyReward;
import policy.EnergyRewardTerminal;
import policy.IPolicy;
import representation.*;
import rl.ILearning;
import rl.QLearning;
import robocode.ScannedRobotEvent;

public class LUTTNinetyRobotTerminal extends QLearningRobot {
    public LUTTNinetyRobotTerminal() {
        super(createLearning());
    }


    public static ILearning createLearning() {
        IActionRepresentation actionRepresentation = new
            TNinetyActionRepresentation();
        IStateRepresentation stateRepresentation = new StateRep();
        IFunctionApproximation functionApproximation = new LUT("
            LUTTNinetyRobot.obj", false);
        IPolicy policy = new EnergyRewardTerminal();
        return new QLearning(
                stateRepresentation,
                actionRepresentation,
                new ConcatenationRepresentation(),
                policy,
                functionApproximation,
                0.8, 0.1, 0.9, 3, false);
    }

    @Override
    public void onScannedRobot(ScannedRobotEvent event) {
        super.onScannedRobot(event);
        System.out.println("HELLLLLOOOOOOO");
    }
}
```

Listing 71: robot/LUTTNinetyRobotTerminal.java

```java
package robot;

/**
 * DISCLAIMER: the code below has been auto-generated by Robocde
 *   http://robocode.sourceforge.net/
 */

import robocode.*;

// API help : http://robocode.sourceforge.net/docs/robocode/robocode/Robot.html

/**
 * MyFirstRobot - a robot by (your name here)
 */
public class MyFirstRobot extends AdvancedRobot
{
    /**
     * run: MyFirstRobot's default behavior
     */
    public void run() {
        // Initialization of the robot should be put here

        // After trying out your robot, try uncommenting the import at the top,
        // and the next line:

        // setColors(Color.red,Color.blue,Color.green); // body,gun,radar

        // Robot main loop
        while(true) {
            // Replace the next 4 lines with any behavior you would like
            ahead(100);
            turnGunRight(360);
            back(100);
            turnGunRight(360);
        }
    }

    /**
     * onScannedRobot: What to do when you see another robot
     */
    public void onScannedRobot(ScannedRobotEvent e) {
        // Replace the next line with any behavior you would like
        fire(1);
    }

    /**
     * onHitByBullet: What to do when you're hit by a bullet
     */
    public void onHitByBullet(HitByBulletEvent e) {
        // Replace the next line with any behavior you would like
        back(10);
    }

    /**
     * onHitWall: What to do when you hit a wall
     */
    public void onHitWall(HitWallEvent e) {
        // Replace the next line with any behavior you would like
        back(20);
    }
}
```

Listing 72: robot/MyFirstRobot.java

```
package robot;

import fa.IFunctionApproximation;
import fa.NN;
import policy.EnergyReward;
import policy.IPolicy;
import representation.*;
import rl.ILearning;
import rl.QLearning;

public class NNTNinetyRobotConfident extends QLearningRobot {
    public NNTNinetyRobotConfident() {
        super(createLearning());
    }


    public static ILearning createLearning() {
        IActionRepresentation actionRepresentation = new
            TNinetyActionRepresentation();
        IStateRepresentation stateRepresentation = new StateRep();
        IFunctionApproximation functionApproximation = new NN("
            NNTNinetyRobot.obj", true);
        IPolicy policy = new EnergyReward();
        return new QLearning(
                stateRepresentation,
                actionRepresentation,
                new ConcatenationRepresentation(),
                policy,
                functionApproximation,
                0.05, 0.1, 0.9, 3, false);
    }

}
```

Listing 73: robot/NNTNinetyRobotConfident.java

```
package robot;

import fa.IFunctionApproximation;
import fa.LUT;
import fa.NN;
import fa.NNLUT;
import policy.EnergyReward;
import policy.IPolicy;
import representation.*;
import rl.ILearning;
import rl.QLearning;
import robocode.Robot;

```

```java
public class NNTNinetyRobot extends QLearningRobot {
    public NNTNinetyRobot() {
        super(createLearning());
    }


    public static ILearning createLearning() {
        IActionRepresentation actionRepresentation = new TNinetyActionRepresentation();
        IStateRepresentation stateRepresentation = new StateRep();
        IFunctionApproximation functionApproximation = new NN("NNTNinetyRobot.obj", false , 20);
//        IFunctionApproximation functionApproximation = new NNLUT();
//        IFunctionApproximation functionApproximation = new LUT("NNTNinetyRobot.obj", false);
        IPolicy policy = new EnergyReward();
        return new QLearning(
                stateRepresentation,
                actionRepresentation,
                new ConcatenationRepresentation(),
                policy,
                functionApproximation,
                0.8, 0.1, 0.9, 3, false);
    }

}
```

Listing 74: robot/NNTNinetyRobot.java

```java
package robot;

import representation.IAction;
import representation.IState;
import rl.ILearning;
import robocode.*;

import java.io.IOException;

public class QLearningRobot extends Robot {
    private ILearning learning;
    private IState state;
    private IState lastState;
    private IAction lastAction;
    private long lastTurn;
    private int turn = 0;
    private StatusEvent lastStatusEvent;

    @Override
    public void run() {
        super.run();
//        setAdjustGunForRobotTurn(true);
//        setAdjustRadarForGunTurn(true);
        while (true) {
            // Replace the next 4 lines with any behavior you would like
//            ahead(100);
//            turnGunRight(360);
//            back(100);
//            turnRadarLeft(360);
```

```
            turnGunRight(360);
            turn ++;
        }
    }

    public int getTurn() {
        return turn;
    }

    public QLearningRobot(ILearning learning) {
        this.learning = learning;
        try {
            this.learning.getFunctionApproximation().load();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    public ILearning getLearning() {
        return learning;
    }


    private IState getLastState() {
        return lastState;
    }

    public IState getState() {
        return this.state;
    }

    @Override
    public void onScannedRobot(ScannedRobotEvent event) {
        super.onScannedRobot(event);
        processEvent(event);
    }

    private void processEvent(Event event) {
        if (event instanceof ScannedRobotEvent && this.getTurn() > this.
           lastTurn || event instanceof WinEvent || event instanceof
           DeathEvent) {
            IState newState = learning.getStateRepresentation().represent(
               getState(), lastStatusEvent);
            newState = learning.getStateRepresentation().represent(newState
               , event);
            //set current state
            setState(newState);
            IAction action = learning.takeStep(getLastState(),
               getLastAction(), getState());
            //take new action
            this.lastTurn = this.getTurn();
            System.out.println("Turn " + this.lastTurn);
            takeAction(action);
        }
    }

```

```java
    @Override
    public void onWin(WinEvent event) {
        super.onWin(event);
        processEvent(event);
    }

    @Override
    public void onDeath(DeathEvent event) {
        super.onDeath(event);
        processEvent(event);
    }

    private IAction getLastAction() {
        return lastAction;
    }

    private void takeAction(IAction action) {
        learning.getActionRepresentation().takeAction(this, action);
        if (action != null) lastAction = action;
    }

    public void setState(IState state) {
        this.lastState = this.state;
        this.state = state;
    }

    @Override
    public void onRoundEnded(RoundEndedEvent event) {
        try {
            learning.getFunctionApproximation().save();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void onStatus(StatusEvent e) {
        lastStatusEvent = e;
    }
}
```

Listing 75: robot/QLearningRobot.java

```java
package robot;

import robocode.Robot;
import robocode.StatusEvent;

public class TopLeftCornerRobot extends Robot {
    @Override
    public void run() {
        super.run();
        ahead(100);
    }

    @Override
    public void onStatus(StatusEvent e) {
        super.onStatus(e);
        var yDistance = getBattleFieldHeight() - getY();
        var xDistance = getX();
        var angle = Math.atan(yDistance / xDistance);
        angle = 270 + Math.toDegrees(angle);
        System.out.printf("%f, %f, %f, %f %n", getY(), getX(), getHeading()
            , angle);
        double absHeadingDiff = Math.abs(getHeading() - angle);
        if (getHeading() < angle) {
            if (absHeadingDiff > 180) {
                turnLeft(360 - absHeadingDiff);
            } else {
                turnRight(absHeadingDiff);
            }
        } else {
            if (absHeadingDiff > 180) {
                turnRight(360 - absHeadingDiff);
            } else {
                turnLeft(absHeadingDiff);
            }
        }
        if (absHeadingDiff < 0.01) {
            ahead(100);
        }
    }
}
```

Listing 76: robot/TopLeftCornerRobot.java

```java
/*
 * Copyright (c) 2001-2021 Mathew A. Nelson and Robocode contributors
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v1.0
 * which accompanies this distribution, and is available at
 * https://robocode.sourceforge.io/license/epl-v10.html
 */
package robot;


import robocode.HitRobotEvent;
import robocode.Robot;
import robocode.ScannedRobotEvent;
import robocode.WinEvent;
import static robocode.util.Utils.normalRelativeAngleDegrees;

import java.awt.*;


/**
 * Tracker - a sample robot by Mathew Nelson.
 * <p>
 * Locks onto a robot, moves close, fires when close.
 *
 * @author Mathew A. Nelson (original)
 * @author Flemming N. Larsen (contributor)
 */
public class Tracker extends Robot {
    int count = 0; // Keeps track of how long we've
    // been searching for our target
    double gunTurnAmt; // How much to turn our gun when searching
    String trackName; // Name of the robot we're currently tracking

    /**
     * run:  Tracker's main run function
     */
    public void run() {
        // Set colors
        setBodyColor(new Color(128, 128, 50));
        setGunColor(new Color(50, 50, 20));
        setRadarColor(new Color(200, 200, 70));
        setScanColor(Color.white);
        setBulletColor(Color.blue);

        // Prepare gun
        trackName = null; // Initialize to not tracking anyone
        setAdjustGunForRobotTurn(true); // Keep the gun still when we turn
        gunTurnAmt = 10; // Initialize gunTurn to 10

        // Loop forever
        while (true) {
            // turn the Gun (looks for enemy)
            turnGunRight(gunTurnAmt);
            // Keep track of how long we've been looking
            count++;
            // If we've haven't seen our target for 2 turns, look left
            if (count > 2) {
                gunTurnAmt = -10;
            }
            // If we still haven't seen our target for 5 turns, look right
            if (count > 5) {
                gunTurnAmt = 10;
            }
            // If we *still* haven't seen our target after 10 turns, find another target
            if (count > 11) {
                trackName = null;
            }
        }
    }

    /**
     * onScannedRobot:  Here's the good stuff
     */
    public void onScannedRobot(ScannedRobotEvent e) {

        // If we have a target, and this isn't it, return immediately
        // so we can get more ScannedRobotEvents.
        if (trackName != null && !e.getName().equals(trackName)) {
            return;
        }

        // If we don't have a target, well, now we do!
        if (trackName == null) {
            trackName = e.getName();
            out.println("Tracking " + trackName);
        }
        // This is our target.  Reset count (see the run method)
        count = 0;
        // If our target is too far away, turn and move toward it.
        if (e.getDistance() > 150) {
            gunTurnAmt = normalRelativeAngleDegrees(e.getBearing() + (getHeading() - getRadarHeading()));

            turnGunRight(gunTurnAmt); // Try changing these to setTurnGunRight,
            turnRight(e.getBearing()); // and see how much Tracker improves...
            // (you'll have to make Tracker an AdvancedRobot)
            ahead(e.getDistance() - 140);
            return;
        }

        // Our target is close.
        gunTurnAmt = normalRelativeAngleDegrees(e.getBearing() + (getHeading() - getRadarHeading()));
        turnGunRight(gunTurnAmt);
        fire(3);

        // Our target is too close!  Back up.
        if (e.getDistance() < 100) {
            if (e.getBearing() > -90 && e.getBearing() <= 90) {
                back(40);
            } else {
                ahead(40);
            }
        }
        scan();
    }

    /**
     * onHitRobot:  Set him as our new target
     */
    public void onHitRobot(HitRobotEvent e) {
        // Only print if he's not already our target.
        if (trackName != null && !trackName.equals(e.getName())) {
            out.println("Tracking " + e.getName() + " due to collision");
        }
        // Set the target
        trackName = e.getName();
        // Back up a bit.
        // Note:  We won't get scan events while we're doing this!
        // An AdvancedRobot might use setBack(); execute();
        gunTurnAmt = normalRelativeAngleDegrees(e.getBearing() + (getHeading() - getRadarHeading()));
        turnGunRight(gunTurnAmt);
        fire(3);
        back(50);
    }

    /**
     * onWin:  Do a victory dance
     */
    public void onWin(WinEvent e) {
        for (int i = 0; i < 50; i++) {
            turnRight(30);
```

```java
                turnLeft(30);
            }
        }
}
```

Listing 77: robot/Tracker.java

```java
package robot;

import fa.IFunctionApproximation;
import fa.LUT;
import policy.EnergyReward;
import policy.GoTopRight;
import policy.IPolicy;
import representation.*;
import rl.ILearning;
import rl.QLearning;

public class TrivialLUTRobotConfident extends QLearningRobot {
    public TrivialLUTRobotConfident() {
        super(createLearning());
    }


    public static ILearning createLearning() {
        IActionRepresentation actionRepresentation = new MoveRepresentation();
        IStateRepresentation stateRepresentation = new CoordinatesRepresentation();
        IFunctionApproximation functionApproximation = new LUT("TrivialLUTRobot.obj", true);
        IPolicy policy = new GoTopRight();
        return new QLearning(
                stateRepresentation,
                actionRepresentation,
                new ConcatenationRepresentation(),
                policy,
                functionApproximation,
                0.05, 0.1, 0.8);
    }
}
```

Listing 78: robot/TrivialLUTRobotConfident.java

```java
package robot;

import fa.IFunctionApproximation;
import fa.LUT;
import policy.EnergyReward;
import policy.GoTopRight;
import policy.IPolicy;
import representation.*;
import rl.ILearning;
import rl.QLearning;

public class TrivialLUTRobot extends QLearningRobot {
    public TrivialLUTRobot() {
        super(createLearning());
    }


    public static ILearning createLearning() {
        IActionRepresentation actionRepresentation = new MoveRepresentation();
        IStateRepresentation stateRepresentation = new CoordinatesRepresentation();
        IFunctionApproximation functionApproximation = new LUT("TrivialLUTRobot.obj", false);
        IPolicy policy = new GoTopRight();
        return new QLearning(
                stateRepresentation,
                actionRepresentation,
                new ConcatenationRepresentation(),
                policy,
                functionApproximation,
                0.8, 0.1, 0.8);
    }
}
```

Listing 79: robot/TrivialLUTRobot.java

```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 80: autograd/VariableTest.java

```java
package fa;

import org.junit.Assert;
import org.junit.Ignore;
import org.junit.Test;
import representation.IRepresentable;

public class TestFunctionApproximation {
    @Ignore
    @Test
    public void TestLUT() {
        LUT lut = new LUT(null, true);
        IRepresentable desiredState = new IRepresentable() {
            @Override
            public double[] toVector() {
                return new double[0];
            }
        };
        IRepresentable otherState = new IRepresentable() {
            @Override
            public double[] toVector() {
                return new double[0];
            }
        };
        double[] desiredResponse = new double[] {10};
        double[] otherDesiredResponse = new double[] {5};
        lut.train(desiredState, desiredResponse);
        Assert.assertArrayEquals(desiredResponse, lut.eval(desiredState), 0.);
        lut.train(otherState, otherDesiredResponse);
        Assert.assertArrayEquals(otherDesiredResponse, lut.eval(otherState), 0.);
        lut.train(desiredState, otherDesiredResponse);
        Assert.assertArrayEquals(otherDesiredResponse, lut.eval(desiredState), 0.);
    }
}
```

Listing 81: fa/TestFunctionApproximation.java

```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 {
        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 MeanSquaredError(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 MeanSquaredError(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 82: nn/NeuralNetworkTest.java

```
package optimization;

import autograd.UniformInitializer;
import dataset.*;
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 MeanSquaredError(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)
```

```java
            );
            var dataSet = new XORBinaryDataSet();
            var optimizer = new GradientDescent(0.2, 0.);
            var loss = new MeanSquaredError(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());
            var optimizer = new GradientDescent(0.2, 0.);
            var loss = new MeanSquaredError(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 MeanSquaredError(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);
    }

    @Ignore("ReLU not working on XOR or buggy")
    @Test
    public void TestBipolarMomentumGDReLU() 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 ReLU(),
                    new UniformInitializer(-0.5, 0.5)
            );
            var dataSet = new BinaryToBipolarWrapper(new XORBinaryDataSet()
                );
            var optimizer = new GradientDescent(0.01, 0.9);
            var loss = new MeanSquaredError(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);
    }


    @Test
    public void TestLookupNeuralNet() throws ExecutionControl.NotImplementedException, IOException {
        var model = Factory.createNeuralNetwork(
                new int[]{12, 1},
                new BipolarSigmoid(),
                new UniformInitializer(-0.5, 0.5),
                false
        );
        double[] input = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1};
        double desired = 2.1;
        DataPointDataSet dataPointDataSet = new DataPointDataSet(new DataPoint(input, new double[]{desired}));
        double toleratedError = 0.01;
        double lossLimit = Math.pow(toleratedError, 2) / 2;
        model.fit(dataPointDataSet, new GradientDescent(0.5, 0.), new MeanSquaredError(model.getOutput()), 1000, lossLimit);
        Assert.assertEquals(model.evaluate(input)[0], desired, toleratedError);
    }
}
```

Listing 83: optimization/GradientDescentTest.java

```java
package rl;

import org.junit.Ignore;
import policy.IPolicy;
import representation.*;
import org.junit.Assert;
import org.junit.Test;
import robocode.Robot;
import robocode.control.BattleSpecification;
import robocode.control.BattlefieldSpecification;
import robocode.control.RobocodeEngine;
import robocode.control.RobotSpecification;
import robocode.control.events.BattleAdaptor;
import robocode.control.events.RoundEndedEvent;
import robocode.control.events.TurnEndedEvent;
import robocode.control.snapshot.IRobotSnapshot;
import robot.TrivialLUTRobot;
import robot.TrivialLUTRobotConfident;

import java.io.File;
import java.util.ArrayList;

public class TestQLearning {

    @Ignore("Focus on testing TNinety")
    @Test
    public void TestTrivialLUTRobot() {
        new File("TrivialLUTRobot.obj").deleteOnExit();
        Robot opponent = new TrivialLUTRobot();
        ArrayList<IState> states = new ArrayList<>();
        TrivialLUTRobotConfident robot = new TrivialLUTRobotConfident();
        System.setProperty("NOSECURITY", "true");
        RobocodeEngine.setLogMessagesEnabled(false);
        RobocodeEngine engine = new RobocodeEngine(new java.io.File(System.getProperty("user.home") + "/robocode/"));
        engine.addBattleListener(new BattleAdaptor() {
            @Override
            public void onTurnEnded(TurnEndedEvent event) {
                super.onTurnEnded(event);
                for (IRobotSnapshot robotSnapshot: event.getTurnSnapshot().getRobots()) {
//                      System.out.println(robotSnapshot.getShortName());
                    if (robotSnapshot.getShortName().equals("TrivialLUTRobotConfident*")) {
                        states.add(new Coordinates(
                                (int) (robotSnapshot.getX() / 100),
                                (int) (robotSnapshot.getY() / 100), 0));
                    }
                }
            }
        });
//        engine.setVisible(true);
        int numberOfRounds = 100;
        BattlefieldSpecification battlefield = new BattlefieldSpecification(800, 600);
        RobotSpecification[] selectedRobots = engine.getLocalRepository(robot.getClass().getCanonicalName() + "*," + opponent.getClass().getCanonicalName() + "*");
        BattleSpecification battleSpec = new BattleSpecification(numberOfRounds, battlefield, selectedRobots);
        engine.runBattle(battleSpec, true); // waits till the battle finishes
        engine.close();

        Assert.assertTrue("Seemingly battle didn't happen. No state is collected.", states.size() > 1);
        IState lastRun = states.get(states.size() - 1);
        IState firstRun = states.get(0);
        IPolicy policy = robot.getLearning().getPolicy();
//        double initialReward = policy.getReward(firstRun);
//        double finalReward = policy.getReward(lastRun);
//        Assert.assertTrue(
//                String.format("Learning wasn't effective, initial reward %f, final reward %f", initialReward, finalReward),
//                initialReward <= finalReward);
    }
}
```

Listing 84: rl/TestQLearning.java

```java
package robot;
```

```

import autograd.UniformInitializer;
import dataset.*;
import fa.LUT;
import jdk.jshell.spi.ExecutionControl;
import nn.*;
import optimization.GradientDescent;
import org.junit.Assert;
import org.junit.Ignore;
import org.junit.Test;
import representation.Concatenation;
import representation.States;
import representation.TNinetyAction;

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.util.ArrayList;

public class TestLUTNN {

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

        LUT lut = new LUT(this.getClass().getClassLoader().getResource("
            LUTTNinetyRobot.obj").getPath(), true);
        lut.load();
        System.out.println(lut.getSize());
        Assert.assertTrue(false);
        IDataSet dataSet = new LookupTableDataSet(lut);

        for (double momentum :
                new double[]{0., 0.5, 0.9}) {
            for (double lr :
                    new double[] {1e-4, 1e-3, 1e-2}) {
                for (int hiddenNeurons :
                        new int[]{5, 15, 30}) {
                    var model = Factory.createNeuralNetwork(
                            new int[]{15, hiddenNeurons, 1},
                            new BipolarSigmoid(),
                            new UniformInitializer(-0.05, 0.05),
                            false
                    );
                    var optimizer = new GradientDescent(lr, momentum);

                    var loss = new MeanSquaredError(model.getOutput());
                    var collector = new ConvergenceCollector();
                    double finalLoss = model.fit(dataSet, optimizer, loss,
                        100, 0.00, collector, true);
                    System.out.println(lr + " " + momentum + " " +
                        hiddenNeurons + " " + finalLoss);
                }
            }
        }
    }

```

```java
    @Ignore
    @Test
    public void BestGraph() throws ExecutionControl.NotImplementedException, IOException, ClassNotFoundException {

        LUT lut = new LUT(this.getClass().getClassLoader().getResource("LUTTNinetyRobot.obj").getPath(), true);
        lut.load();
        IDataSet dataSet = new LookupTableDataSet(lut);

        var model = Factory.createNeuralNetwork(
                new int[]{15, 15, 1},
                new BipolarSigmoid(),
                new UniformInitializer(-0.05, 0.05),
                false
        );
        var optimizer = new GradientDescent(0.001, 0.5);

        var loss = new MeanSquaredError(model.getOutput());
        var collector = new ConvergenceCollector();
        for (int i = 0; i < 30; i++) {
            double finalLoss = model.fit(dataSet, optimizer, loss, 100, 0.00, collector, true);
            System.out.println((100 * i + 100) + " " + Math.sqrt(finalLoss / lut.getSize()));
        }
    }
}
```

Listing 85: robot/TestLUTNN.java

```java
package robot;

import fa.IFunctionApproximation;
import fa.LUT;
import fa.NN;
import jdk.jshell.spi.ExecutionControl;
import org.junit.Ignore;
import org.junit.Test;
import representation.IState;
import robocode.Robot;
import robocode.control.BattleSpecification;
import robocode.control.BattlefieldSpecification;
import robocode.control.RobocodeEngine;
import robocode.control.RobotSpecification;
import robocode.control.events.BattleAdaptor;
import robocode.control.events.BattleCompletedEvent;
import robocode.control.events.RoundEndedEvent;
import robocode.control.events.TurnEndedEvent;
import robocode.control.snapshot.IRobotSnapshot;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Objects;

public class TestTNinetyRobot {
```

```java

//     @Ignore
    @Test
    public void TestTrivialLUTRobot() {
//        testRobot(new LUTTNinetyRobot0(), 1, 100);
//        testRobot(new LUTTNinetyRobotOnline(), 1, 100);
//        testRobot(new LUTTNinetyRobotTerminal(), 500, 100);
//        testRobot(new LUTTNinetyRobot05(), 100, 100);
//        testRobot(new LUTTNinetyRobot(), 500, 2);
//        NN functionApproximation = (NN) (new NNTNinetyRobot()).getLearning().getFunctionApproximation();
//        var dataSet = functionApproximation.getDataSet();
//        for (int i = 0; i < dataSet.getX().size(); i++) {
//            double[] x = dataSet.getX().get(i);
//            for (int j = 0; j < x.length; j++) {
//                System.out.print(x[j] + " ");
//            }
//            System.out.println(dataSet.getY().get(i)[0]);
//        }

        new File("NNTNinetyRobot.obj").deleteOnExit();
        new File("NNTNinetyRobot.LUT").deleteOnExit();
        new File("NNTNinetyRobot.NN").deleteOnExit();
        testRobot(new NNTNinetyRobotw1g9(), 50, 100);
//        testRobot(new NNTNinetyRobotw10g9(), 50, 100);
//        testRobot(new NNTNinetyRobotw10g5(), 50, 100);
//        testRobot(new NNTNinetyRobotw10g1(), 50, 100);
//        testRobot(new NNTNinetyRobotw20g9(), 50, 100);
    }

    private void testRobot(Robot trainRobot, int rounds, int battles) {

        String outputFileName = "doc/" + trainRobot.getClass().getName() + ".tex";
//        new File(outputFileName).deleteOnExit();
        ArrayList<IState> states = new ArrayList<>();
        NNTNinetyRobotConfident testRobot = new NNTNinetyRobotConfident();
        Corners opponent = new Corners();
        System.setProperty("NOSECURITY", "true");
        RobocodeEngine.setLogMessagesEnabled(false);
        RobocodeEngine engine = new RobocodeEngine(new File(System.getProperty("user.home") + "/robocode/"));
        engine.addBattleListener(new BattleAdaptor() {

            double energy = 0;
            double enemyEnergy = 0;
            int battles = 0;
            TurnEndedEvent turnEndedEvent;

            @Override
            public void onTurnEnded(TurnEndedEvent event) {
                super.onTurnEnded(event);
                turnEndedEvent = event;
            }

            @Override
            public void onRoundEnded(RoundEndedEvent event) {
                super.onRoundEnded(event);
                IRobotSnapshot[] robots = turnEndedEvent.getTurnSnapshot().
                    getRobots();
                for (IRobotSnapshot robot :
                        robots) {
                    if (robot.getName().equals("robot.
                        NNTNinetyRobotConfident*")) {
                        energy += robot.getEnergy();
                    } else {
                        enemyEnergy += robot.getEnergy();
                    }
                }
            }

            @Override
            public void onBattleCompleted(BattleCompletedEvent event) {
                super.onBattleCompleted(event);
                boolean shouldPrint = false;
                try {
                    FileWriter of = new FileWriter(outputFileName, true);
                    for (var result :
                            event.getIndexedResults()) {
                        if (Objects.equals(result.getTeamLeaderName(),
                            testRobot.getClass().getCanonicalName() + "*"))
                            {
                            battles += 1;
                            of.write((battles * rounds) + " " + result.
                                getFirsts() + " " + (energy / 100) + " " + (
                                enemyEnergy / 100) + " ");
                            System.out.print(result.getFirsts() + " ");
                            shouldPrint = true;
                        }
                    }
                    if (shouldPrint) {
                        NN functionApproximation = (NN) (new
                            NNTNinetyRobotConfident()).getLearning().
                            getFunctionApproximation();
                        System.out.println(functionApproximation.getSize()
                            + "\n");
                        double loss = functionApproximation.getLoss();
                        System.out.println(loss + "\n");
                        of.write(loss + "\n");
                    }
                    of.close();
                } catch (IOException | ExecutionControl.
                    NotImplementedException e) {
                    e.printStackTrace();
                }
                energy = 0;
                enemyEnergy = 0;
            }
        });
        for (int i = 0; i < battles; i++) {
            int numberOfRounds = rounds;
            BattlefieldSpecification battlefield = new
                BattlefieldSpecification(800, 600);
            Robot robot;
            if (i % 2 == 0) {
                engine.setVisible(false);
                robot = trainRobot;
            } else {
                numberOfRounds = 100;
                engine.setVisible(false);
                robot = testRobot;
            }
            RobotSpecification[] selectedRobots = engine.getLocalRepository(robot.getClass().getCanonicalName() + "*," + opponent.getClass().getCanonicalName() + "*");
            BattleSpecification battleSpec = new BattleSpecification(numberOfRounds, battlefield, selectedRobots);
            engine.runBattle(battleSpec, true); // waits till the battle finishes
        }
        engine.close();
    }
}
```

Listing 86: robot/TestTNinetyRobot.java