CPEN502 Assignment 3

Authors

Xuechun Qiu, 55766737

Tao Ma, 13432885

Questions

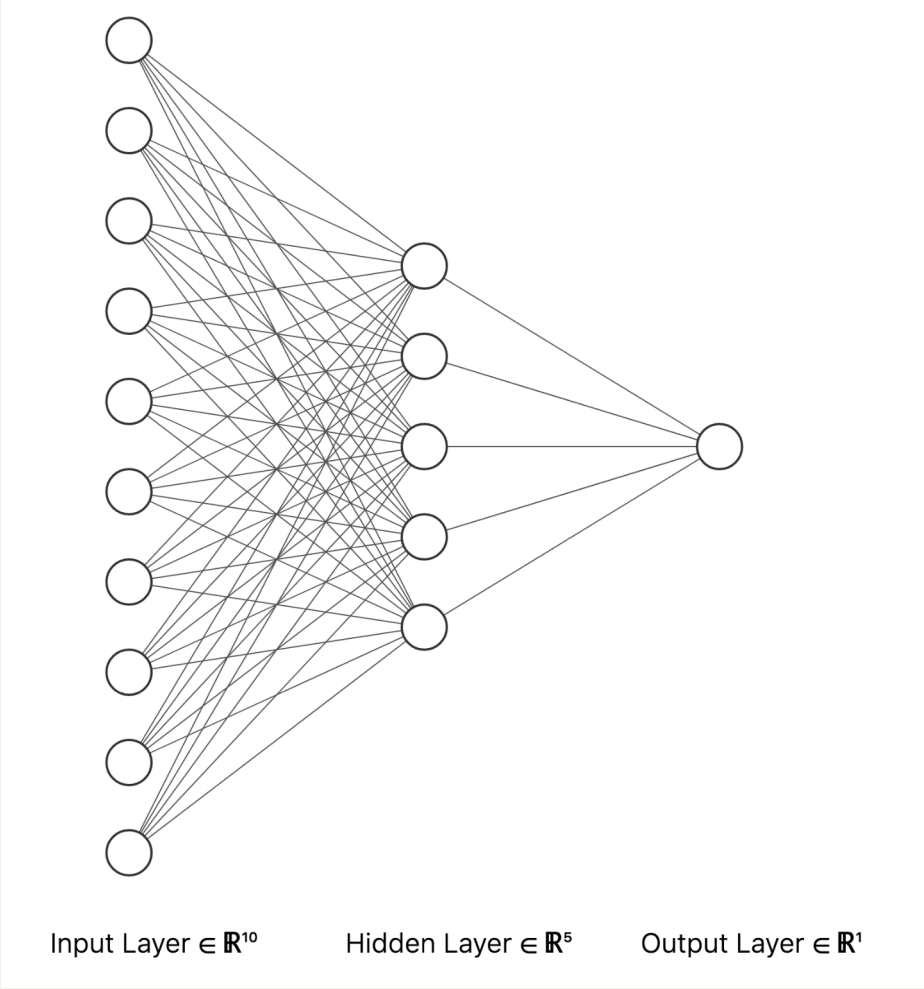
(4)

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

a)

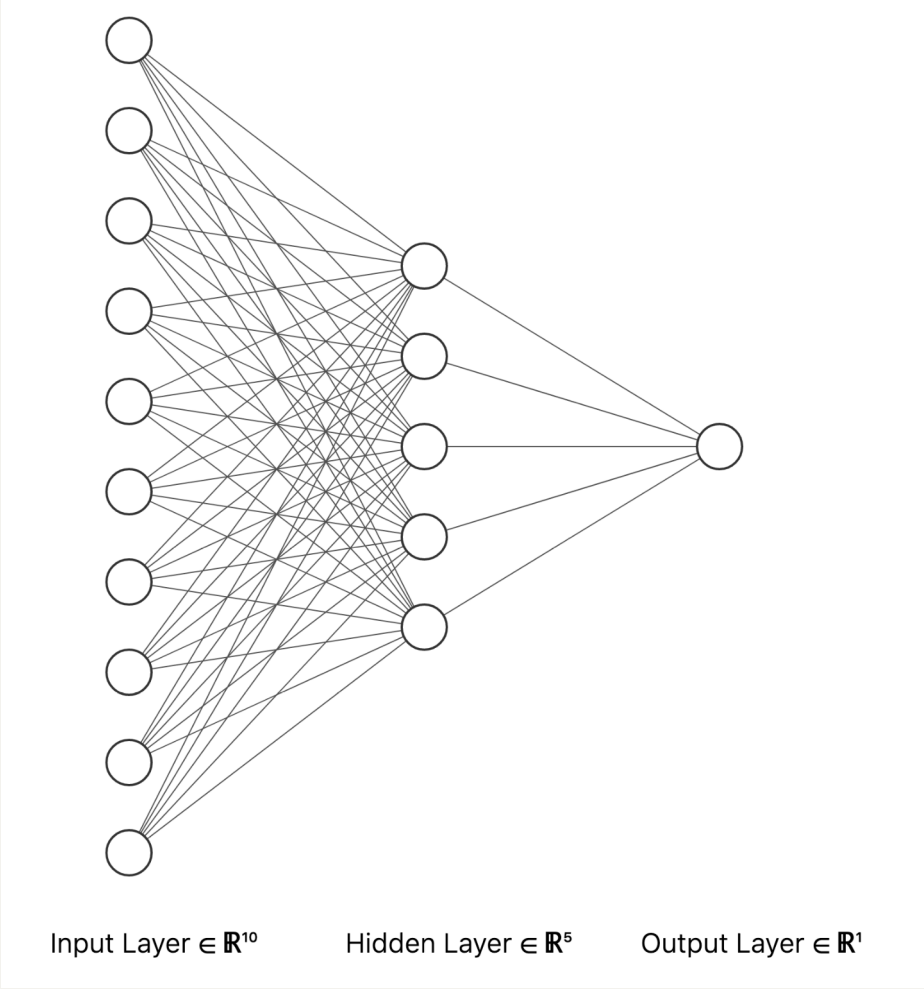
Question: There are 3 options for the architecture of your neural network. Describe and draw all three options and state which you selected and why. (3pts)

Option 1



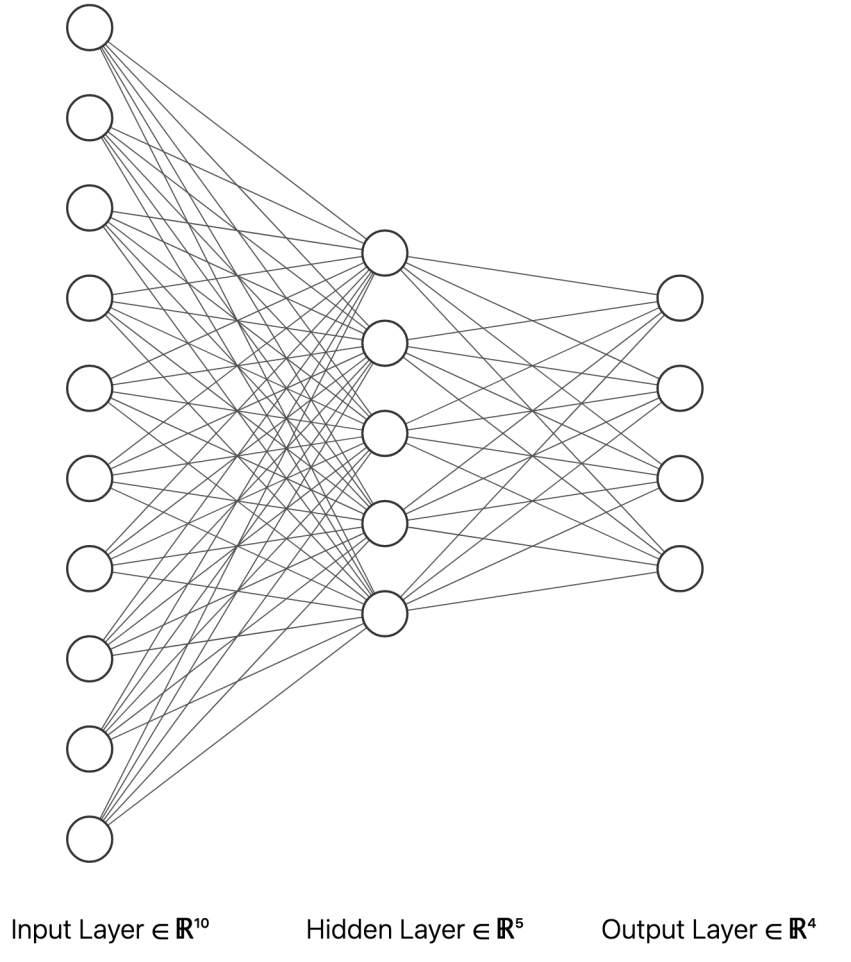
The first option is to utilize a straightforward neural network as a form of predictive modelling in order to derive the Q value for the state action pair input. To be more specific, this option takes in an input vector that contains a state vector and an action vector, and it then generates a Q value that is associated with the state-action pair.

Option 2



The second option, combining the previous two options, utilize a straightforward neural network as a form of predictive modelling in order to derive the Q value of an action for the state input. This ensemble method prepare n neural networks for n possible action. When a state vector is being fed into the model, n neural networks work together to compute the $Q\ value \in \mathbb{R}^{Num\ of\ Actions}$ for each action individually.

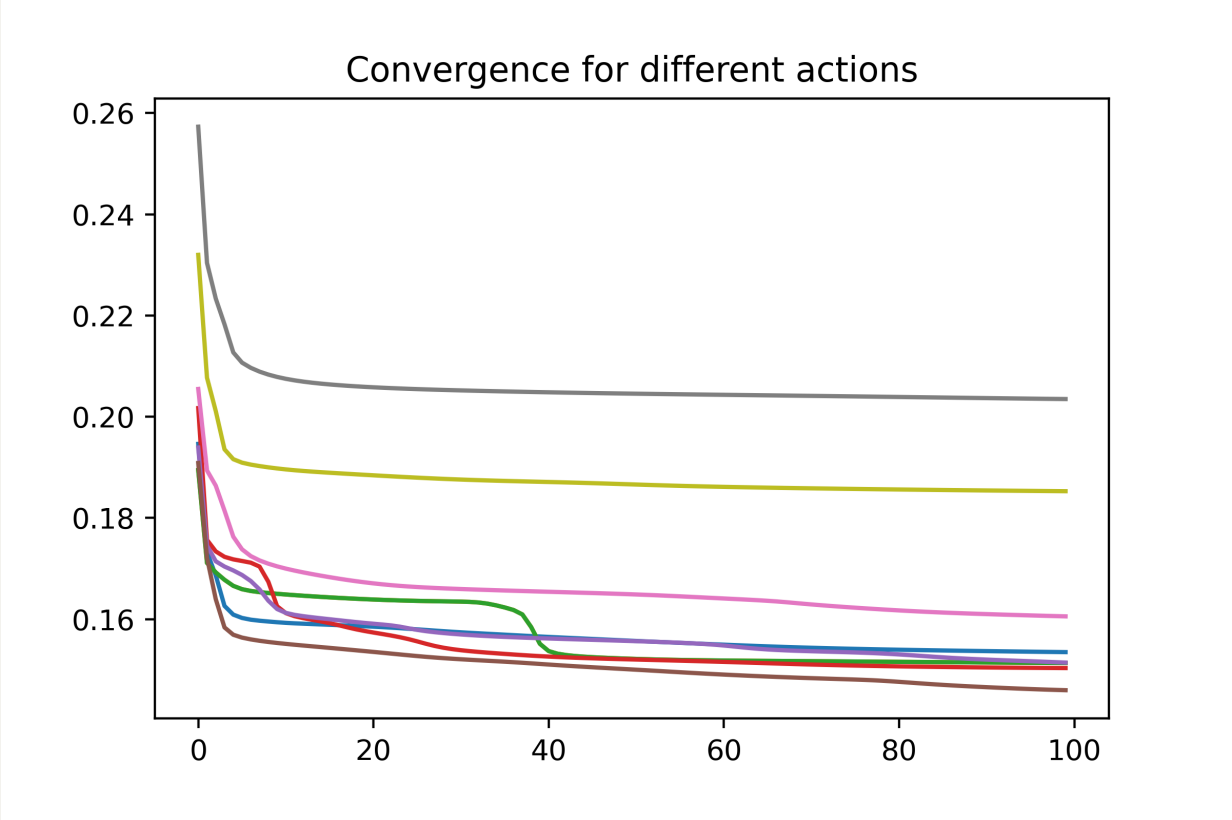
Option 3



The third option is to utilize a simple neural network to compute the corresponding Q value vectors of all the possible actions for the corresponding state input. More specifically, this neural network computes a vector of $Q\ value \in \mathbb{R}^{Num\ of\ Actions}$ from a state vectors consist of $state \in \mathbb{R}^{Num\ of\ States}$. This architecture also enable adapting the neural network into a multi-label classification problem. To be more explicit, rather than considering the neural network as a model that predicts the Q values for each action, we may view the neural network as an action selector that chooses the best action based on the state input. This allows for a more accurate representation of the network's capabilities. However, it is important to take into consideration that the final layer should be changed to a softmax layer in this scenario.

In the context of this assignment, we have implemented all three of them, and we have decided to report using Option 3 as our model. In this part of the article, we will begin by discussing these three possibilities and explaining the thought process that went into reaching our decision. These three choices each come with their own set of advantages to consider. For instance, Option 1 enables an easier implementation, and the computation is uncomplicated; however, concatenating the state vector and the action vector together could result in an issue due to the fact that they come from distinct distributions. Option 3 is intended to be a standardized approach to the problem of multi-label prediction; but,

by choosing this option, you will lose more granular control over each operation. The accompanying graphic demonstrates that each action converges at a different rate (see below graph), and it is possible that combining several actions will result in unexpected behaviour. In the end, we decided that Option 3 would be the best model to report. Option 2 has some severe complexity issues, such as the fact that we have to run n times more predictions for n actions, but it does provide clearer illustrations for the analysis of each action and provides for better control over the activities. However, Option 2 ignores the competition behind actions. For example, imaging you are the robot and you are running into a state, you need to compare action to find the best action. Option 2 only consider action individually but a robot need to consider the whole action set to make decision.



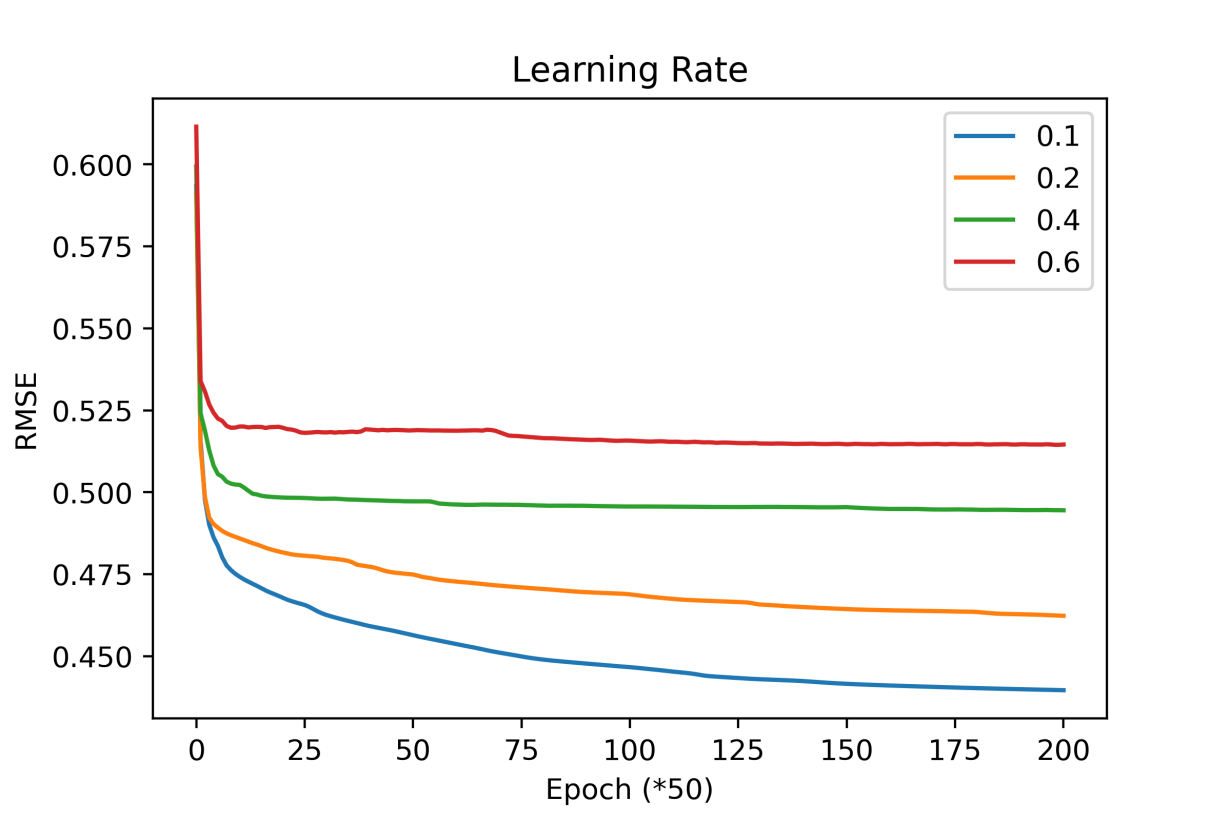
b)

Question: 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)

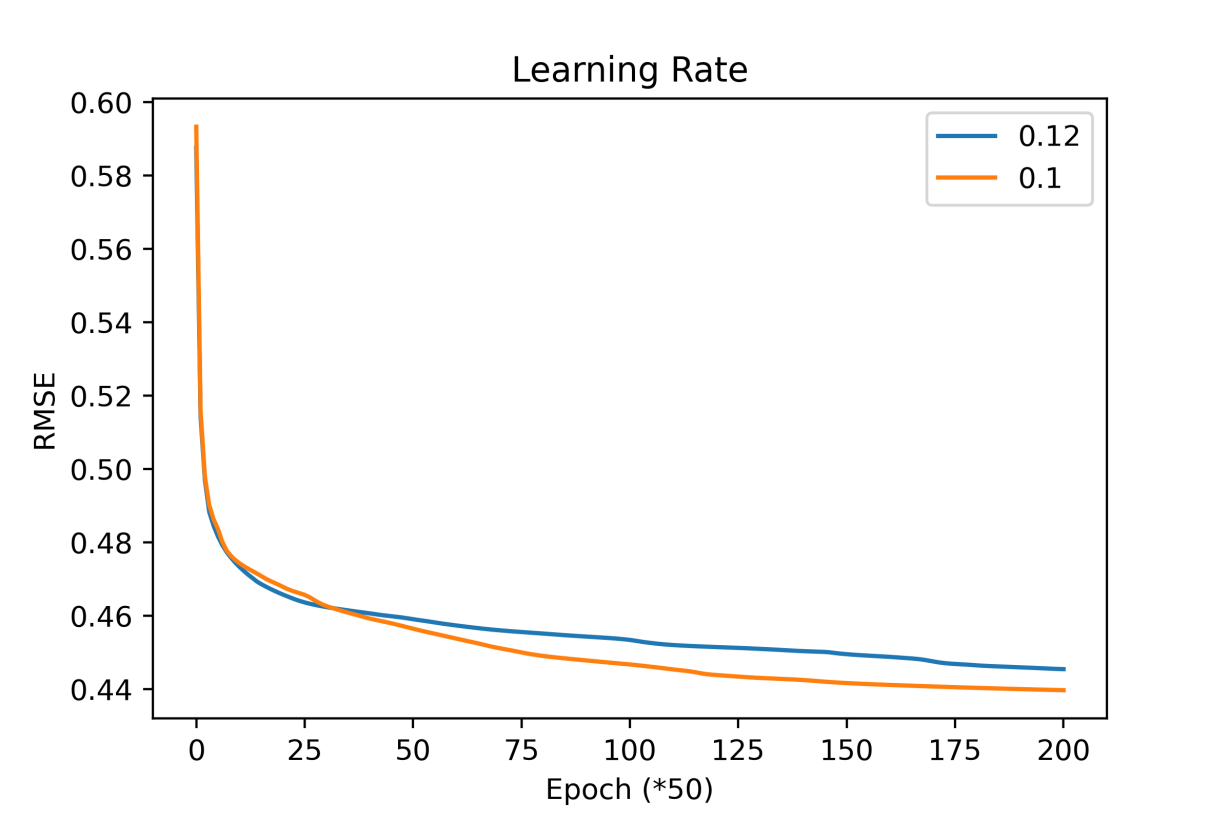
Learning Rate

- Momentum = 0.0, Hidden = 10

LEARNING RATES	0.1	0.2	0.4	0.6
RMS Error	0.439	0.462	0.494	0.514



When the learning rate is held constant, we are able to see that the error produced by the network continues to get progressively better. However, we discovered that a decline in performance occurs if the rate of learning is increased. We hypothesise that the huge step of tested learning rate is to blame for the low performance because this is the most likely explanation. We may imagine that the value of the learning rate hyperparameter affects how big of a step the model takes while descending down the hill towards where the derivative points to. This is controlled by the learning rate hyperparameter, which governs the rate or speed at which the model learns. When the learning rate is excessively high, the model may arrive at a final set of weights that is less than ideal very rapidly. To show that it's the case, I draw another graph with Learning Rate 0.125 and 0.1 shown below

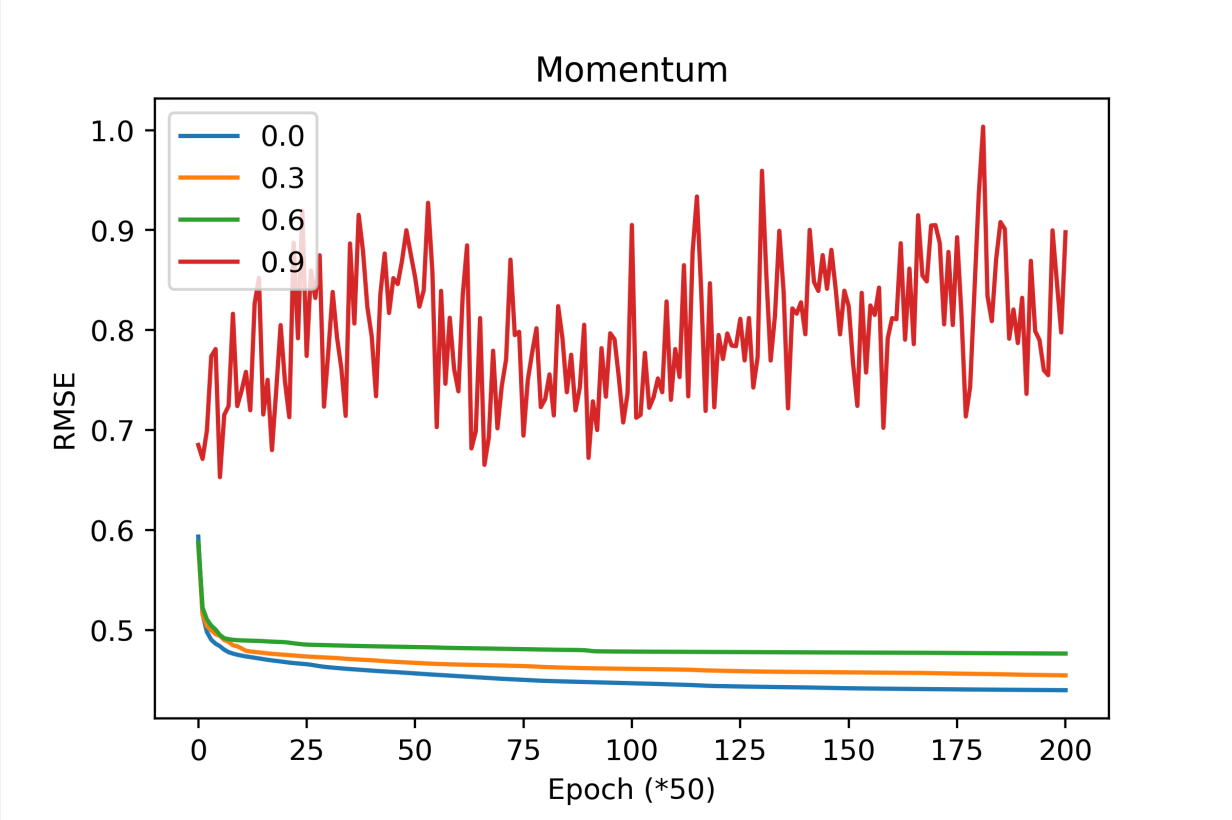


At around Epoch 12, the model with learning rate 0.12 converges quicker than the model with learning rate 0.1. From that point on, however, the model with a learning rate of 0.12 remained in a suboptimal position and was surpassed by the model with a learning rate of 0.1.

Momentum

- Learning rate = 0.1, Hidden = 10

MOMENTUM	0.0	0.3	0.6	0.9
RMS Error	0.439	0.454	0.476	0.902

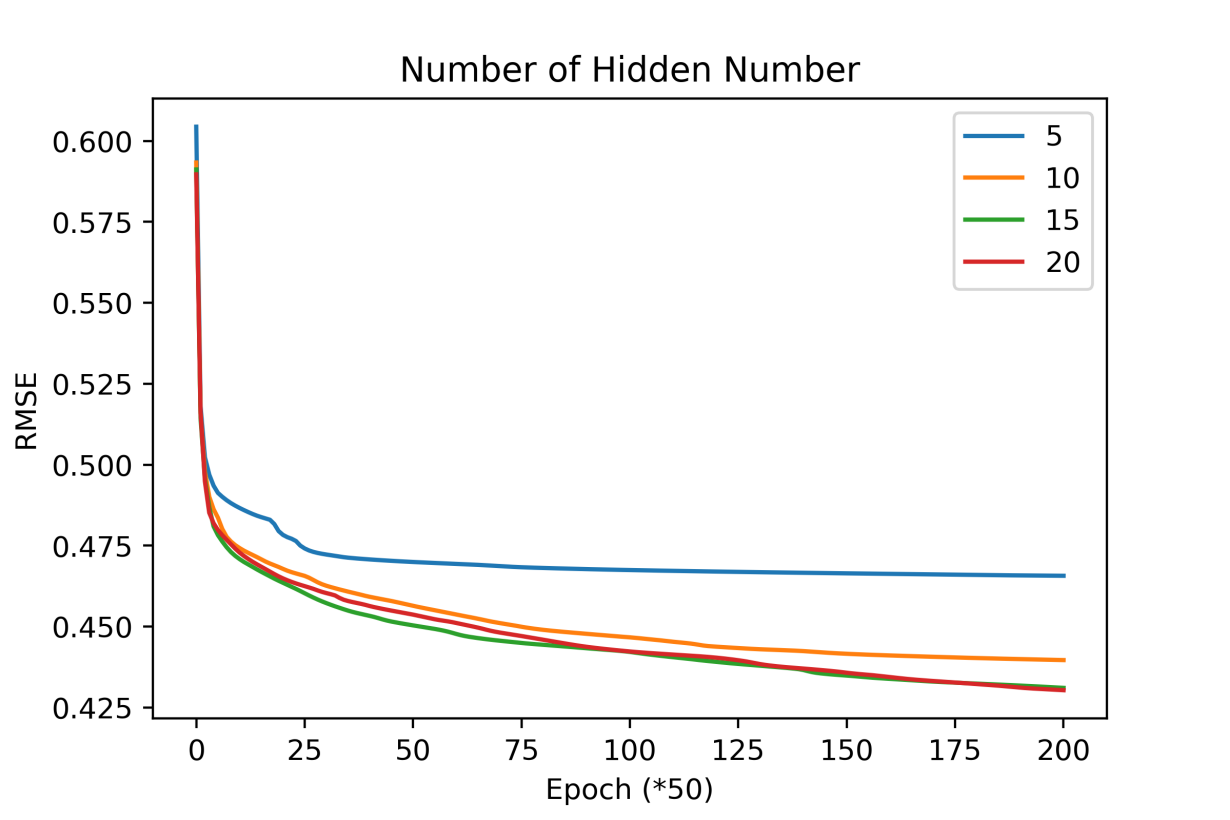


Momentum is a method for accelerating model convergence. As seen in the graph, as momentum increases, initial convergence speed increases, but models with greater momentum values quickly become stuck in suboptimal states. When momentum is set to 0.9, the gradient of each learning step is significantly influenced by the gradient of the previous step. $(\text{momentum} \times \Delta_{ij}^{t-1})$. Consider also that the model used a learning rate of 0.1 in this experiment, resulting in the model exhibiting bizarre random behaviour because it was heavily influenced by momentum and the effect of momentum already outweighed learning rate significantly.

Hidden neurons

- Learning rate = 0.1, Momentum = 0.0

NUMBERS OF HIDDEN NEURONS	5	10	15	20
RMS Error	0.465	0.439	0.431	0.427



The number of hidden neurons has a direct impact on the model's performance. With more hidden neurons, the model can represent more complex input and generalise more effectively.

In this assignment, our selection of hyperparameters was based on trial and error. In particular, we first obtain a q table by applying our model from the prior assignment (30K rounds). Then, for each of the hyperparameters, we selected certain values to serve as candidates for the hyperparameters and chose the hyperparameters with the best performance. Due to the fact that we only analyse a single variable at a time, it is important to note that basic trial and error is suboptimal. Due to time constraints, we reserve for the future certain more effective tuning options. For instance, evaluating the hyperparameters with grid search, in which the entire collection of hyperparameters is viewed simultaneously, is a superior method.

c)

Question: 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)

In assignment 2, our state is being set as below

STATE	DESCRIPTION	DISCRETE SPACE DIMENSION
HP	Robot's current energy	3
Enemy HP	Enemy's current energy	3
Distance To Enemy	Robot's distance to enemy	3
Distance To Wall	Robot's distance to wall	3
Enemy Robot Heading	Enemy robot's heading (degree)	5
X	Robot's x axis position	3
Y	Robot's y axis position	3

Also, our action space contains 7 different action (AHEAD_LEFT, AHEAD_RIGHT, BACK_LEFT, BACK_RIGHT, AHEAD, BACK, HEAVY_FIRE, LIGHT_FIRE). Therefore, to record all states and actions in a table, the table size need to be

$$3^6 \times 5 \times 7 \tag{1}$$

Table of this size is huge and it's going to cause a lot of IO pressure on the local machine.

When a neural network is used, however, the amount of space that is required is drastically cut down. The fundamental process of learning in a neural network is function mapping. To be more specific, when put in the current context, the neural network is assuming that there exists some relationship between states/actions and their respective Q values in an attempt to approximate the relationship. This assumption is being made while the neural network is using the current context. In light of the aforementioned, the neural network's sole objective is to collect the parameters (weights, bias, etc.) that constitute the relationship it employs to approximate the actual relationship between states/actions and the Q values that are associated with them. When trained with only offline data, a neural network is able to pick up a continuous space input and output the Q values that correspond to it. This is true despite the fact that our state configuration only assumes discrete state.

(5)

Question: 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)

Question: 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)

We identified Winning Rate and RMS Errors as the metrics for evaluating the performance with online training. We use following set up for our online training.

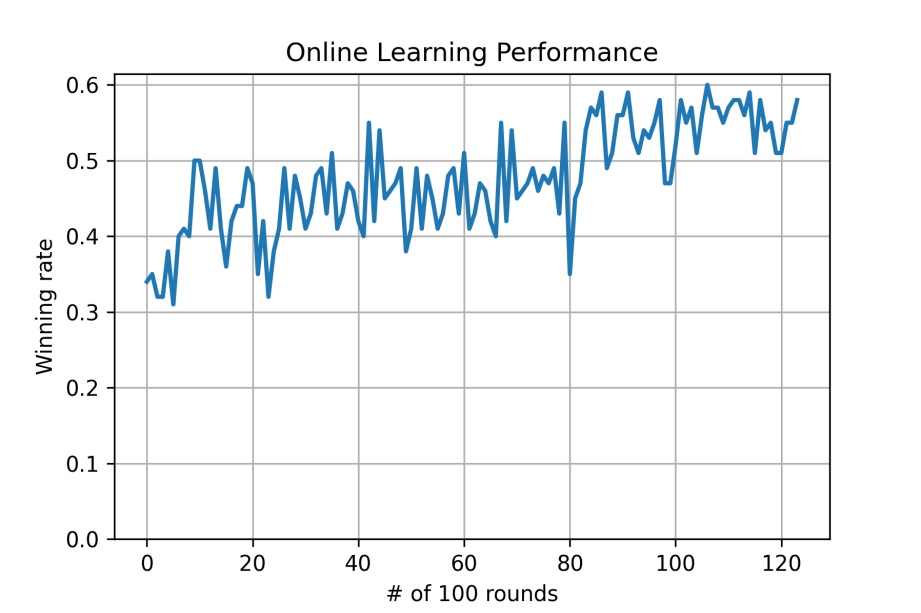
HYPERPARAMETER	VALUE
Learning Rate	0.1
Momentum	0.0
Number of Hidden Neurons	25
α	0.1
γ	0.3
ϵ	0.1
Mini Batch Size	100

Metric 1: Winning Rates

Winning rate is defined as the ratio for the robot's wining. To be more specific, it's being defined as follow:

$$\text{Winning Rate} = \frac{\text{Number of Winning rounds}}{\text{Total Rounds}} \tag{2}$$

However, to showcase the learning process of our model, we modified the metric to be "winning rounds for each 100 rounds" such that we recorded the metric winning rates for each 100 rounds our robot played and we plot a graph to show the changing of winning rate of our robot during online training.

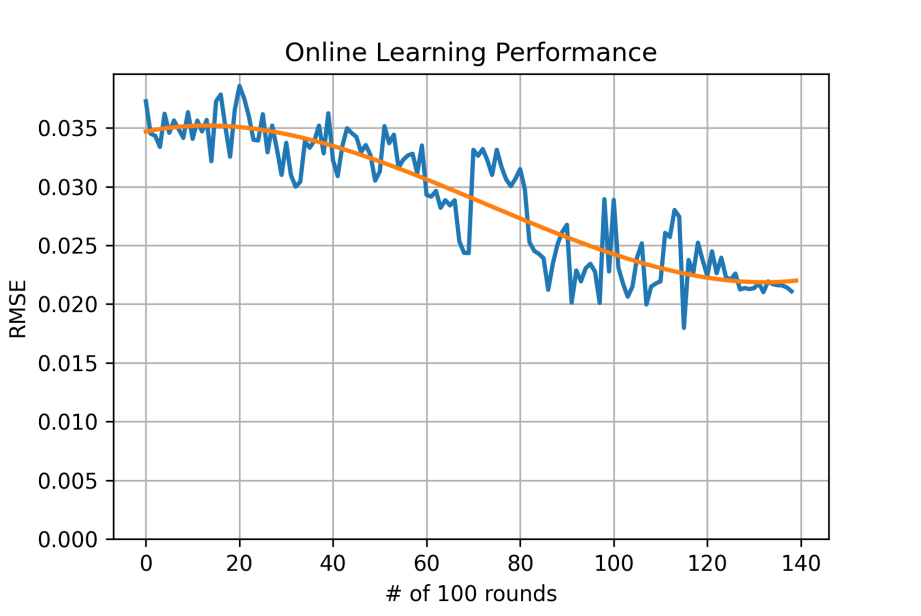


Metric 2: RMS Errors

Root Mean Square (RMS) Error is one of the standard way measuring the errors in a neural network. Following equation shows the way of calculating RMS Error

$$\mathrm{RMSE} = \sqrt{\frac{\sum_{i=1}^{N}(x_i - x_i^{\mathrm{target}})^2}{N}} \tag{3}$$

RMS Error is a popular metric for evaluating the performance of a model, especially in the context of regression tasks. It is a measurement of the average mistake in a model's predictions and can be read as the standard deviation of the residuals (the difference between the predicted values and the true values). RMS error is a valuable metric for evaluating the overall performance of a model since it provides a straightforward and interpretable measure of the error's size. It is crucial to remember, however, that RMS error alone may not provide a whole picture of a model's performance; other measures, such as mean absolute error or mean absolute percentage error, may also be worth evaluating. In addition, the utility of RMS error as a performance statistic might vary depending on the particular context and objectives of a given model. However, due to the time constraint of this assignment, we only consider these two metrics.

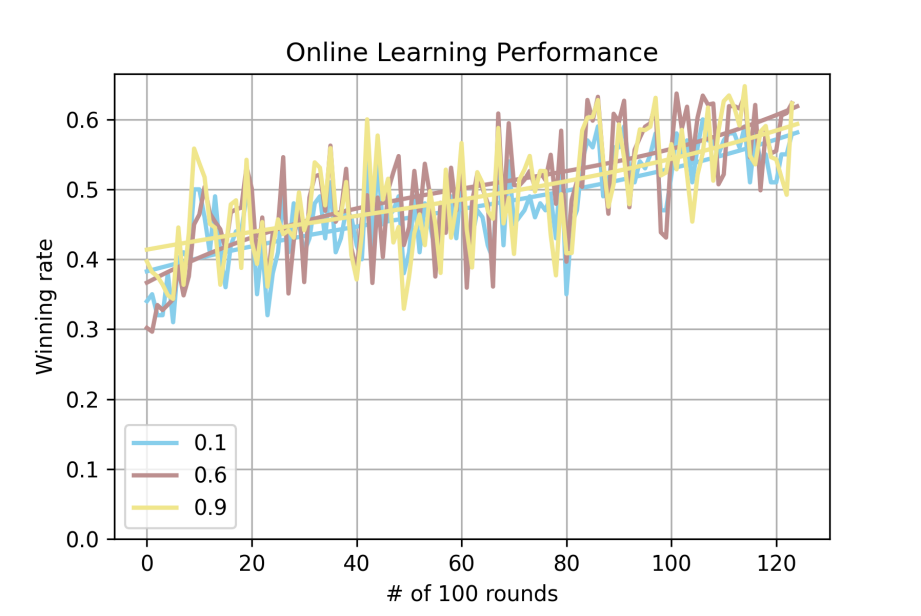


We can observe that the model performance is relatively poorly. Even though converge by a bit, the model still suffers from serious performance issue. There are several reasons for this and we tend to discuss them in this section.

- **Neural Network Architecture**: The architecture of our neural model is relatively simple. Even though we have 25 neurons, we didn't try adding the number of layers of the neural model which means we can only have few non-linear transformation which might affect our overall performance.
- **Hyperparameters Tuning**: In this assignment, our selection of hyperparameters was based on trial and error and this is rather naive and lead to sub-optimal solution. Due to time constraints, we reserve for the future certain more effective tuning options. For instance, evaluating the hyperparameters with grid search, in which the entire collection of hyperparameters is viewed simultaneously, is a superior method.
- **Trainings**: In this project, due to time constraint, we only trained the model with around 10K ~ 20K epochs and it's not enough for our model to pick up useful knowledge.

b)

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



In my opinion, both higher and lower value could impact the model in different ways. The discount factor is used to determine the discounted future value of a state, which is defined as the sum of the immediate reward and the discounted future value of the next state. A higher discount factor will give future benefits greater weight, whereas a lower discount factor will give future rewards less weight. Therefore, in order to train the robot with the best hyperparameters, we need to test with different gamma value

c)

Question: Theory question: With a look-up table, the TD learning algorithm is proven to converge – i.e. will arrive at a stable set of -values for all visited states. This is not so when the -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)

The TD learning algorithm is a model-free reinforcement learning algorithm that updates the values of states in a lookup table based on the observed rewards and transitions using the Bellman equation. According to the Bellman equation, the value of a state equals the expected sum of future rewards, discounted by a factor to account for the time value of rewards. The TD learning algorithm is shown to converge with a lookup table because the values of states are directly updated based on observed rewards and transitions. Since the Bellman equation is exact in this instance, the state values will eventually converge to their true values if the algorithm is given enough time to learn.

However, things could have changed when using neural network to approximate the look-up table.

$$\text{Bellman equation for value of state s: } V(s) = max_a(R(s,a) + \gamma V(s')) \quad (4)$$

And when using neural network as the look-up table, the loss function / convergence target is

$$L(\Theta) = E_{(s,a,r,s')}[(r + \gamma \times max_{a'} Q(s', a'; \Theta^-) - Q(s, a; \Theta))^2] \quad (5)$$

Convergence is no longer guaranteed when the function is approximated with a neural network. This is due to the fact that the neural network function approximation introduces errors and approximations into the value update procedure, which can result in instability and divergence of the values. In particular, the neural network's approximations can cause state values to oscillate or diverge, rather than converge to their true values. Consequently, it is typically more difficult to achieve convergence when approximating the function with a neural network.

d)

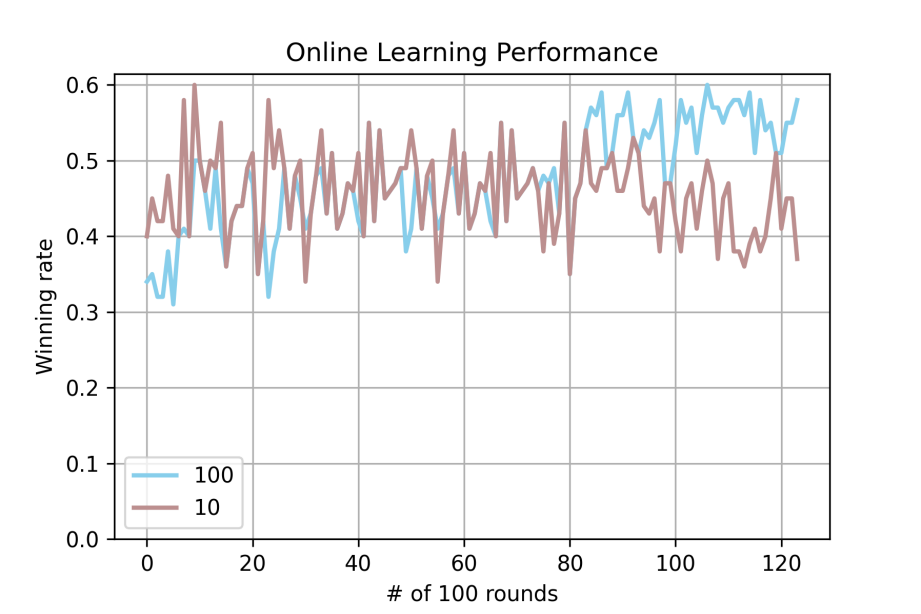
Question: When using a neural network for supervised learning, performance of training is typically measured by computing a total error over the training set. When using the NN for online learning of the -function in robocode this is not possible since there is no a-priori training set to work with.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

It's true that it may not be possible to directly assess the total error across a predefined training set when employing a neural network for online learning of the function in a robocode environment, as the available data may be produced on-the-fly and may not be readily accessible or labelled. In this situation, one method for monitoring the learning performance of the neural network could consist of evaluating the network's performance on a tiny fraction of the data created throughout the learning process. This subset of the available data could be chosen at random and used to compute various performance metrics, such as the mean squared error or classification accuracy, to evaluate the network's learning progress. The process is being called *Replay Memory*. With that being said, different metrics to evaluate the NN's performance can be applied during the process of replaying memory.

- Winning Rates: Winning rate is defined as the ratio for the robot's wining. This metric is unique to our context since the context allows us to determine whether or not the robot is winning.
- Neural Network Metrics: Many metrics (Mean Squared Error, Root Mean Square (RMS) Error…) are this kind of metric and can be used to evaluate any neural network's performance. To monitor the performance of NN during online learning with these metrics, we only need to calculate these metrics in the fixed frame.

e)

Question: 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)



The value of n is directly related to the performance of the model. With a larger value of n, the neural network function approximation is able to observe a variety of training data during a single training process, resulting in enhanced performance. According to the graph, the model is incapable of learning and is not converging. However, as n increases, the winning percentage of the robot will converge to some degree.

(6)

Question: Overall Conclusions

a)

Question: 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)

Throughout this project, I have learnt how to apply Neural Network (back propagation) and Reinforcement Learning in a real world problem. More importantly, throughout the process of implementing the robot with both RL & NN, I had chances to explore and understand the very basic of these two techniques.

In the project, I also observe some insights that I am able to offer with regard to the practical issues surrounding the application of RL & BP to the problem.

- **Model Design**: The design of the architecture is one of the most important things. The design of model can involve following different properties:
    - Neural Net Architecture Design: The design of the neural net architecture. For example, in this particular assignment, we have designed and implemented 3 different design options and we have chosen the most reasonable one.
    - Reinforcement Learning Environment Design: The design of the Markov environment is crucial to the performance of the model, in my opinion. In the second assignment, for instance, we are permitted to define the state and action space. This experience taught us that determining an acceptable state and action is extremely difficult. On the one hand, we would prefer state and action space to be continuous so that they can capture the entirety of reality. Alternatively, we must reduce the size of the state and action space so that training does not take forever!

- **Training Data Quality**: One restriction I discovered in this project is the limitation of training data quality. According to my observations, machine learning models rely heavily on task-specific training data. Regardless of whether the model employs supervised, unsupervised, or reinforcement learning, the model is simply attempting to learn from the data and then apply the acquired knowledge to answer the unobserved data. Therefore, machine learning models are constrained by the capacity of the training data to represent information. For instance, after training with robocode data, if the training data set only comprises states in the bottom left portion of the playground, the robot will be overfitted to the lower left portion of the playground and will perform badly in the other portions of the playground. With that being said, the quality of training data would directly impact the performance of the ML model. When training our neural model to approach the Q table offline, it's then crucial to have our robot first explore different states so that the static training data could cover more states for the ML model to learn. There are also other techniques for enhancing the training data quality and one such example is data enhancement. Data enhancement typically refers to the process of enriching the first-party data with external context by generating similar data from the current data set. For example, if we have one data point: $\text{state}, \text{Q values}$, we can assume that Q values might not be changed that much for a state that is close to the current state. That way we can approximate another data point $\text{state}', \text{Q values}'$ that is close to the data point we have in the data set. Due to time constraint, we didn't have chance for applying this kind of technique.
- **Data Preprocess**: Data preprocessing is the use of procedures to input data before to its use in training or testing. One use case that comes to mind for our model is preprocessing the state input prior to Neural Network training. Despite the fact that all values are dubbed "state," they derive from distinct distributions. For instance, the robot's current energy and the distance to the enemy robot are from distinct distributions, and we can preprocess them to normalize those numbers under the same distribution so that different distributions do not become entangled.
- **Hyperparameter Tuning**: As stated in previous section, our selection of hyperparameters for this assignment was based on trial and error. Specifically, we obtain a q table by applying our model from the preceding assignment (30K rounds). Then, for each of the hyperparameters, candidate values were picked, and the best performing hyperparameters were chosen. Due to the fact that we only examine one variable at a time, it is essential to emphasise that trial and error is suboptimal. Due to time constraints, we reserve certain more effective

tweaking possibilities for the future. For example, evaluating the hyperparameters with grid search, where the complete set of hyperparameters is viewed simultaneously, is a preferable way.

- **Computational Capability**: The term computational capability limitation refers to the lack of appropriate computing power as the complexity of the computation task increases. The complexity of the computation task increases if the dimension of the data or the architecture of the model changes.

We have identified several insights from this project and now we tend to answer the sample questions.

What could you do to improve the performance of your robot?

Based on the aforementioned insight, the following steps could be taken to enhance the performance of our robot.

- Redesign the state and action space of Markov to include more "critical" states. To determine which state played a significant role in the decision-making process, we can use state derivatives or a salient map to determine which state is crucial. The reasoning behind this is that some states may be irrelevant when determining an action.
- Increase training data set by having robot from previous Assignment to explore as much as state and action pair.
- Input data should be preprocessed and normalised to reduce the degree to which a neural network suffers from having data from different distributions.

How would you suggest convergence problems be addressed?

We recommended following steps to address the convergence problem specifically.

- Incorporate more complex architecture of the neural network. In our current implementation, we only have one layer of hidden layers therefore the input data was only transformed by 2 non-linear transformation and this might be one of the bottleneck for the current implementation.
- Tune the set of hyperparameters with better techniques. For example, evaluating the hyperparameters with grid search, where the complete set of hyperparameters is viewed simultaneously.
- Implement changing hyperparameters instead of fixed hyperparameters. The

intuition behind this is that we found having a large learning rate at initial stages of the training helps accelerating the training process but it can easily find a sub-optimal if it keeps the same learning rate. Therefore, we can implement a dynamically changing learning rate such that it gets decreased as the training progress. This technique can also be applied to other hyperparameters.
- Train the model with more epoch and more data.

What advice would you give when applying RL with neural network based function approximation to other practical applications?

To applying RL with neural network based function approximation to other practical applications, we should pay attention to all insights that we observed. Specifically, we should do the following.

- Carefully construct the markov states and actions space and experiment with different neural network architecture.
- Pay attention on data collection process to ensure that the data is balanced and rich since the training data is basically the best your model can reach.
- Remember to preprocess training data to deal with any data problems (data missing, data distribution problems…).
- Tune the set of hyperparameters with better techniques.

b)

Question: 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 concerns of training such closed-loop control system for automatically delivering anesthetic to a patient under going surgery including the difficulties, correctness and privacy. In this section, I will discuss those concerns individually and then report one of the variations that I found practical.

Difficulties

- **Representation of the environment**: To implement a closed-loop control system, as in this project, we must represent the robot's operating environment. However, in a complex environment such as performing surgery, anything could influence the outcome; consequently, the representation of the state can be extremely complex and difficult to define. Likewise, the action space can be incredibly complex. Moreover, it is extremely difficult for us to formulate end state and reward function. There are intuitive formulations for end state and reward function. As an illustration, we can use the patient's life status as one of the most crucial aspects of end state and reward functions. However, these intuitive methods for representing states can have coarse granularity and are likely to produce undesirable outcomes.
- **Collection of training data**: As observed in this project, the quantity of training data is extremely important. The model is merely attempting to acquire knowledge from the observed data and then apply it to the unobserved data. Therefore, having more training data is crucial to the system's performance, especially for such a complex system. However, in this context, training data may be difficult to obtain due to the limited number of similar surgeries performed.

Correctness

Even if we have successfully trained a robot to perform surgery, the consequences of incorrect predictions are severe because they directly affect human life. Not only the algorithm, but also the software engineering aspect of the robot can have an effect on the surgery. During this project, I once forgot to change one of the parameters from class variable to `static`, causing it to be altered continuously. Errors such as these can have catastrophic effects on the surgery.

Privacy

Assuming the model is well-formulated and sufficient training data is available, the issue of data privacy is a concern. To make accurate predictions, our model requires vast amounts of data. This data contains a significant amount of personal and confidential information and should not be disclosed. Nonetheless, as we train our model, it essentially extracts information from the data, encodes it in the gradient, and uses it to update itself. If an adversary is able to monitor the gradient or the data itself, the confidentiality of the data is significantly compromised.

Variation

One variation of the strategy is to limit the robot's capabilities and include a human in the loop. Specifically, instead of having a robot perform the entire operation, we can have a robot predict the correct morphine dosage or even the amount of food given to the patient. Such sub-problems clearly identify their inputs and outputs, thereby reducing the burden of training a model to fit such a complex environment. Human-in-the-loop is a further modification that replaces robots in the decision-making process with human doctors who can examine and even override the decision.

Appendix

org.homework.neuralnet.EnsembleNeuralNet

```java
package org.homework.neuralnet;

import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import org.homework.neuralnet.matrix.Matrix;
import org.homework.robot.model.Action;
import org.homework.robot.model.ImmutableState;
import org.homework.robot.model.State;

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.Random;

@NoArgsConstructor
@Getter
@Setter
public class EnsembleNeuralNet {
    private static final double ALPHA = 0.1;
    private static final double GAMMA = 0.5;
    private static final double RANDOM_RATE = 0.1;
    private static final boolean KEEP = false;
    private final List<Double> errorLog = new ArrayList<>();
    public NeuralNetArrayImpl[] neuralNetArrays;
    public int actionSize;
    private double qTrainSampleError = 0;

    public EnsembleNeuralNet(final int actionSize) {
```

```
        this.actionSize = actionSize;

        this.neuralNetArrays = new NeuralNetArrayImpl[actionSize];

        for (int i = 0; i < actionSize; i++) {
            this.neuralNetArrays[i] =
                    new NeuralNetArrayImpl(
                            Action.values()[i].name(),
ImmutableState.builder().build());
        }
    }

    double[] forward(final State state) {
        final double[] res = new double[this.neuralNetArrays.length];
        for (int i = 0; i < res.length; i++) {
            res[i] = this.neuralNetArrays[i].forward(state).data[0]
[0];
        }
        return res;
    }

    void train(final Map<State, double[]> qTable) {
        final double[][][] trainX =
                new double[this.neuralNetArrays.length]
[qTable.size()]

[ImmutableState.builder().build().getIndexedStateValue().length];

        final double[][][] trainY = new
double[this.neuralNetArrays.length][qTable.size()][1];

        int curCount = 0;

        for (final Map.Entry<State, double[]> entry :
qTable.entrySet()) {
            for (int i = 0; i < this.neuralNetArrays.length; i++) {
                trainX[i][curCount] =
entry.getKey().getTrainingData(KEEP);
```

```
                trainY[i][curCount] = new double[] {entry.getValue()
[i]};
            }
            curCount++;
        }

        for (int i = 0; i < this.neuralNetArrays.length; i++) {
            this.neuralNetArrays[i].train(trainX[i], trainY[i]);
        }
    }

    void train(final Map<State, double[]> qTable, final int epoch,
final double errorThreshold) {
        int elapsedEpoch = 0;
        double totalError = 0;
        double curEpochError;
        double rmsError;
        do {
            curEpochError = 0;
            for (final Map.Entry<State, double[]> entry :
qTable.entrySet()) {
                for (int i = 0; i < this.neuralNetArrays.length; i++)
{
                    curEpochError +=
                            this.neuralNetArrays[i].train(

 entry.getKey().getTrainingData(KEEP),
                                    new double[] {entry.getValue()
[i]});
                }
            }

            rmsError = this.rmse(curEpochError, qTable.size());

            totalError += rmsError;

            if (elapsedEpoch % 50 == 0) this.errorLog.add(rmsError);
```

```
            curEpochError /= new
Integer(qTable.size()).doubleValue();

            if (elapsedEpoch++ % 1000 == 0)
                System.out.printf(
                        "NN: Current Epoch: %d %n Current Error: %f%n
Current RMSE: %f%n",
                        elapsedEpoch, curEpochError, rmsError);

        } while (elapsedEpoch++ < epoch && rmsError >
errorThreshold);

        System.out.printf(
                "NN trained for %d epochs, reached error per
eentry.getKey().getTrainingData(KEEP)poch = %.2f, best error:
%.2f%n",
                elapsedEpoch, totalError / elapsedEpoch, rmsError);
    }

    public int chooseAction(final State currentState) {
        if (Math.random() < RANDOM_RATE) {
            return Action.values()[new
Random().nextInt(Action.values().length)].ordinal();
        }

        double max = Double.NEGATIVE_INFINITY;
        Action action = Action.AHEAD;

        final double[] curQ = this.forward(currentState);

        for (int i = 0; i < Action.values().length; i++) {
            if (max < curQ[i]) {
                action = Action.values()[i];
                max = curQ[i];
            }
        }

        return action.ordinal();
    }
```

```
    public void qTrain(
            final double reward,
            final State prevState,
            final Action prevAction,
            final State curState) {

        final double[] prevQ = this.forward(prevState);
        final double curQ =
Arrays.stream(this.forward(curState)).max().orElse(0);
        final double loss = ALPHA * (reward + GAMMA * curQ -
prevQ[prevAction.ordinal()]);
        final double updatedQ = prevQ[prevAction.ordinal()] + loss;

        this.qTrainSampleError +=
                this.neuralNetArrays[prevAction.ordinal()]
                        .loss(
                                new Matrix(
                                        new double[][] {
                                            new double[]
{prevQ[prevAction.ordinal()]}
                                        }),
                                new Matrix(new double[][] {new
double[] {updatedQ}}),
                                0.5)
                        .sumValues();

        this.neuralNetArrays[prevAction.ordinal()].backpropagation(
                new Matrix(new double[][] {new double[]
{prevQ[prevAction.ordinal()]}}),
                new Matrix(new double[][] {new double[]
{updatedQ}}));
    }

    private double rmse(final double sampleError, final int
sampleSize) {
        return Math.sqrt(sampleError / new
Integer(sampleSize).doubleValue());
    }
```

```
    public void load(final File dataFile) throws IOException {}
}
```

org.homework.neuralnet.NeuralNetArrayImpl

```
package org.homework.neuralnet;

import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import org.homework.neuralnet.matrix.Matrix;
import org.homework.robot.model.Action;
import org.homework.robot.model.ImmutableState;
import org.homework.robot.model.State;

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.Random;

@NoArgsConstructor
@Getter
@Setter
public class EnsembleNeuralNet {
    private static final double ALPHA = 0.1;
    private static final double GAMMA = 0.5;
    private static final double RANDOM_RATE = 0.1;
    private static final boolean KEEP = false;
    private final List<Double> errorLog = new ArrayList<>();
    public NeuralNetArrayImpl[] neuralNetArrays;
    public int actionSize;
    private double qTrainSampleError = 0;

    public EnsembleNeuralNet(final int actionSize) {
```

```
        this.actionSize = actionSize;

        this.neuralNetArrays = new NeuralNetArrayImpl[actionSize];

        for (int i = 0; i < actionSize; i++) {
            this.neuralNetArrays[i] =
                    new NeuralNetArrayImpl(
                            Action.values()[i].name(),
ImmutableState.builder().build());
        }
    }

    double[] forward(final State state) {
        final double[] res = new double[this.neuralNetArrays.length];
        for (int i = 0; i < res.length; i++) {
            res[i] = this.neuralNetArrays[i].forward(state).data[0]
[0];
        }
        return res;
    }

    void train(final Map<State, double[]> qTable) {
        final double[][][] trainX =
                new double[this.neuralNetArrays.length]
[qTable.size()]

[ImmutableState.builder().build().getIndexedStateValue().length];

        final double[][][] trainY = new
double[this.neuralNetArrays.length][qTable.size()][1];

        int curCount = 0;

        for (final Map.Entry<State, double[]> entry :
qTable.entrySet()) {
            for (int i = 0; i < this.neuralNetArrays.length; i++) {
                trainX[i][curCount] =
entry.getKey().getTrainingData(KEEP);
```

```
                trainY[i][curCount] = new double[] {entry.getValue()
[i]};
            }
            curCount++;
        }

        for (int i = 0; i < this.neuralNetArrays.length; i++) {
            this.neuralNetArrays[i].train(trainX[i], trainY[i]);
        }
    }

    void train(final Map<State, double[]> qTable, final int epoch,
final double errorThreshold) {
        int elapsedEpoch = 0;
        double totalError = 0;
        double curEpochError;
        double rmsError;
        do {
            curEpochError = 0;
            for (final Map.Entry<State, double[]> entry :
qTable.entrySet()) {
                for (int i = 0; i < this.neuralNetArrays.length; i++)
{
                    curEpochError +=
                            this.neuralNetArrays[i].train(

 entry.getKey().getTrainingData(KEEP),
                                    new double[] {entry.getValue()
[i]});
                }
            }

            rmsError = this.rmse(curEpochError, qTable.size());

            totalError += rmsError;

            if (elapsedEpoch % 50 == 0) this.errorLog.add(rmsError);
```

```
            curEpochError /= new
Integer(qTable.size()).doubleValue();

            if (elapsedEpoch++ % 1000 == 0)
                System.out.printf(
                        "NN: Current Epoch: %d %n Current Error: %f%n
Current RMSE: %f%n",
                        elapsedEpoch, curEpochError, rmsError);

        } while (elapsedEpoch++ < epoch && rmsError >
errorThreshold);

        System.out.printf(
                "NN trained for %d epochs, reached error per
eentry.getKey().getTrainingData(KEEP)poch = %.2f, best error:
%.2f%n",
                elapsedEpoch, totalError / elapsedEpoch, rmsError);
    }

    public int chooseAction(final State currentState) {
        if (Math.random() < RANDOM_RATE) {
            return Action.values()[new
Random().nextInt(Action.values().length)].ordinal();
        }

        double max = Double.NEGATIVE_INFINITY;
        Action action = Action.AHEAD;

        final double[] curQ = this.forward(currentState);

        for (int i = 0; i < Action.values().length; i++) {
            if (max < curQ[i]) {
                action = Action.values()[i];
                max = curQ[i];
            }
        }

        return action.ordinal();
    }
```

```
    public void qTrain(
            final double reward,
            final State prevState,
            final Action prevAction,
            final State curState) {

        final double[] prevQ = this.forward(prevState);
        final double curQ =
Arrays.stream(this.forward(curState)).max().orElse(0);
        final double loss = ALPHA * (reward + GAMMA * curQ -
prevQ[prevAction.ordinal()]);
        final double updatedQ = prevQ[prevAction.ordinal()] + loss;

        this.qTrainSampleError +=
                this.neuralNetArrays[prevAction.ordinal()]
                        .loss(
                                new Matrix(
                                        new double[][] {
                                            new double[]
{prevQ[prevAction.ordinal()]}
                                        }),
                                new Matrix(new double[][] {new
double[] {updatedQ}}),
                                0.5)
                        .sumValues();

        this.neuralNetArrays[prevAction.ordinal()].backpropagation(
                new Matrix(new double[][] {new double[]
{prevQ[prevAction.ordinal()]}}),
                new Matrix(new double[][] {new double[]
{updatedQ}}));
    }

    private double rmse(final double sampleError, final int
sampleSize) {
        return Math.sqrt(sampleError / new
Integer(sampleSize).doubleValue());
    }
```

```
    public void load(final File dataFile) throws IOException {}
}
```

org.homework.neuralnet.matrix.Matrix

```
package org.homework.neuralnet.matrix;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import org.homework.neuralnet.matrix.op.ElementTransformation;
import org.homework.util.Util;

import java.util.Arrays;

@AllArgsConstructor
@NoArgsConstructor
@Getter
@Setter
public class Matrix {
    public int rowNum;
    public int colNum;
    public double[][] data;

    public Matrix(final double[][] _data) {
        this.rowNum = _data.length;
        this.colNum = _data[0].length;
        this.data = _data;
    }

    public Matrix(final int r, final int c) {
        this(new double[r][c]);
    }

    public static Matrix initZeroMatrix(final int r, final int c) {
        return new Matrix(r, c, new double[r][c]);
```

```
    }

    public static Matrix initRandMatrix(final int r, final int c) {
        return new Matrix(r, c).elementWiseOp(0, (a, b) ->
Math.random());
    }

    /**
     * return a matrix which is the result of this matrix dot
products with other matrix
     *
     * @param matrix matrix on the rhs
     * @return a matrix which is the result of this matrix dot
products with other matrix
     */
    public Matrix mmul(final Matrix matrix) {
        final double[][] newData = new double[this.data.length]
[matrix.data[0].length];
        for (int i = 0; i < newData.length; i++) {
            for (int j = 0; j < newData[i].length; j++) {
                newData[i][j] = this.multiplyMatricesCell(this.data,
matrix.data, i, j);
            }
        }
        return new Matrix(newData);
    }

    private double multiplyMatricesCell(
            final double[][] firstMatrix,
            final double[][] secondMatrix,
            final int row,
            final int col) {
        double cell = 0;
        for (int i = 0; i < secondMatrix.length; i++) {
            cell += firstMatrix[row][i] * secondMatrix[i][col];
        }
        return cell;
    }
```

```
    public Matrix transpose() {
        final double[][] temp = new double[this.data[0].length]
[this.data.length];
        for (int i = 0; i < this.rowNum; i++)
            for (int j = 0; j < this.colNum; j++) temp[j][i] =
this.data[i][j];
        return new Matrix(temp);
    }

    public Matrix mul(final double x) {
        return this.elementWiseOp(x, (a, b) -> a * b);
    }

    public Matrix mul(final Matrix matrix) {
        return this.elementWiseOp(matrix, (a, b) -> a * b);
    }

    public Matrix add(final double x) {
        return this.elementWiseOp(x, Double::sum);
    }

    public Matrix add(final Matrix matrix) {
        return this.elementWiseOp(matrix, Double::sum);
    }

    public Matrix sub(final double x) {
        return this.elementWiseOp(x, (a, b) -> a - b);
    }

    public Matrix sub(final Matrix matrix) {
        return this.elementWiseOp(matrix, (a, b) -> a - b);
    }

    public Matrix elementWiseOp(
            final double operand, final ElementTransformation
elementTransformation) {
        final double[][] dataClone =
Util.getDeepArrayCopy(this.data);
        for (int i = 0; i < dataClone.length; i++) {
```

```
            for (int j = 0; j < dataClone[0].length; j++) {
                dataClone[i][j] =
elementTransformation.op(dataClone[i][j], operand);
            }
        }
        return new Matrix(dataClone);
    }

    public Matrix elementWiseOp(
            final Matrix matrix, final ElementTransformation
elementTransformation) {
        final double[][] dataClone =
Util.getDeepArrayCopy(this.data);
        for (int i = 0; i < dataClone.length; i++) {
            for (int j = 0; j < dataClone[0].length; j++) {
                dataClone[i][j] =
elementTransformation.op(dataClone[i][j], matrix.data[i][j]);
            }
        }
        return new Matrix(dataClone);
    }

    public double sumValues() {
        return
Arrays.stream(this.data).flatMapToDouble(Arrays::stream).sum();
    }

    @Override
    public boolean equals(final Object obj) {
        if (obj == this) return true;

        if (!(obj instanceof Matrix)) return false;

        final Matrix matrix = (Matrix) obj;

        return Arrays.deepEquals(this.data, matrix.data);
    }

    @Override
```

```
    public int hashCode() {
        return Arrays.deepHashCode(this.data);
    }
}
```

org.homework.neuralnet.matrix.op.ElementTransformation

```
package org.homework.neuralnet.matrix.op;

public interface ElementTransformation {
    double op(double x, double y);
}
```

org.homework.robot.model.Memory

```
package org.homework.robot.model;

import org.immutables.value.Value;

@Value.Immutable
public abstract class Memory {
    @Value.Default
    public State getPrevState() {
        return ImmutableState.builder().build();
    }

    @Value.Default
    public State getCurState() {
        return ImmutableState.builder().build();
    }

    @Value.Default
    public double getReward() {
        return 1.0;
    }

    @Value.Default
```

```
    public Action getPrevAction() {
        return Action.values()[0];
    }

    @Override
    public String toString() {
        return String.format("Prev: %s %n Current: %s %n",
this.getPrevState(), this.getCurState());
    }
}
```

org.homework.robot.model.State

```
package org.homework.robot.model;

import org.homework.util.Util;
import org.immutables.value.Value;

import java.util.Arrays;

@Value.Immutable
public abstract class State {
    @Value.Default
    public StateName.HP getCurrentHP() {
        return StateName.HP.MID;
    }

    @Value.Default
    public StateName.ENEMY_HP getCurrentEnemyHP() {
        return StateName.ENEMY_HP.MID;
    }

    @Value.Default
    public StateName.DISTANCE_TO_ENEMY getCurrentDistanceToEnemy() {
        return StateName.DISTANCE_TO_ENEMY.MID;
    }

    @Value.Default
```

```
    public StateName.DISTANCE_TO_WALL getCurrentDistanceToWall() {
        return StateName.DISTANCE_TO_WALL.MID;
    }

    @Value.Default
    public StateName.ENEMY_ROBOT_HEADING
getCurrentEnemyRobotHeading() {
        return StateName.ENEMY_ROBOT_HEADING.MID;
    }

    @Value.Default
    public StateName.X getX() {
        return StateName.X.MID;
    }

    @Value.Default
    public StateName.Y getY() {
        return StateName.Y.MID;
    }

    @Value.Default
    public double[] getDequantizedState() {
        return new double[7];
    }

    @Value.Default
    public double[] getTrainingData(final boolean keep) {
        return keep
                ?
Util.getDoubleArrayFromIntArray(this.getIndexedStateValue())
                : this.getDequantizedState();
    }

    @Value.Default
    public int[] getIndexedStateValue() {
        return new int[] {
            this.getCurrentHP().ordinal(),
            this.getCurrentEnemyHP().ordinal(),
            this.getCurrentDistanceToEnemy().ordinal(),
```

```
            this.getCurrentDistanceToWall().ordinal(),
            this.getCurrentEnemyRobotHeading().ordinal(),
            this.getX().ordinal(),
            this.getY().ordinal()
        };
    }

    @Override
    public String toString() {
        return String.format(
                "Current State Array: %s",
Arrays.toString(this.getIndexedStateValue()));
    }

    @Override
    public boolean equals(final Object obj) {
        if (obj == this) return true;

        if (!(obj instanceof State)) return false;

        final State state = (State) obj;

        return Arrays.equals(this.getDequantizedState(),
state.getDequantizedState());
    }

    @Override
    public int hashCode() {
        return Arrays.hashCode(this.getDequantizedState());
    }
}
```

org.homework.robot.NNRobot

```
/*
 * Copyright (c) 2001-2022 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 org.homework.robot;

import org.homework.neuralnet.NeuralNetArrayImpl;
import org.homework.replaymemory.ReplayMemory;
import org.homework.robot.model.Action;
import org.homework.robot.model.ImmutableMemory;
import org.homework.robot.model.ImmutableState;
import org.homework.robot.model.Memory;
import org.homework.robot.model.State;
import robocode.BattleEndedEvent;
import robocode.RoundEndedEvent;
import robocode.ScannedRobotEvent;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

/**
 * 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 NNRobot extends AIRobot {
    private static final NeuralNetArrayImpl neuralNetArray =
            new NeuralNetArrayImpl(
                    "Default Neural Net",
                    ImmutableState.builder().build(),
                    25,
                    Action.values().length,
```

```
                    0.1,
                    0.0,
                    true);
    private static final int MINI_BATCH_SIZE = 20;
    private static final int MEMORY_SIZE = 200;
    private static final String PREV_TRAINED_WEIGHT = "weights.txt";
    private static final ReplayMemory<Memory> replayMemory = new
ReplayMemory<>(MEMORY_SIZE);
    private static final List<Double> qTrainRMSE = new ArrayList<>();

    @Override
    public void run() {
        this.setAdjustGunForRobotTurn(true);
        this.setAdjustRadarForGunTurn(true);
        this.load();
        while (true) {
            this.setTurnRadarLeftRadians(2 * Math.PI);
            this.scan();
            this.setCurrentAction(this.chooseCurrentAction());
            this.act();
            this.updateQValue();
            this.setReward(.0);
        }
    }

    @Override
    public Action chooseCurrentAction() {
        return Action.values()
[neuralNetArray.chooseAction(currentState)];
    }

    @Override
    /** Update q value */
    public void updateQValue() {
        final Memory memory =
                ImmutableMemory.builder()
                        .curState(currentState)
                        .prevState(prevState)
                        .reward(this.getReward())
```

```
                        .prevAction(this.getCurrentAction())
                        .build();
        replayMemory.add(memory);
        if (replayMemory.sizeOf() >= MINI_BATCH_SIZE) {
            neuralNetArray.setQTrainSampleError(0);
            int curTrainSize = 0;
            for (final Object object :
replayMemory.randomSample(MINI_BATCH_SIZE)) {
                final Memory experienceBatch = (Memory) object;
                neuralNetArray.qTrain(
                        experienceBatch.getReward(),
                        experienceBatch.getPrevState(),
                        experienceBatch.getPrevAction(),
                        experienceBatch.getCurState());
                curTrainSize++;
            }
            qTrainRMSE.add(

 neuralNetArray.rmse(neuralNetArray.getQTrainSampleError(),
curTrainSize));
        }
    }

    @Override
    public void onRoundEnded(final RoundEndedEvent event) {
        totalRound++;
        this.logWinStatue(100);
        if (totalRound % 100 == 0) {
            log.writeRMSEToFile(this.getDataFile("rmse.log"),
qTrainRMSE, 100, true);
            qTrainRMSE.clear();
        }
    }

    /**
     * Get current scanned state
     *
     * @param event The event of scan
     * @return State represent current
```

```
     */
    @Override
    public State findCurrentState(final ScannedRobotEvent event) {
        this.setBearing(event.getBearing());
        return ImmutableState.builder()
                .dequantizedState(
                        this.getEnergy() / 100.0,
                        event.getEnergy() / 100.0,
                        event.getDistance() / 1000.0,
                        this.getDistanceFromWall(this.getX(),
this.getY()) / 1000.0,
                        event.getBearing() / 360.0,
                        this.getX() / 800.0,
                        this.getY() / 600.0)
                .build();
    }

    @Override
    public void load() {
        try {

 neuralNetArray.load(this.getDataFile(PREV_TRAINED_WEIGHT));
            System.out.println("Loaded");
        } catch (final IOException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public void onBattleEnded(final BattleEndedEvent event) {
        neuralNetArray.save(this.getDataFile((PREV_TRAINED_WEIGHT)),
true);
    }
}
```

org.homework.neuralnet.EnsembleNeuralNetTest

```
package org.homework.neuralnet;

import org.homework.rl.LUTImpl;
import org.homework.robot.model.Action;
import org.homework.robot.model.ImmutableState;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.io.IOException;

class EnsembleNeuralNetTest {

    @BeforeEach
    void setUp() {}

    @Test
    void QTRAIN_WORK() {
        final EnsembleNeuralNet mockEnsembleNeuralNet =
                new EnsembleNeuralNet(Action.values().length);

        mockEnsembleNeuralNet.qTrain(
                0.1,
                ImmutableState.builder().build(),
                Action.AHEAD,
                ImmutableState.builder().build());
    }

    @Test
    void ONE_NN_FOR_EACH_ACTION() throws IOException {
        // lut q table gained from AI robot
        final String offlineTrainingDate = "robot-log/AIRobot-crazy-
robot.txt";
        final LUTImpl lut = new
LUTImpl(ImmutableState.builder().build());
        lut.load(offlineTrainingDate);
```

```
        final EnsembleNeuralNet ensembleNeuralNet = new
EnsembleNeuralNet(Action.values().length);

        ensembleNeuralNet.train(lut.qTable);

        for (int i = 0; i < ensembleNeuralNet.neuralNetArrays.length;
i++) {
            System.out.printf(
                    "%s error log: %s%n",
                    Action.values()[i].name(),
ensembleNeuralNet.neuralNetArrays[i].getErrorLog());
        }
    }

    @Test
    void ONE_NN_FOR_EACH_ACTION_AGGREGATE_TRAIN() throws IOException
{
        // lut q table gained from AI robot
        final String offlineTrainingDate = "robot-log/AIRobot-crazy-
robot.txt";
        final LUTImpl lut = new
LUTImpl(ImmutableState.builder().build());
        lut.load(offlineTrainingDate);

        final EnsembleNeuralNet ensembleNeuralNet = new
EnsembleNeuralNet(Action.values().length);

        ensembleNeuralNet.train(lut.qTable, 10000, 0.01);

        System.out.println(ensembleNeuralNet.getErrorLog());
    }
}
```

org.homework.neuralnet.NeuralNetArrayImplTest

```
package org.homework.neuralnet;

import org.homework.rl.LUTImpl;
import org.homework.robot.model.Action;
import org.homework.robot.model.ImmutableState;
import org.homework.robot.model.State;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.io.IOException;
import java.util.Map;

import static org.homework.TestUtil.getRandomInputVector;
import static org.homework.TestUtil.printNNTrainState;
import static org.junit.jupiter.api.Assertions.assertEquals;

class NeuralNetArrayImplTest {

    NeuralNetArrayImpl neuralNetArray;

    int testArgNumInputs = 2;

    int testArgNumHidden = 4;

    int testArgNumOutputs = 1;

    @BeforeEach
    void setUp() {
        this.neuralNetArray =
                new NeuralNetArrayImpl(
                        this.testArgNumInputs,
                        this.testArgNumHidden,
                        this.testArgNumOutputs,
                        .1,
                        .0,
                        1,
                        0,
```

```
                        false);
    }

    @Test
    void TEST_TRAIN_XOR_RANDOM() {
        final double[][] x = getRandomInputVector(10,
this.testArgNumInputs);
        final double[][] yHat = getRandomInputVector(10,
this.testArgNumOutputs);
        this.neuralNetArray.train(x, yHat);
        printNNTrainState(x, yHat,
this.neuralNetArray.batchForward(x).getData());
    }

    @Test
    void TEST_TRAIN_XOR_THEN_APPLY() {
        final double[][] x = new double[][] {{0, 0}, {0, 1}, {1, 0},
{1, 1}};
        final double[][] yHat = new double[][] {{0}, {1}, {1}, {0}};
        this.neuralNetArray.train(x, yHat);
        printNNTrainState(x, yHat,
this.neuralNetArray.batchForward(x).getData());
    }

    @Test
    void TEST_TRAIN_XOR_BIPOLAR() {
        this.neuralNetArray.setBipolar(true);
        final double[][] x = new double[][] {{-1, -1}, {-1, 1}, {1,
-1}, {1, 1}};
        final double[][] yHat = new double[][] {{-1}, {1}, {1},
{-1}};
        this.neuralNetArray.train(x, yHat);
        printNNTrainState(x, yHat,
this.neuralNetArray.batchForward(x).getData());
    }

    @Test
    void TEST_NN_FOR_DIFFERENT_DIMENSION() {
        final State mockState = ImmutableState.builder().build();
```

```
        this.neuralNetArray = new NeuralNetArrayImpl("mockNN",
mockState);
        assertEquals(
                Action.values().length,
                this.neuralNetArray
                        .forward(new double[1]
[mockState.getIndexedStateValue().length])
                        .getColNum());
    }

    @Test
    void OFFLINE_TRAINING_PROCESS() throws IOException {
        final NeuralNetArrayImpl nn =
                new NeuralNetArrayImpl("NN",
ImmutableState.builder().build(), 10, 0.3, 0.1, true);

        // lut q table gained from AI robot
        final String offlineTrainingDate = "robot-log/AIRobot-crazy-
robot.txt";
        final LUTImpl lut = new
LUTImpl(ImmutableState.builder().build());
        lut.load(offlineTrainingDate);

        final double[][] trainX =
                new double[lut.qTable.size()]

[ImmutableState.builder().build().getIndexedStateValue().length];

        final double[][] trainY = new double[lut.qTable.size()]
[Action.values().length];

        int count = 0;

        for (final Map.Entry<State, double[]> entry :
lut.qTable.entrySet()) {
            trainX[count] = entry.getKey().getTrainingData(true);
            trainY[count++] = entry.getValue();
        }
```

```
        nn.train(trainX, trainY);

        final String nnWeightFile = "./Weights";

        nn.save(nnWeightFile);

        final NeuralNetArrayImpl nnTestLoad =
                new NeuralNetArrayImpl(
                        "NNTest", ImmutableState.builder().build(),
10, 0.3, 0.1, true);

        nnTestLoad.load(nnWeightFile);

        assertEquals(nn.getInputToHiddenWeight(),
nnTestLoad.getInputToHiddenWeight());
        assertEquals(nn.getHiddenToOutputWeight(),
nnTestLoad.getHiddenToOutputWeight());
        assertEquals(nn.getHiddenLayerBias(),
nnTestLoad.getHiddenLayerBias());
        assertEquals(nn.getOutputLayerBias(),
nnTestLoad.getOutputLayerBias());
    }
}
```