**FINAL PROJECT REPORT: MILESTONE 3 – REINFORCEMENT LEARNING WITH BACKPROPAGATION**

**Abbreviations**

BP - Backpropagation

LUT – Lookup Table

NN – Neural Net

RL – Reinforcement Learning

RMS – Root Mean Square

TD – Temporal Different Learning Algorithm

**Introduction and Background**

At the end of milestone 2, I had implemented a robot (“FizzBotFinal”) in Robocode that learned how to battle an enemy robot (“TrackFire”) through reinforcement learning (RL) using the Temporal Difference (TD) learning algorithm. The most important data from the trained robot was saved in a lookup table (LUT). The term “QValues” will be used to refer to this data in this report, where each QValue in the LUT represents the amount of reward the robot can expect to receive in a certain state – denoting how good that state is for the robot’s long-term goal: to beat the enemy robot and win the game.

In milestone 2, once FizzBotFinal had been trained (through either on or off policy training), it could then play games against the TrackFire by querying the LUT for the maximum Qvalue given its current state, and choose the best action to take next.

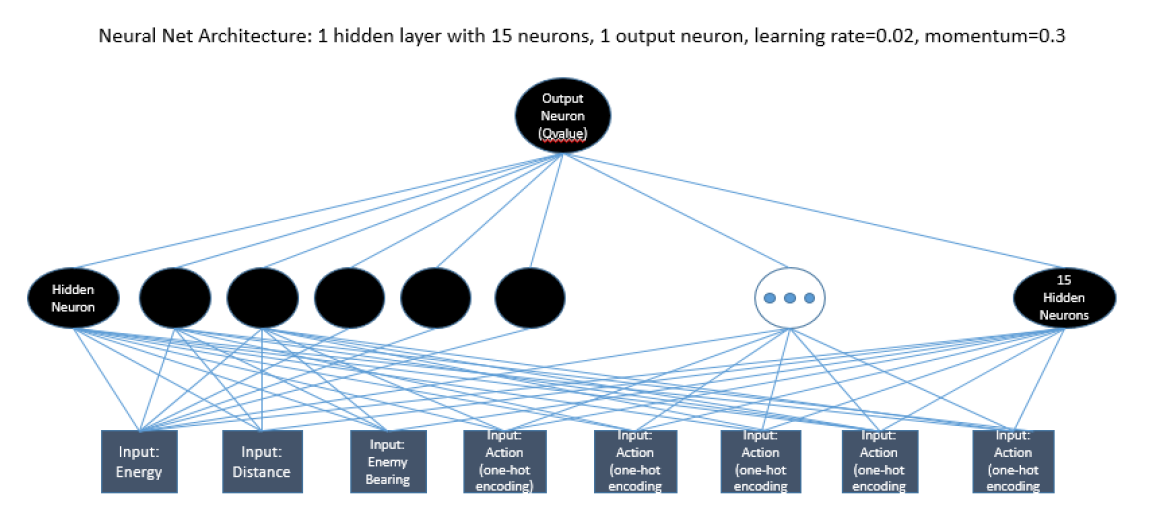
The intent of Milestone 3 is to replace the LUT we defined through TD training with a neural net (NN). In order to make the RL-LUT approach tractable, state-space reduction was applied to ensure that the LUT did not blow up in size. A large LUT means a couple of things:

- More memory is required to house the LUT

- RL-TD takes longer to converge to the optimal value function

However, doing too much state space reduction can result in a sparsely populated LUT which is also not a good thing because there is less accurate state data, and this in turn can reduce the quality of learning. In real life applications, QValue function approximation is attempted as a way of bridging the gap between a LUT that is too large and that takes too long to converge, and a LUT that is too sparsely populated and contains too little information to actually be effective. In this milestone, we will attempt to approximate the value function provided by my LUT from part 2 with a. That is, FizzBotFinal in Robocode should use the NN as its *brain* (instead of the LUT as was done before) while it continues to learn how to battle against the enemy robot. We are interested to see how well the RL-NN approach can *approximate* the value function when there is no state-space reduction applied to the input state-action vectors.

*1a) Describe the architecture of your neural network and how the training set captured from Part 2 was used to “offline” train it mentioning any input representations that you may have considered. Note that you have 3 different options for the high-level architecture. A net with a single Q output, a net with a Q output for each action, separate nets each with a single output for each action. Draw a diagram for your neural net labeling the inputs and outputs.*



**Figure 1: Finalized Neural Net architecture for milestone 3**

Figure 1 shows the final NN architecture I am using for FizzBotFinal. In section 1b I would explain how I arrived at this specified number of hidden neurons, and the values for learning rate and momentum. One of the main advantages to having the LUT from part 2 is that I have a set of training data to apply to my NN. With this training data, I can evaluate different parameters for the NN to find out which set gives me the best root mean square (RMS) error. Note: RMS is a measure for how well the NN predicts the output for each new input data vector. The lower the RMS, the better prediction accuracy the NN will provide [1]. More of this will be discussed in section 1b.

In this NN, there is one output neuron representing the QValue for the input state-action feature vector. During online RL, I enumerate all actions for the current state of the robot (as seen by Robocode) and select the action that corresponds to the highest QValue output to perform next; this is called a “greedy” policy. Online training to the NN is applied by the same TD algorithm used in milestone 2, and the NN itself uses backpropagation to update its weights. Since milestone 2 I have revised my robot to have the state actions described in Table 1.

**Table 1: Input Representations for LUT and Neural Net:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **ENERGY(0-100+)** | **DISTANCE(0-1000)** | **ENEMY\_BEARING**  **(-180, 180)** | **ACTION** |
| RL-LUT Input (Milestone 2) | Intervals of 10  0 - 11 | Intervals of 10  0 - 11 | Intervals of 10 degrees, 0-35 | Aim and Fire  Strafe  Circle  Diagonal Left  Diagonal Right  0-5 |
| RL-NN Input during offline training | 1.0 – 11.0 | 1.0 – 11.0 | -1.8 to +1.8 | -1 -1 -1 -1 +1  -1 -1 -1 +1 -1  -1 -1 +1 -1 -1  -1 +1 -1 -1 -1  +1 -1 -1 -1 -1 |
| RL-NN Input during online training | 1.1, 1.2, 1.3, ...-11.0  (Continuous values, with significant digits up to a full Java double) | 1.1, 1.2, 1.3, ...-11.0  (Continuous values, with significant digits up to a full Java double) | -1.80, -1.79…-1.8  (Continuous values, with significant digits up to a full Java double) | Same as above |

Note, with the RL-LUT, I just enumerated state action pairs, and did not care about the actual input representations of the state-action. I had just used the state-action enumerations as indexes to my LUT. However, since I am using the LUT to train my NN, the input representations do matter. Here are some things I did to update the input representation for the NN (summarized in row 3 of Table 1):

- Energy, Distance and Enemy Bearing are doubles and all within the same order of magnitude of each other. For example, energy = energy/10, distance = distance/100, and enemy\_bearing = enemy\_bearing/10. This is done because I do not want the NN to be susceptible to huge changes in orders of magnitude – this would require the weights of my neural net to account for this variation. An alternative option is to normalize the data, but I thought this representation would be fine for my purposes.

- Action is not an ordinal value, and so I should enumerate the action values using 1-5. I choose to represent actions using a “one-hot-encoded” scheme, as this is one good way to represent categorical data.

- Finally, I try not to have inputs that are zero based where possible. I.e. I use a bipolar format for action. We learned in milestone 1 that the neurons apply an activation to the weighted sum of their inputs. If the inputs are zero based, then training can take longer.

The NN requires an activation function for the neurons. In this scheme, I choose to keep the bipolar activation function, and pre-process the output QValues from the RL-LUT to scale between [-1, 1] using the following formulae:

1. 𝑛𝑜𝑟𝑚𝑎𝑙𝑖𝑧𝑒𝑑𝑉𝑎𝑙𝑢𝑒 =

(2) 𝑠𝑐𝑎𝑙𝑒𝑑𝑉𝑎𝑙𝑢𝑒 = 𝑛𝑜𝑟𝑚𝑎𝑙𝑖𝑧𝑒𝑑𝑉𝑎𝑙𝑢𝑒∗(𝑏𝑖𝑝𝑜𝑙𝑎𝑟𝑚𝑎𝑥−𝑏𝑖𝑝𝑜𝑙𝑎𝑟𝑚𝑜𝑛)+𝑏𝑖𝑝𝑜𝑙𝑎𝑟𝑚𝑖𝑛

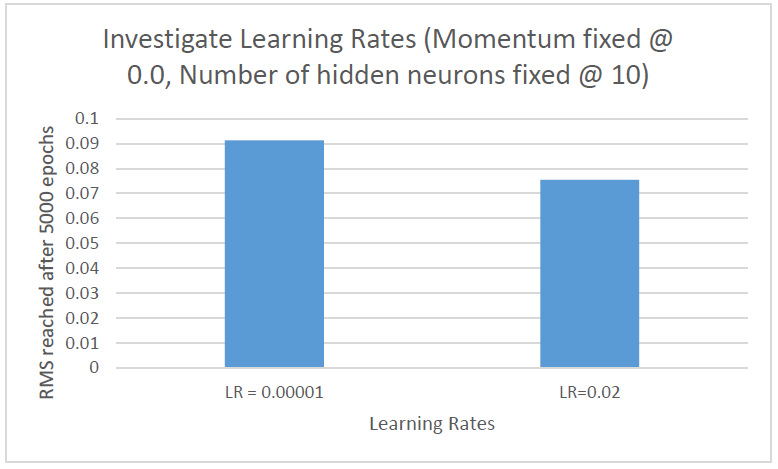
Where Qmin and Qmax are the minimum and maximum QValues from the trained LUT. In my case they were -17.8618 and 1.35524 respectively. Bipolar min and max are -1 and +1 respectively. By doing this, I can continue to use the bipolar sigmoid activation function in the NN and backpropagation algorithms.

*1b) Show (as a graph) the results of training your neural network using the contents of the LUT from Part 2. You may have attempted learning using different hyper-parameter values (i.e. momentum, learning rate, number of hidden neurons). Include graphs showing which parameters best learned your LUT data. Compute the RMS error for your best results.*

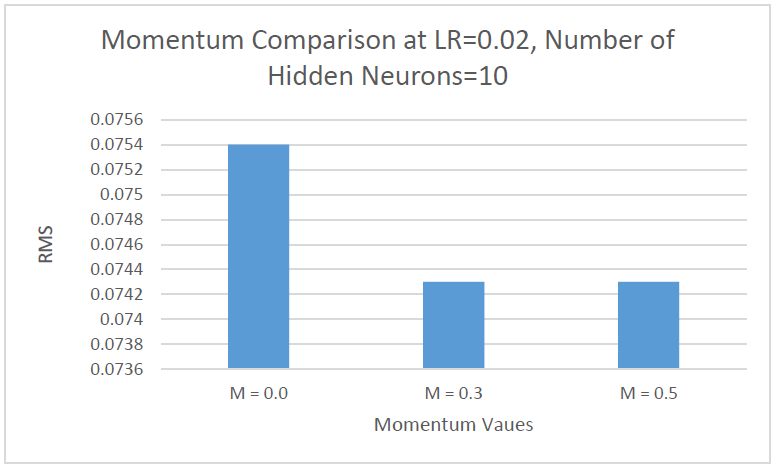
My method for finding parameters for the NN is purely experimental and based on the best RMS results achieved when training the NN offline using the LUT training data. Initially, I keep the number of hidden layer neurons and momentum fixed, and test different learning rates. Once I find the best performing learning rate, I fix that and test the other parameters one by one. I continue with this method until I settle at all parameters for my NN. I recognize that this experimental method may not be the best way to define my NN architecture because the decisions are order dependent, however since I still manage to find a reasonable RMS error rate, I decided to continue and test this NN with Robocode.

Aside: for future work, I would consider evaluating these parameters in an order-independent way, i.e. I would test each value for each parameter with all other values from all other parameters that would result in a 3D chart – from here I would choose the parameters that result in the lowest RMS error as well.

Since my state-action feature vector length is 8, I first started testing 10 hidden neurons in order to gauge the RMS error for different learning rate and momentum values. 10 hidden neurons seemed like a reasonable number to start with since I do not want to over-fit the data (which can happen when the number of hidden neurons is much larger than the number of input neurons) [2], but rather approximate the QValue function if possible. Figure 2 shows a comparison of 2 different learning rates used on the NN for 4000 epochs. The training set is the normalized data from the LUT. Here we can see that a learning rate of 0.02 performs better, and so now I can investigate different values for the number of hidden neurons and momentum.

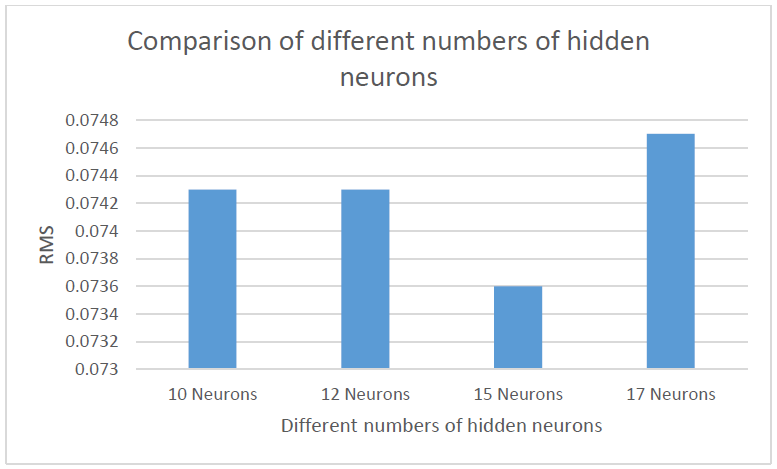


**Figure 2: Learning Rate Experiment – 0.02 achieves a better RMS in 4000 training epochs.**



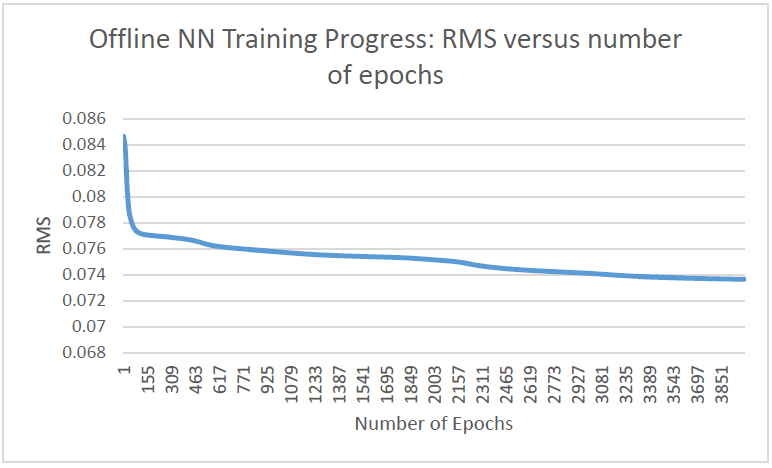
**Figure 3: With learning rate fixed at 0.02, and hidden neurons at 10, a momentum of 0.3 results in the best RMS value of 0.0743 over 4000 epochs.**

In figure 3, I keep learning rate fixed at 0.02, and the number of hidden neurons fixed at 10 while investigating different momentum values. At 4000 epochs, we can see that a momentum of 0.3 results in the best RMS value of 0.0743. RMS is a measure for how much error there is in each NN prediction.



**Figure 4: With learning rate fixed at 0.02, and momentum fixed at 0.3, we investigate the performance of different numbers of hidden neurons. This experiment shows 15 hidden neurons achieves the best RMS value of 0.0736**

In Figure 4, we fix learning rate and momentum to the best performing values already found, and test different numbers of hidden neurons. Here we can see that 15 hidden layer neurons achieves the best RMS value of 0.0736 after 4000 epochs.



**Figure 5: This graph shows the RMS value as the NN is trained offline using the LUT training data set.**

In figure 5 you can see that training is stopped at 4000 epochs. 4000 epochs was chosen as the upper bound in the interest of time and completing all of these experiments. Ideally training should continue until the RMS value stabilizes. In this graph it looks like the RMS is still decreasing, however a closer look at the variance of the RMS values shows that it is quite small at 4000 epochs, making this a reasonable stopping point.

*1c) Try mitigating or even removing any quantization or dimensionality reduction (henceforth referred to as state space reduction) that you may have used in part 2. A side-by-side comparison of the input representation used in Part 2 with that used by the neural net in Part 3 should be provided. (Provide an example of a sample input/output vector). Compare using graphs, the results of your robot from Part 2 (LUT with state space reduction) and your neural net based robot using less or no state space reduction. Show your results and offer an explanation.*

Within Robocode, a state-action feature vector that has not been quantized looks like:

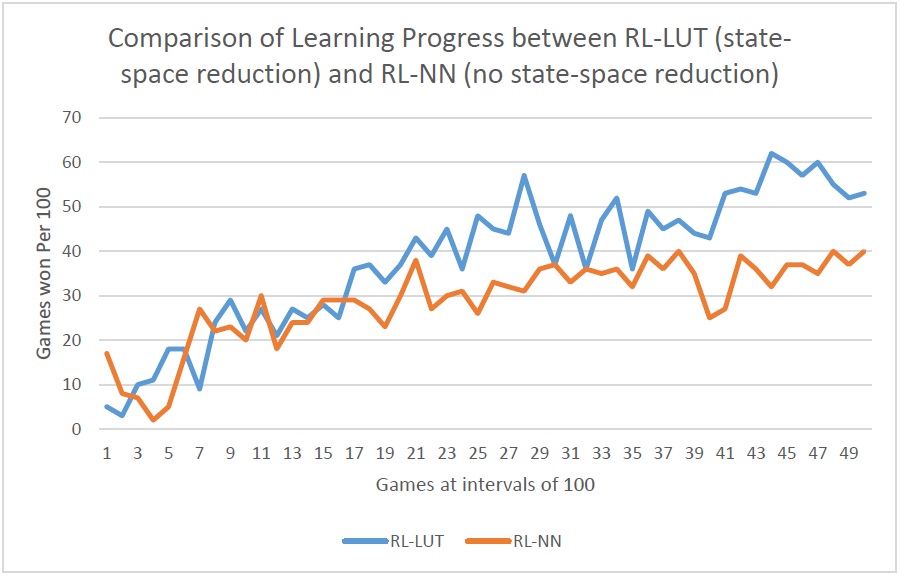
<Energy=**100.0**, Distance=**749.0**, Bearing=**-123.4**, Action=**3**>.

When training the robot using the LUT with state space reduction, the above feature vector is quantized to become:

<Energy=**10**, Distance=**7**, Bearing=-123.4/10+18 = **5**, Action=**3**>.

When training the NN during live gameplay, we now drop all quantization and apply the pre-processing described in Table 1. The same feature vector above would be an input to the NN that looks like:

<Energy=**10.0**, Distance=**7.49**, Bearing**=-1.23**, Action**=-1 +1 -1 -1 -1**>.



**Figure 6: Training performance from Milestone 2 (Blue line) versus RL with NN-Backpropagation and no state-space reduction over 5000 games.**

In Figure 6, I initialize the NN with random weights, because I want to directly compare the learning progress with the progress from Milestone 2 (RL-LUT series). Note, the RL-LUT series begins with a LUT initialized with random QValues, so having the NN initialized with random weights is better for direct comparison purposes. Notice that while initially the two learning trends are close, the overall trend from the RL-NN series shows a slower increasing curve than RL-LUT. This is to be expected for a couple of reasons:

-Since RL-NN is not using state-space reduction there are a lot more “training patterns” exposed to the NN than there was for the LUT meaning that the NN will take a longer time to be trained.

- In 1b we discovered that a learning rate of 0.02 works best for my NN architecture. This is a small step size for the backpropagation algorithm, and the smaller the step size the longer the training time for the NN.

Therefore, the slower learning trend shown by the RL-NN series makes sense. In 2b I will investigate online training for the RL-NN over a longer period of games (10,000 instead of 5000), **as well as what the learning trend looks like when I initialize the NN with weights determined from the offline training** process (using the LUT training data).

*1d) 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.*

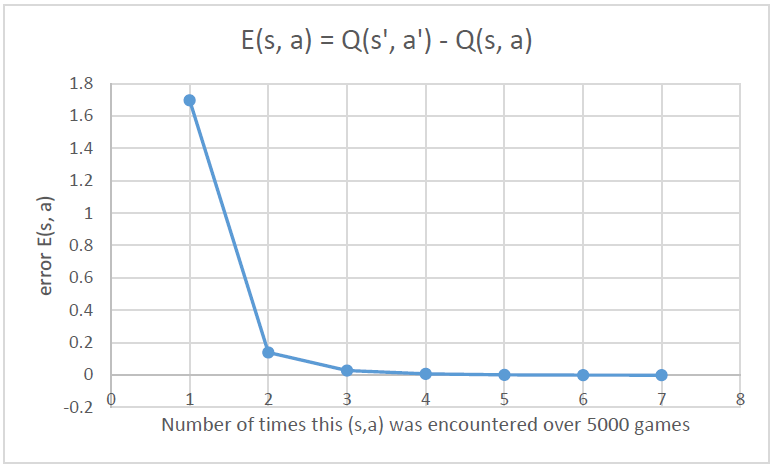
Recall our reason for using state space reduction in milestone 2. Given my current state-action feature vector of energy, distance, enemy bearing and number of actions, if I had not used discretization my LUT table would have been 100 x 1000 x 360 x 5 = 1.8 x 10^8 states. In order for TD to converge, we first would have needed to store this large table and then visit each state several times in order for the QValues to stabilize to their true values. This would not have been practical. When using a NN to approximate the Q-function, our goal is to save a set of weights that have been updated during TD learning and Backpropagation, which is much less data to save than a fully populated LUT. Training data (the state-action vectors and rewards) are sent to the NN online as it is generated in Robocode, instead of being saved in a large memory intensive table. Therefore, the NN does not require the same level of state-space reduction since we are not saving all the state-action vectors possible, but only the weights that try to approximate the value function.

*2a) What was the best win rate observed for your tank? Describe clearly how your results were obtained? Measure and plot e(s) (compute as Q(s’,a’)-Q(s,a)) for some selected state-action pairs. Your answer should provide graphs to support your results. Remember here you are measuring the performance of your robot online. I.e. during battle.*

The best observed win rate for FizzBotFinal was 95% when the NN was initialized with the weights saved from the offline training process. This will be shown in section 2b.

When training the NN online, I want to measure the performance of training. E(s,a) = Q(s’, a’) – Q(s, a) is the difference in QValues for the same state-action pair over several games. I measure this by selecting the first state-action encountered in the first game and saving this vector for all other games. Anytime this state-action vector is encountered again, I query the NN for the forward propagation of this input vector and calculate the difference between the current QValue and the last QValue seen for this state-action vector.

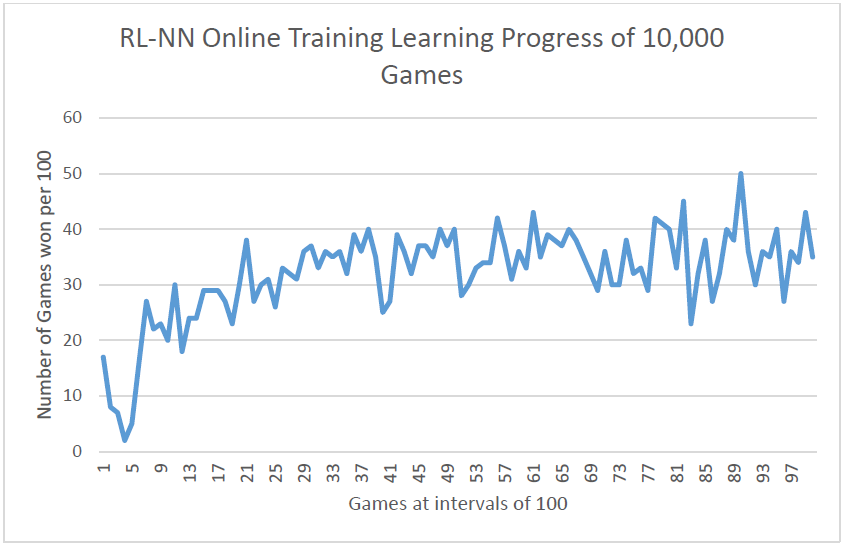
In my experiments, I did not encounter this state-action vector many times as can be seen in the graph below. This data was gathered over 5000 games.



**Figure 7: This graph shows the decreasing variation of the NN output for input (s, a) over time.**

In Figure 7, we do notice that the E(s, a) decreases over time which is a good thing, this means that the NN weights are stabilizing such that the outputs are more deterministic for the same inputs. However, since my experiments for different state action vectors over 5000 games show that the state-action was only visited a couple of times, this goes to show the magnitude of the state space now that we are no long discretizing the input.

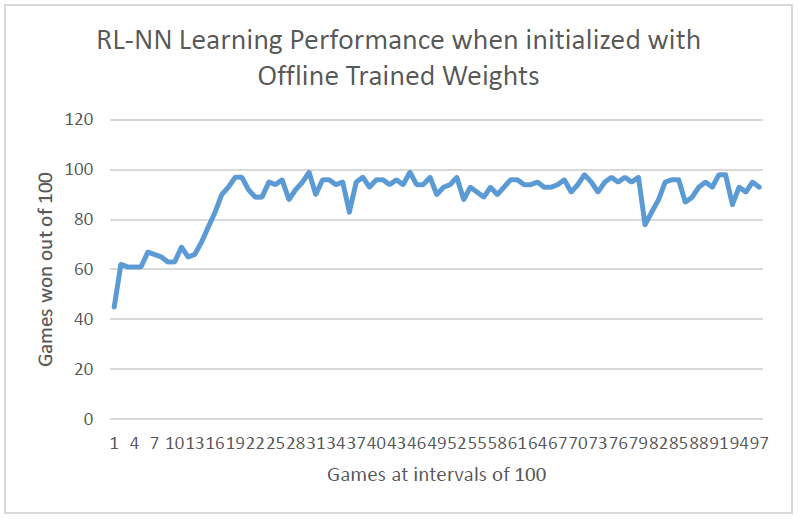
*2b) Plot the win rate against number of battles. As training proceeds, does the win rate improve asymptotically?*



**Figure 8: This is the online learning progress of the NN when initialized with random weights. Note: learning is continuing to take place during live game play because I continue to perform TD algorithm updates as we did during RL in milestone 2.**

In Figure 6, we trained the NN initialized with random weights over 5000 games and saw that the win rate was lower than the RL-LUT performance (around a 15% difference). Now with 10,000 games, we can see that the learning trend of the NN with randomly initialized weights reaches a win rate of 50%, a 10% improvement when trained with 5000 extra games. However, at 5000 games, the RL-LUT achieved a 60% win rate. It looks like while the NN is learning, it is not performing as well as the RL-LUT. It is possible that the NN-Backpropagation for this set of initial random weights is settling at a local minima which is why the win rate does not reach the same RL-LUT win rate.

Now I test how the NN will perform when I initialize it using the weights saved during offline training. I expect that since the NN starts with some knowledge, it will continue to improve in its learning when provided state-action vectors with no data discretization, and continued TD updates.



**Figure 10: This graph shows the learning trend of the NN initialized with weights saved from the offline LUT training, and then playing online for 10,000 games.**

In the above graph, when the NN is initialized with the weights saved from the offline training using the LUT data, the NN starts at a better win rate (45%) than previously (15% with random initial weights). It then continues to learn from the online non-discretized state-action pairs until it reaches a win rate of 95%. This is quite good, and it shows us that when trained offline, the NN reaches a better win rate faster because it has already started with some knowledge of the learning environment. I surmise that the NN is also beating the win rate achieved with the RL-LUT approach because we are no longer applying state-space reduction to the inputs, instead the NN is receiving more accurate input data to match with expected rewards and can therefore make smarter/more accurate decisions. When I observe FizzBotFinal with the NN brain, I notice that it does a different move that I didn’t notice with the LUT:

- The winning move with the LUT was for FizzBotFinal to continually circle the TrackFire (when it was not bounded by the walls). FizzBotFinal would circle the TrackFire faster than the TrackFire could aim and shoot and TrackFire would end up spending all of its energy on ammunition and missing. However the maximum win rate for the RL-LUT approach was only 60% during Q-learning.

- When I observe FizzBotFinal with the RL-NN approach, I notice that in addition to circling the TrackFire, it also moves in tight circles beside TrackFire. TrackFire loses in a similar fashion as before because it continually repositions its gun while FizzBotFinal just circles beside it and misses all the bullets. This was an interesting tactic I didn’t notice with the RL-LUT approach. Perhaps the non-discretized state space applied to the NN provided it with more accurate state-rewards information, allowing it to win much more against the TrackFire robot

*2c) 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 Qfunction is approximated. Explain this convergence in terms of the Bellman equation and also why when using approximation, convergence is no longer guaranteed.*

The Bellman Equation is as follows:

(1) 𝑉(𝑠𝑡) = 𝑟𝑡+ 𝛾𝑉(𝑠𝑡+1)

Where V(st) represents the value function at time t, rt is the reward for time t and 𝛾 is the discount factor for future states. Note, in this equation we can see that V(st) is dependent on V(st+1), or a successive state. This will become important when we show how the value function converges. In RL, at the start of learning we initialize all the states in the LUT to random QValues. The goal as we learn over time is to improve our value function guesses until we reach the optimal value function given by (2):

(2) 𝑉∗(𝑠𝑡)=𝑟𝑡+ 𝛾𝑉∗(𝑠𝑡+1)

Since we initialize the value function with random values, we can write our value function as follows:

(3) 𝑉(𝑠𝑡)=𝑉∗(𝑠𝑡)+𝑒(𝑠𝑡)

Where e(st) is the error separating the value function at each state from the optimal value function. Let us take a closer look at e(st):

Since the Bellman function is recursive in nature, we can write

(4) 𝑉(𝑠𝑡+1)= 𝑉∗(𝑠𝑡+1)+𝑒(𝑠𝑡+1).

Now we can substitute equations (3) and (4) into equation (1) as follows:

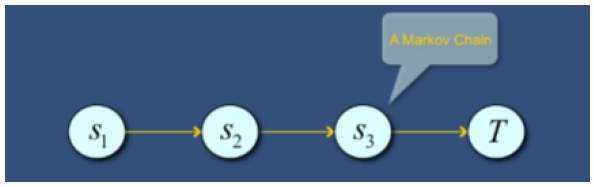
𝑉∗(𝑠𝑡)+𝑒(𝑠𝑡)=𝑟𝑡+ 𝛾(𝑉∗(𝑠𝑡+1)+𝑒(𝑠𝑡+1)) => 𝑉∗(𝑠𝑡)+𝑒(𝑠𝑡)=𝑟𝑡+ 𝛾𝑉∗(𝑠𝑡+1)+𝛾𝑒(𝑠𝑡+1)

In the above equation, term 1 on the LHS and terms 2 and 3 on the RHS are the optimal value function. If we collect the rest of the terms we can see that:

𝑒(𝑠𝑡)=𝛾𝑒(𝑠𝑡+1)

Which means that the errors of successive states are related to each other, just like how the value functions of successive states are related to each other!

If we take a look at the Markov Decision Process in Figure 11 below, we can see that a chain of states eventually end in a terminal state at time T. Terminal states have the property that we actually know the reward beforehand, and so there is no error associated with the reward at a terminal state.



**Figure 11: MDP Diagram** [3]

That is, e(sT) = 0. Coming back to TD that we are applying in Robocode, we are propagating future rewards backwards, one state at a time. Meaning that over multiple iterations of learning, the error from the terminal state (e(sT) = 0) will also be propagated back and so:

𝑒(𝑠𝑡)=𝛾𝑒(𝑠𝑡+1) -> 0

Will eventually decrease until it becomes zero. This is how the Bellman value function converges over time to stable states dictated by the optimal value function.

Now: this convergence is only proven to happen when we perform RL with the LUT based approach. As in, there are a finite number of states that will be visited numerous times until a stable value is reached for each visited state. If we are now trying to approximate the value function with a neural net, we no longer have a finite number of states (and so visiting them all is unlikely, let alone visiting them all multiple times), and there is an additional error from the value function approximation that must be accounted for.

(5) 𝑉(𝑠𝑡)=𝑟𝑡+ 𝛾𝑉(𝑠𝑡+1)+𝑒(𝑠𝑡)+𝑒(𝑓𝑢𝑛𝑐𝑡𝑖𝑜𝑛𝑎𝑝𝑝𝑟𝑜𝑥𝑖𝑚𝑎𝑡𝑖𝑜𝑛)

Here, the value function written is the same as before, except now there is an extra error term “e(function\_approximation)”. When using a NN to approximate the value function, backpropagation and gradient descent is used to update the weights such that they allow the NN to reach the target value for each input vector. However, gradient descent does not guarantee that the NN solution will reach a global minima, it is possible for the NN to settle at a local minima. This means we cannot guarantee e(function\_approximation) to become zero, and so value function approximation may not converge to the optimal value function as it did before in the RL-LUT approach.

Referenced [3] for material related to this.

*2d) 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. Hint: Readup on experience replay.*

Learning performance in online NN training can be measured in a couple of ways:

1. We can use the E(s,a) = Q(s’, a’) – Q(s, a) approach that we used in 2a. This can be generalized to any application of online RL-NN training: E(x) = Q(x’) – Q(x). The reason this is a good measure is because we can see if the NN output for input x converges (E(x) approaches zero over time) or diverges (E(x) is volatile over time).
2. We can also gather statistics from the application RL-NN is being used in to determine whether the NN is generating good value. For example in Robocode, I have chosen to gather the winning rate data. Since I can see the win rate increase over time, I can determine that my NN is in the correct solution space of gradient descent.

**Overall Conclusion**

*3a) 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?*

From what we learnt, RL works well when an agent is learning in a specific environment, and the agent will continue to operate in this environment. In Robocode, this means that if I train FizzBotFinal against the TrackFire enemy robot, then FizzBotFinal will learn based on this environment only. If I now want FizzBotFinal to battle against another robot, it must be re-trained in this new environment. In other practical applications, if an agent is trained using RL, it should be understood that the trained agent may perform well only in the environment it was trained on, not in other environments (unless it is re-trained and the rewards are adjusted to match the new environment).

Two other practical issues surrounding RL is the time needed for training, and the state space used when using a LUT. If the state space is too discretized (sparsely populated LUT), then the quality of the training data may not be realistic enough, however the training time for convergence will be shorter. If there is not enough state-space reduction, then more memory is required and training time will be longer in order for the QValues to converge. Therefore, some smart decisions in quantization need to be made when using the RL-LUT approach to ensure sufficient but manageable data. In terms of RL with a NN, I used the LUT data to train the NN offline in order to determine the best parameters for the NN (i.e. learning rate, momentum, and the number of hidden neurons). In other practical situations, it seems like going through RL with a LUT is very useful in order to do Q-function approximation later with a NN. Without the LUT data, it would have been much more difficult to design and evaluate the NN before using the NN in online training. If we wanted to have better quality offline training data, we would have used a larger LUT with more state-action enumerations, and this would have resulted in much longer training time for the NN – another practical issue to consider.

It is also important to note that Q-function approximation is not guaranteed to converge, as was already discussed in 1d and 2c. But it is also important to remember that the backpropagation algorithm suffers from potentially stopping at local minima, meaning it is also not guaranteed to find the optimal solution.

In terms of improving the performance of FizzBotFinal, there are a couple of things I could have done. If we recall my experimental set up of investigating the best parameters for the NN, I used an order dependent evaluation. This is perhaps not the best method, and I could have done better by testing all values of each parameter with all values of other parameters – an *order independent* evaluation.

Other methods could also be used such as completely different NN architectures. For example, more hidden layers, or even separate neural nets for each action. When more hidden layers are used in a NN, this is called Deep Learning, and deep learning nets have proven themselves to be good at feature extraction [4]. Perhaps a deep neural net trained with a more detailed LUT could extract patterns better than my current NN architecture. In terms of separate NN’s for each action, this might also improve the NN Qvalue function approximation as now each NN is only concerned with the outcomes for a single action, instead of needing to approximate the outcomes for all the actions which could result in higher error approximation.

Furthermore, I used a bipolar sigmoid as my activation function in my NN. The RELU function is another popular activation function for deep neural nets that has biological inspiration and could potentially do well in this setup and other practical applications [3]. Lastly, in our TD algorithm we perform backups after every step of the robot. Perhaps performing backups with more steps would provide better historical data for both RL and the NN.

If I were to give advice to someone else who is interested in applying RL with a NN for Q-function approximation, I would definitely encourage them to obtain a training set of data using RL with a LUT as we did for this project. Not only does this help with NN architecture evaluation, it also provides the engineer with confidence that their TD algorithm works as expected.

*3b) 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.*

This is a tricky scenario for RL since the environment includes live patients. Recall that the RL-TD algorithm learns by exploration and reward evaluation based on all moves taken in a current state. It would be completely unethical to train an RL agent on live patients. In this case, using RL would only be possible if the agent could be trained in a simulated environment that contains the reactions of multiple patients of different body builds, ethnicities and perhaps even allergies. For this to work, patient data must first be accrued manually (with their consent). A simulator must then be developed to train the RL agent, and if it does well based on the collected data, only then should it be used on live patients.

**References**

|  |  |
| --- | --- |
| [1] | S. Sarkaria, "Backpropagation: EECE592 Course Notes," September 2016. [Online]. |
| [2] | S. Sarkaria, "Introduction to Neural Nets: EECE 592 Course Notes," September 2016. [Online]. |
| [3] | S. Sarkaria, "Reinforcement Learning Part 2: EECE592 Course Notes," October 2016. [Online]. |
| [4] | S. Sarkaria, "Deep Learning: EECE592 Course Notes," [Online]. |
| [5] | S. Sarkaria, "Reinforcement Learning Part 1, EECE592 Course Notes," October 2016. [Online]. |
| [6] | L. Fausett, Fundamentals of Neural Networks: Architectures, Algorithms and Applications, Pearson, 1992. |

**APPENDIX**

Code for FizzBotFinal incorporated with Neural Net

**package** my.rl.robot;

**import** robocode.AdvancedRobot;

**import** java.util.Arrays;

**import** java.util.Random;

**import** my.rl.robot.NeuralNet;

**import** robocode.\*;

**public** **class** FizzBotFinal **extends** AdvancedRobot {

**static** String *laptop\_path* = "C:\\Users\\Faizan Mansuri\\My Data\\";

**static** String *online\_weights\_file* = *laptop\_path*+"online\_trained\_weights.txt";

**static** String *offline\_weights\_file* = *laptop\_path*+"trained\_weights\_15\_0.02\_0.3.txt";

// learning versus trained

**private** **static** **boolean** *Q\_LEARNING* = **true**;

**private** **static** **double** *EPSILON\_INIT* = 0.0;

**private** **static** **double** *EPSILON* = *EPSILON\_INIT*;

**private** **static** **boolean** *EXPLORATION* = **false**;

**private** **static** **boolean** *ON\_POLICY* = **false**;

**private** **static** **int** *MAX\_GAMES* = 50000;

// actions

**enum** Action {***AIMANDFIRE***, ***STRAFE***, ***CIRCLE***, ***DIAGRIGHT***, ***DIAGLEFT***}

// rewards // rewards

**private** **static** **final** **double** ***WALL\_COLLISION\_REWARD*** = -0.9;

**private** **static** **final** **double** ***AWAY\_FROM\_WALL*** = 0.1;

**private** **static** **final** **double** ***ROBOT\_COLLISION\_REWARD*** = -0.8;

**private** **static** **final** **double** ***HIT\_TARGET\_REWARD*** = 0.8;

**private** **static** **final** **double** ***I\_AM\_HIT\_REWARD*** = -1.2;

**private** **static** **final** **double** ***WIN\_GAME\_REWARD*** = 1;

**private** **static** **final** **double** ***LOSE\_GAME\_REWARD*** = -1;

**private** **static** **final** **double** ***LOW\_DIST\_TO\_ENEMY\_REWARD*** = -0.6;

**private** **static** **final** **double** ***MID\_DIST\_TO\_ENEMY\_REWARD*** = 0.5;

// end rewards

**private** **static** **double** *ALPHA* = 0.2;

**private** **static** **double** *GAMMA* = 0.5;

// stats to save on the enemy when I get to scan him

**double** enemyDistance = 0.0;

**double** enemyEnergy = 0.0;

**double** enemyHeading = 0.0;

**double** enemyBearing = 0.0;

// for auto aim

**double** enemyBearingRadians = 0.0;

**double** enemyVelocity = 0.0;

**double** enemyHeadingRadians = 0.0;

**long** lastScanTime = 0;

Action act;

**int** actionToTake;

//set corresponding index in feature vector

**public** **static** **final** **int** ***MY\_ENERGY*** = 3;

**public** **static** **final** **int** ***DIST*** = 2;

**public** **static** **final** **int** ***ENEMY\_BEARING*** = 1;

**public** **static** **final** **int** ***ACTION*** = 0;

**public** **static** **boolean** *normalized* = **true**;

**public** **static** **boolean** *isOfflineTraining* = **true**;

**public** **static** **int** *numActions* = 5;

**public** **static** **int** *featureVectorSize* = 3 + *numActions*;

**public** **static** **int** [] *featureNumValues* = {5, 37, 11, 11};

// initialized backwards!

//args to NN: int numInputs, int numHiddenNeurons, double learningRate, double momentum, boolean isOfflineTraining, weights\_file

//the NN that showed learning: public static NeuralNet nn = new NeuralNet(featureVectorSize, 15, 0.00001, 0.0, true, false, offline\_weights\_file);

**public** **static** NeuralNet *nn* = **new** NeuralNet(*featureVectorSize*, 15, 0.02, 0.3, *isOfflineTraining*, *normalized*, *offline\_weights\_file*);

**int** direction = 1;

**private** **double** [] prevState = **new** **double**[*featureVectorSize* - *numActions*];

**private** **double** [] currState = **new** **double**[*featureVectorSize* - *numActions*];

**private** **double** [] prevStateAction = **new** **double**[*featureVectorSize*];

**private** **static** **double** [] *sa* = **new** **double**[*featureVectorSize*];

**private** **static** **boolean** *qsaSaved* = **false**;

**private** **double** qsa = -1;

**private** **double** qPrimeSa = -1;

**private** **double** [] currStateAction = **new** **double**[*featureVectorSize*];

**private** **double** [] currStateActionGreedy = **new** **double**[*featureVectorSize*];

**public** **double** rewardPerTurn;

// learning stats

**public** **static** **double** *rewardPerHundred* = 0.0;

**public** **static** **int** *numTotalGames* = 0;

**public** **static** **int** *numWinGamesPerHundred* = 0;

**public** **static** **int** *wallMargin* = 60;

**public** **void** run()

{

setAdjustRadarForRobotTurn(**true**); // keep the radar still while we turn

setAdjustGunForRobotTurn(**true**); // keep the gun still while we turn

turnRadarRight(180);

prevState = getState();

/\*\*\* Q-Learning with NN \*\*\*/

**if** (*Q\_LEARNING*)

{

**while** (**true**)

{

turnRadarRight(90);

// choose a from s using policy

System.***out***.println("Choose Action");

prevStateAction = chooseAction(prevState);

**if** (!*qsaSaved*)

{

System.*arraycopy*(prevStateAction, 0, *sa*, 0, prevStateAction.length);

*qsaSaved* = **true**;

}

**if** (Arrays.*equals*(prevStateAction, *sa*))

{

computeErrorSA();

}

// take action a

performAction(actionToTake);

// observe some rewards r

**if** (enemyDistance <= 250)

{

rewardPerTurn += ***LOW\_DIST\_TO\_ENEMY\_REWARD***;

}

**else** **if** (enemyDistance > 250 && enemyDistance <= 333)

{

rewardPerTurn += ***MID\_DIST\_TO\_ENEMY\_REWARD***;

}

**double** x, y;

x = getX();

y = getY();

**if** ( (x <= *wallMargin*) || (x >= (getBattleFieldWidth() - *wallMargin*)) || (y <= *wallMargin*) || (y >= (getBattleFieldHeight() - *wallMargin*)) )

{

rewardPerTurn += ***WALL\_COLLISION\_REWARD***;

}

**else**

{

rewardPerTurn += ***AWAY\_FROM\_WALL***;

}

// observe s'

currState = getState();

currStateAction = chooseAction(currState);

updateNN(prevStateAction, currStateAction);

System.***out***.println("rewardperTurn: "+rewardPerTurn);

*rewardPerHundred* +=rewardPerTurn;

rewardPerTurn = 0; // reset reward for next state transition

// prevState = currState

System.*arraycopy*(currState, 0, prevState, 0, prevState.length);

}

}

**else**

{

// playing from trained table

prevState = getState();

**while**(**true**)

{

**if** (getTime() - lastScanTime > 5)

{

turnRadarRight(90);

}

prevStateAction = chooseAction(prevState);

// take action a

performAction(actionToTake);

rewardPerTurn = 0;

prevState = getState();

}

}

}

**public** **void** computeErrorSA()

{

qPrimeSa = *nn*.outputFor(*sa*);

**double** errorSa = qPrimeSa - qsa;

*nn*.saveErrorSA(errorSa, *numTotalGames*);

qsa = qPrimeSa;

}

// all mapping for feature vector state occurs here (i.e. int indexes)

**public** **double** [] getState()

{

/\* In order from MSB to LSB \*/

// energy: 0.0, 0.1...100.0

// distance to enemy: 0.0, 0.1...1000.0

// bearing: -1.80, -1.79, ... 1.80

// action: one hot encoded

**double** [] state = **new** **double**[*featureVectorSize* - *numActions*];

// energy (low, medium high)

**double** energy = getEnergy();

**if** (energy > 100)

energy = 100.0;

state[***MY\_ENERGY*** - 1] = energy/10.0;

// distance to enemy (low, med, far)

**double** distance = enemyDistance;

state[***DIST*** - 1] = distance/100.0;

// enemy bearing

state[***ENEMY\_BEARING*** - 1] = enemyBearing/100.0;

System.***out***.println("Energy: "+energy+" Dist: "+distance+" Bearing: "+enemyBearing);

System.***out***.println("Quantized Energy: "+state[***MY\_ENERGY*** - 1]+" Dist: "+state[***DIST*** - 1]+" Bearing: "+state[***ENEMY\_BEARING*** - 1]);

**return** state;

}

**public** **double**[] getCategoricalActionRepresentation(**int** index)

{

**double** [] representation = **null**;

**if** (index == 0) representation = **new** **double**[]{-1.0, -1.0, -1.0, -1.0, 1.0};

**else** **if**(index == 1) representation = **new** **double**[]{-1.0, -1.0, -1.0, 1.0, -1.0};

**else** **if** (index == 2) representation = **new** **double**[]{-1.0, -1.0, 1.0, -1.0, -1.0};

**else** **if** (index == 3) representation = **new** **double**[]{-1.0, 1.0, -1.0, -1.0, -1.0};

**else** **if** (index == 4) representation = **new** **double**[]{1.0, -1.0, -1.0, -1.0, -1.0};

**return** representation;

}

/\*\*

\* \* returns state-action in expected NN representation \*/

**public** **double** [] chooseAction(**double**[] prevState2)

{

**double** qValue = 0.0;

**double** maxQValue = Double.***NEGATIVE\_INFINITY***;

**int** actionToTake = 0;

Random number = **new** Random();

**double** rand = number.nextDouble();

**double** [] fullFeatureVector = **null**; // 3 + actions

**double** [] qValueSet = outputQValuesForCurrentState(prevState2);

**assert**(qValueSet.length == *featureNumValues*[0]);

/\* determine random or greedily chosen action \*/

**if** (rand < *EPSILON* && *EXPLORATION*)

{

// choose random

actionToTake = (**int**) (0 + (qValueSet.length - 0) \* number.nextDouble());

**this**.actionToTake = actionToTake;

**if** (!*ON\_POLICY*)

{

// choose greedy for off policy update later

**for** (**int** i = 0; i < qValueSet.length; i++)

{

qValue = qValueSet[i];

**if** (qValue > maxQValue)

{

maxQValue = qValue;

actionToTake = i;

}

}

**this**.actionToTake = actionToTake;

fullFeatureVector = concat(reverseArray(prevState2), getCategoricalActionRepresentation(**this**.actionToTake));

System.*arraycopy*(fullFeatureVector, 0, currStateActionGreedy, 0, fullFeatureVector.length);

}

}

**else**

{

// choose greedy

**for** (**int** i = 0; i < qValueSet.length; i++)

{

qValue = qValueSet[i];

**if** (qValue > maxQValue)

{

maxQValue = qValue;

actionToTake = i;

}

}

**this**.actionToTake = actionToTake;

fullFeatureVector = concat(reverseArray(prevState2), getCategoricalActionRepresentation(**this**.actionToTake));

System.*arraycopy*(fullFeatureVector, 0, currStateActionGreedy, 0, fullFeatureVector.length);

}

**return** fullFeatureVector;

}

**public** **double**[] outputQValuesForCurrentState(**double** [] state)

{

**double** [] qValues = **new** **double**[*numActions*];

**double** [] revState = reverseArray(state);

**double** [] stateAction = **null**;

**for** (**int** i = 0; i<*numActions*; i++)

{

stateAction = concat(revState, getCategoricalActionRepresentation(i));

qValues[i] = *nn*.outputFor(stateAction);

}

**return** qValues;

}

**public** **double**[] reverseArray(**double** [] arr)

{

**double**[] toReverse = **new** **double**[arr.length];

System.*arraycopy*(arr, 0, toReverse, 0, arr.length);

**for**(**int** i = 0; i < toReverse.length / 2; i++)

{

**double** temp = toReverse[i];

toReverse[i] = toReverse[toReverse.length - i - 1];

toReverse[toReverse.length - i - 1] = temp;

}

**return** toReverse;

}

**public** **double**[] concat(**double**[] a, **double**[] b)

{

**int** aLen = a.length;

**int** bLen = b.length;

**double**[] c= **new** **double**[aLen+bLen];

System.*arraycopy*(a, 0, c, 0, aLen);

System.*arraycopy*(b, 0, c, aLen, bLen);

**return** c;

}

**public** **void** onScannedRobot(ScannedRobotEvent e)

{

System.***out***.println("Scanned enemy robot!");

lastScanTime = getTime();

// update state info on enemy

enemyDistance = e.getDistance();

enemyEnergy = e.getEnergy();

enemyHeading = e.getHeading();

enemyBearing = e.getBearing();

enemyBearingRadians = e.getBearingRadians();

enemyVelocity = e.getVelocity();

enemyHeadingRadians = e.getHeadingRadians();

// auto aim

**double** absBearing=e.getBearingRadians()+getHeadingRadians(); //enemies absolute bearing

**double** latVel=e.getVelocity() \* Math.*sin*(e.getHeadingRadians() - absBearing); //enemies later velocity

setTurnRadarLeftRadians(getRadarTurnRemainingRadians());//lock on the radar

// turn gun to face enemy

**double** gunTurnAmt = robocode.util.Utils.*normalRelativeAngle*(absBearing- getGunHeadingRadians()+latVel/22);

//amount to turn our gun, lead just a little bit

setTurnGunRightRadians(gunTurnAmt); //turn our gun

execute();

}

**public** **void** updateNN(**double**[] prevStateAction, **double**[] currStateAction)

{

**double** prevQValue = *nn*.outputFor(prevStateAction);

**double** currQValueTaken = *nn*.outputFor(currStateAction);

**if** (*ON\_POLICY*)

{

// make update based on the action that was performed

*nn*.trainModified(prevStateAction, ((1-*ALPHA*)\*prevQValue + *ALPHA*\*(rewardPerTurn + *GAMMA*\*currQValueTaken)));

}

**else**

{

// make update based on the greedy action regardless of whether you took it

**double** currQValueGreedy = *nn*.outputFor(currStateActionGreedy);

*nn*.trainModified(prevStateAction, ((1-*ALPHA*)\*prevQValue + *ALPHA*\*(rewardPerTurn + *GAMMA*\*currQValueGreedy)));

}

}

**public** **double** normalizeQValue(**double** input)

{

**double** result;

**double** max = 1.3552442660534507;

**double** min = -17.86177741523035;

**double** bipolar\_min = -1.0;

**double** bipolar\_max = 1.0;

result = (input-min)/(max-min);

result = result\*(bipolar\_max - bipolar\_min) + bipolar\_min;

**return** result;

}

**public** **void** performAction(**int** action)

{

act = Action.*values*()[action];

**switch** (act)

{

**case** ***AIMANDFIRE***:

System.***out***.println("Firing");

//Automatically aim and fire on scanning an enemy robot

**double** absBearing = enemyBearingRadians+getHeadingRadians();//enemies absolute bearing

**double** latVel = enemyVelocity \* Math.*sin*(enemyHeadingRadians - absBearing);//enemies later velocity

setTurnRadarLeftRadians(getRadarTurnRemainingRadians());//lock on the radar

// turn gun to face enemy

**double** gunTurnAmt = robocode.util.Utils.*normalRelativeAngle*(absBearing- getGunHeadingRadians()+latVel/22);

//amount to turn our gun, lead just a little bit

setTurnGunRightRadians(gunTurnAmt); //turn our gun

setFire(Math.*min*(400 / enemyDistance, 3));

// if e farther away, use less fire power. Increase fire power if e is closer.

execute();

**break**;

**case** ***STRAFE***:

System.***out***.println("Strafe");

setTurnRight(enemyBearing + 90);

setAhead(direction \* 100);

execute();

direction \*= -1;

setAhead(direction \* 100);

execute();

**break**;

**case** ***CIRCLE***:

System.***out***.println("Circle");

setTurnRight(enemyBearing + 90);

setAhead(direction \* 100);

execute();

**break**;

**case** ***DIAGRIGHT***:

System.***out***.println("DiagRight");

setTurnRight(45);

setAhead(direction \* 100);

execute();

**break**;

**case** ***DIAGLEFT***:

System.***out***.println("DiagLeft");

setTurnLeft(45);

setAhead(direction \* 100);

execute();

**break**;

**default**:

**break**;

}

**while**(getDistanceRemaining() != 0 || getTurnRemaining() != 0)

{

execute();

}

}

**public** **void** onHitWall(HitWallEvent e)

{

System.***out***.println("Robot hit a wall");

rewardPerTurn += ***WALL\_COLLISION\_REWARD***; // change direction auto on wall hit

direction \*= -1;

setAhead(100 \* direction);

execute();

}

/\*\*

\* \* Occurs when my robot collides with another robot \*/

**public** **void** onHitRobot(HitRobotEvent e)

{

System.***out***.println("Robot hit another robot");

rewardPerTurn += ***ROBOT\_COLLISION\_REWARD***;

}

/\*\*

\* \* One of my bullets hit the enemy robot \*/

**public** **void** onBulletHit(BulletHitEvent e)

{

System.***out***.println("Robot shot the enemy");

rewardPerTurn += ***HIT\_TARGET\_REWARD***;

}

/\*\*

\* \* I am hit by a bullet \*/

**public** **void** onHitByBullet(HitByBulletEvent e)

{

System.***out***.println("I got shot");

rewardPerTurn += ***I\_AM\_HIT\_REWARD***;

}

**public** **void** onWin(WinEvent e)

{

System.***out***.println("I won");

rewardPerTurn += ***WIN\_GAME\_REWARD***;

updateNN(prevStateAction, currStateAction);

*numTotalGames*++;

System.***out***.println("Num total games "+*numTotalGames*);

*numWinGamesPerHundred*++;

writeToFile();

}

**public** **void** onDeath(DeathEvent e)

{

System.***out***.println("I died");

rewardPerTurn += ***LOSE\_GAME\_REWARD***;

updateNN(prevStateAction, currStateAction);

*numTotalGames*++;

System.***out***.println("Num total games "+*numTotalGames*);

writeToFile();

}

**public** **void** writeToFile()

{

**if** ((*numTotalGames* % 100) == 0)

{ // output to file numWinGamesPerHundred

*nn*.saveGameStats(*numWinGamesPerHundred*, *rewardPerHundred*);

*numWinGamesPerHundred* = 0;

*rewardPerHundred* = 0;

}

**if** (getRoundNum() + 1 == *MAX\_GAMES*)

{

*nn*.save(*online\_weights\_file*);

}

}

/\*\*

\* \* normalizes a bearing to between +180 and -180 \*

\* **@param** angle \* **@return**

\*/

**double** normalizeBearing(**double** angle)

{

**while** (angle > 180) angle -= 360;

**while** (angle < -180) angle += 360;

**return** angle;

}

}

Code for Neural Net

**package** my.rl.robot;

**import** java.io.BufferedReader;

**import** java.io.File;

**import** java.io.FileReader;

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** java.io.Writer;

**import** java.nio.file.Files;

**import** java.nio.file.Paths;

**import** java.util.List;

**public** **class** NeuralNet **implements** NeuralNetInterface

{

**public** **int** numInputs;

**public** **double**[] weights;

**public** **static** **final** **int** ***INPUTSIZE*** = 8;

**public** **double** errorPerPattern;

**protected** **int** numHiddenNeurons;

**protected** Neuron[] hiddenNeurons;

**protected** Neuron outputNeuron;

**protected** **double** learningRate;

**protected** **double** momentum;

**public** **static** Writer *trained\_weights\_wr*;

**public** **static** Writer *game\_stats\_wr*;

**public** **static** Writer *error\_stats\_wr*;

**static** String *laptop\_path* = "C:\\Users\\Faizan Mansuri\\My Data\\";

**public** NeuralNet (**int** numInputs, **int** numHiddenNeurons, **double** learningRate, **double** momentum, **boolean** isOfflineTraining, **boolean** normalizedSet, String weights\_file)

{

**this**.numInputs = numInputs;

**this**.numHiddenNeurons = numHiddenNeurons;

**this**.learningRate = learningRate;

**this**.errorPerPattern = 0;

hiddenNeurons = **new** Neuron[**this**.numHiddenNeurons];

**for** (**int** i = 0; i < **this**.numHiddenNeurons; i++)

{

hiddenNeurons[i] = **new** Neuron(**this**.numInputs, learningRate, momentum, normalizedSet);

}

outputNeuron = **new** Neuron(**this**.numHiddenNeurons, learningRate, momentum, normalizedSet);

**if** (isOfflineTraining)

{

initializeWeights();

}

**else**

{

// online training

**try**

{

load(weights\_file);

}

**catch** (IOException e)

{

e.printStackTrace();

}

}

}

@Override

/\*\*

\* \* Computes the output of the NN without training (forward pass).

\* \* **@param** X - the input vector, an array of doubles.

\* \* **@return** Y - Sigmoid threshold output value.

\* \* **@throws** IllegalArgumentException if the input vector length is incorrect \*/

**public** **double** outputFor(**double**[] X) **throws** IllegalArgumentException

{

**if** (X.length != ***INPUTSIZE***)

{

**throw** **new** IllegalArgumentException("input vector length is " + X.length + "; expected " + ***INPUTSIZE***);

}

/\* Perform forward propagation pass - compute weighted Sum Si and activation Ui for all cells \*/

**double**[] hiddenRes = **new** **double**[**this**.numHiddenNeurons];

**double** result;

**double** sum;

**for** (**int** i = 0; i < **this**.numHiddenNeurons; i++)

{

sum = hiddenNeurons[i].computeWeightedSum(X);

hiddenRes[i] = hiddenNeurons[i].applyActivation(sum);

}

**if** (hiddenRes[0] == 0.0 && hiddenRes[1] == 0.0 && hiddenRes[2] == 0.0 && hiddenRes[3] == 0.0)

{

System.*exit*(1);

}

sum = outputNeuron.computeWeightedSum(hiddenRes);

result = outputNeuron.applyActivation(sum);

**return** result;

}

**public** **double** trainModified(**double**[] X, **double** target)

{

// 1. Perform forward propagation step and calculate error for the pattern

**double** nnActualOuput = outputFor(X);

// 2. Perform backward propagation of error signals and compute weight udpates right away

**double** [] hiddenErrSignals = **new** **double**[**this**.numHiddenNeurons];

**double** outputErrSignal = **this**.outputNeuron.applyGradientDescent()\*(target - nnActualOuput);

**double** [] activatedInputsToOutput = **new** **double**[**this**.numHiddenNeurons];

**for** (**int** i = 0; i < **this**.numHiddenNeurons; i++)

{

activatedInputsToOutput[i] = **this**.hiddenNeurons[i].lastActivatedValue;

}

**this**.outputNeuron.updateWeights(outputErrSignal, activatedInputsToOutput);

**for** (**int** i = 0; i < **this**.numHiddenNeurons; i++)

{

hiddenErrSignals[i] = hiddenNeurons[i].applyGradientDescent()\*outputErrSignal\***this**.outputNeuron.weights[i];

hiddenNeurons[i].updateWeights(hiddenErrSignals[i], X);

}

/\* Calculate total error for this patten and return \*/

**this**.errorPerPattern = errorPerPattern(nnActualOuput, target);

**return** **this**.errorPerPattern;

}

@Override

/\*\*

\* \* This method is used to update the weights of the neural net

\* \* **@param** X - input vector

\* \* **@param** target - the target value for this input vector

\* \* **@return** double - the error used to train (error before the update) \*/

**public** **double** train(**double**[] X, **double** target)

{

// 1. Perform forward propagation step and calculate error for the pattern

**double** nnActualOuput = outputFor(X);

// 2. Perform backward propagation of error signals

**double** [] hiddenErrSignals = **new** **double**[**this**.numHiddenNeurons];

**double** outputErrSignal = **this**.outputNeuron.applyGradientDescent()\*(target - nnActualOuput);

**for** (**int** i = 0; i < **this**.numHiddenNeurons; i++)

{

hiddenErrSignals[i] = hiddenNeurons[i].applyGradientDescent()\*outputErrSignal\***this**.outputNeuron.weights[i];

}

// 3. Update the weights for all neurons

**double** [] activatedInputsToOutput = **new** **double**[**this**.numHiddenNeurons];

**for** (**int** i = 0; i < **this**.numHiddenNeurons; i++)

{

activatedInputsToOutput[i] = **this**.hiddenNeurons[i].lastActivatedValue;

}

**this**.outputNeuron.updateWeights(outputErrSignal, activatedInputsToOutput);

**for** (**int** i = 0; i < **this**.numHiddenNeurons; i++)

{

hiddenNeurons[i].updateWeights(hiddenErrSignals[i], X);

}

/\* Calculate total error for this patten and return \*/

**this**.errorPerPattern = errorPerPattern(nnActualOuput, target);

**return** **this**.errorPerPattern;

}

**public** **void** save(String path)

{

**double** [] weights;

**try** {

path.replaceAll("\\\\", "/");

System.***out***.println(path);

*trained\_weights\_wr* = **new** FileWriter(path);

System.***out***.println("File created");

// print hidden neuron weights

**for** (**int** i = 0; i<**this**.numHiddenNeurons; i++)

{

weights = **this**.hiddenNeurons[i].getWeight();

**for** (**int** j = 0; j < weights.length; j++)

{

*trained\_weights\_wr*.append(String.*valueOf*(weights[j]+" "));

}

*trained\_weights\_wr*.append(System.*getProperty*("line.separator"));

}

// print output neuron weights

weights = **this**.outputNeuron.getWeight();

**for** (**int** j = 0; j < weights.length; j++)

{

*trained\_weights\_wr*.append(String.*valueOf*(weights[j]+" "));

}

*trained\_weights\_wr*.close();

}

**catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

@Override

**public** **void** load(String path) **throws** IOException

{

System.***out***.print("Neural Net loading weights from file "+path);

BufferedReader br = **new** BufferedReader(**new** FileReader(path));

**int** count = 0;

String [] tokens;

**double** [] weights = **null**;

List<String> lines = Files.*readAllLines*(Paths.*get*(path));

**for** (String line : lines)

{

tokens = line.split(" ");

weights = **new** **double**[tokens.length];

**for** (**int** i=0; i<tokens.length; i++)

{

weights[i] = Double.*valueOf*(tokens[i]);

}

**if** (count != **this**.numHiddenNeurons)

{

**this**.hiddenNeurons[count].setWeight(weights);

count ++;

}

**else**

{

**this**.outputNeuron.setWeight(weights);

}

}

br.close();

checkWeights();

}

**public** **void** checkWeights()

{

**for** (**int** i = 0; i<**this**.numHiddenNeurons; i++)

{

**this**.hiddenNeurons[i].printWeights();

}

**this**.outputNeuron.printWeights();

}

/\*\*

\* \* **@return** f(x) = 2 / (1+e(-x)) - 1 \*/

**public** **double** bipolarSigmoid(**double** x)

{

**return** ((2.0 / (1.0 + Math.*exp*(-x))) - 1.0);

}

**public** **double** binarySigmoid(**double** x)

{

**return** (1.0 / (1.0 + Math.*exp*(-x)));

}

**public** **double** errorPerPattern(**double** actual, **double** target)

{

**double** diff = actual - target;

**return** Math.*pow*(diff, 2) \* 0.5;

}

@Override

/\*\*

\* \* Initialize the weights to random values.

\* \* For say 2 inputs, the input vector is [0] & [1]. We add [2] for the bias.

\* \* Like wise for hidden units. For say 2 hidden units which are stored in an array.

\* \* [0] & [1] are the hidden & [2] the bias.

\* \* We also initialize the last weight change arrays. This is to implement the alpha term. \*/

**public** **void** initializeWeights()

{

**for** (**int** i = 0; i < **this**.hiddenNeurons.length; i++)

{

hiddenNeurons[i].initializeWeights();

}

outputNeuron.initializeWeights();

}

/\*\* \*

\* \*/

**public** **void** saveGameStats(**int** numWinGamesPerHundred, **double** rewardsPerHundred)

{

**try**

{

String path = *laptop\_path*+"robocodeNN\_game\_stats.txt";

path.replaceAll("\\\\", "/");

System.***out***.println(path);

*game\_stats\_wr* = **new** FileWriter(path, **true**);

System.***out***.println("File created");

*game\_stats\_wr*.append(String.*valueOf*(numWinGamesPerHundred+" "+rewardsPerHundred) + System.*getProperty*("line.separator"));

*game\_stats\_wr*.close();

}

**catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**public** **void** saveErrorSA(**double** error, **int** gameNumber)

{

**try** {

String path = *laptop\_path*+"robocodeNN\_errorSA\_stats.txt";

path.replaceAll("\\\\", "/");

System.***out***.println(path);

*error\_stats\_wr* = **new** FileWriter(path, **true**);

System.***out***.println("File created");

*error\_stats\_wr*.append(String.*valueOf*(error) +" "+String.*valueOf*(gameNumber)+ System.*getProperty*("line.separator"));

*error\_stats\_wr*.close();

}

**catch** (IOException e)

{

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

@Override

**public** **void** save(File argFile) {

// **TODO** Auto-generated method stub

}

@Override

**public** **double** sigmoid(**double** x) {

// **TODO** Auto-generated method stub

**return** 0;

}

@Override

**public** **double** customSigmoid(**double** x) {

// **TODO** Auto-generated method stub

**return** 0;

}

@Override

**public** **void** zeroWeights() {

// **TODO** Auto-generated method stub

}

}

Code for Neuron [Each Block of Neural Net]

**package** my.rl.robot;

**import** java.util.Random;

**public** **class** Neuron

{

**double** [] weights; /\* neurons only concerned with input weights \*/

**double** [] previousWeights;

**int** numWeights;

**boolean** isBinary; /\* if false, use bipolar sigmoid \*/

**double** lastActivatedValue = 0.0;

**double** lastWeightedSum = 0.0;

**double** learningRate;

**double** momentum;

**boolean** normalized;

**final** **double** bias = 1.0;

**public** Neuron (**int** numWeights, **double** learningRate, **double** momentum, **boolean** normalized)

{

**this**.numWeights = numWeights + 1; // each neuron has a bias input weight

**this**.weights = **new** **double**[**this**.numWeights];

**this**.previousWeights = **new** **double**[**this**.numWeights];

**this**.learningRate = learningRate;

**this**.momentum = momentum;

**this**.normalized = normalized;

}

**public** **void** setWeight(**double**[] weights)

{

**assert**(weights.length == **this**.numWeights);

System.*arraycopy*(weights, 0, **this**.weights, 0, **this**.numWeights);

zeroWeights();

}

**public** **double**[] getWeight()

{

**return** **this**.weights;

}

/\*\*

\* \* Initializes weights to a random value between [-0.5, 0.5] \*/

**public** **void** initializeWeights()

{

Random number = **new** Random();

**double** min = -0.5;

**double** max = 0.5;

**for** (**int** i = 0; i < **this**.numWeights; i++)

{

**this**.weights[i] = min + (max - min) \* number.nextDouble();

**assert**(**this**.weights[i] <= 0.5 && **this**.weights[i] >= -0.5);

}

zeroWeights(); /\* Init prev weights to zero \*/

}

**public** **void** zeroWeights()

{

**for** (**int** i = 0; i < **this**.numWeights; i++)

{

**this**.previousWeights[i] = 0.0;

}

}

/\*\*

\* \* **@param** double[] X - input vector to neuron

\* \* **@return** double S = W\*X for the one neuron \*/

**public** **double** computeWeightedSum(**double**[] inputs)

{

**double** result = 0.0;

**assert**(inputs.length == **this**.numWeights - 1); // minus 1 for bias

**for** (**int** i = 0; i < inputs.length; i++)

{

result += **this**.weights[i] \* inputs[i];

}

result += **this**.weights[**this**.numWeights - 1] \* bias;

**this**.lastWeightedSum = result;

**return** result;

}

**public** **void** printWeights()

{

System.***out***.println("printing weights!");

**for** (**int** i=0; i < **this**.weights.length; i++)

{

System.***out***.print(**this**.weights[i] + " ");

**assert**(!Double.*isNaN*(**this**.weights[i]));

}

}

/\*\*

\* \* **@param** errorSignal - for the neuron

\* \* **@param** inputs[] - the inputs to the neuron from the layer below \*/

**public** **void** updateWeights(**double** errorSignal, **double**[] inputs)

{

**double** [] tempWeights = **new** **double**[**this**.numWeights];

System.*arraycopy*(**this**.weights, 0, tempWeights, 0, **this**.numWeights);

**for** (**int** i = 0; i < **this**.numWeights - 1; i++)

{

**this**.weights[i] = **this**.weights[i] + **this**.learningRate \* errorSignal \* inputs[i] + **this**.momentum\*(**this**.weights[i] - **this**.previousWeights[i]);

}

**this**.weights[**this**.numWeights - 1] = (**this**.weights[**this**.numWeights - 1] + **this**.learningRate \* errorSignal \* bias + **this**.momentum\*(**this**.weights[**this**.numWeights - 1] - **this**.previousWeights[**this**.numWeights - 1]));

**this**.previousWeights = tempWeights;

}

**public** **double** applyActivation(**double** weightedSum)

{

**if** (**this**.normalized)

{

**this**.lastActivatedValue = bipolarSigmoid(weightedSum);

}

**else**

{

**this**.lastActivatedValue = customSigmoid(weightedSum);

}

**return** **this**.lastActivatedValue;

}

**public** **double** applyGradientDescent()

{

**double** result = 0.0;

**if** (normalized)

{

result = bipolarSigmoidPrime(**this**.lastActivatedValue);

}

**else**

{

result = customSigmoidPrime(**this**.lastWeightedSum);

}

**return** result;

}

/\*\* \* Return a binary sigmoid

\* \* **@param** x The input weighted sum of neuron

\* \* **@return** f(x) = 1 / (1 + e(-x)) \*/

**public** **double** binarySigmoid(**double** x)

{

**return** (1.0 / (1.0 + Math.*exp*(-x)));

}

/\*\*

\* \* Return a derivative of binary sigmoid

\* \* **@param** x - Ui (output activated value of cell)

\* \* **@return** f'(Si) = Ui(1-Ui) \*/

**public** **double** binarySigmoidPrime(**double** x)

{

**return** (x \* (1.0 - x));

}

//(6/(1+e^-x)) - 4.45

**public** **double** customSigmoid (**double** x)

{

**return** ((21.0 / (1.0 + Math.*exp*(-x))) - 19.0);

}

**public** **double** customSigmoidPrime(**double** x)

{

**return** (12.0\*Math.*exp*(-x) / Math.*pow*(1.0 + Math.*exp*(-x), 2));

}

/\*\* \* Return a bipolar sigmoid

\* \* **@param** x The input weighted sum of neuron

\* \* **@return** f(x) = (2 / (1 + e(-x))) - 1 \*/

**public** **double** bipolarSigmoid(**double** x)

{

**return** ((2.0 / (1.0 + Math.*exp*(-x))) - 1.0);

}

**public** **double** bipolarSigmoidPrime(**double** x)

{

**return** (0.5)\*(1.0 + x)\*(1.0 - x);

}

}

Code for Neural Net Offline Training [Using LUT Data to train NN]

**package** my.rl.robot;

**import** java.io.BufferedReader;

**import** java.io.FileReader;

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** java.io.Writer;

**import** java.nio.file.Files;

**import** java.nio.file.Paths;

**import** java.util.List;

**public** **class** NeuralNetTest

{

**double**[][] input;

**double** [] output;

**int** state\_action\_len;

**int** num\_patterns;

**static** String *laptop\_path* = "C:\\Users\\Faizan Mansuri\\My Data\\";

**public** **static** Writer *nn\_stats*;

**public** NeuralNetTest(**int** featureVectorSize, String file) **throws** IOException

{

**this**.state\_action\_len = featureVectorSize;

**this**.num\_patterns = getNumPatterns(file);

**this**.input = **new** **double**[**this**.num\_patterns][**this**.state\_action\_len];

**this**.output = **new** **double**[**this**.num\_patterns];

**try**

{

loadTrainingSet(file);

}

**catch** (IOException e)

{

e.printStackTrace();

}

}

**public** **int** getNumPatterns(String file) **throws** IOException

{

BufferedReader br = **new** BufferedReader(**new** FileReader(file));

List<String> lines = Files.*readAllLines*(Paths.*get*(file));

**int** num\_patterns = lines.size();

br.close();

**return** num\_patterns;

}

**public** **void** trainNet(**int** numInputs, **int** numHiddenNeurons, **double** learningRate, **double** momentum, **boolean** isOfflineTraining, String stats\_file, String weights\_file, **boolean** normalized) **throws** IOException

{

NeuralNet nnBipolar = **new** NeuralNet(numInputs, numHiddenNeurons, learningRate, momentum, isOfflineTraining, normalized, weights\_file);

**double** rms = 10.0;

**double** totalError = 0;

**int** epochs = 0;

**int** same\_rms = 0;

**double** rms\_prev;

**try**

{

stats\_file.replaceAll("\\\\", "/");

System.***out***.println(stats\_file);

*nn\_stats* = **new** FileWriter(stats\_file, **true**);

System.***out***.println("File created");

}

**catch** (IOException e)

{

e.printStackTrace();

}

**while** (epochs < 4000)

{

rms\_prev = rms;

totalError = 0.0;

**for** (**int** i = 0; i < **this**.num\_patterns; i++)

{

totalError += nnBipolar.trainModified(**this**.input[i], **this**.output[i]);

}

rms = rootMeanSquare(totalError);

epochs++;

//System.out.println("totalError: "+totalError+" RMS: "+rms);

*nn\_stats*.append(String.*valueOf*(totalError+" "+rms) + System.*getProperty*("line.separator"));

**if** (rms == rms\_prev)

{

same\_rms ++;

**if** (same\_rms == 500) **break**;

}

}

System.***out***.println("totalError: "+totalError+" RMS: "+rms +" Epochs: "+epochs);

nnBipolar.save(weights\_file);

*nn\_stats*.close();

}

/\*\*

\* \* RMS is the prediction error/accuracy of the NN; i.e. output

\* \* will be +/- RMS Value. In XOR case of RMS(0.05)=+/- 0.158 \*/

**public** **double** rootMeanSquare(**double** totalErrorPerTrainingSet)

{

**double** rms = Math.*sqrt*(2\*totalErrorPerTrainingSet/**this**.num\_patterns);

**return** rms;

}

**public** **void** loadTrainingSet(String file) **throws** IOException

{

BufferedReader br = **new** BufferedReader(**new** FileReader(file));

List<String> lines = Files.*readAllLines*(Paths.*get*(file));

**int** pattern\_ind = 0;

**this**.num\_patterns = lines.size();

**for** (String line : lines)

{

String[] tokens = line.split(" ");

// set input data

**for** (**int** i=0; i<**this**.state\_action\_len; i++)

{

**this**.input[pattern\_ind][i] = Double.*valueOf*(tokens[i]);

}

// set output data

**this**.output[pattern\_ind] = Double.*valueOf*(tokens[**this**.state\_action\_len]);

pattern\_ind++;

}

br.close();

}

**public** **void** printTrainingSet()

{

**int** pattern\_ind = 0;

String str = "";

**while** (pattern\_ind < **this**.num\_patterns)

{

**for** (**int** i = 0; i < **this**.state\_action\_len; i++)

{

str += **this**.input[pattern\_ind][i]+" ";

}

str += **this**.output[pattern\_ind];

System.***out***.println(str);

str = "";

pattern\_ind++;

}

}

**public** **void** printMaxAndMinQvalue()

{

**double** max = **this**.output[0];

**int** maxd = 0;

**for** (**int** i = 0; i < **this**.output.length; i++)

{

**if** (**this**.output[i] > max)

{

max = **this**.output[i];

maxd = i;

}

}

System.***out***.println("max: "+max +" at "+maxd);

**double** min = **this**.output[0];

**int** mind = 0;

**for** (**int** i = 0; i < **this**.output.length; i++)

{

**if** (**this**.output[i] < min)

{

min = **this**.output[i];

mind = i;

}

}

System.***out***.println("min: "+min+" at "+mind);

}

**public** **static** **void** main(String[] args)

{

**int** numHiddenNeurons = 17;

**double** learningRate = 0.02;

**double** momentum = 0.3;

**boolean** isOfflineTraining = **true**;

**boolean** normalized = **true**;

NeuralNetTest nnt = **null**;

String lut\_file = **null**;

**int** featureVectorSize;

**if** (normalized)

{

lut\_file = *laptop\_path* +"normalized\_lut\_table.txt";

}

**else**

{

lut\_file = *laptop\_path* +"processed\_lut\_table.txt";

}

String offlinestats\_file = *laptop\_path*+"trainingNN\_stats.txt";

String weights\_file = *laptop\_path*+"trained\_weights.txt";

featureVectorSize = 8; // not including qvalue

**try**

{

nnt = **new** NeuralNetTest(featureVectorSize, lut\_file);

}

**catch** (IOException e)

{

e.printStackTrace();

}

//nnt.printTrainingSet();

nnt.printMaxAndMinQvalue();

**try**

{

nnt.trainNet(featureVectorSize, numHiddenNeurons, learningRate, momentum, isOfflineTraining, offlinestats\_file, weights\_file, normalized);

}

**catch** (IOException e)

{

e.printStackTrace();

}

}

}