A Java-built Reinforcement Learning Agent in Robocode

Ali Magzari

02/15/2022

I. Introduction

Robocode is a programming game where one could build their robot tank and play against other tanks. A tank is made of the vehicle itself, a gun and a radar (see figure below). All three components are allowed to move independently of each other. The basic possible actions are as follow:

1. Scan enemies with radar
2. Fire bullets with power of 1, 2 or 3
3. Move ahead, back a certain number of pixels
4. Turn right or left at a specific angle

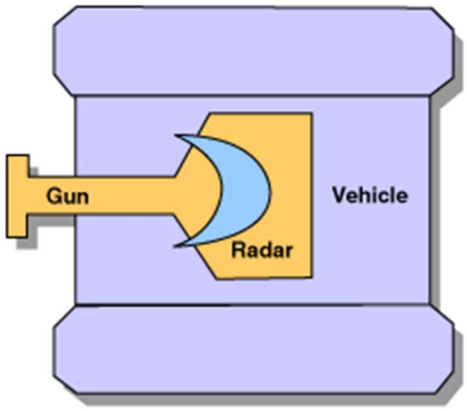


Figure 1: Robcode tank

This project consists of designing a reinforcement learning (RL) robot battle tank in Robocode environment. Unlike in supervised and unsupervised learning, there is no available data before the start of a battle. The agent interacts with its environment, and is trained through interaction (see diagram below). The agent probes the current state it is in, performs an action, and receives a reward based on the state-action combination. The reward could either be

positive or negative, and that is how the agent knows which action it should and should not perform at a specific state.

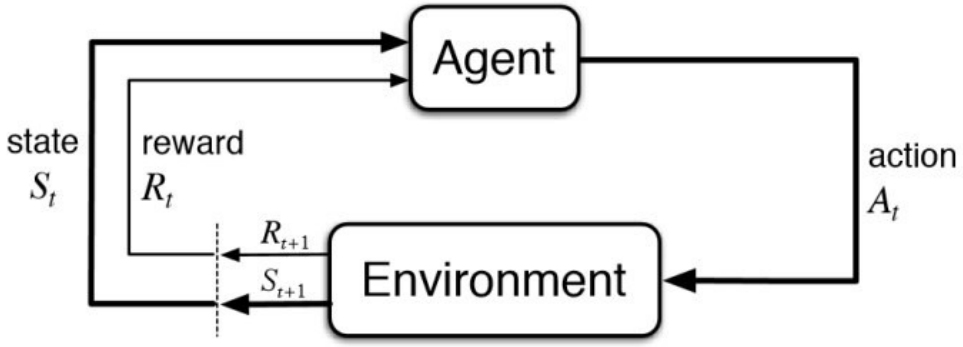


Figure 2: Reinforcement learning diagram

II. Methods

Temporal-difference learning is the method used to train the agent. As a matter of fact, both on-policy (Sarsa) and off-policy (Q-learning) strategies are used. A look-up table is gradually populated where several state-action pairs are visited and generate a specific Q-value.

1. Agent and Robocode environment

The framework is defined as follow:

1. The agent is the designed tank
2. The environment is the battle ring (shown in the figure below)
3. Every change in the environment creates a new state. In the case of this project, the state is defined as:
   a. The robot current x position (in pixels)
   b. The robot current y position (in pixels)
   c. The bearing to the enemy robot, relative to our robot's heading (in degrees)
   d. The distance between our robot and the enemy robot (in pixels)
4. The reward system is as follow:
   a. A small positive reward when our robot bullet hits the enemy
   b. A small negative reward when our robot is hit by an enemy bullet
   c. A large positive reward after our robot wins a round (a battle is composed of many rounds)
   d. A large negative reward after our robot loses a round

5. The agent was allowed to choose from five different actions:
   a. Turn right 90 degrees and move forward 100 pixels
   b. Turn right 90 degrees and move back 100 pixels
   c. Turn left 90 degrees and move forward 100 pixels
   d. Turn left 90 degrees and move back 100 pixels
   e. Points the gun at enemy position and fire a bullet with a power of 3

The first four actions were created with the purpose to allow the agent to move and dodge enemy bullets, while the latter was defined with the sole purpose to accumulate points and lower the enemy's energy.

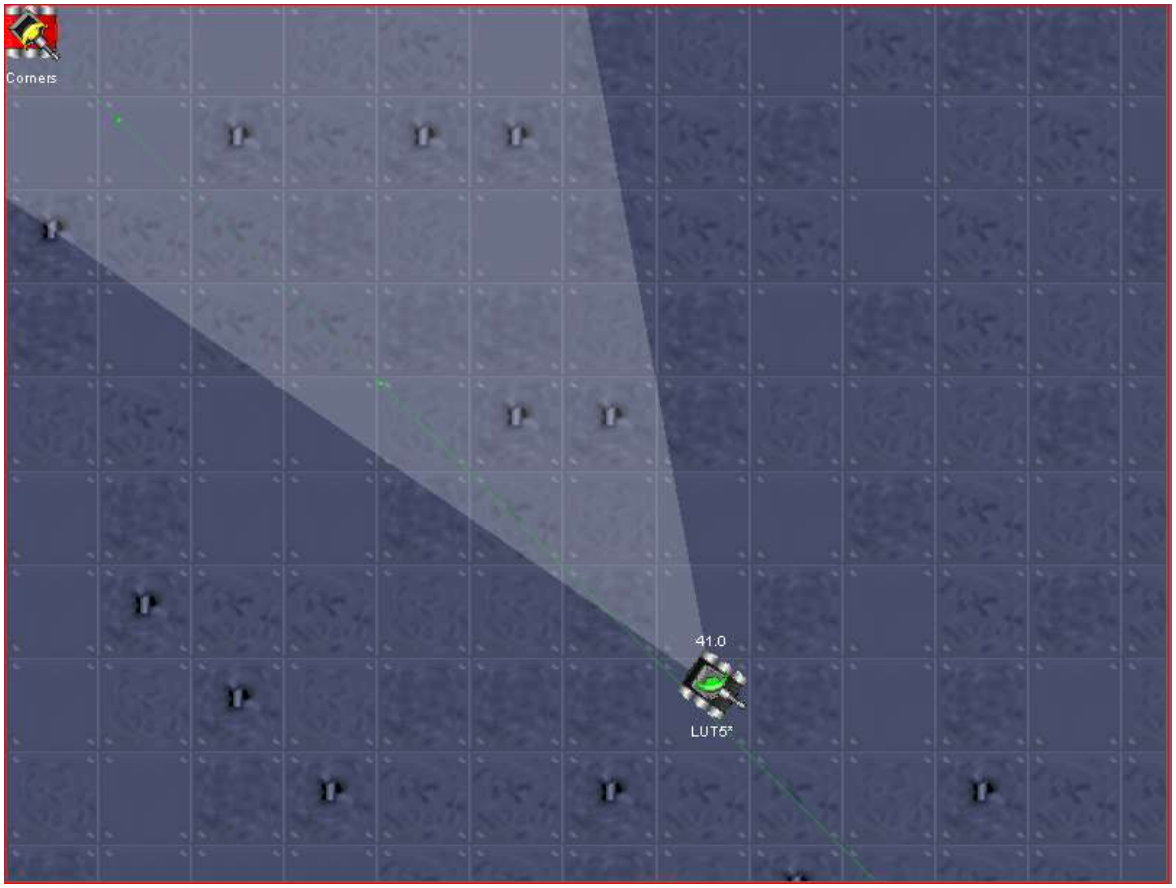
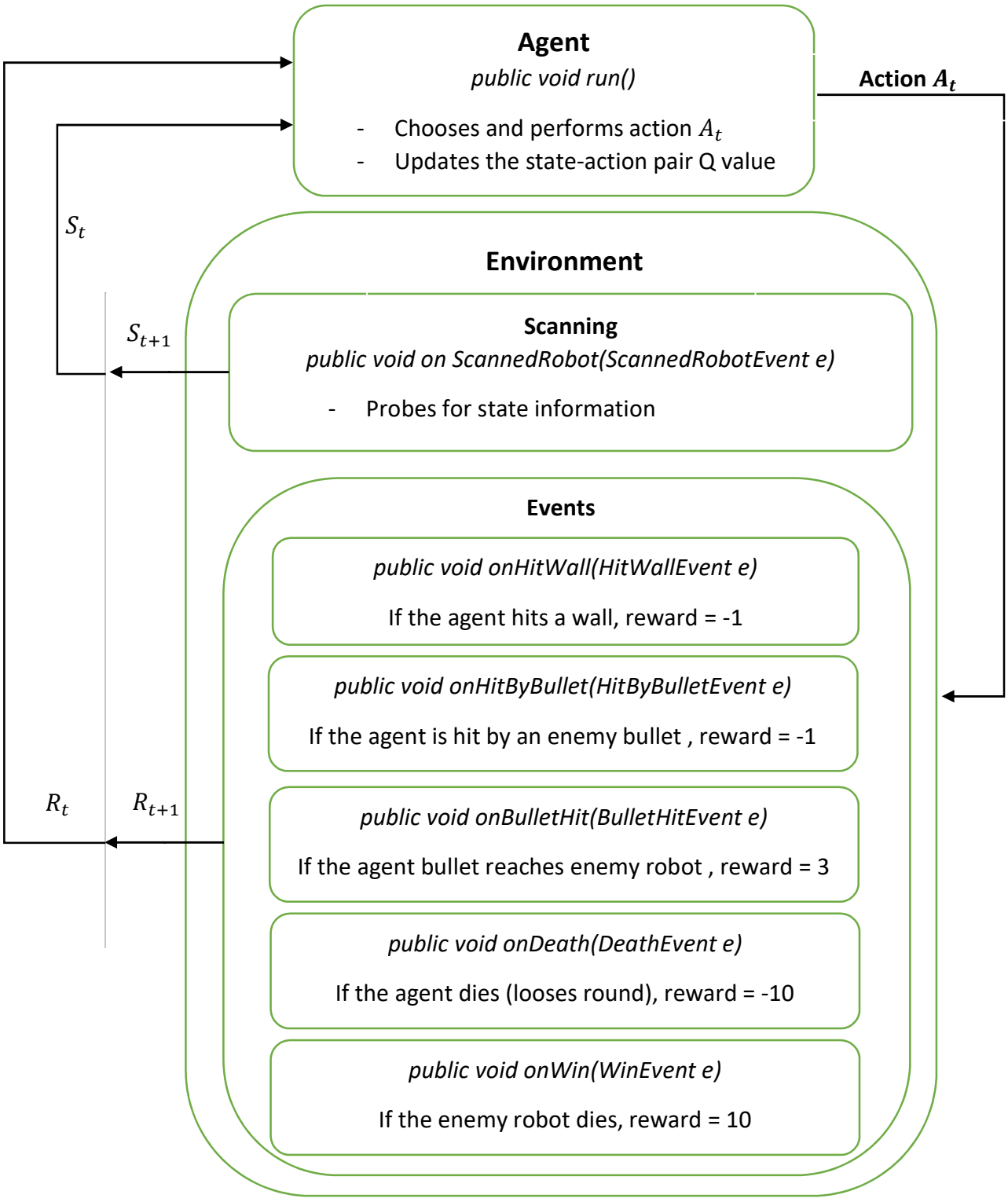


Figure 3: Robocode battle ring

A Robocode battle comprises a certain number of rounds set by the user. Each of these rounds is a series of interactions between the agent and its environment (battle ring and opponent robot). In TD jargon, a round is qualified as an episode, which is defined as an alternating sequence of states and state-action pairs [1]:

…	State *t*	Action *t*	Reward *t+1*	State *t+1*	Action *t+1*	Reward *t+2*	State *t+2*	…

In terms of Robocode, the reinforcement learning diagram is as follow:



where S_t, the state at time t is the vector containing the robot x position (in pixels), its y position (in pixels), the enemy's bearing relative to the agent heading (in degrees), and the distance between the designed robot and its enemy (in pixels), all at time t.

In an effort to reduce the dimensions of the Q-table (look-up table in which every state-action pair is assigned a specific value), the state elements are quantized as follow:

Table 1: Quantization of the state vector elements

X position	Y position	Distance	Bearing
- Division of the 0-1000 pixels continuum into 11 segments (0-100; 101-200; …; 901-1000) - Each segment is coded separately (0; 1; …; 10)			- Division of the unit circle into 8 eights - Each portion is coded separately (0; 1; 7)

A look-up table is gradually populated where several state-action pairs are visited and generate a specific Q-value.

2. Sarsa (on-policy TD)

Sarsa stands for state-action-reward-state-action. The heart of this algorithm lies in the value of the state-action pair, defined below:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

where:

- $Q(S_t, A_t)$ is the Q-value (cumulative discounted reward) at the current state S_t and action A_t,
- α, the learning rate,
- γ, the discount factor,
- $Q(S_{t+1}, A_{t+1})$, the expected future reward

The pseudo code for on-policy TD control is shown below:

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figure 4: Sarsa pseudo code

The factor *epsilon* controls the policy greediness, and is hence considered an exploration-exploitation tradeoff. For instance a 0-greedy policy would mean no exploration, and therefore full exploitation. This is a dangerous approach as the agent would not learn potential environment changes, but rather exploit rewards stemming from previously visited state-action combinations. On the other extreme, a full exploration policy would keep exploring all the possible actions possible at a given state, but never quite take advantage of a high reward that could be generated from a known state-action pair. This explains the importance of tradeoff, where both exploration and exploitation are crucial for the agent to learn and capitalize on its learning to obtain positive rewards.

3. Q-learning (off-policy TD)

This algorithm follows the exact same principles that Sarsa uses, except in its Q-update. While Sarsa chooses the next action and updates the concerned Q-value using the same epsilon-greedy policy, Q-learning is different. Q-learning does in fact choose the next action based on the epsilon-greedy policy, but updates the Q-value using the action that maximizes the reward for the visited state. This difference is behind considering Q-learning an off-policy TD algorithm.

The pseudo code for off-policy TD control is shown below:

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
    $S \leftarrow S'$
  until S is terminal

Figure 5: Q-learning pseudo code

III. Results

This section assesses the agent learning depending on different variables: on versus off-policy, immediate versus terminal rewards, and level of greediness. The learning performance that was used to evaluate the learning progress was the winning rate expressed by the equation below:

$$Rate_{Winning} = {}^{N_{Wins}}/_{N_{Rounds}}$$

where N_{Wins} is the number of rounds that the agent won, and N_{Rounds} is the number of rounds played.

1. Sarsa versus Q-learning

The graph below compares the learning progress using both Q-learning (off-policy) and Sarsa (on-policy).

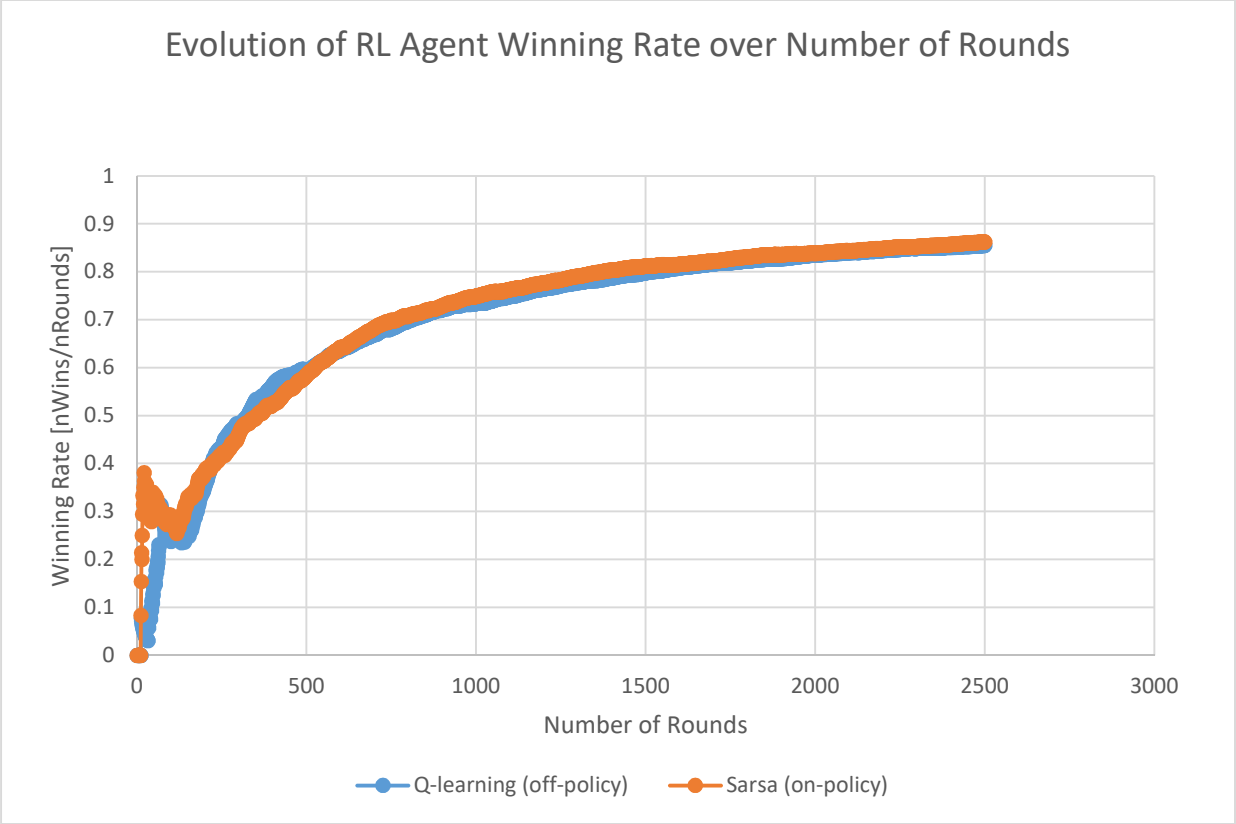


Figure 6: Learning Progress of RL agent with on and off-policy

It was expected that the on-policy learning curve would be consistently and significantly higher than off-policy after a certain number of episodes as would suggest the graph on *Reinforcement Learning, Sutton and Barto* [2]. Our generated data does not show such gap, but nevertheless, it can be seen that the on-policy shows a subtle progress increase compared to off-policy after the 500 rounds mark.

2. Intermediate versus terminal rewards

This section tries to determine if feeding the agent rewards after every step necessarily improves its learning compared to an approach that strictly rewards it at the end of every round.

The obtained results are what was expected. Intermediate rewards are applied several times during a battling round, which helps the agent learn and frequently update its Q-values. On the other hand, the terminal rewards are only granted after the end of a battle. This slows the agent learning since it can only receive feedback after a win or a loss.

The graph below compares the evolution of winning rate using intermediate and terminal rewards only.

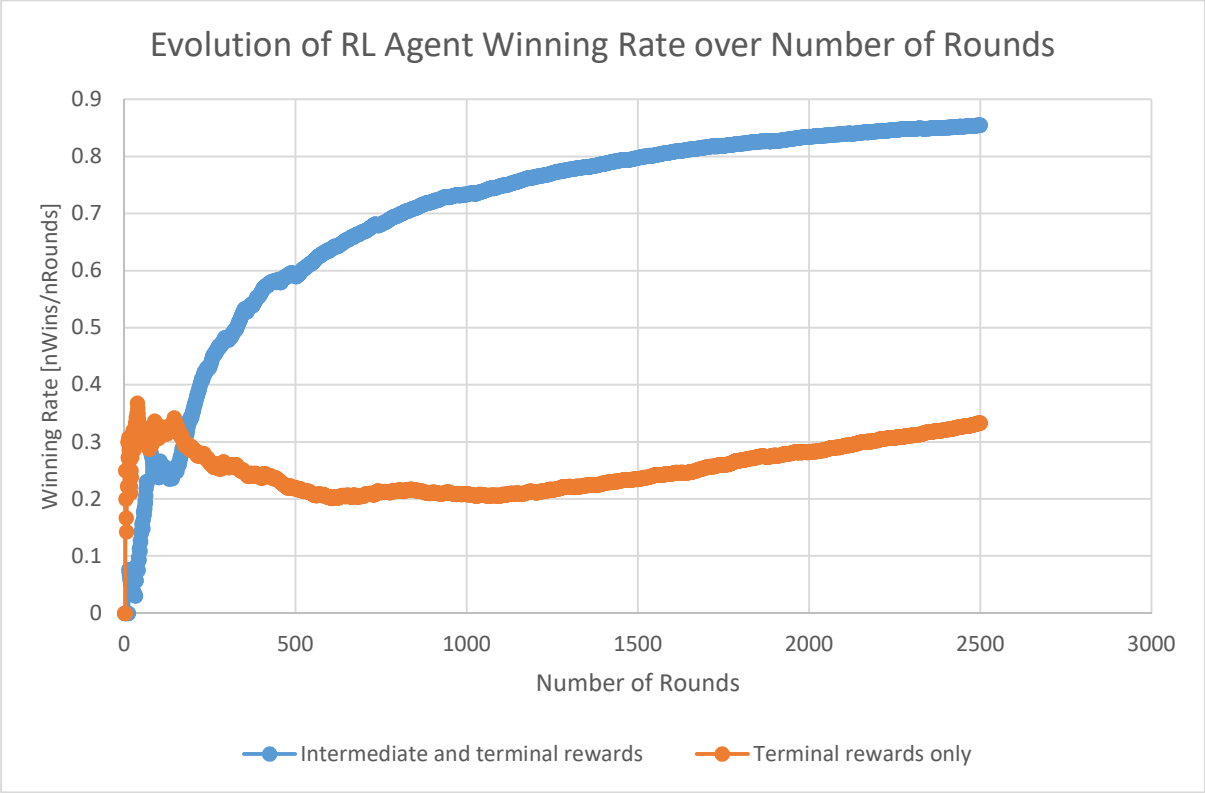


Figure 7: Learning progress of RL agent with and without intermediate rewards

3. Impact of greediness level

This sections compares the training performance using different values of epsilon, including no exploration at all. The graph below shows the learning curves of our RL agent under different epsilon values.

As depicted in the graph, the full-exploitation curve (epsilon = 0) pains to learn, but improves after 500 rounds and is even able to rise over its counterparts, with the exception of epsilon-2 curve which achieves the highest performance.

On the other hand, the full-exploration curve (epsilon = 1) is able to learn at a substantial rate before reaching a performance plateau after the 240th round. The rest of the curves (epsilon = 0.4; 0.6; 0.8) end up achieving performances that lie in between the full-exploration and full-exploitation curves.

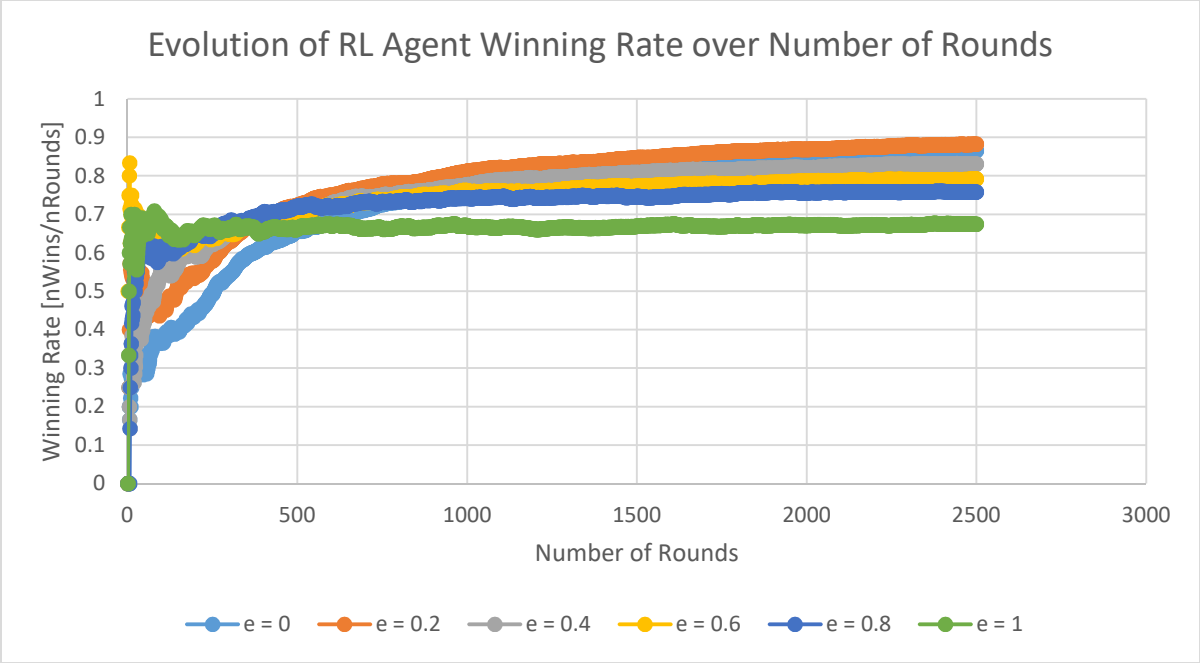


Figure 8: Learning progress of RL agent at various greediness levels

IV. Possible improvements

One of the improvements that could benefit the agent performance and learning time is to choose more powerful actions. For example besides firing, the remaining four actions were a combination of turning right/left with a certain degree and moving back/ahead a certain number of pixels. Although this approach still managed to generate a high winning rate, I believe that more studied actions would have been more adequate. For instance:

- Action 1: *Fire with power of 1*. This is a good approach to use when the enemy is far as a bullet with low power would travel faster and therefore allow us to obtain more points.
- Action 2: *Fire with power of 3*. This is a good approach when the enemy is close. A slow bullet of power 3 would not be much of an issue, and will also cause more damage to the enemy and provide us with more energy.
- Action 3: *Dodge bullets*. An intelligent macro action to allow the agent to move in an unpredictable fashion (similar to Crazy or Spinbot) to dodge the enemy bullets. This would be useful especially when the enemy just sits at one position and keeps a shooting thread (Corners for instance).
- Action 4: *Advance*. This action would be useful in the case when the enemy robot has low energy and ramming (coupled with firing at power of 3) would cause a lot of damage in a short amount of time.

An efficient implementation of wall smoothing would have proven helpful for our agent to immediately leave the blocked position and save itself from enemy attack.

V. Bibliography

Bibliography

[1] R. S. Sutton and A. G. Barto, "Temporal-Difference Learning," in *Reinforcement learning*, The MIT Press, 2018, p. 129.

[2] R. S. Sutton and A. G. Barto, "Q-learning: Off-policy TD Control," in *Reinforcement learning*, The MIT Press, 2018, p. 132.