Adaptive Intelligence - Assignment 2

Aleksandra Kulbaka
The University of Sheffield, Sheffield, United Kingdom
awkulbaka1@sheffield.ac.uk

***Abstract*—This report investigates the effectiveness of a Reinforcement Learning SARSA algorithm implemented using Neural Network on a homing problem. The choice for the optimal parameters, the possible improvements to the implementation and the disadvantages of the algorithm are summarised and discussed.**

I. Introduction

Machine Learning is one of the fastest-growing research areas. It investigates how computers can learn and improve based on data. There exist three general categories of machine learning. Supervised learning takes a set of labelled data as an input and learns how to correctly map data onto the labels [2]. Unsupervised learning given a data set with no labels needs to understand how to cluster/group the data based on the identified properties [6]. Finally, Reinforcement Learning takes the state-action pairs as an input with a goal of maximizing future rewards over many time steps in a given environment [2]. The last machine learning technique is the topic of this report.

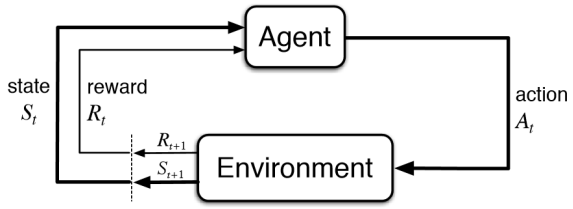

Fig. 1. RL Learning Loop by Sutton and Barto [10]

In Reinforcement Learning, an agent learns continually from its environment by interacting with it and receiving positive or negative reward based on its actions [8]. This learning loop is shown in Figure 1. In order to create a Reinforcement Learning system, the following sub-elements need to be identified: *environment* (the world where the agent exists), *actions* (moves the agent can make in the environment), *policy* π (strategy to determine next action based on the current state), *reward* R (immediate positive or negative reward to evaluate the last action), *action-value* Q (the total amount of reward an agent can expect to receive in the future, starting from given state s, taking action a under policy π) [7], [10]. RL algorithms usually differ in policy or action-value (Q-value) implementation.

II. Methods

A. Agent, Actions, Environment & Reward

Robot (agent) moves in a grid world of 10×10 squares. It has to learn a "homing" task of returning to a particular reward location (end) which is chosen at random at the beginning of the experiment. It is assumed that the robot has been familiarised with the environment and therefore it has some internal representation of its own position in the space, but no explicit memory of the reward location. The experiment contains 1,000 trials over which the robot is allowed to explore its environment until it finds the reward. The robot is allowed to move North, South, East and West. Positive reward $r = 1$ is received at the end point. Negative reward $r = -1$ is received when the predefined number of steps is exceeded.

B. Policy & Q-value

There exist two main policy strategies that can be implemented in the RL model. An on-policy agent learns the value Q based on its current action derived from the currently used policy, whereas the off-policy agent learns it based on the action derived from another policy like greedy policy [5], [7].

SARSA is an on-policy algorithm. Q-value is updated based on the current state s_t, current action a_t, reward obtained r_{t+1}, next state s_{t+1}, and next action a_{t+1} [5]:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\delta \tag{1}$$

$$\delta = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \tag{2}$$

where α is a learning rate that determines to what extent newly acquired information overrides old information, and γ is a discount factor that determines the importance of future rewards [1].

C. Implementation

The SARSA algorithm was implemented using Neural Network. The following pseudo-code shows the steps of the algorithm alongside the update rules for the neurons weights:

Algorithm 1: SARSA with Neural Networks

Initialize weights w and Q
for *all trials* **do**
  $s \leftarrow$ start state
  Choose a from s based on policy from Q
  while $s \neq end$ *and* $steps < maxSteps$ **do**
    Take action a, observe r, s'
    $Q = 1/(1 + e^{-w_{s,a}})$
    Choose a' from s' based on policy from Q
    $w(s,a) \mathrel{+}= \alpha[r + \gamma Q(s',a') - Q(s,a)]$
    if $s = end$ **then**
      $w(s,a) = w(s,a) + \gamma(r - Q(s,a))$
    end
    $s \leftarrow s'$
    $a \leftarrow a'$
  end
end

D. Exploration vs Exploitation

One of the challenges in Reinforcement Learning is the trade-off between exploration and exploitation. To obtain high total reward, an agent must prefer actions that shown to be effective in producing reward. However, to discover such actions, it has to try actions that has not been selected before. The agent has to *exploit* what it has already experienced in order to obtain reward, but it also has to *explore* in order to make better action selections in the future [10].

SARSA, being an on-policy algorithm does not explore new actions but always chooses the action with the highest expected reward - it exploits the environment. A way to include exploration is to choose the action with the highest expected reward with probability $1 - \epsilon$, and the random action the rest of the time. This policy is known as ϵ-greedy [4].

Learning of a robot can be shown using a learning curve - graphical representation of the efficiency of an agent (number of steps it took the robot to reach its target) versus its current experience (trial number). Figure 2 shows the learning curve averaged over 20 runs of the SARSA algorithm with the ϵ-greedy policy and $\alpha := 0.5$, $\gamma := 0.9$, $\epsilon := 0.2$, max steps := 50 and randomly initialised weights.

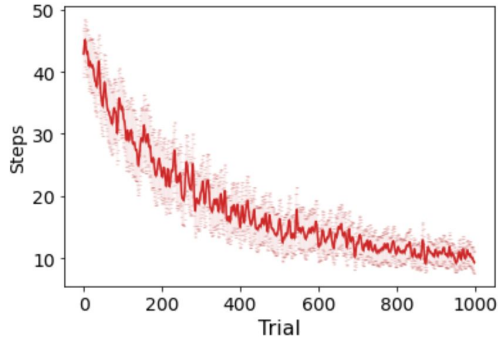

Fig. 2. Learning curve of SARSA

E. Improvements

Number of experiments has shown that instead of initialising weights as random float values from a univariate Gaussian distribution of mean 0 and variance 1, it is better to set them to 0. Random weights make robot more biased to choose the directions that may be away from the *end* point, thus make him less effective.

Another improvement was added to the allowed steps in the grid. In the initial implementation, if a robot chose the forbidden action (like going North at the top of the grid), it was moved back to its previous position, but the step was added to the total count. This made the agent less efficient. To prevent it, the dictionary of allowed moves for all cells has been created. In the new implementation, at any state, the robot can only choose new action from the set of allowed moves ($actions$) so it cannot move outside the grid:

$$action = \begin{cases} max(Q[actions]) & \text{if } greedy \\ random(actions) & \text{otherwise} \end{cases} \quad (3)$$

Figure 3 shows that the improvements accelerated the learning in most all the trials.

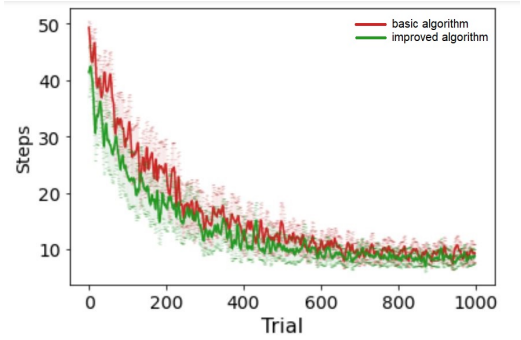

Fig. 3. Learning Curves of initial and improved implementations

III. Results and Discussion

A. Optimising the parameters

Having developed the SARSA ϵ-greedy algorithm, the set of experiments was conducted to find the optimal values for all the parameters. The parameters have been set one at a time, in the following order: learning rate, discount factor, exploration factor. Having found the optimal values for all of them, the experiments were ran again to ensure the validity of the obtained results. Each value was tested 15 times. The number of extra steps over the optimal path per trial averaged over the run was used as a success measure of the learning.

The algorithm with parameters: $\alpha := 0.7$, $\gamma := 0.99$, $\epsilon := 0$, max steps := 200 gave the best results of around 2.03 extra steps on the average trial and these values have been used in all the following experiments.

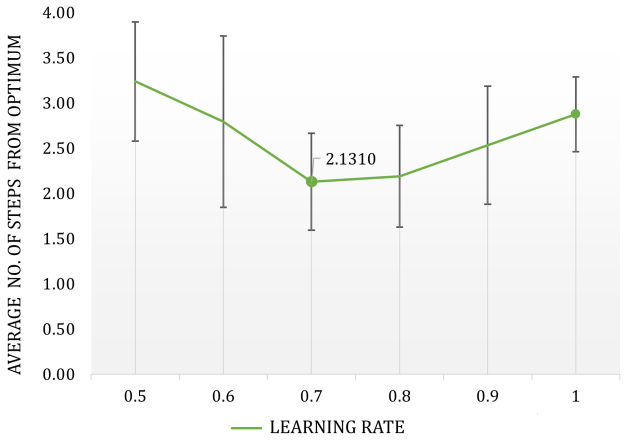

Fig. 4. Optimal Learning Rate

1) Learning Rate α: determines how important to the agent's learning is the new information. $\alpha = 0$ means that the agent will not learn anything, while $\alpha = 1$ will make the agent consider only the most recent information [9]. As shown in Figure 4, based on the experiments the best value for α is 0,7. Since the world is very simple, with only 2 rewards, setting α to 0.7 is reasonable - the risk of unstable training process is relatively low, and the training process is fast and efficient [3].

2) Discount Factor γ: models the significance of future rewards over the immediate rewards. Its value should be between 0 and 1, where 0 means that the robot only considers the current rewards. The closer to 1, the more important the long-term reward is [9]. Figure 5 shows, that the best value of γ in the experiment is 0.99. This happens because all the states except one have no reward at all. Hence the robot needs to value the expected reward much more than the current one.

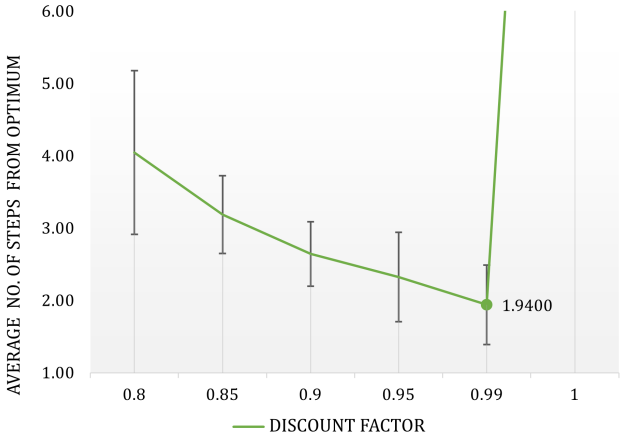

Fig. 5. Optimal Discount Factor

3) Exploration Factor ϵ: states the probability of robot making the random decision to explore more states [4]. As shown in Figure 6, the best value for ϵ is 0. The start position of the agent is chosen at random during each of the 1,000 trials, which makes the robot explore the whole search space. Therefore, introducing even more exploration by making some decisions at random is unnecessary and decreases the algorithm's performance.

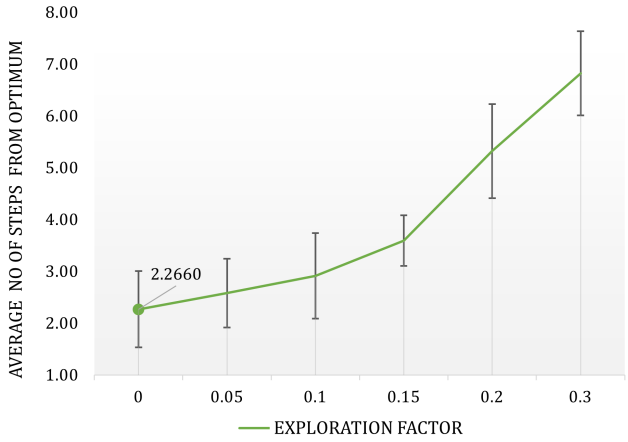

Fig. 6. Optimal Exploration Factor

In order for ϵ-greedy strategy to be more efficient than greedy, the start state should be set at the beginning of the experiment and remain constant for all the trials or the world should become more complex.

B. Plotting the preferred direction

At the end of the run, the weights store the probability of robot going North, East, South and West from a given

state. This data can be used to plot the robot's preferred direction for all the states in the world. Such graph can be implemented in various ways, e.g. using heat maps. In this report, the preferred direction is plotted using a Quiver plot. One possible implementation is plotting the step a robot takes in each state. This is done by showing just the direction with the largest weight for every state in the world. Such graph is useful for plotting data from a single algorithm run. In Figure 7, the *end* state is indicated by a green X. It can be seen that from any state in the world, the robot would move closer to the reward.

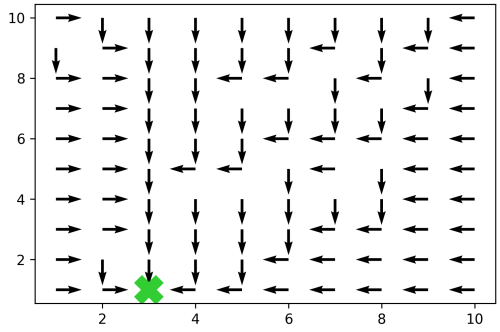

Fig. 7. Map of the chosen actions at each state

When one wants to plot the preferred direction averaged over multiple runs or to see how certain the robot was to move in the preferred direction, the second implementation is more appropriate. The final direction of the arrow at each state is computed by subtracting South vector from North vector and East vector from West vector (weights values). As shown in Figure 8, in most of states the robot hesitated between two directions. For example, in states located at the top right part of the world (coordinates: 2...10, 4...10), the correct action was either going South or West which is indicated in the graph by the arrows' directions.

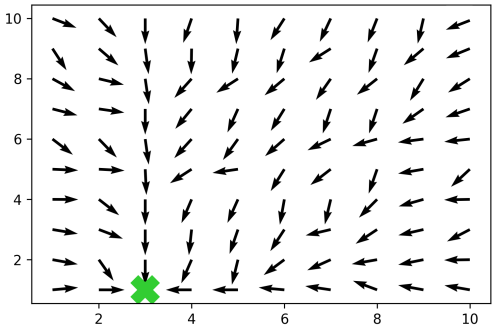

Fig. 8. Map of the preferred directions at each state

C. Adding negative rewards to the world

To assess the performance of SARSA algorithm in more complex world, the negative rewards are introduced. There are 12 cells with negative reward $r = -0.1$ which form 3 walls in the world.

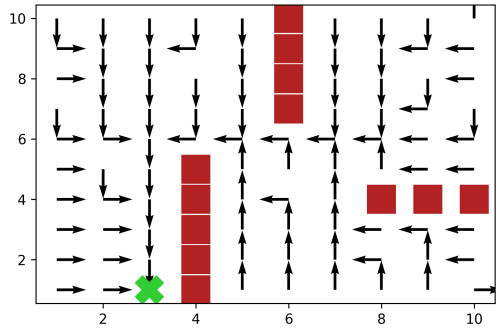

Fig. 9. Map of chosen actions in a world with negative rewards

In Figure 9, the red squares indicate the walls and the green X - *end* state. The arrows show the preferred direction from any state in the world. Based on the graph, it can be derived that robot learnt to avoid walls (no arrow is pointing at any wall) and it found the shortest path to the *end* state.

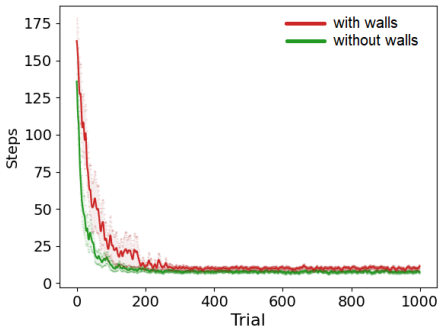

Fig. 10. Comparison of Learning Curves: different worlds

Finding the optimum path in the more complex world takes longer than in world with no obstacles. Figure 10 compares the learning curves for these two cases.

Another observation is that in more complex world the impact of maximum number of steps allowed in each trials is much more significant - the algorithm with smaller number of maximum steps takes longer to find an optimal path (see: Appendix A for a graph). This leads to the conclusion that all the parameters in the algorithm should be tuned after every change to the implementation or to the world.

D. Eligibility Trace

Almost any temporal-difference method, (such as SARSA), can be combined with eligibility traces to obtain a more general method that may learn more efficiently [10]. The main idea behind this is instead of updating one state-action pair after getting a reward, all Q-values are updated once the reward is received. In order for this to be achieved, an eligibility trace needs to be maintained for each state-action pair. This trace indicates how much the current reward will influence the Q-values [4].

Adding the eligibility trace did not have much impact on the overall performance of the algorithm, regardless of the value of λ. The learning curve of the initial implementation on the simple world with no walls was even slightly better than the one produced by the improved version (see: Appendix B for the graph). Main reason for that may be the simplicity of the world and the limited number of states and actions. Eligibility Trace becomes very beneficial with complex worlds where it is impossible to visit all states sufficient number of times [10]. Figure 11 seems to confirm that theory. Applying the Eligibility trace to the world with negative rewards improved the overall performance of algorithm and made the robot find the optimal route faster.

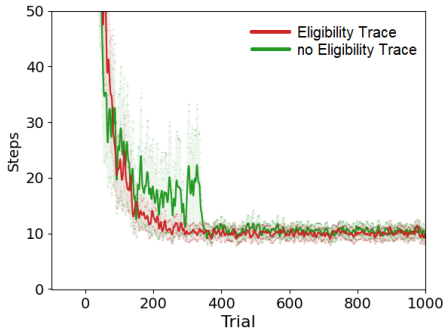

Fig. 11. Comparison of learning curves: impact of Eligibility Trace

E. Expanding the world

Simple Reinforcement Learning Algorithms like SARSA are very effective for worlds with limited number of states to explore and small number of actions to take. Increasing the search space even by a small number can be very time and memory-consuming - the problem is not just the memory needed for large tables, but the time and data needed to fill them accurately [10]. Expanding the world to the 100 $\times$ 100 grid, increased the runtime of the developed algorithm from a few seconds to almost 50 minutes. Increasing the number of states to 1,000,000 (1,000 $\times$ 1,000) has caused the Memory Error (*"MemoryError: Unable to allocate 7.28 TiB for an array with shape (1000000, 1000000) and data type float64"*). The reason for the poor performance of SARSA is that the agent has to visit each state many times in order to find the best action for a given node. To ensure that, the number of trials and the maximum number of steps need to be sufficiently large (orders of millions).

In problems with large state space the goal is to find a good approximate solution using limited computational resources. Almost every state encountered by the robot will never have been seen before. Therefore to make sensible decisions in such states it is necessary to generalize actions from previously visited states that are similar to the current one [10].

Possible solution to such problems is to use Eligibility Traces which may speed up the computations. For the problem described in this report, where vast majority of states is similar, the best method to use may be Real-time Dynamic Programming. RTDP is guaranteed to find an optimal policy on the relevant states without visiting every state infinitely often, or even without visiting some states at all [10].

IV. Conclusions

The report explored the performance of SARSA algorithm on the homing problem. With parameters $\alpha := 0.7$, $\gamma := 0.99$, $\epsilon := 0$, max steps := 200, the algorithm achieves the performance of around 2.03 extra steps on average trial in a world with no obstacles and of around 8.11 extra steps in a world with more negative rewards. Experiments have shown that the parameters need to be tuned after every change to the implementation or to the world.

The effectiveness of Eligibility Trace and ϵ-greedy strategy have been examined. Based on the results it can be derived that these improvements may decrease performance of SARSA algorithm on a very simple world. They become beneficial once the number of states or the number of rewards in the world has increased.

Further research could compare the results obtained in this report with the performance of other RL algorithms like Q-learning. Another research question is to examine the performance of SARSA and its improvements in a world where every step has a negative reward.

Acknowledgement

The code has been based on the material from Lab 8 and Lab 9 of COM3240 Adaptive Intelligence created by Dr Matthew Ellis.

References

[1] Reinforcement learning. In *Machine Learning Tutorials*. UNSW Sydney, 2018. Online Course.

[2] Mit 6.s191 (2020): Reinforcement learning. MIT, 2020. Online Course.

[3] BROWNLEE, J. Understand the impact of learning rate on neural network performance, 2020. Online, Accessed: 15- May- 2020.

[4] ELLIS, M. Lectures 8-9: Reinforcement learning. In *COM3420 Adaptive Intelligence*. University of Sheffield, 2021. Course.

[5] GUPTA, A. Sarsa reinforcement learning, 2019. Online, Accessed: 15- May- 2020.

[6] HEIDENREICH, H. What are the types of machine learning?, 2018. Online, Accessed: 15- May- 2020.

[7] KUNG-HSIANG, H. Introduction to various reinforcement learning algorithms. part i (q-learning, sarsa, dqn, ddpg), 2018. Online, Accessed: 15- May- 2020.

[8] PIONEERLABS. The three types of machine learning algorithms, 2021. Online, Accessed: 15- May- 2020.

[9] SORGE, V. Lecture 12: Reinforcement learning. In *Intoduction to AI*. University of Birmingham, 2018. Course.

[10] SUTTON, R. S., AND BARTO, A. G. *Reinforcement Learning: An Introduction*, second ed. The MIT Press, 2018.

Appendix A - Maximum Number of Steps

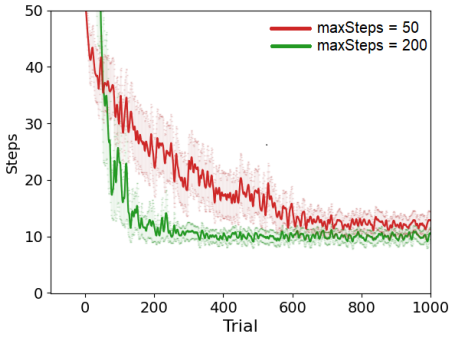

Fig. 12. Comparison of Learning Curves: impact of changing the maximum number of steps allowed

Appendix B - Eligibility Trace

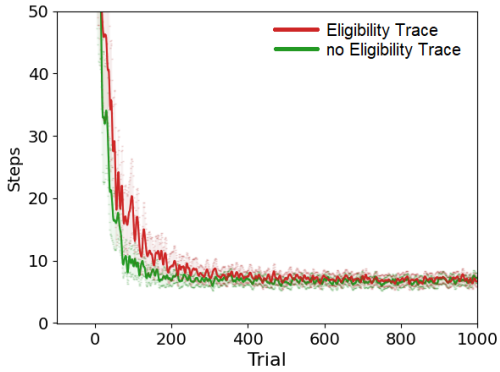

Fig. 13. Comparison of learning curves: impact of Eligibility Trace in a world with no obstacles

Appendix C - Code

A. *Code structure and how to reproduce the results*

The SARSA algorithm has been implemented inside the *homing_task.py* file alongside with methods to plot the learning curves and the preferred directions. *experiments.py* shows how to repeat the experiments described above. To run the experiments described in different tasks in the assignment brief, the desired functions need to be invoked. Code from both files is shown below.