Deep Reinforcement Learning Report

GitHub Repository: https://github.com/nicofinzi/Deep-Reinforcement-Learning

1 BASIC

1.1 Define Environment & Present Problem to be Solved

The environment that was built for this project can be seen in Figure 1 and consists in a 5x5 grid space, which simulates a maze. The agent, represented by the maze runner, starts in the top left corner and its goal is to get to the treasure, which is situated at the bottom right corner. Along the way that leads to the treasure, there are two traps and a key, which the agent has to collect in order to open the treasure and finish the episode. If the maze runner reaches the treasure without the key, then he gets penalised for attempting to end the episode early and the task continues until he exceeds the maximum number of steps or if he goes back to grab the key and is able to make it to the treasure again. Additionally, the above goal has to be executed in the fewest possible number of steps.

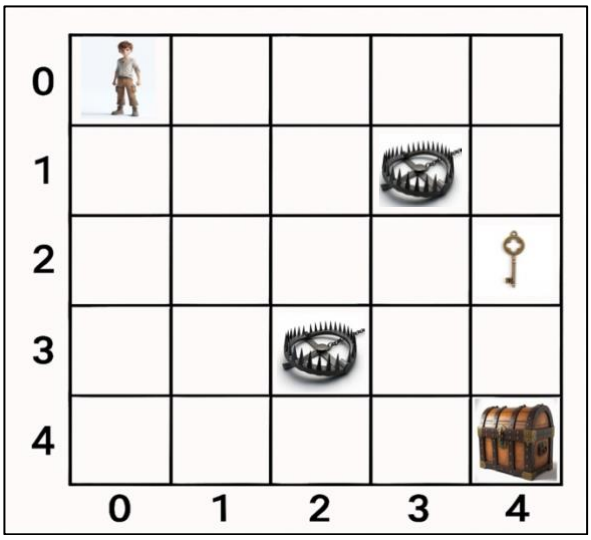

Figure 1. 5x5 maze runner grid space environment with agent, traps, key, and treasure [1-5].

The problem described in the above environment is going to be resolved with the utilisation and implementation of the Q-learning algorithm.

1.2 State Transition & Reward Functions

The state transition function describes how the agent's state changes in response to an action and is defined as $s_{t+1} = \boldsymbol{t}(s_t, a_t)$, while the reward function is dependent on the cell on which the agent moves to and is specified as $r_{t+1} = \boldsymbol{r}(s_t, a_t)$, where s_t belongs to a finite set of states S and a_t belongs to a finite set of actions A, at any discrete time t; the agent is only aware of s_{t+1} and r_{t+1}, but not of $\boldsymbol{t}$ or $\boldsymbol{r}$ [6].

In the case of the above explained environment, the state is represented by the position of the agent on a 5x5 grid space, which means there are $5 * 5 = 25$ possible agent states. The starting state of the agent is at the cell with coordinates (0,0), the traps are positioned in cells (1,3) and (3,2), the key is situated in cell (2,4) and finally, the treasure is at (4,4). The just mentioned coordinates are of the 2D type but could also be expressed as 1D integer numbers, if flattened, using the following formula: $state = y * grid\_size + x$, where in our case $grid\_size = 5$. The modified states can be observed in Table 1, these are easier to refer to and to work with.

Object Reached	Agent state with (y,x)-coordinates	Agent state with integer-coordinates
Trap #1	(1,3)	8
Trap #2	(3,2)	17
Key	(2,4)	14
Treasure	(4,4)	24

Table 1. Relevant agent state coordinates when reaching each object.

The agent can move in 4 possible directions – up, down, right or left, performing only one step at a time. Depending on the action executed by the maze runner, he's going to receive a reward accordingly. For example, every step that the agent takes is rewarded negatively, with -1, to incentivise the agent to come up with the shortest possible path to reach the treasure (while also collecting the key). Only the steps which make the agent land on an object are not rewarded with -1, as they're given the reward attributed to the object in question. All other actions and their corresponding rewards can be noted in Table 2. Generally, rewards are used as a means to improve the learning trajectory of the agent to achieve the required goal.

Action	Reward
Agent takes a step in any direction	-1.0
Agent collects key	+15.0
Agent encounters any of the two traps	-10.0
Agent gets to the treasure without the key	-3.0
Agent gets to the treasure with the key	+30.0

Table 2. Every possible action type and its corresponding negative/positive reward.

1.3 Q-Learning Parameters & Policy

The initially defined and utilised policy for the Q-learning algorithm was the epsilon (ε)-greedy policy. All of its relevant parameters with their corresponding definitions and starting values are reported in Table 3.

Parameter	Definition	Value
Alpha (α)	Agent's learning rate	0.7
Gamma (γ)	Discounting factor for future rewards [7]	0.95
Max Epsilon (ε)	Exploration probability at the very beginning	1.0
Min Epsilon	Minimum possible exploration probability	0.05
Decay Rate	Exponential decay rate of the exploration probability	0.0005

Table 3. Definition and value of every hyperparameter used with the ε-greedy policy.

Additionally, also the number of training episodes and maximum number of steps per episode were specified as 1500 and 100 respectively.

1.4 Q-Learning Algorithm Run & Performance

The experiment with the above-described parameters was run and its initial performance was decent, but not optimal. The latter can be seen by the line graphs represented in Figure 2, more specifically, by the fact that the maximum number of steps is very often reached by the agent and the line graph of the average number of steps per episode, never plateaus to a specific value, which means that the agent is not able to find the shortest path that achieves its goal. Additionally, it can be observed that the average reward per episode does not steadily increase but fluctuates

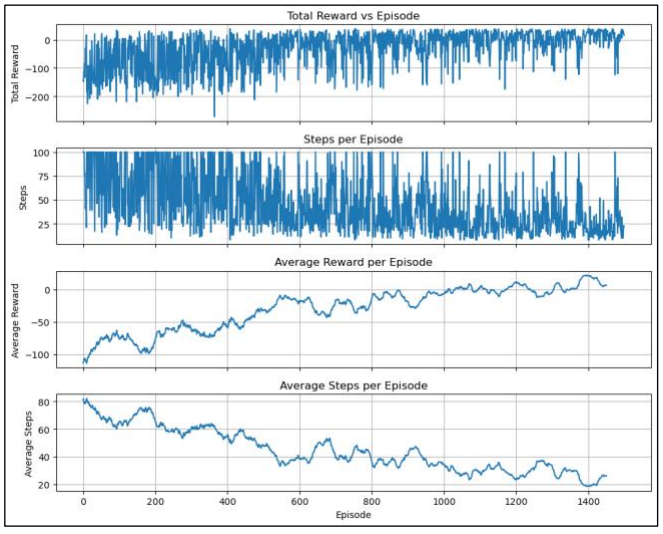

Figure 2. Line graphs representing total reward / steps / average reward / average steps per episode.

instead, while still increasing, implying that the agent is not able to consistently maximise the reward.

1.5 Experiment with Different Parameter Values & Policies

Following the mediocre results from the initial experiment, different hyperparameter values were tested with the scope of improving the agent's performance. The parameters that were changed were the following: learning rate, discounting factor and decay rate. When one parameter value was changed, the others were fixed with their starting values, which are reported in Table 3. The newly tested parameter values are recorded in Table 4. As it can be seen, it was decided to not vary the epsilon parameter, that is because as long as a suitable epsilon decay rate is found, the initial exploration probability can be set to its maximum, i.e. 1.

Parameter	Experimented Values
Alpha (α)	[0.5, 0.9]
Gamma (γ)	[0.90, 0.99]
Decay Rate	[0.001, 0.005]

Table 4. New hyperparameter values, tested to enhance performance.

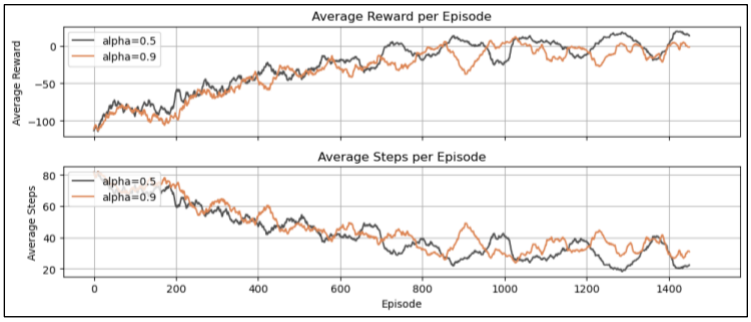

Figure 3. Line graphs representing average reward and average number of steps per episode, with varying alphas.

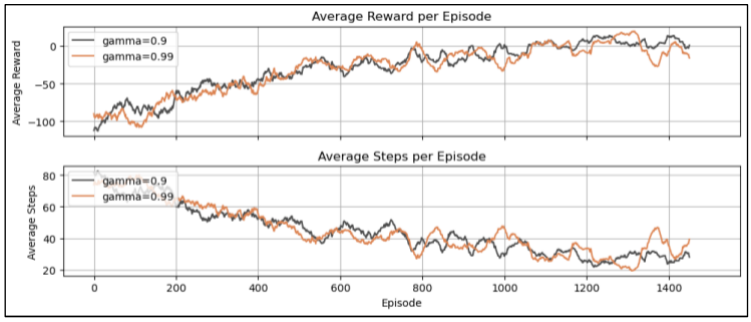

Figure 4. Line graphs representing average reward and average number of steps per episode, with varying gammas.

The line graphs shown in Figures 3, 4 and 5 represent the average reward per episode and the average number of steps taken by the agent every episode, with varying parameter values when the ε-greedy policy was implemented. The analysis of the results' performance is going to be discussed in the next section.

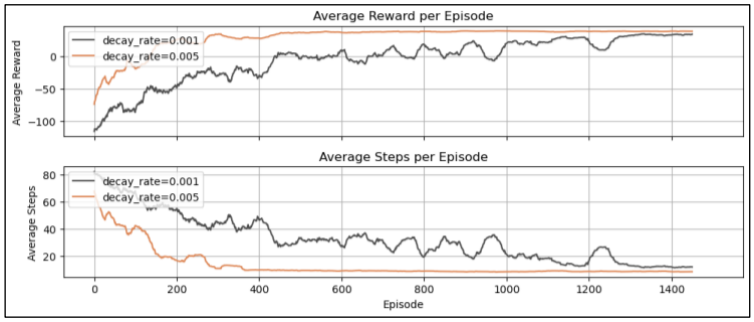

Figure 5. Line graphs representing average reward and average number of steps per episode, with varying decay rates.

After plotting the above figures, the optimal combination of hyperparameters, which yielded the best agent performance when the values of all other parameters in section 1.3 were kept fixed, came out to be the following:

- Alpha = 0.7;
- Gamma = 0.95;
- Decay Rate = 0.005.

After modifying the above hyperparameters, the next step was to change policy, so the experiment was repeated, but this time, while employing the SoftMax policy. Performance results are reported in Figures 6, 7 and 8.

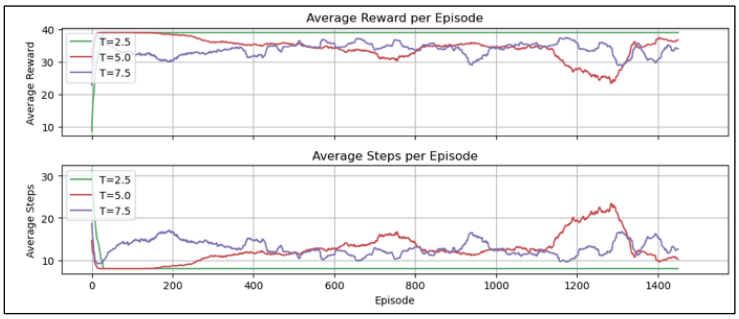

Figure 6. Line graphs representing average reward and average number of steps per episode, with varying temperatures.

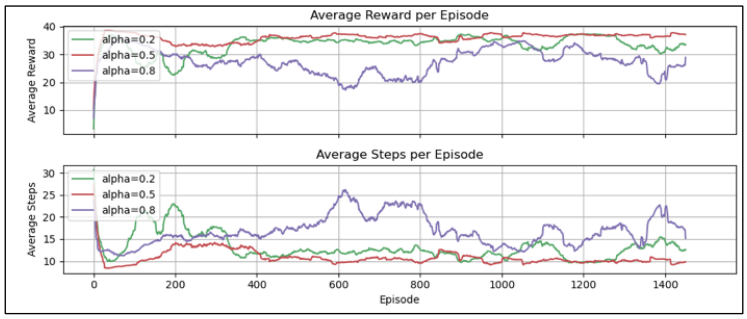

Figure 7. Line graphs representing average reward and average number of steps per episode, with varying alphas.

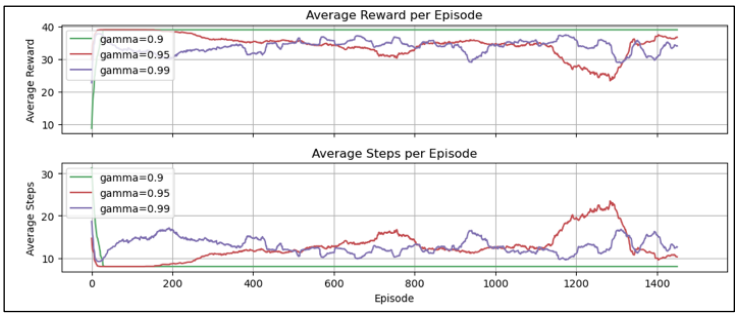

Figure 8. Line graphs representing average reward and average number of steps per episode, with varying gammas.

The SoftMax policy differs from the ε-greedy policy under various aspects, one of them being the fact that the parameters which can be tuned are different. In the case of the SoftMax policy, the parameters that were varied were temperature, learning rate and discounting factor. The same process was executed as when the ε-greedy policy was implemented, which means that two of the three parameters in question were kept fixed while the other one was changed. Almost all parameter values, reported in section 1.3, were fixed as before, apart from alpha, which this time was set to $\alpha = 0.5$ and of course temperature, which was not previously present, was set to $T = 2.5$. The change in a lower alpha was mainly due to the fact that the SoftMax policy does not have a decay rate, so the learning rate could not be put as high as before.

As it can be observed by the shown figures, the behaviour of the line graphs, when the SoftMax policy is implemented, isn't as unstable as it was for the ε-greedy policy. Moreover, the optimal hyperparameter combination, which was identified, consists in the following values:

- Temperature = 2.5;
- Alpha = 0.5;
- Gamma = 0.9.

1.6 Analysis of Results

The first policy that was implemented was the epsilon-greedy policy. The goal of the latter is to balance between the agent's exploration-exploitation rate. This is done by setting the ε value between 0 and 1. If the value is closer to 0, then the agent, in our case the runner, exploits what he learned, while if the value is closer to its maximum, i.e. 1, then the agent tends to explore more of the environment. Having a good equilibrium between exploration and exploitation is crucial in the agent's learning. In this experiment, the starting value of ε was always set to 1, as it was wanted for the agent to prefer exploring the environment at the very beginning. This was also done because there is another parameter which controls epsilon's decay rate and hence adjusts the initial value of ε throughout the experiment. The two parameter values go hand-in-hand and have to be chosen wisely, which is what was done after the initial Q-algorithm run, as it was not up to a good enough standard. Therefore, it was decided to do some hyperparameter tuning.

At first, the learning rate was changed, and no real improvement was observed, on the contrary, performance deteriorated, which can be seen from the significant decrease in average reward per episode in Figure 3, especially when $\alpha = 0.9$. In that case, the mean reward came out to be -100, which means that the agent, very often, reached the maximum number of possible steps, without getting to the key or the treasure. The situation does not get better when experimenting with the discounting rate either. Again, the agent is not able to consistently maximise reward and continues to arrive to the limit of possible steps that can be taken in every episode – when $\gamma = 0.99$, the mean reward is once more -100. This changes completely when the decay rate is varied. Both values which are experimented with, output much better results, in particular when the decay rate is equal to 0.005. In this instance, the mean reward is +39, which means that the agent learned to take the shortest path to the goal, while also collecting the key, that is $-8 + 2 + 15 + 30 = 39$. 8 are the steps taken, but 2 of those happen when the agent lands on an object, therefore they are not rewarded negatively, while 15 and 30 are the rewards for the key and treasure. This improvement in performance can also be seen from how both curves, which represent the average steps and average reward when ε-decay rate equals 0.005, in Figure 5, plateau to a value. Therefore, as previously stated, the best combination of hyperparameters, especially useful for reproducibility, is $\alpha = 0.7$, $\gamma = 0.95$, $\varepsilon - decay = 0.005$.

Next, the SoftMax policy was employed and the agent's performance, tracked. The newly implemented policy is represented by the following Boltzmann equation [6]:

$$P(s,a) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_{b \epsilon A} e^{\frac{Q(s,b)}{T}}}. \quad (1)$$

The above describes the probability of taking action a in a state s. $Q(s,a)$ is the expected return of taking a specific action, and T is the temperature parameter, which controls the randomness of each action contained in a finite set of actions A. The range of the parameter T is $(0, \infty)$. As T approaches infinity, exploration is preferred by the agent and actions are chosen uniformly, while as T gets closer to zero, the policy becomes more 'greedy' and actions which give the highest rewards are selected, favouring exploitation.

The agent's performance, when the SoftMax policy was employed can be observed in Figures 6, 7 and 8. The optimal hyperparameter combination can be very easily spotted to be $T = 2.5$, $\alpha = 0.5$, $\gamma = 0.9$. This can be seen from how quickly the agent learns when those parameters are selected, in particular the green curves in Figures 6 and 8 and the red curve in Figure 7, show how the agent manages to come up with the shortest path to achieve its goal, while also repeatedly maximising reward in almost every episode. Meanwhile, the other curves are a lot more unstable and tend to fluctuate between values. The mean reward of the just mentioned, best combination, came out to be +39, same as with the ε-greedy policy. Hence, even if in the end, the mean reward output was the same when the two best hyperparameter combinations for both policies are looked at, the quickest one to achieve a positive average reward was the SoftMax.

2 ADVANCED

2.1 DQN Implementation with 2 Improvements

2.1.1 What is a Deep Q-Network? [8]

A Deep Q-Network (DQN) is described to be a reinforcement learning model that puts together the Q-learning algorithm with a deep Convolutional Neural Network (CNN). The scope of the latter is to train a network to estimate the value of the Q-function, which maps every state-action pair to their anticipated discounted return. The inputs into the NN are the state variables, while the outputs are defined by the Q-values. A representation of this can be seen in Figure 9.

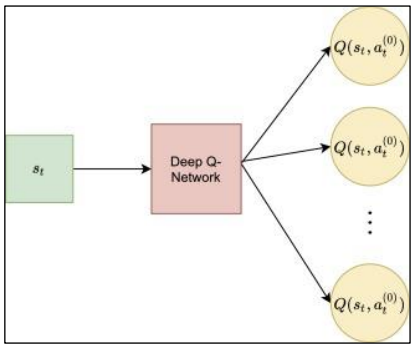

Figure 9. Deep Q-Network inputs and outputs [9].

2.1.2 Environment Description & Motivation

The environment that was chosen for the implementation of the DQN is called 'CartPole-v1' and consists in a pole balancing experiment. A pole is standing on top of a cart, and the cart has two possible actions, i.e. it can move either left or right. A value of 0 is assigned to the former action, while 1 is allocated to the latter. The state in the environment is described by 4 distinct values, representing cart position, cart velocity, pole angle, and pole angular velocity. The goal of the cart is to adjust, so that the pole stays in equilibrium and doesn't fall. [10]

One of the reasons for choosing to employ the DQN in the above-outlined environment is the fact that DQNs are made for discrete action spaces and as stated above, this environment has 2 discrete actions, which make it a good fit. Moreover, it is mostly deterministic, which facilitates the DQN to learn significant Q-value estimates.

2.1.3 DQN Implementation & 1st Improvement – Double DQN

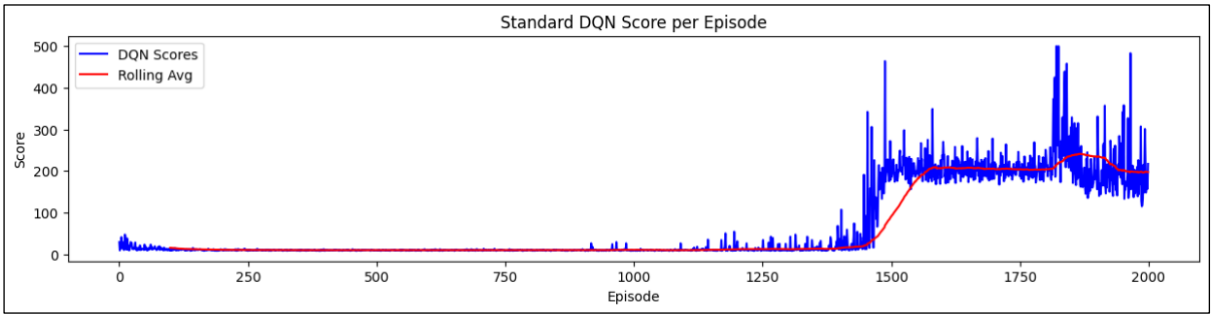

Figure 10. Line graph of the score per episode of a standard DQN, with a rolling average.

The Q-network was built using a simple Multi-layer Perceptron (MLP) model with two hidden layers and a rectified linear unit (ReLU) activation function. The number of hidden neurons per hidden layer was chosen to be 64 – it was wanted for the model to neither overfit nor underfit and it looked to perform particularly well with this number of hidden neurons. The optimiser function used was the Adam optimiser: the learning rate, alpha, was set to 0.001, the default exponential decay rate parameters, i.e. 'betas', were fixed to 0.9 and 0.999 respectively and no L2 regularisation was applied, hence $weight\_decay = 0$. Moreover, the initial exploration probability, epsilon and discounting factor, gamma were both set to $\varepsilon_{max} = \gamma = 0.99$, while the minimum exploration probability was $\varepsilon_{min} = 0.01$. And finally, the experience replay's buffer size was set to 100,000 so that the agent could have access to a very large set of past experiences which could help it to stabilise learning down the line.

Additionally, two Q-learning update functions were defined – one was used for the standard DQN, while the other one was utilised for the double DQN. Everything else was mainly kept

the same for both DQNs in order to make a fair comparison so that the improvement came down to the actual algorithm and not the parameters employed. The main difference between the two consists in how the agent selects the next action. The former chooses and evaluates the action which yields the maximum Q-value using the target network, while the latter utilises the local network to select the best action, but evaluates it, in the target network to get the maximum Q-value. What was just described is at the core of what differentiates a standard DQN from a double DQN.

The two models were trained accordingly, and their results were plotted and are reported in Figures 10 and 11. When observing these, bear in mind that their target score was set to 195.

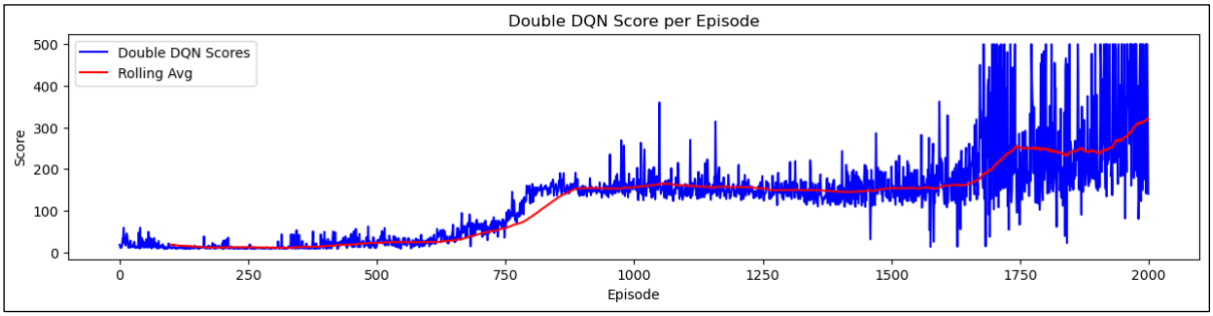

Figure 11. Line graph of the score per episode of a double DQN, with a rolling average.

2.1.4 Implementation of 2nd Improvement – Multi-step Learning with Double DQN

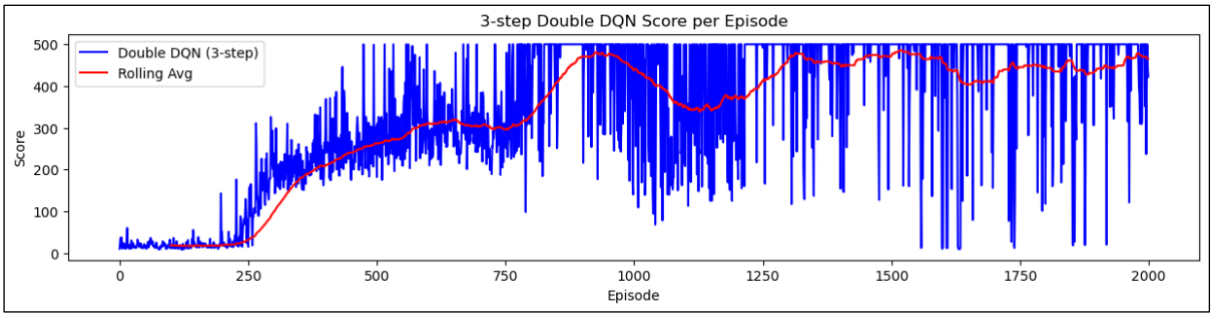

Figure 12. Line graph of the score per episode of a 3-step double DQN, with a rolling average.

The second improvement that was implemented was the multi-step double DQN, which is a variation of the double DQN model described in the section above. The biggest structural difference is the addition of the multi-step parameter, which states how many steps into the future are used to calculate the return and is employed both in the experience replay buffer and when computing the Q-value. Everything else, including all parameters in common between standard, double and multi-step double DQN were kept the same.

To gauge a better understanding of this difference, take a look at the following equations. Equation (2) describes both the standard and double DQN, while equation (3) defines what is happening in the multi-step double DQN.

$$Q_{target} = r + \gamma * Q(s', a') \tag{2}$$

$$Q_{target} = r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots + \gamma^{n-1} r_n + \gamma^n * Q(s_{n+1}, a_{n+1}) \tag{3}$$

Where r_i is the reward obtained after taking action a_i, γ is the discounting factor for future rewards, $Q(s_{n+1}, a_{n+1})$ is the value estimate of the next state and best next action and n represents the number of steps into the future that are going to be used to calculate return. The most significant difference between (2) and (3) is the fact that in the former, the Q_{target} is calculated using immediate reward, while in the latter, the agent combines multiple rewards to estimate the value of the next state.

The plot of the second improvement is shown in Figure 12. Again, the target score was fixed to 195 and the number of steps in the future were set to $n = 3$.

2.2 Analysis of Results

Figures 10, 11 and 12 display the performance of each of the three implemented DQNs. In particular, they show every DQN's score output per episode, with a score rolling average curve. The target score was set to 195 for all of the DQNs employed and it can be observed how the target is reached just before episode 1500 for the standard DQN, in the vicinity of episode 750 for the double DQN and right before episode 250 for the 3-step double DQN. Just this fact could be enough to state that the 3-step improvement is the best performing one, as the environment would be considered to be solved, but let's analyse them further and look at their rolling averages before and after they hit the target, for instance. The rolling average of the score in the standard DQN is flat for the great majority of the episodes and starts increasing when the target score is hit, stabilising at around the 200-score mark until the last episode. In the double DQN, it increases significantly at approximately episode 800, next it stabilises at around episode 1700 and finally increases again and hits the 300-score mark towards the end. Finally, the 3-step double DQN's score rolling average steadily grows between episode 250 and 750, at which point it spikes to its average maximum score, just under the 500-score mark, to then decrease, increase again and lastly stabilise in the vicinity of the 400-score mark until the final episode. Even if its performance was more unstable than the other two, the rolling average score never went below the 300-score mark after the first spike, therefore, it is clear that the best DQN is the last one that was implemented, i.e. the 3-step double DQN.

2.3 Application of RLLib Algorithm to Atari Environment Learning

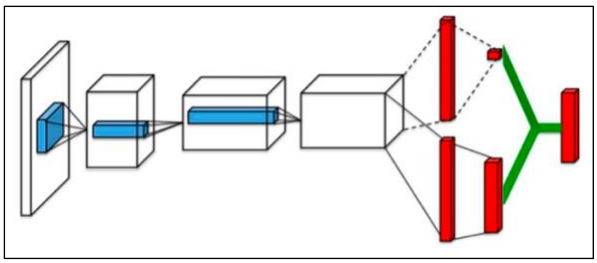

Figure 13. Dueling Q-network's double stream [11].

It was decided to apply the Dueling DQN algorithm to the Atari Learning Environment called Pong. When the Dueling DQN is implemented, computation is separated into two different streams. One produces a scalar, State Value Stream V(s), which estimates the value of state s, independently of the actions, while the other one produces a vector, Advantage Stream A(s,a), which estimates how much improved is action a, compared to the average action in state s [11]. In Figure 13, an example of this can be seen.

One of the benefits of the above-explained algorithm is that repetitiveness is significantly reduced in learning, since in states where there are a lot of similar actions, the network will focus on V(s) and hence avoid overfitting on A(s,a). Furthermore, the algorithm is going to favour learning for V(s) when actions have negligible impact, especially in environments that

have lots of actions. Additionally, the importance of critical states is going to be captured by V(s) independently of actions, which results in the critical states being learned faster. [11]

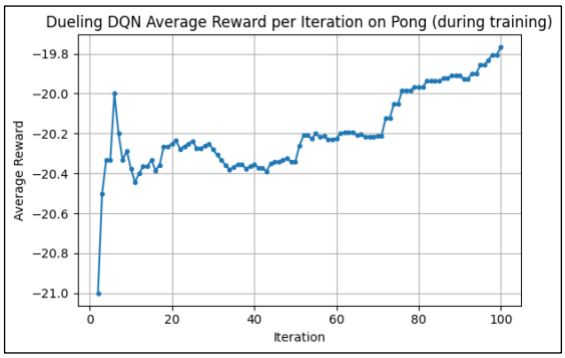

Figure 14. Dueling DQN average reward per training iteration.

The Dueling DQN was implemented from RLLib in the 'Pong-ram-v0' environment. The first step was to configure the Dueling DQN, so that the agent could be trained. In the configuration, the maximum number of steps per episode was set to 500 and the agent was trained over 100 iterations. The average reward per training iteration can be observed in Figure 14. Next, the agent was evaluated, still over 100 episodes and the net score between the agent and the opponent was tracked every 10 episodes, which can be seen in Figure 15, as net score and average reward per evaluated episode come out to be the same since for every point a value of +1 is assigned and for every point taken, the assigned value is -1.

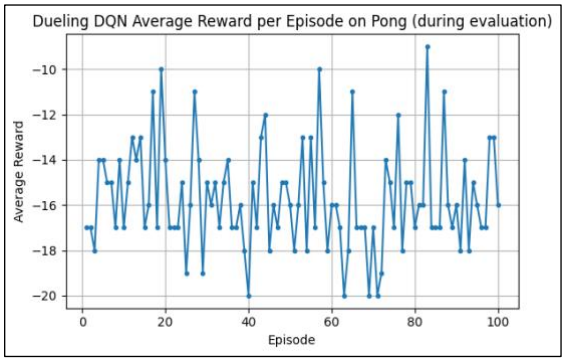

Figure 15. Dueling DQN average reward per evaluated episode.

2.4 Analysis of Results

Before analysing the above-reported results, it need be said that more parameters were tried to be employed in the Dueling DQN configuration for a better performance of the agent, such as the exploration configuration type, initial and final epsilon, and number of epsilon time steps. But this came out to be too computationally expensive and time consuming, therefore it was opted against it.

Moving on to Figure 14, it can be seen how the agent is slowly but steadily learning as the reward is increasing almost every game iteration. Probably, if more iterations were employed, by the end of training, the agent could've reached a positive average reward. On the contrary, in Figure 15, where the agent's performance is evaluated on 100 episodes, it can be observed that the reward is very unstable but still hits higher peaks than when in training mode, i.e. -9. The instability may be very well cause by the fact that the agent wasn't fully trained yet.

References

[1] OpenAI, *5x5 empty grid with labeled rows and columns*, ChatGPT, OpenAI, San Francisco, CA, USA, May 2025. [Online]. Available: https://chat.openai.com/.

[2] chest., "Closed Treasure Chest Images – Browse 18,882 Stock Photos, Vectors, and Video," *Adobe Stock*, 2025. https://stock.adobe.com/search?k=closed+treasure+chest (accessed May 07, 2025).

[3] "3D Animated Newt from Maze Runner in Pixar Style | AI Art Generator | Easy-Peasy.AI," *Easy-Peasy.AI*, 2024. https://easy-peasy.ai/ai-image-generator/images/3d-animation-newt-maze-runner-pixar-style (accessed May 07, 2025).

[4] in, "Trap Images – Browse 469,659 Stock Photos, Vectors, and Video," *Adobe Stock*, 2025. https://stock.adobe.com/uk/search?k=trap (accessed May 07, 2025).

[5] *Istockphoto.com*, 2024. https://www.istockphoto.com/photos/treasure-key (accessed May 07, 2025).

[6] E. Alonso, 2025. Lecture 3 – Reinforcement Learning The Solution. Lecture notes distributed in INM707 Deep Reinforcement Learning. 11 February 2025, City St. George's, University of London.

[7] E. Alonso, 2025. Lecture 2 – Reinforcement Learning The Problem. Lecture notes distributed in INM707 Deep Reinforcement Learning. 4 February 2025, City St. George's, University of London.

[8] D. Lamba, W. H. Hsu, and M. Alsadhan, "Chapter 1 - Predictive analytics and machine learning for medical informatics: A survey of tasks and techniques," *ScienceDirect*, Jan. 01, 2021. https://www.sciencedirect.com/science/article/abs/pii/B9780128217771000239.

[9] A. Tsantekidis, N. Passalis, and A. Tefas, "Deep reinforcement learning," Academic Press, 2022, pp. 117–129. doi: https://doi.org/10.1016/B978-0-32-385787-1.00011-7.

[10] "Gymnasium Documentation," *gymnasium.farama.org*. https://gymnasium.farama.org/environments/classic_control/cart_pole/.

[11] E. Alonso, 2025. Lecture 6 – Rainbow DQN. Lecture notes distributed in INM707 Deep Reinforcement Learning. 11 March 2025, City St. George's, University of London.