Q-Learning Algorithm and Dueling Double Deep Q Network Applied in Reinforcement Learning Tasks

Konstantinos Gkolias, Kimon Iliopoulos

konstantinos.gkolias@city.ac.uk, kimon.iliopoulos@city.ac.uk

Basic Part: Domain and task

Artificial Intelligence (AI), Deep Learning (DL) and Reinforcement Learning (RL), have been under the spotlight during the recent years, through sectors like autonomous driving (Shalev-Shwartz, Shammah and Shashua, 2016), real time strategy-games (Vinyals *et al*., 2019) and decision-making games (Silver *et al.*, 2017). A RL setting is composed of the environment, which is the state and action space, and at least one agent, who extrapolates its behavior through trial-and-error interactions, aiming to map the environments' states and actions that bear the most reward.

For the basic part of this project, a static deterministic environment was designed, grounded on OpenAI's "Frozen Lake" (OpenAI). The environment is a Gridworld consisting of tiles, mapped to a game setting. The agent always starts from a particular tile and has to navigate to the goal tile in the most efficient way, while dodging certain states (holes), which automatically terminate the game as the agent falls into them. Finally, negative or positive rewards are being given to the agent until it finds a path to the goal state. For the solution of this task, we used the Q-Learning algorithm (Watkins and Dayan, 1992).

State, Transition and Reward functions

The environment is a 4x4 grid, with states named alphabetically from A to P. The agent can move along the two-dimensional cartesian system coordinates "UP", "DOWN", "RIGHT" and "LEFT", until it reaches the goal-state 'P', where the environment is solved, or a terminal-state, where the agent loses. In both scenarios, the episode is completed and a new one is initialized with the agent placed at the start-state 'A'. Every time the agent falls into a Terminal-state it receives a reward of -100, while every time the agent reaches the goal-state it receives a reward of +100. For every other state transition, the agent receives a reward of -1, indicating the loss of energy. The environment is reckoned to be solved when the average cumulative reward of 100 sequential episodes is equal to, or greater than 83. This project purports to ascertain the policy for which the agent takes the smaller number of episodes to attain the aforementioned average reward over 100 episodes.

Policies

A Policy is a set of regulations that controls the agents' behavior. The actions that the agent chooses in each state are aftereffects of the agents' policy. More precisely, the policy π is defined as the probability distribution over all actions A_t for every possible state s.

$$\boldsymbol{\pi(a|s) = P[A_t = a|S_t = s]}$$

The **Exploration – Exploitation** tradeoff has bemused lots of researches in recent years. The idea behind it is to substantiate whether it is better to take the best already experienced solution at a certain state (**Exploitation**), or if it is worth trying something new that you do not know its outcome (**Exploration**). In the RL setting, it might be better for the agent to sacrifice a short-term reward to further explore the state space, aiming to descry states with a higher long-term reward. For the purposes of this project, we tested the Boltzmann policy, two variations of the Epsilon Greedy Policy, Epsilon Decay and Constant, and the Random policy.

Random and Epsilon-Greedy Constant policy

During the Random policy, the agent chooses random actions exclusively, without learning from its experiences, leading to a constant exploration of the state space. Epsilon-greedy constant policy, a simple yet effective policy for trivial tasks, takes the best available action (exploitation), in terms of short-term rewards, with a probability of a constant number *epsilon* ($0 \leq \varepsilon \leq 1$), while a random action (exploration) is selected with $1-\varepsilon$.

Epsilon-Greedy Decay policy

Epsilon-Greedy Decay policy is a more sophisticated variation of the epsilon-greedy method. That is, starting with an epsilon equal to 1 and linearly decaying over the episodes up to a lower bound after which it remains constant. This technique ensures that in the beginning random actions (**Exploration**) will be chosen more frequently than the highest rewarded ones (**Exploitation**). As the time goes by and the agent becomes more experienced and aware of the state-space, exploitation will emerge continually, and exploration will be limited.

Boltzmann exploration policy

Boltzmann exploration policy deals with the exploration-exploitation dilemma without the need of the epsilon. Instead of choosing the optimal (highest valued) action or a random action, an action is chosen based on weighted probabilities over the available actions. To accomplish that, it employs the SoftMax activation function over the guesstimate values of the actions. In this way, the best estimated action is more likely to be selected, albeit the other actions are not treated as equally probable to be chosen, but each of them has a unique probability that corresponds to its estimated value. Based on our findings, Boltzmann Policy demonstrated the best results among the aforementioned policies and is going to be harnessed for the purpose of our project.

Problem representation and Reward Matrix

The problem is being represented in *Figure 1*. More specifically, the fixed start-state is “A” (green), the goal-state is “P” (orange), “F”, “H”, “J” and “M” are the terminal states (red) and the rest are the neutral walkable states (white). The arrows represent the accessible transitions from every state. A total number of 4 actions are permitted: ‘UP’, ‘DOWN’, ‘RIGHT’, ‘LEFT’.

The reward function is created from the feedback an agent receives, by moving from one state to another and is defined by the following function:

$$\mathbf{R_{t+1} = R(s_t\, ,\, a_t\,)}$$

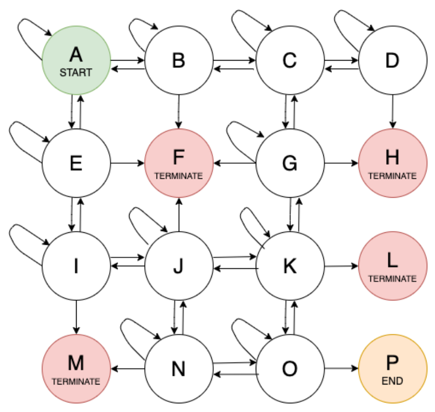


Figure 1. Environment representation Graph

{A}	→	{A, B, E}
{B}	→	{A, B, C, F}
{C}	→	{B, C, D, G}
{D}	→	{C, D, H}
{E}	→	{A, E, F, I}
{F}	→	{terminate}
{G}	→	{C, F, G, H, K}
{H}	→	{terminate}
{I}	→	{E, I, J, M}
{J}	→	{F, I, J, K, N}
{K}	→	{G, J, K, L, O}
{L}	→	{terminate}
{M}	→	{terminate}
{N}	→	{J, M, N, O}
{O}	→	{K, N, O, P}
{P}	→	{end}

Figure 2. Available Transitions per State

The reward matrix is a 16x16 table that defines the immediate reward that the agent receives for every available transition from one state to another. As we can see from *Table 1*, the agent receives -100 every time it falls into a red state (“F”, “H”, “L”, “M”), +100 for reaching the goal-state “P” and -1 for every other state transition, representing the energy usage that the agent necessitates to move around the grid. In this way, we encourage the agent to discover the shortest path to the goal-state “P”.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
A	-1	-1			-1											
B	-1	-1	-1			-100										
C		-1	-1	-1			-1									
D			-1	-1				-100								
E	-1				-1	-100			-1							
F						-1										
G			-1			-100	-1	-100			-1					
H								-1								
I					-1				-1	-1			-100			
J						-100			-1	-1	-1			-1		
K							-1			-1	-1	-100			-1	
L												-1				
M													-1			
N										-1			-100	-1	-1	
O											-1			-1	-1	100
P																🟥

Table 1: Reward Matrix

Parameter values for Q-Learning

The values for each parameter of the Q-Learning algorithm were chosen after careful consideration. Different combinations have been tried, which will be analyzed in the next sections.

Alpha (α) represents the learning rate, ranging between 0 and 1. When its value is set to 0, the agent is not learning anything because the rewards it receives for every action do not affect its behavior. When it is set to 1 then the agent will always choose certain actions for any given state, without any randomness at all. After several tests, the learning rate was set to 0.2.

Gamma (γ) is the discount factor of the future rewards ranging between 0 and 1. More precisely, it is the importance that the agent assigns to future rewards against immediate rewards. If gamma is set to 0 then the agent cares for the immediate reward exclusively and if set to 1 the agent cares for the future rewards more than the immediate ones. In our case, gamma was set to 0.99.

To deeply understand the aforementioned parameters and the Q-Learning algorithm itself, it is better to present the actual **Q-Learning update rule:**

$$\mathbf{Q_{new}(s_t,a_t) \leftarrow Q_{old}(s_t\,,a_t) + \alpha.[R(s_t,a_t) + \gamma.argmax_{\alpha}Q(s_{t+1},\,A) - Q_{old}(s_t,a_t)]}$$

Where s is the current state, s' is the next state, a is the action taken from state s, α is the learning rate, γ is the discount factor, A are the available actions from state s, R is the reward function and finally $\alpha.[R(s_t,a_t) + \gamma.argmax_a Q(s_{t+1}, A) - Q_{old}(s_t,a_t)]$ is the temporal difference error.

Q matrix updates

In this section we will describe the procedure of the updates that take place in the Q table and a potential first step in our environment:

1. **Initialize Q table** (Size = 16x4) with zeros
2. **The Agent selects an action** according to its policy: The Agent begins from state 'A' and selects the action "RIGHT"
3. **Action Execution**: The agent moves from state 'A' to state 'B' with action 'RIGHT'
4. **The Agent receives the reward** for the specific state given from the reward table: The Agent receives a negative reward of -1
5. **Update the Q table** based on the update rule stated above:
    - ✔ $\alpha = 0.2$, $\gamma = 0.99$, Q_{old}("A", "right") = 0, R("A", "right") = -1
    - ✔ Q_{new}("A", "right") = Q_{old}("A", "right") + 0.2.[-1 + 0.99.argmax$_{actions}${Q("B","up"), Q("B", "down"), Q("B", "right"),Q("B", "left")} – Q_{old}("A", "right")] = 0 + 0.2[-1 + 0.99.0] = **-0.2**

The aforementioned steps will be repeated for a minimum of 6 steps, until the agent reaches the goal-state "P".

Performance representation

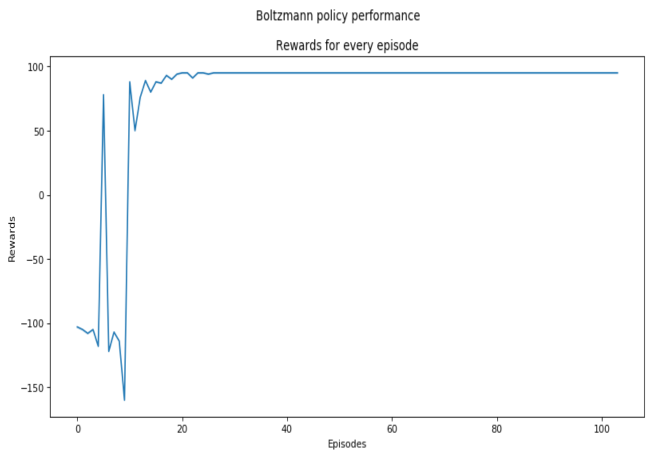


Figure 3. Agent performance with Boltzmann policy (Reward/Episode)

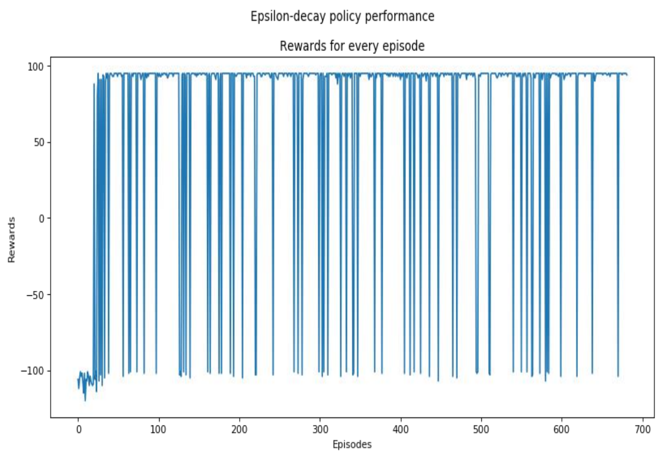
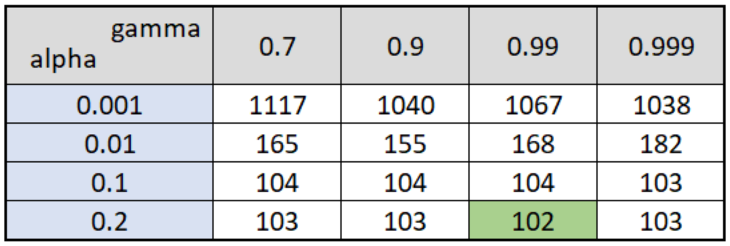


Figure 4. Agent performance with Epsilon Greedy Decay policy (Reward/Episode)

In the Figures presented above (3, 4), we can clearly observe the difference between the Learning Progress of the two most efficient policies; Boltzmann (Softmax) and Epsilon-Greedy-Decay Policy. More specifically, the former policy outperforms the latter, which is proven to be highly unstable until it converges after almost 700 Episodes. Boltzmann Policy is highly efficient for the purposes of our task, starting with low rewards which reach a minimum value of -150, but after less than 20 episodes the agent learns to find the target. Thus, after the initial Exploration stage, the Agent reaches high reward values (over 70), until it finally converges when the average reward function becomes greater than or equal to 83, during the Episode 102.

Different parameter values

Table 2 demonstrates the number of episodes required to solve the environment for different combinations of alpha and gamma, using the Boltzmann's policy. We observed that for smaller values of alpha, the efficiency of the agent was not as good as for larger ones. The optimal hyperparameter value combination was (alpha=0.2, gamma= 0.99), with 102 Episodes until convergence.



gamma / alpha	0.7	0.9	0.99	0.999
0.001	1117	1040	1067	1038
0.01	165	155	168	182
0.1	104	104	104	103
0.2	103	103	102	103

Table 2: Number of episodes required to solve the environment for alpha, gamma combinations of the Boltzmann exploration policy

Quantitative and qualitative analysis

In this section we will analyze the results of our experiment quantitatively and qualitatively. The former will break down the role of each parameter and discuss the antecedents for the agents' performance change for different parameter values. The latter will anatomize the two leading policies and argue about which performed better and why.

Quantitative Analysis

The policy to be used in a RL problem is the most decisive aspect of the task, as it defines which action will be chosen at each time-step by the agent. Several policies were implemented; Random, Epsilon Greedy, Epsilon Greedy Decay and Boltzmann policy. The first two were outscored by the last two, and thus will not be furtherly discussed. Epsilon decay required from frequently 600, to rarely 1200 episodes, for the same parameters, to solve the environment, while Boltzmann necessitated only 102 episodes.

We tested the agents' performance for 16 different combinations of learning rate and discount factor (*Table 2*). The learning rate has been proven to play a crucial role in the policy optimization process, using the Boltzmann exploration method. For low values of alpha (0.001, 0.01) the agent achieved the worst performance. Nevertheless, the increase of gamma for the given alpha value slightly improved the agents' performance. As the alpha was furtherly increased the agents' performance ameliorated dramatically, with minor effects from the changes in gamma, up to the point where the minimum of 102 episodes are required to solve the environment. This result is outstanding because, as it we can see from *Figure 3*, the agent manages to find the optimal path to the goal-state in less than 20 episodes.

Qualitative Analysis
Epsilon decay policy is widely used and although it is unadorned, it often achieves decent results. However, in our case it displayed fluctuating outcomes. High variance of its performance is probably generated by the randomness of the actions that the agent chooses in the early episodes of the training, where epsilon is close to 1. Thus, it is out of the author's control if the agent is going to select "better" or "worse" actions in the beginning. These controversial selections influence the agents' performance up to the final solution of the environment.

Boltzmann policy is more sophisticated than epsilon decay. It seems that the weighted probabilities assigned to each action available in every state, based on their estimated value, assist the agent to realize that there is no point of making an action that will force it to stay in the same state. This is achieved by assigning lower probabilities to suboptimal actions, meaning that they are treated individually. For every state there is one optimal action, and three suboptimal actions which can cause distinct negative results to the agents' performance. This property is ignored by the epsilon decay policy as it seems that the agent is often trapped to corners of the state space where there is almost 50% chance of selecting an action that will force it to stay put. This happens because it classifies the actions into two categories, one action will be the optimal, and the rest are considered suboptimal, each of which will have the same probability to be chosen. This is not effective as suboptimality can differentiate from action to action and this point seems to be effectively addressed by the Boltzmann method.

Advanced Part: Dueling Double Deep Q Networks (DDDQN)

Some tasks might often be too complex to represent state-action values through lookup tables. In these cases, we opt to use more powerful algorithms, that require the application of DL methods and tools, like Deep Q Network (Mnih *et al.*, 2015).

Deep Q Network (DQN)

DQN is using a Deep Neural Network (DNN) to act as a function approximator by passing as input a state-space or raw pixel data to the network which then produces an output vector of action values. Another crucial part of the learning progress of DQNs is the experience replay. It stores experience tuples of state, action, reward and next state values as the agent interacts with the environment and randomly samples a small batch of these tuples in order to learn, leading to an increase of data efficiency (Wang *et al.*, 2016). Sequential experience tuples can be highly correlated and by obtaining random batches from the experience buffer this correlation is reduced.

Deep Q Network limitation

However, even though DQN has been introduced as a solution to these problems, it yields some limitations. The temporal difference target in the update rule of the Q-learning algorithm is delivered by getting the best action from a given state. The accuracy of these Q-values depends on what actions have been tried and what neighboring states have been explored. This results in an overestimation of the Q-values since we always pick the maximum among a set of noisy numbers (Thrun and Schwartz, 1993). More specifically, this is a known adverse effect of Q-learning, since the same value network is being used to select and evaluate an action.

Double Deep Q Network

Double Deep Q Network (DDQN) (Van Hasselt, Guez and Silver, 2016) has been shown to work very well in practice by dealing with the overestimation bias. It consists of two networks; an online network which is utilized to select the best action using a set of parameters θ, and a target network which is employed to evaluate the selected action using a different set of parameters θ^-. In this way, we harness two separate function approximators that must agree on the best action (Van Hasselt, Guez and Silver, 2016). The new temporal difference target is ciphered from the equation below:

$$\mathbf{TDtarget_t^{DoubleDQN} = R_{t+1} + \gamma . Q(S_{t+1}, argmax_a Q(S_{t+1}, a; \theta_t), \theta_t^-)}$$

Dueling Deep Q Network

Dueling Deep Q Network (DDQN) (Wang *et al.*, 2016) is a powerful network architecture for model free reinforcement learning, which is based on the idea that the estimation of the value of each action choice for all states is not mandatory. The intuition behind this idea is that the values of some states do not vary across actions and so it is better to directly estimate the state-values (Wang *et al.*, 2016). However, we still need to acquire the difference that each action makes in each state. Two value estimations are being used for this purpose; the state-value “V”, which evaluates the efficiency of an agent being in a specific state and the advantage-value “A”, which showcases the importance of each possible action (Wang *et al.*, 2016). Finally, the Q-values are obtained through the following function:

$$\mathbf{Q^{\pi}(s, a) = A^{\pi}(s, a) + V^{\pi}(s)}$$

In Neural Networks this is being translated as a Dueling Architecture (*Figure 6*), during which two streams are being created; the first one produces a scalar (V) and the second one produces a vector (A), (Figure 5). However, the two streams share a common convolutional feature learning module and can be merged through an aggregating function, to produce an estimate of the state-action value (Q) (Wang et al. 2016). As a result, the network creates independent estimates of the state value function, one per action, with no supervision (Wang et al. 2016).

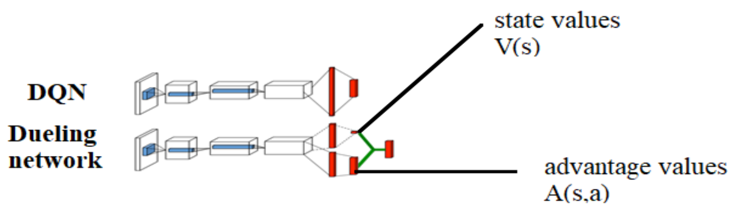


Figure 5. Difference between DQN and DDQN architectures | Source: Wang et al., (2016)

Dueling Double Deep Q Network

For the advanced part of this project, it was decided to apply the DQN algorithm boosted with its two aforementioned extensions, resulting to the Dueling Double Deep Q Network (DDDQN). DDDQN algorithm was applied in the OpenAI's "LunarLander-v2" (OpenAI) environment where the goal is to land the lunar to the landing pad. When the lunar is moving away of the landing pad the agent receives a negative reward and the episode ends only if it either crashes, where the agent receives -100 reward, or it comes to rest where the agent receives +100 reward. When each leg ground to contact the agents receives +10 points and for firing the main engine is -0.3 points for each frame. Four discrete actions are available: (1) do nothing, (2) fire main engine, (3) fire right engine and (4) fire left engine. The environment is considered solved when the mean cumulative reward of the last 100 episodes is equal or greater than 200 (OpenAI).

As for the DDDQN's architecture we omitted the convolutional layers that the prototype DQN algorithm in Mnih *et al.*, (2015) includes in its setting, as there is no need for image preprocessing for our environment. As a result, we harness a feedforward network with backpropagation which is classified into two parts. The first part is formed of three fully connected layers, of 256, 128 and 128 units respectively. In the second part, the network is split into two streams, one for the state values (V) and one for the advantage values (A) which both are comprised of two fully connected layers. The first layer of the state value stream has 128 nodes and the output layer has one node, indicating the unique value of the given state. The advantage stream has one layer with 128 nodes and an output layer of 4 nodes, equal to the number of possible actions in each state. Afterwards, the two output layers of each stream are aggregated resulting to 4 Q-values. After each layer, a Rectified Linear Unit (ReLU) activation function is applied and the parameters of alpha and gamma are set to 0.0005 and 0.996 accordingly. The optimization algorithm used for the adjustment of the network's parameters is the Adam optimizer, that backpropagates the network's obtained Mean Squared Error (MSE), which has proven to achieve significant results while, at the same time, it is computationally efficient and necessitates little memory requirements (Kingma and Ba, 2015). The target network's parameters (θ^-) are updated every 4 time-steps using the soft update function, a method introduced by Lillicrap *et al.*, (2016), with a value of tau equal to 0.01. Lastly, the experience replay buffer has a total memory size of 1,000,000 tuples, the batch size is 64 experience tuples and is being randomly sampled.

Quantitative analysis

As we can see from *Table 3*, the results of the conducted experiments were different each time we ran the python script. This is an aftereffect of the partial randomness that characterizes our model, which we will analyze in the next section. In order to represent unbiased results, we decided to run ten experiments using the same parameter values and calculate the average number of episodes and elapsed time obtained. The following table (*Table 3*) presents the results of these experiments.

Experiment	Number of Episodes	Average Payoff Convergence (Final Loop)	Elapsed Time (min)
1	335	200.34	8.04
2	331	203.22	7.15
3	463	202.8	11.68
4	289	204	6.36
5	272	200.14	6.52
6	528	200.86	14.87
7	536	200.52	12.33
8	574	200.45	19.92
9	323	203.69	11.04
10	337	201.66	10.86
Average Number of Episodes	399		
Average Elapsed Time (min)	10.88		

Table 3. Experiments Results

From *Table 3*, we observe that the Average Number of Episodes needed by the agent to reach convergence is 399 episodes and the Average Elapsed Time is 10.88 minutes. More specifically, the minimum number of episodes was recorded for the fifth (5th) experiment and was 272 episodes, but interestingly the shortest time has been recorded during the fourth (4th) experiment, where more episodes (289) were needed to reach convergence. Moreover, the maximum number of episodes has been recorded during the eighth (8th) experiment with 574 episodes and 19.92 minutes. We are confident to state that the agent solves the environment relatively fast.

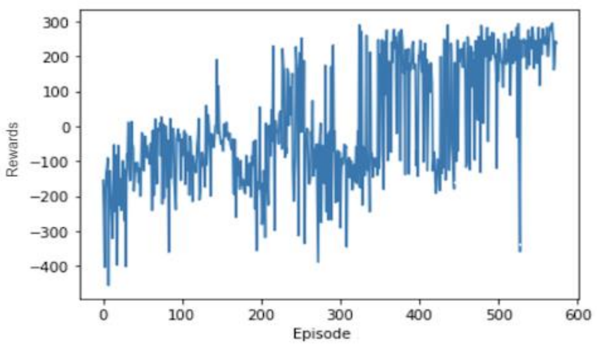


Figure 6. Least efficient experiment (8th)

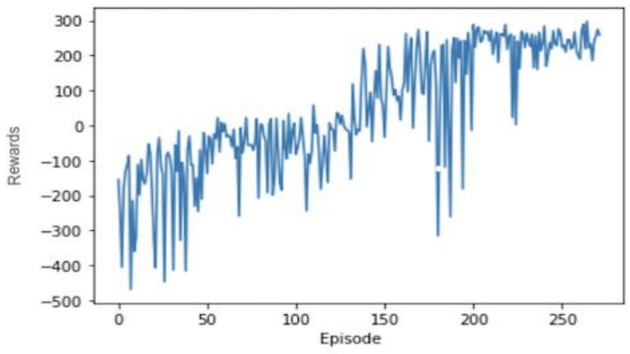


Figure 7. Most efficient experiment (5th)

Figure 6 (8th experiment) and *Figure 7* (5th experiment) show the Learning Progress of the least efficient experiment and the most efficient experiment, respectively. We can clearly detect the severe imbalance in the Rewards, during the 8th experiment, which causes the average score to drop. This effect can especially be observed during the 300th episode, where a dropping tense to the learning progress can be spotted. Additionally, the effects of the epsilon greedy decay policy are witnessed, as the agent explores in the first episodes, where the rewards have negative values ranging from -400 to 0, and then as the epsilon value decreases, exploitation takes action. Along with the exploitation phase, the agent starts receiving higher cumulative reward per episode, that gradually lead to the final solution of the environment (Reward up to 300).

Qualitative analysis

In this section we will explain the agent's behavior, that leads to the results displayed and analyzed in the previous section. The variance in both time and episodes required for the agent to solve the environment was something that initially puzzled the authors. After researching and discussing the obtained results, we concluded that the observed variance is prompted to the randomness that occurs in three key factors of our model. At this point it is important to say that it was decided to not display the results for different parameter values, as in the basic task, because we would not be able to argue impartially on them due to the influence of randomness in the agents' performance.

Firstly, each network's parameters are initialized randomly. This leads some experiments to begin with more suitable weights having less MSE loss than others and so minimize it and eventually solve the environment faster. Secondly, by using the epsilon-greedy decay policy, the explorative actions of the agent are chosen randomly. Thus, the agent's decisions during exploration influence the rest of the training, as it can either visit more "good" states, that are going to assist it along its journey, or more "bad" states that are going to sabotage its learning and trap it to local maxima. Fortunately, after a significant number of trials, we can say that no matter how "bad" the initially explored states are, the agent will eventually find the solution. Thirdly, the Replay Buffer plays a crucial role to this behavior as the experience tuples are randomly sampled. This factor is strongly connected to the epsilon-greedy decay policy. Let us assume that the agent, during the exploration phase, visits lots of "bad" states. This means that the model will store the "bad" experienced tuples in the Replay Buffer and when the sampling stage starts these "bad" tuples are going to have the same probability to be selected as the "good" ones. In this way, the agent will be taking actions based on "bad" choices that made in the beginning of its training. This problem can be addressed with a different sampling method of the Experienced Buffer that was introduced in Schaul *et al.*, (2016) where the agent prioritizes the experiences that returned high reward to be selected more frequently than the ones that did not. In order to limit the randomness in our project, we set a specific seed every time we initialized the environment, that helped stabilize our results more. Finally, another reason that could also cause this problem is the fact that ReLU units are used within our network architecture, which are vulnerable to collapsing, when reaching the regime of zero outputs (Lu *et al.*, 2019).

All in all, our model displayed decent results and we consider some of its aspects to be crucial to this performance. By using a Double DQN temporal difference target prevents the overestimation of the Q-values and Dueling networks helps getting rid of states that no matter what action the agent takes, they lead to a low reward. Lastly, we initially set the buffer size to 10,000 experience tuples, which proved to be small. When we set it to 1,000,000 the agent's performance was improved as it could randomly sample through a larger amount of data, which was more representative. In conclusion, these factors are driving our agent to convergence faster.

References

Van Hasselt, H., Guez, A. and Silver, D. (2016) ‘Deep reinforcement learning with double Q-Learning’, in *30th AAAI Conference on Artificial Intelligence, AAAI 2016*.

Kingma, D. P. and Ba, J. L. (2015) ‘Adam: A method for stochastic optimization’, in *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*.

Lillicrap, T. P. *et al.* (2016) ‘Continuous control with deep reinforcement learning’, *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*.

Lu, L. *et al.* (2019) ‘Dying ReLU and Initialization: Theory and Numerical Examples’, 107, pp. 1–32. Available at: http://arxiv.org/abs/1903.06733.

Mnih, V. *et al.* (2015) ‘Human-level control through deep reinforcement learning’, *Nature*. doi: 10.1038/nature14236.

OpenAI (no date a) *FrozenLake-v0*. Available at: https://gym.openai.com/envs/FrozenLake-v0/.

OpenAI (no date b) *LunarLander-v2*. Available at: https://gym.openai.com/envs/LunarLander-v2/.

Schaul, T. *et al.* (2016) ‘Prioritized experience replay’, in *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*.

Shalev-Shwartz, S., Shammah, S. and Shashua, A. (2016) ‘Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving’, *arXiv preprint arXiv:1610.03295v1*.

Silver, D. *et al.* (2017) ‘Mastering the game of Go without human knowledge’, *Nature*. doi: 10.1038/nature24270.

Thrun, S. and Schwartz, A. (1993) ‘Issues in Using Function Approximation for Reinforcement Learning’, *Proceedings of the 4th Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*.

Vinyals, Oriol and Babuschkin, Igor and Chung, Junyoung and Mathieu, Michael and Jaderberg, Max and Czarnecki, Wojtek and Dudzik, Andrew and Huang, Aja and Georgiev, Petko and Powell, Richard and Ewalds, Timo and Horgan, Dan and Kroiss, Manuel and Danihel, D. (2019) *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. Available at: https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii.

Wang, Z. *et al.* (2016) ‘Dueling Network Architectures for Deep Reinforcement Learning’, in *33rd International Conference on Machine Learning, ICML 2016*.

Watkins, C. J. C. H. and Dayan, P. (1992) ‘Q-learning’, *Machine Learning*. doi: 10.1007/bf00992698.

Personal Individual Contribution

Our cooperation was very efficient, as this is the second time that we have worked together for a project during the MSc Data Science. In person meetings were organized every week from the beginning of this coursework, which after the COVID-19 measures were replaced from frequent Skype calls, to strategically organize our plan and goals.

Basic Part

Both members of the team, me, Kimon Iliopoulos, and Konstantinos Gkolias, contributed equally for the Basic Part of the project. We both worked for the implementation of the policies, Boltzmann, Epsilon Greedy Decay and Constant, and Random as well as the general context of the code. I participated in all the tasks of the basic part by creating the environment, researching the literature about the implemented methods, writing code for the model, testing different parameter values and writing the submitted report.

The report was split into two parts, that we were working on at the same time in order to save time while additionally it was considered more effective in terms of quality of writing. Then each member gave feedback and made corrections to the other member’s part that helped finding mistakes faster. After we ensured that they are valid we merged them and created the basic part as it can be seen in this report.

Advanced Part

Following the same pattern, the advanced part was conducted with high cooperation performance that lead to a relatively fast-convergent environment for the agent using the DDDQN algorithm. Similarly, this part demanded significant research to be made both for the environment to be used and the modelling phase. We both contributed equally to the implementation and test of the code, and to the experiments needed to the final arrangement of the best parameters to be used. Having two people run experiments simultaneously lead to an apace discovery of robust hyperparameter values that they achieved a relatively fast convergence of the agent in the LunarLander-v2 Environment.

After completing the code, we implemented several experiments to reassure that the results we were getting, were stable enough. Again, both of us contributed on that as we reported through continuous updates the results we got. Moreover, 10 experiments were decided to be conducted and documented which helped us calculate the averages of the Episodes to Convergence and Elapsed Times, for the purposes of a more robust analysis. This phase was split, as one member run the experiments and the other created the graphs and the tables used in the report. Additionally, the architecture of the network was chosen after discussing the needs of our problem and the computational time required to be solved with an additional empirical study by documenting each architecture’s results.

To sum up, the project was fulfilled with high cooperation between the members, helping each other when one got stuck and was split, in some cases, to equal difficulty parts in order to save time when we needed to.