Reinforcement Learning- Final Project

Zuriya Ansbacher 208532515 and Akiva Bruno Melka 332629393

Submitted as final project report, BIU, Semester A, 2022

1 Introduction

Machine learning approaches are traditionally divided into three broad categories, depending on the nature of the "signal" or "feedback" available to the learning system: 1) Supervised learning, where the computer is presented with example inputs and their desired outputs given by a "teacher" and the goal is to learn a general rule that maps inputs to outputs, 2) Unsupervised learning, where no labels are given to the learning algorithm, leaving it on its own to find structure in its input, and finally, 3) Reinforcement learning (RL), where a computer program interacts with a dynamic environment in which it must perform a certain task (such as driving a vehicle or playing a game against an opponent). As it navigates its problem space, the program is provided feedback that's analogous to rewards, which it tries to maximize (1). Applications of reinforcement learning were in the past limited by weak computer architectures. Nowadays the rise of computational technologies is opening the way to completely new inspiring applications. For instance, autonomous cars are utilizing RL to acquire the optimal driving policy which will enhance customer experience and reduce risks. Moreover, reinforcement learning is widely used in the gaming industry, recent works were able to solve more than hundreds of Atari games (2; 6) and the research in that field is constantly evolving.

Reinforcement learning is intuitively the closest to the way people learn tasks. Indeed, one does not always have labels or a teacher to correct or point in the right direction. Nevertheless, one's experience or even experience from others significantly improves the learning process. The two major components of RL are the agent and the environment. The environment determines the world in which the agent evolves with a set of states, actions, and rewards. The execution of an action by the agent given a current state will lead (deterministically or stochastically) to a new state and a reward. The agent tries to find the chain of actions that will grant the highest reward or simply accomplish the task required by the environment. In order to do that, the agent goes through a training process with trial and error.

Ideally, the agent wants to learn how to achieve the goal in a minimum number of trials, especially if those trials are "expensive". Several challenges arise when determining the best course of action. For instance, always taking the best next action might not be efficient in the long term. Only Looking at the last actions performed might also induce bias. Therefore, several algorithms have been developed to palliate those caveats. Two different types of problems are mostly encountered: model-based and model-free. We will

focus on solving model-free problems. Within this framework, three main approaches can be used: Q-learning which consists in computing the value of action in order to determine the next step, Policy optimization, and a combination of both.

1.1 Problems to solve

In this project, we solve the continuous Lunar lander and the Bipedal walker hardcore problems and compare the performance of different algorithms. Those two environments were developed by OpenAI Gym (4).

1.1.1 LunarLanderContinuous-v2

The main objective of the agent is to land the spaceship between two flags located on the moon's surface as fast and as smooth as possible (Fig. 1 (a)). The ship has 3 throttles in it, one throttle points downward and the other two are pointing to the left and right directions. With the help of these, the agent is expected to control the spaceship. A typical observation vector looks as follows: [Position X, Position Y, Velocity X, Velocity Y, Angle, Angular Velocity, Is left leg touching the ground, Is right leg touching the ground].

The Action vector is composed of two floats: [main engine, left-right engines]. The Reward for moving from the top of the screen to the landing pad with zero speed is around 100-140 points. If the lander moves away from the landing pad it loses reward back. The episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing the main engine costs -0.3 points in each frame. The game is considered solved if the agent achieves a score of 200 in 100 consecutive games.

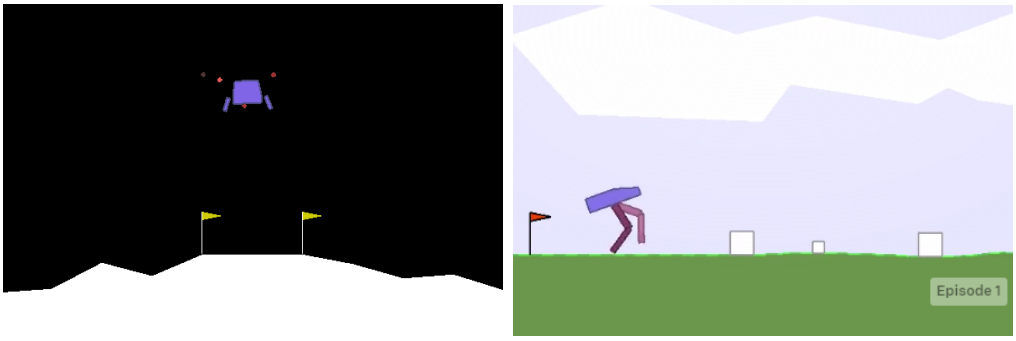
1.1.2 BipedalWalkerHardcore-v2

The goal is to train a 2D bipedal walker to walk through rough and to move the robot forward as much as possible (Fig. 1 (b)). the hardcore version contains ladders, stumps, and pitfalls in terrain. Terrain is randomly generated at the beginning of each episode. In the hardcore version, time limit is increased due to obstacles. Reward is given for moving forward, total +300 points up to the far end. If the robot falls, it gets -100. Applying motor torque costs a small amount of points, more optimal agent will get better score.

State consists of hull angle speed, angular velocity, horizontal speed, vertical speed, position of joints and joints angular speed, legs contact with ground, and 10 lidar rangefinder measurements. There's no coordinates in the state vector. In the action space, the robot has two legs with two joints at knee and hip. Torque is provided to knee and pelvis joints of both legs. The game is considered solved if the agent achieves a score of 300 in 100 consecutive games.

1.2 Related Works

The most basic designs of RL models rely on Tabular methods, whether the problem is model-based or model-free. Those methods are expensive in terms of computations, especially if



(a) Lunar lander (b) Bipedal Walker - Hardcore

Figure 1

there are many states and actions. The convergence time can also be quite long. With the emergence of strong computational tools and machine learning techniques, new algorithms were developed using Q-learning (QL) such as DQN and its derivatives or actor-critic. They rely on real samples from the environment and never use generated predictions of the next state and next reward to alter behavior (although they might sample from experience memory reply).

1.2.1 Deep Q-Learning (DQN) and its variants (DDQN nd D3QN)

DQN approximates a state-value function $Q(s, a)$ in a QL framework with a neural network. Instead of keeping the possible states and predicted Q-values in a table, the neural network takes as input the current state of the environment and outputs the predicted Q-values for each possible action (5).

The original implementation of the DQN consisted of a single neural network in which both the Q-values and the target-values shift during the Q-learning update, leading to an unstable model struggling to converge. Therefore, a well-known improvement consists in using a "frozen" target network, which is a clone of the standard policy network and is used to calculate $Q(S_{t+1}; a)$. During the update step (i.e. every k steps, k being a hyper-parameter) the target network is cloned once again and the weights are refreshed. This makes the training more stable by preventing short-term oscillations from a moving target (6).

Apart from the oscillation issue, another drawback is that DQN chooses the action with the highest Q-value for the next state. In most cases, this leads to an overestimation and, as a result, to an unstable model or a slow convergence. To overcome this issue, Double DQN (**DDQN**) makes a double estimation: the greedy policy for the next states is evaluated using the online (standard) policy network and the target network is used to estimate its value. Then, these two estimations are used to compute the final Q-value (7).

For many states, the estimation of the value for each action is superfluous. Therefore, the advantage function was introduced: $A(S_t; a_t) = Q(S_t; a_t) - V(S_t)$ in the Dueling DQN (**D3QN**). By subtracting the value function from the Q-function, the advantage function obtains a measure of the relative importance of each action. Practically, it is done by the neural network dueling architecture. Two streams separately estimate state value and the

advantages for each action. They share a learning module and are combined via a special aggregating layer (8).

1.2.2 Deep Deterministic Policy Gradient algorithm (DDPG)

Another approach in the model-free field is policy optimization. The most recent works combine both the Q-learning approach and policy optimization approach. **DDPG** (9) assumes continuous action space and is an actor-critic algorithm based on deterministic policy gradient. Both actor and critic have their own target and policy networks, and they are trained separately with two different optimizers. As such, DDPG is an expansion of DQN to continuous environments.

We also implemented an advanced version of DDPG: FORward-looKing Actor for model-free (FORK) (11). It is a new training algorithm, based on the actor-critic structure that improves the policy by considering not only the action-value of the current state (or states of the current mini-batch) but also future states and action forecasts using a learned system model and a learned reward model. We resorted to this model for the Bipedal Walker Hardcore since it required a stronger model.

2 Solution

2.1 General approach

We here present solutions to both problems described above. As mentioned, this implies getting an average of 200 for the Lunar Lander and 300 for the Bipedal Walker over 100 consecutive trials. In order to achieve that goal, we built agents using the different algorithms described above and trained them on several episodes until the completion of the tasks. We recorded the best parameters and models weights for each agent.

We applied a bottom-up approach, implementing first the most basic algorithm and slightly improving it up to the most sophisticated one. As mentioned above, Tabular methods proved way too fastidious and expensive in terms of computation, especially in a continuous environment. Therefore, we started with DQN. For the reasons we already described above, DQN can oscillate and overestimate the Q-value. As a result, its convergence might be fuzzy. Hence, we also implemented DDQN and D3QN. Since the action space in the two problems is continuous and DQN class algorithms only receive a finite number of actions, we had to discretize the action space as will be further discussed. Finally, we implemented DDPG since it works with a continuous action space and is, as such, more general. Its convergence is also less sensitive to parameters.

Beyond solving the problems, we kept track of the number of episodes required by each algorithm and determined which performed best. We also observed their sensitivity to the parameters.

2.2 Design

2.2.1 General features

We implemented our code in python using the pytorch package. We implemented a super Agent class that is called upon by the learning program. Each of the algorithms we implemented derives from this class except for the DDPG, which requires more parameters and, therefore has its own agent class.

Apart from the general features, the super Agent class receives all the parameters and computes the train function with a loop over the episodes. We tracked the average reward over the last 100 episodes. If at any point this average reaches 210 (we increased the goal to ensure convergence even during the test), we simply run 100 episodes without learning to test whether we indeed completed the task and achieve an overall average reward of 200.

The train function calls the learn function, in which, for each state, target, prediction, and loss are computed. The learn function also performs the backward propagation. Each inherited class has its own target computation with respect to the architecture of the algorithm as further described.

2.2.2 Fully observable

In the training process, we calculated Velocity X and Velocity Y by computing the differential between the Position X and Position Y values of the current state and the next state obtained from the environment. After a few experiments, we received estimated constants that give an approximation of the real velocities.

2.2.3 Discrete action space

While DDPG can handle a continuous action space, DQN type algorithms require a discrete action space. For the Lunar lander, the action space is of dimension 2 (main engine and left-right engine) with values between -1 and 1. We, therefore, split each dimension into 5 intervals for a total of 25 actions. For the Bipedal Walker, we only implemented DDPG and, hence, did not require discretization.

2.2.4 Agents architecture

Although basic DQN technically only requires one network, we implemented the Agent class for all DQN type algorithms with two networks (target and policy). For the basic DQN, the update frequency would simply be 1. DQN and DDQN networks have 2 layers. The number of atoms in each layer is obtained as a hyper-parameter. The D3QN networks have an additional layer common to the valuation and the advantage functions. DDPG has 4 networks, target, and policy (similar to the DQN networks), two for the actor and two for the critic.

2.2.5 ϵ -greedy policy

During the testing phase, the policy is to produce the action yielding the highest reward. However, during the training phase, one wants the agent to explore all possibilities at the

beginning, and then, as the decision process becomes more and more astute, the agent will tend to choose the most efficient action. To accomplish such a learning curve we used an ϵ-greedy policy for exploration with an exponential decay over time down to a minimum value.

2.2.6 Memory replay

To alleviate the problems of correlated data and non-stationary distributions, we use an experience replay mechanism that randomly samples previous transitions, and thereby smooths the training distribution over many past behaviors. Experience replay means that one records the steps of each episode into an experience buffer, which consists of the current state, current action, reward, next state, and whether we are done or not. Then, the samples are drawn randomly from this replay memory. This demonstrates the balance between the exploration and the exploitation and is used for all algorithms (10).

2.2.7 hyper-parameters optimization - NNI

All algorithms require hyper-parameters. Some of them we fixed such as the memory replay buffer size (10000), the maximum (1) and minimum (0.01) ϵ value, or the maximum number of episodes during the training (800).

Other parameters such as the time decay γ, the number of atoms in the network layers, ϵ decay, the learning rate, the target update, or the learning frequency can be optimized to achieve better convergence. We, therefore, used Microsoft NNI for the tuning of those parameters. We obtained parameters through the NNI by optimizing the minimum number of episodes required to solve the environment (see Fig. 2).

After we received the best parameters combinations from the NNI, we run them again on the models in order to ensure stability.

3 Experimental results

The best hyper-parameters obtained by NNI tuning along with the minimum number of episodes until convergence (i.e. training learned sufficiently good model weights for the testing to achieve an average above the goal over 100 consecutive episodes) are displayed in Table 1. As discussed further, models are relatively sensitive to hyper-parameters. Nevertheless, the parameters we fixed such as the memory size or the maximum number of episodes did not influence too much. Even the discretization size was not of major significance.

Fig. 3 and 4 show the learning processes and evaluation phases for the Lunar Lander and the Bipedal Walker respectively. The left column corresponds to the training process. In each graph, the blue line corresponds to the score in each episode and the yellow line corresponds to the average over the last 100 episodes (after more than 100 episodes and all episodes before). The right column corresponds to the testing phase with the score over 100 episodes. Each line corresponds to a different model.

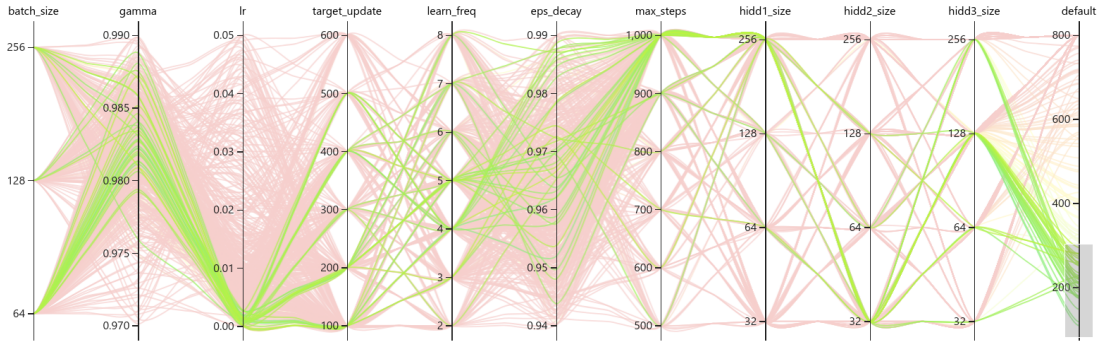


Figure 2: Parameters tuning by NNI. The green routes indicate the best parameters combinations which lead to the fastest convergence.

	Lunar Lander Continuous				**Bipedal Walker Hardcore**
	DQN	**DDQN**	**D3QN**	**DDPG**	**DDPG FORK**
batch size	256	128	64	256	100
gamma	0.9791	0.9856	0.9828	0.9849	0.99
learning rate	0.0015	0.0020	0.0019	Actor: 0.0021 Critic: 0.0032	0.0003
target update	400	100	100	10	-
learn freq	3	8	4	-	2
epsilon decay	0.9590	0.9826	0.9519	0.0008	-
hidden size layers	256, 64	64, 64	256, 32, 128	Actor: 256, 128 Critic: 128, 128	400, 300
max steps	900	500	1000	800	3000
tau	-	-	-	0.0087	0.02
Episodes until convergence	**158**	**320**	**107**	**124**	**4109**

Table 1: Description of the parameters and performance obtained with all implemented algorithms.

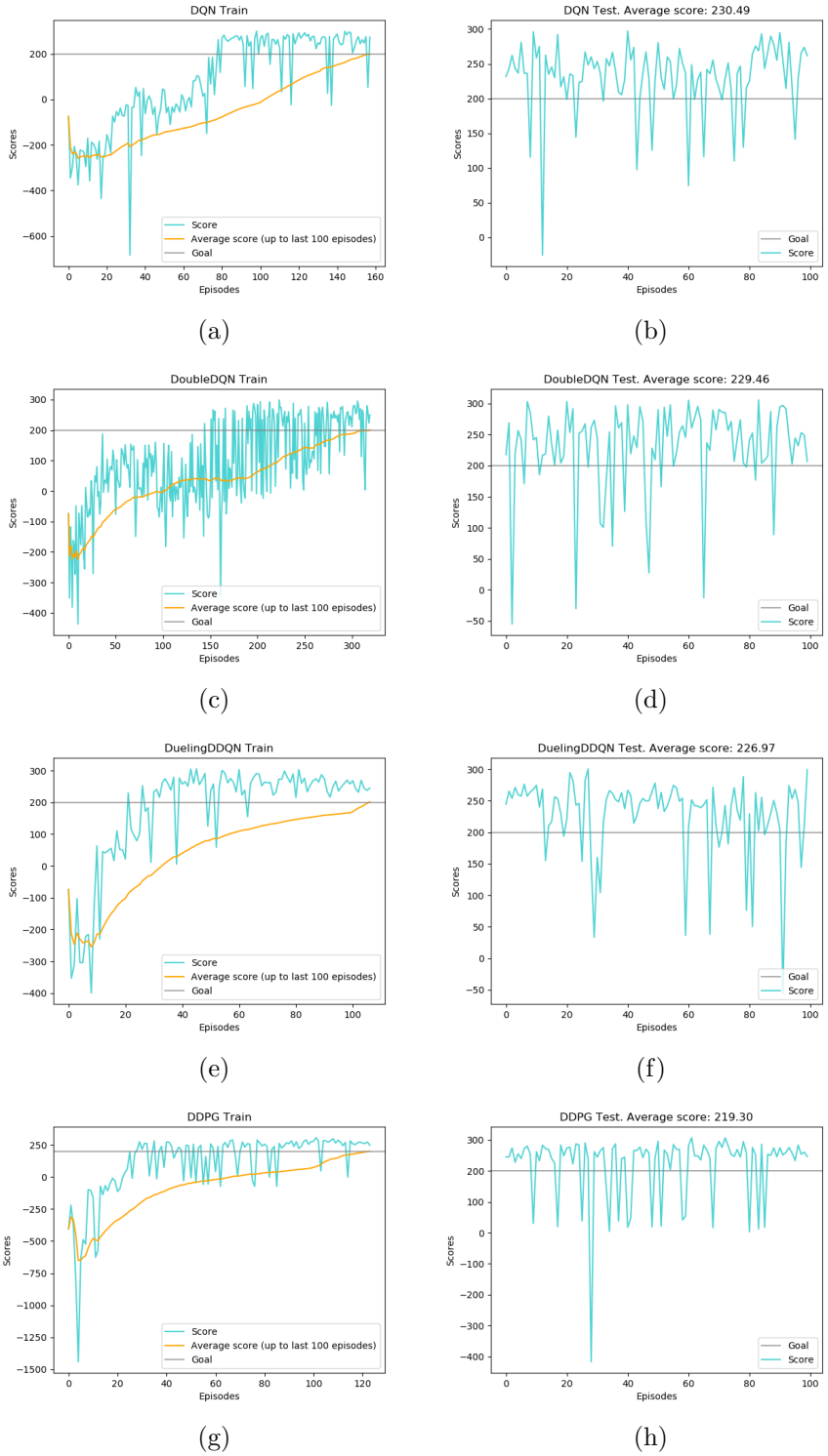
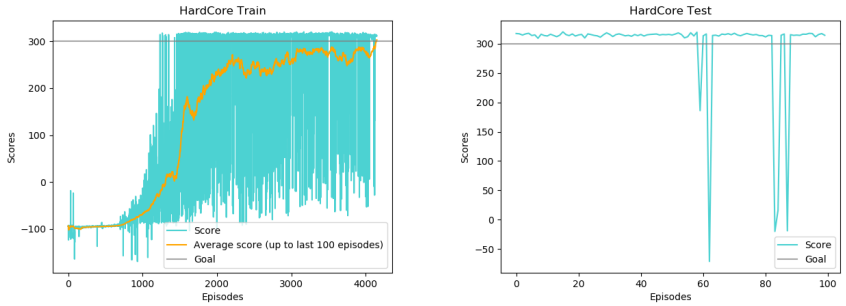


Figure 3: Lunar Lander results. The left column corresponds to the training process. In each graph, the blue line corresponds to the score in each episode and the yellow line to the average over the last 100 episodes (after more than 100 episodes and all episodes before). The right column corresponds to the testing phase with the score over 100 episodes. The average is displayed above each graph. Each line corresponds to a different model (DQN, DDQN, D3QN, and DDPG from top to bottom.



(a) (b)

Figure 4: Bipedal Walker Hardcore results. The left plot corresponds to the training process. The blue line corresponds to the score in each episode and the yellow line to the average over the last 100 episodes. The right plot corresponds to the testing phase with the score over 100 episodes. The model used was FORK.

Although all models have a rather noisy training process, DDQN has the noisiest one for the Lunar Lander, even noisier than DQN. The number of episodes until convergence is also higher than the other models for DDQN. Apart from DDQN which required more than 300 episodes, all DQN variants and DDPG achieved convergence between 100 and 200 episodes. DQN performed surprisingly well. We infer that the good performance of DQN, similar to other models, is due to the simplicity of the action space (only two dimensions) and the fact that we optimized the parameters through NNI. Nevertheless, we observed that even the slightest tweak in the parameters could seriously compromise the convergence making the model unstable and extremely sensitive to parameters. DDQN and more sophisticated models were, however, less sensitive to parameters. To ensure the stability of our parameters, we choose some of the best sets of parameters obtained by NNI and trained our models with those parameters, ensuring that they were indeed optimal.

The FORK model is by far the noisiest. This is obviously due to the extremely high complexity of the Bipedal Walker Hardcore game. We only achieved convergence after more than 4000 episodes (more than the 3000 episode limit imposed by the assignment).

4 Discussion

In this project, we implemented different reinforcement models in order to solve gaming environments such as the Lunar Lander Continuous and the Bipedal Walker Hardcore tasks. We used varied methods: from the different variants of Deep Q-Learning (DQN, DDQN, D3QN) to the Deep Deterministic Policy Gradient algorithm (DDPG), and its advanced version: DDPG FORK.

We succeeded to achieve convergence with all models. Against common beliefs, simple models also performed well for the Lunar Lander problem. We explain this by the relative simplicity of the action space. We, indeed, guess that in more advanced environments, the gap would be more significant. The Bipedal Walker Hardcore is indeed a good example of that claim since we resorted to a much more advanced model that accounts for future states and actions to solve it.

While DDPG is more advanced than D3QN since it solves continuous action space, D3QN performed better. It seems that, whenever achievable, a discrete action space results in a better convergence. Parameters tuning was also critical and there were quite different behaviors and convergence rates between models. Since we computed the hyper-parameters through NNI, the number of episodes required for convergence is optimal and quite low, which also explains the small differences between models.

Through this project, we were able to implement sophisticated methods of Reinforcement learning. We observed how powerful those tools are and how they can solve tricky problems. We were also able to gauge the issues and difficulties at hand.

5 Code

The code is available at the following links:

Link to GitHub
Link to a short video on YouTube. (Train and then Test of Lunar Lander)
Link to colab netbook 1 (DQN, DDQN, D3QN)
Link to colab netbook 2 (DDPG)
Link to colab netbook 3 (DDPG FORK)

References

[1] Bishop, Christopher M and Nasrabadi, Nasser M, Pattern recognition and machine learning, **4** (2006)

[2] Kaiser, Lukasz et. al, Model-based reinforcement learning for atari, arXiv preprint arXiv:1903.00374, (2019)

[3] Mnih, Volodymyr et. al, Human-level control through deep reinforcement learning, Nature **518** 529 (2015)

[4] Brockman, Greg et. al, Openai gym, arXiv preprint arXiv:1606.01540 (2016)

[5] Mnih, Volodymyr and Kavukcuoglu, Koray and Silver, David and Graves, Alex and Antonoglou, Ioannis and Wierstra, Daan and Riedmiller, Marti, Playing atari with deep reinforcement learning, arXiv preprint arXiv:1312.5602 (2013)

[6] Mnih, Volodymyr and Kavukcuoglu, Koray and Silver, David and Rusu, Andrei A and Veness, Joel and Bellemare, Marc G and Graves, Alex and Riedmiller, Martin and Fidjeland, Andreas K and Ostrovski, Georg and others, Human-level control through deep reinforcement learning, Nature **518** 529 (2015)

[7] Van Hasselt, Hado and Guez, Arthur and Silver, David, Deep reinforcement learning with double q-learning, Proceedings of the AAAI conference on artificial intelligence **30** (2016)

[8] Wang, Ziyu and Schaul, Tom and Hessel, Matteo and Hasselt, Hado and Lanctot, Marc and Freitas, Nando, Dueling network architectures for deep reinforcement learning, International conference on machine learning 1995 (2016)

[9] Lillicrap, Timothy P and Hunt, Jonathan J and Pritzel, Alexander and Heess, Nicolas and Erez, Tom and Tassa, Yuval and Silver, David and Wierstra, Daan, Continuous control with deep reinforcement learning, arXiv preprint arXiv:1509.02971 (2015)

[10] De Bruin, Tim and Kober, Jens and Tuyls, Karl and Babuška, Robert, The importance of experience replay database composition in deep reinforcement learning, Deep reinforcement learning workshop, NIPS, (2015)

[11] Wei, Honghao and Ying, Lei, Fork: A forward-looking actor for model-free reinforcement learning, 2021 60th IEEE Conference on Decision and Control, 1554 (2021)