COMP 412 - Autonomous Agents (Term Project)
School of Electrical and Computer Engineering
Technical University of Crete
Fall 2021-2022

# Landing Spaceships with Q-Learning

by

**Kariotakis Emmanouil**
ID: 2017030126

*Instructor: Michail G. Lagoudakis*
27 February, 2022

# Contents



# List of Figures



# List of Tables

# 1 Introduction

The purpose of this project is the understanding of Q-Learning, a Reinforcement Learning Algorithm, and the implementation of a Deep Q-Learning agent for solving a control task. The task we sought to solve is "LunarLander-v2", by *OpenAI Gym* [1]. The goal of this task is to land a small spacecraft in between two of two flags.

# 2 Theoretical Background

## 2.1 Reinforcement Learning

Reinforcement Learning (RL) is a field of machine learning that is concerned with how intelligent agents ought to take actions in an environment. There is the agent, a set of states $S$ and a set of actions $A$, per state. Execution of an action in a specific state provides a reward to the agent. The main goal of the agent is to maximize its total reward.

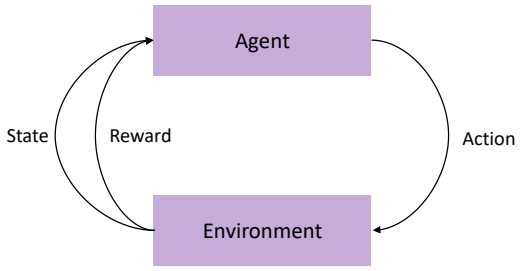



Figure 1: Reinforcement Learning

## 2.2 Markov Decision Process (MDP)

A noteworthy point is that each state of the environment is a consequence of the previous states. Storing all this information, even for small environments, is infeasible. To resolve this problem, we can assume that the each state depends only to each previous state and that it is independent from all those before that (Markov Property). We can assume the same for the rewards.

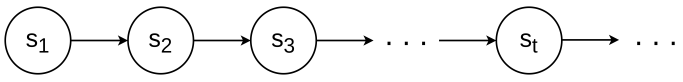



Figure 2: Markov Chain of States

## 2.3 Q-Learning

Q-Learning is a model-free reinforcement learning algorithm that learns the value of an action in a particular state. This algorithm does not require a model for the environment (model-free). For a finite MDP, Q-learning can find an optimal action-selection policy to

maximize the expected value of the total reward over any and all successive steps, starting from the current state.
Now, we can define some variables that will be useful in our analysis.

- $S$: state space
- $A$: action space
- $P$: transition model
- $R$: reward model
- $r$: reward
- $\gamma$: discount factor ($0 < \gamma \leq 1$)
- $D$: initial state distribution

The basic idea is that if the agent knows the expected reward of each action at every step, it would perform the sequence of actions that would maximize the total reward. This total reward is called Q-value and can be formalized as:

$$Q^{\pi^*}(s_t, a_t) = R(s_t, a_t) + \gamma \sum_{s_{t+1} \in S} P(s_{t+1}|s_t, a_t) \max_{a_{t+1} \in A} Q^{\pi^*}(s_{t+1}, a_{t+1}).$$

The Q-Learning update equations proposed by Watkins et al. 1989 [2], are:

- for each $(s_t, a_t, r, s_{t+1})$ sample:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left( r_t + \gamma \max_{a_{t+1} \in A} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right),$$

- if $s_{t+1}$ is a terminal state:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left( r_{t+1} - Q(s_t, a_t) \right),$$

where $\alpha$ is the learning rate.

# 3 Deep Q-Learning

## 3.1 Q-Learning Issues

Q-learning is a simple and powerful algorithm that creates a Q-table. This table helps the agent to figure exactly which action to perform. Unfortunately, Q-learning requires the Q-table to contain an entry of all possible states the environment can take. In the case that the environment contains a small amount of discrete state elements, then the traditional Q-learning algorithm can be a good solution. On the other hand if we have a problem with $|S|$ states and $|A|$ actions, the size of the table that should be created is $|S| \times |A|$.
Now, imagine that we have continuous states. To store them, we must discretize them. This operation can result with thousands or millions of state values and it breeds two main problems:

i. the amount of memory required to save and update that table will increase significantly
ii. and the amount of time required to explore each state, to create the required Q-table would be unrealistic

where $\alpha$ is the learning rate.

### 3.2 Deep Q-Networks (DQNs)

How can we overcome the problems stated above? We can create a neural network (NN) and approximate the Q-values with it. This was the basic idea behind DeepMind's algorithms [3], [4].

In deep Q-learning we use neural networks to replace the Q-table. Unlike a teable, a neural network does not represent every combination of state and action. NNs can generalize these states and learn commonalities. To a DQN is given as input the state and hte Q-value of all possible actions is generated as the output. The DQN effectively becomes a function that accepts a state and suggests an action, by returning the expected reward for each of the possible actions. Figure 3 compares Q-learning with deep Q-learning.

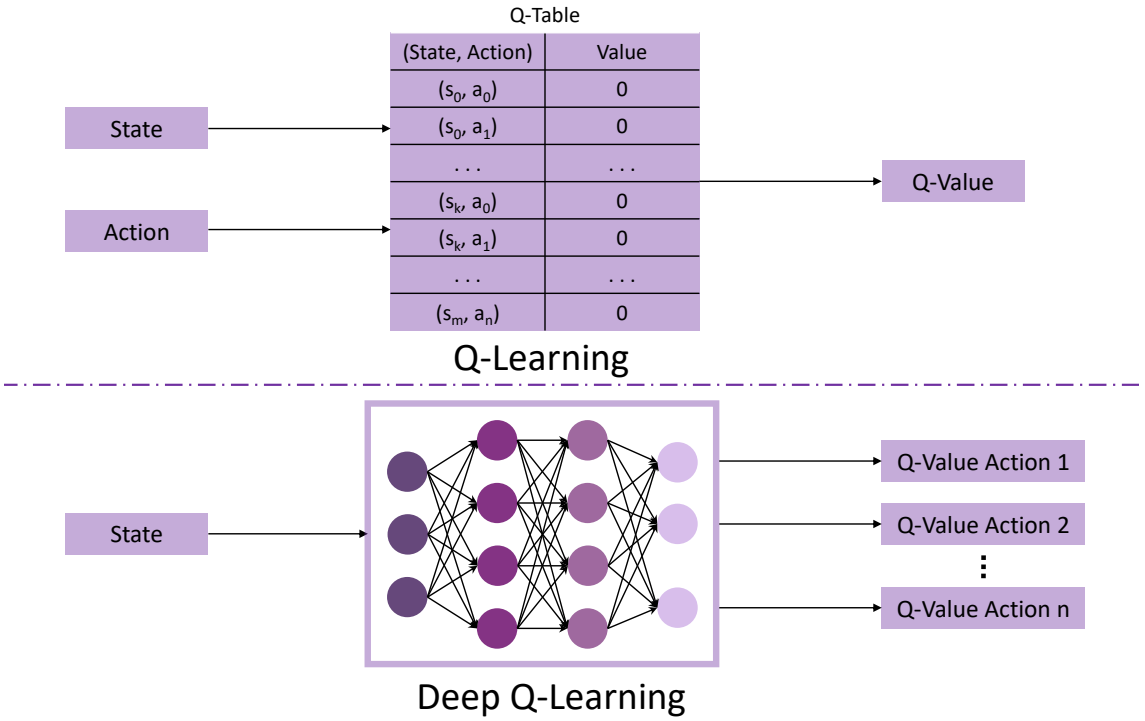



Figure 3: Q-Learning & Deep Q-Learning

The basic steps of RL using DQNs are [5]:

1. Store all past experience in memory.
2. Determine the next action by the maximum output of Q-Network.
3. Compute the loss function as the mean squared error of the predicted Q-value and the target Q-value. However, because of the nature of the problem, there is no known target value. If we observe the Q-value update equation, we can define the following target value.

$$Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha(\underbrace{r_t + \gamma \max_a Q\left(s_{t+1}, a_{t+1}\right)}_{\text{target value}} - Q\left(s_t, a_t\right))$$

### 3.3 Deep Q-Learning Challenges

In classic deep learning algorithms the target is constant and thus we have a stable training. This is not true for reinforcement learning. It is clear that the target value we defined above, is changing in each iteration of the algorithm. As the agent explores the environment, it understands more about it and get more knowledge about the states and the values, thus the output is also changing. Below, we present some tricks that can be used to create a more stable model that converges to a good solution.

#### 3.3.1 Target Network

An important component of DQN is the use of a target network, which was introduced to stabilize learning. In Q-learning, the agent updates the value of executing an action in the current state, using the values of executing actions in a successive state. This procedure often results in an instability because the values change simultaneously on both sides of the update equation. A target network is a copy of the estimated value function that is held fixed to serve as a stable target for some number of steps. [6]
The update procedure of the target network value is being done according to the following equation,

$$v_{target} = \tau \cdot v_{predict} + (1 - \tau) \cdot v_{target},$$

where $v_{target}$ and $v_{predict}$ are the values of target and predict networks, accordingly, and $\tau$ is just an update factor that defines how the target values are going to be updated.

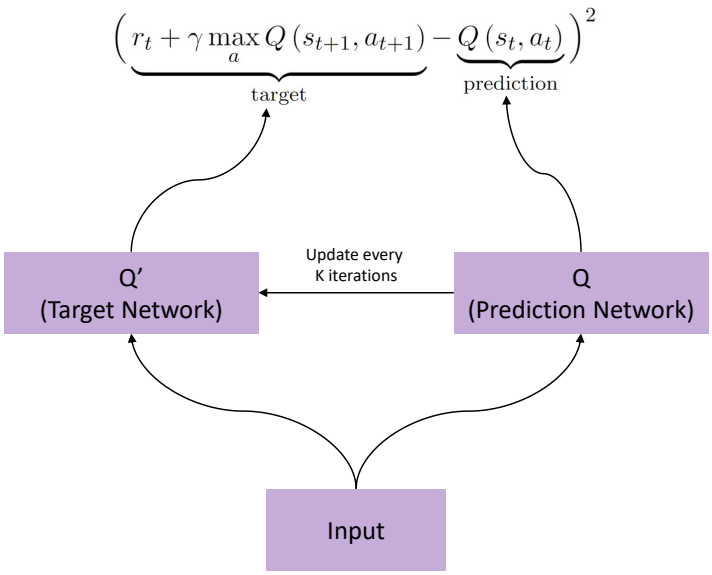



Figure 4: Target Network Policy

### 3.3.2 Experience Replay

Another modification that can be applied to DQN algorithm is experience replay. The main idea behind this policy is that instead of training the agent after each episode, we store its experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ in a table, called Replay Buffer. Then, during learning, we apply updates on Q-value using samples or batches this table. We can also use previous experience, by learning it multiple times, thing very important when gaining real-world experience is costly. This can happen because Q-learning update are incremental and do not converge quickly, so multiple passes with the same data can be beneficial, especially when there is low variance in immediate outcomes, reward and next state, given the same state and action pair. Also, by storing the experiences, we can break the dependence of consecutive episodes and making our selections randomized. This can approximate i.i.d. data which are assumed in most supervised learning convergence proofs.

### 3.3.3 Epsilon-Greedy Action Selection

In RL there is always the struggle between exploration and exploitation. This means that, if you always select the actions that seem the best (exploitation), you might never discover that there are better alternatives. Also, if you explore too much (exploration), you may never reach the better solution.
Epsilon-greedy action selection, has an $\epsilon$ value in $[0, 1]$, determining the amount of exploration vs exploitation. For 0, agent always chooses the action that seems the best and for 1, agent always chooses a random action. At the first episodes of the algorithm, the $\epsilon$ value is high, close to 1, and gradually decreases. The reason we do that, is that in the beginning the agent does not know anything about the environment, and it must explore it to discover the effect of its actions, and as it starts to learn it can exploit good strategies.

## 3.4 Dueling Deep Q-Network

Another network architecture proposed by Google DeepMind in 2016, is the Dueling Q-Network [7]. This architecture, explicitly separates the representation of state values and (state-dependent) action advantages. The dueling architecture consists of two streams that represent the value and advantage functions, while sharing a common feature learning module. The two streams are combined via a special aggregation layer to produce an estimate of the state-action value function $Q$, as shown in figure 5 (this figure represents a convolutional neural network, because it is taken unchanged from the related paper). This dueling network should be understood as a single Q-network with two streams, that replaces the single-stream Q-network which was described previously.
Intuitively, the dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state. This is particularly useful in states where its actions do not affect the environment in any relevant way.
Given a policy $\pi$, the action and state value are defined as, respectively:

$$Q^{\pi}(s, a) = \mathbb{E}[R_t | s_t = s, a_t = a, \pi],$$
$$V^{\pi}(s) = \mathbb{E}_{a \sim \pi(s)}[Q^{\pi}(s, a)].$$

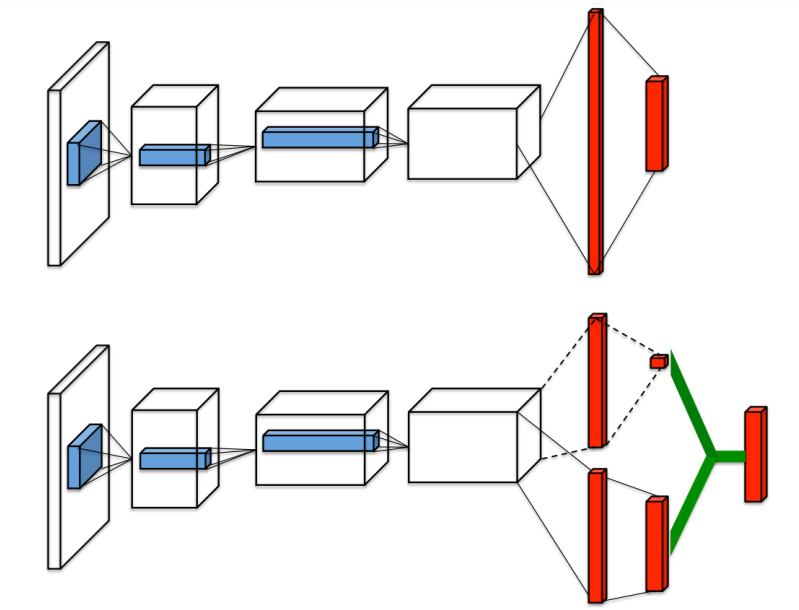


Figure 5: Single Stream Q-Network (**top**)
Dueling Q-Network (**bottom**)

The advantage is defined as:

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

Intuitively, the value function V measures the how good it is to be in a particular state s. The Q function measures the the value of choosing a particular action when in this state. The advantage function subtracts the value of the state from the Q function to obtain a relative measure of the importance of each action.
The first stream computes the state value, $V$, and the second stream the advantage value, $A$. In [7] there is an extended analysis of dueling network architecture, and there is a proposed way to connect the two streams to the output of the network, as:

$$Q(s,a) = V(s) + \left( A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a') \right),$$

where $|\mathcal{A}|$ is the number of available actions.

# 4 Environment

## 4.1 OpenAI Gym

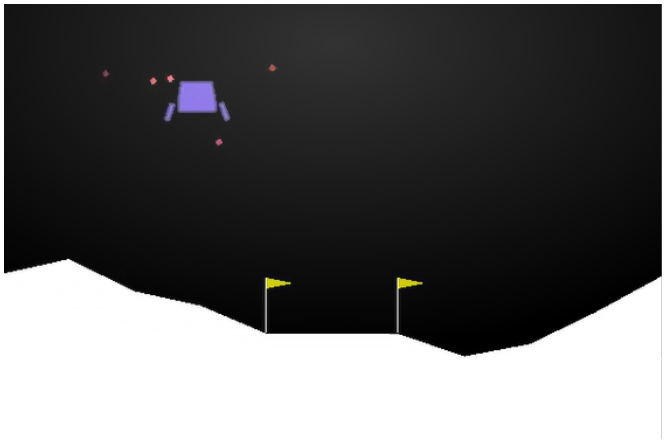
OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. The gym[1] library includes various environments that one can use to work out their algorithms. Their is also OpenAI Gym Leaderboard[2], which tracks performance of user

[1] `https://gym.openai.com/`
[2] `https://github.com/openai/gym/wiki/Leaderboard`

algorithms and where anyone can submit their scores, by providing a write-up with sufficient instructions for reproducing the results and optionally a video showing thos results.

### 4.2 LunarLander-v2

One of many provided environments is the "LunarLander-v2". The goal of this task is to land the lander between the two yellow flags without crashing it.



The lander has 4 discrete actions available:

- fire left orientation engine
- fire main engine
- fire right orientation engine
- do nothing

and there are 8 observation states:

- x-axis coordinates
- y-axis coordinates
- x-axis linear velocity
- y-axis linear velocity
- angle
- angular velocity
- left leg in contact with the ground (boolean)
- right leg in contact with the ground (boolean)

The rewards have as follows:

- Reward for moving from the top of the screen to the landing pad and coming to rest is about 100-140 points. (If the lander moves away from the landing pad, it loses reward.)
- If the lander crashes, it receives an additional $-100$ points.

- If it comes to rest, it receives an additional +100 points.
- Each leg with ground contact is +10 points.
- Firing the main engine is −0.3 points each frame.
- Firing the side engine is −0.03 points each frame.
- Solved is 200 points.

The episode terminates if:

- the lander crashes (the lander body gets in contact with the moon),
- the lander gets outside of the viewpoint (x-coordinate is greater than 1),
- the lander is not awake (doesn't move and doesn't collide with any other body).

# 5 Implementation

The implementation of the agent was done in Python 3.9.10, using PyTorch 1.10.2. The environment of the game is provided by OpenAI Gym, as described in the previous section. The installation and execution instructions are included in the provided GitHub repository.

## 5.1 Structure

The project is divided in two basic folders, the `dqn` and `dueling_dqn`.
The first, implements the simple DQN agent and contains the following 3 files:

- `dqn_agent.py`: This file contains 2 Classes, the `Agent` and the `ReplayBuffer`. The `Agent` class implements the basic functionality of the deep Q-learning agent as was previously described. The `ReplayBuffer` class implements the table that is used to store experiences by the experience replay policy.
- `model.py`: This file implements the deep neural network that is used in DQN. The NN we created is fully connected, with 4 layers. The input layer has $|S| = 8$ nodes, the two hidden layers have 64 each one of them and the output layer has $|A| = 4$ nodes.
- `main_dqn.ipynb`: This jupyter notebook file, is basically the "main" file of the project. From this file one can train the agent and visualize the results.

The second, implements the dueling-DQN agent and contains similar files to the first one. The hyperparameters that are used in the project are being described in table 1.

| ***Hyperparameter*** | ***Value*** |
|---|---|
| Replay Buffer Size | $10^5$ |
| Batch Size | 64 |
| Discount Factor $\gamma$ | 0.99 |
| Update Factor $\tau$ | $10^{-3}$ |
| Learning Rate $\alpha$ | $5 \cdot 10^{-4}$ |
| $\epsilon$-start | 1 |
| $\epsilon$-end | 0.01 |
| $\epsilon$-decay | 0.995 |

Table 1: Hyperparameters

The implementation is based on a core implementation provided by Udacity Courses.

## 6 Results

The performance of our agent is being measured using the reward values from each episode. At each episode of training, the agent performs 1000 steps or stops and goes to the next episode if the environment returns the variable `done` as true. The sum of all rewards from each step is stored in a variable called `score`. This score is our metric of performance. We observed that the problem is solved when score is about 250.

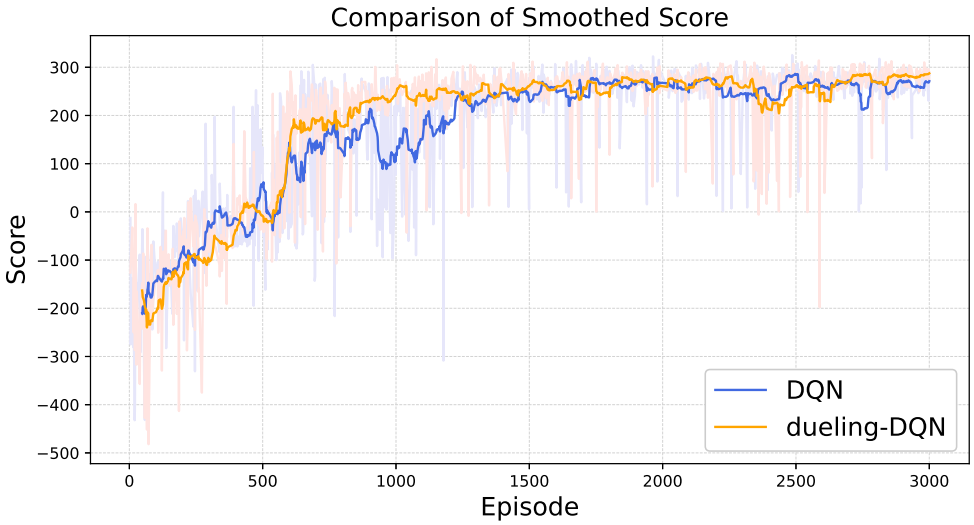



Figure 6: DQN vs Dueling-DQN

We can see from figure 6, that the dueling-DQN has a bit better performance than the simple DQN. Especially, it gets faster to a score value of around 250, at the first 1000 episodes. From there it is mostly stable. On the other hand, the simple DQN agent has to run 1500 episodes to get to a score value of 250. The difference is not that big in our problem because the available actions for the agent are only 4. As stated in [7], when we

increase the number of actions, the dueling architecture performs clearly better than the traditional Q-network.

# 7 Conclusion

## 7.1 Advantages - Disadvantages

The main advantage of the model that we created is that it does not need to have any knowledge about the environment. During training it just observes the rewards for specific actions and makes its decisions. Another great advantage of the method is that it can be used for solving plenty different problems with changing only the number of states and actions, based on the given environment.
On the other hand, it certainly has some disadvantages. The agents needs many episodes to be trained properly and in environments with large state and action spaces it may even take days of training to solve the given problem. Also, the DQN model is not able to solve problems that require a continuous action space. A way to overcome this is to discretize the action space, but this will lead to an extremely large number of actions, that will make the training impossible, or to less number of action that will be unable to describe the necessary space.

## 7.2 Challenges

A challenge that one can face when is dealing with reinforcement learning problems is the tweaking of hyperparameters. There can be various changes in those parameters which can alter the time needed in training or performance of the agent. This can be assumed to be an art, that the engineer must master to create better agents.
Here we chose a well-created ready environment. In real problems, one should create the environment and choose wisely the policy of the rewards that are given to the agent. This can be a very challenging task and one could argue that the hardest part to reinforcement learning is actually in the engineering of your environment's observations and rewards for the agent.

## References


[1] G. Brockman, V. Cheung, L. Pettersson, *et al.*, "Openai gym," *CoRR*, vol. abs/1606.01540, 2016. arXiv: 1606.01540. [Online]. Available: http://arxiv.org/abs/1606.01540.

[2] C. Watkins, "Learning from delayed rewards," Jan. 1989.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. DOI: 10.1038/nature14236. [Online]. Available: https://doi.org/10.1038/nature14236.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. arXiv: 1312.5602. [Online]. Available: http://arxiv.org/abs/1312.5602.

[5] A. Choudhary. "A hands-on introduction to deep q-learning using openai gym in python." (2019), [Online]. Available: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/.

[6] S. Kim, K. Asadi, M. Littman, and G. Konidaris, "DeepMellow: Removing the need for a target network in deep q-learning," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, International Joint Conferences on Artificial Intelligence Organization, Aug. 2019. DOI: 10.24963/ijcai.2019/379. [Online]. Available: https://doi.org/10.24963/ijcai.2019/379.

[7] Z. Wang, N. de Freitas, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," *CoRR*, vol. abs/1511.06581, 2015. arXiv: 1511.06581. [Online]. Available: http://arxiv.org/abs/1511.06581.