Deep Reinforcement Learning: with applications to robotic control

Niels Verleysen

Supervisor(s): prof. dr. ir. Aleksandra Pizurica, Laurens Meeus and dr. ir. Tim Waegeman (RoboVision)

Abstract— **The research field of reinforcement learning (RL) is evolving very quickly. State-of-the-art algorithms already perform better than humans in a number of tasks. The most powerful part of these algorithms is how easily they can be applied to different tasks. By only stating what results are good and bad, RL-algorithms can, independent from human interaction, learn to behave as to maximise its rewards. In this thesis we study multiple variants of the recent Deep Q-Network algorithm. We then introduce an alternative, easier way to prioritise experiences. This alternative way is then used with these algorithms to solve different tasks. These tasks can be divided in two parts. First, these different versions of the algorithm are tested and evaluated in different environments from OpenAI Gym. In this part we show that it is very important to continue exploration to keep the policy optimal when the learning process is not stopped. Secondly, the Double Deep Q-Network algorithm is used to control an industrial robotic arm in a simulated environment. The goal is here to control the movements of the robotic arm to pick up items in front of it.**

Keywords— **Reinforcement Learning, Q-Learning, Deep Q-Network, Double Deep Q-Network, Dueling Double Deep Q-Network, Robotics, Picking tasks**

I. Introduction

A Reinforcement Learning (RL) algorithm learns a strategy, called a policy, in an environment and uses its experiences in this environment to adapt its policy [1]. The algorithm strives for an optimal policy, which gives the maximum cumulative reward that can be obtained. The wanted behaviour can thus be determined by engineering this reward. This has both its advantages and its disadvantages. A big advantage is that the algorithm can find new strategies, which could give us new and better insights in the environment in which this algorithm has trained. On the other hand this could also lead to unwanted behaviour, that may have not been foreseen.

Two of the most well known state-of-the-art algorithms are AlphaGo [2] and OpenAI Five [3]. Both have found new strategies to play the games Go and Dota 2 respectively. With these new strategies, both algorithms were able to beat the best human players in their respective games. Nowadays OpenAI Five is used by professional Dota 2 teams to find new strategies to beat their opponents. However, RL algorithms can also be used to learn a multitude of tasks in the real world [4] [5] [6]. In this thesis we want to apply a state-of-the-art RL algorithm in an industrial setting and let it learn to control a robotic arm and use it to perform picking tasks.

To reach this goal we study different forms of the Deep Q-Network (DQN) algorithm [7], which is a form of Q-learning. These algorithms have already been used to defeat the best human players in a multitude of computer games only based on the accumulated score [7] [8] [9]. Thus proving that these algorithms are very powerful and could be used for controlling a robot. First we test and evaluate these algorithms on two computer games in the OpenAI Gym environment. While doing this we introduce an alternative way of organising the memory, which can lead to better results in the learning process. We also show that it is important for the algorithm to keep exploring as to keep the policy optimal. This exploration allows the algorithm to keep on learning, even if the optimal policy has been found. This in turn enables the algorithm to adapt its policy if the environment would be dynamic.

Playing computer games is only a little part of what can be accomplished with RL. In the second part of this thesis one version of the DQN algorithm is used for picking tasks with an industrial robotic arm. In previous research these algorithms were successful in learning a good strategy [10] [11] [12]. In these cases a setup of multiple physical robots was used and it took months to gather enough data for training. An alternative method could be to use a simulation of the robot and then transfer the learned policy to a real robot. This would allow training to be much faster and it would remove the need of a setup with multiple real robots. It was shown in [13] that this is possible, but not without difficulties. When transferring the learned policy in a simulation to a robot in the real world, the algorithm moved the gripper to close states and remained there instead of picking up the object. When starting the robotic arm with the object immediately in its grasp, it did pick up the object.

In this thesis we use a powerful simulator from the industry to train a model of an industrial robotic arm. The learned policy can then be applied to a real life model of the robot.

First we develop a modular framework for training robots to manipulate objects in the Gazebo simulator. We then use this framework with the Double DQN algorithm, with and without the dueling architecture, to control the Denso vs060 industrial robotic arm and Robotiq Hand-e gripper to learn picking strategies. As the learning process is very slow, the algorithm could not learn to pick up the object consistently. However, it was already able to move the gripper towards the object. When starting with the object in its grasp, the algorithm was able to pick up the object. This shows that our method can, with more training, perform this task consistently.

Section II gives the relevant background information for this thesis. In section III we introduce our alternative way of prioritising experiences. We then combine this method with the studied RL algorithms and test them on two computer games in section IV. We then describe our framework for training robots

and use it to learn picking tasks in section V. Finally we give our conclusions in section VI.

II. Background

Before RL can be used to solve a problem, this problem needs to be transformed to a Markov Decision Process (MDP) [14]. A MDP exists out of multiple parts. First, the environment has to be divided in a limited number of states S. Secondly, there is a possibility in every state to chose an action a after which the state of the environment can be changed by the transition function T. Thirdly, a reward function R gives rewards or punishments after choosing an action. Finally, a policy π determines which action is chosen in each state. To solve a MDP, a policy which gives the maximum possible cumulative reward needs to be found. However, in most practical cases a policy which displays the wanted behaviour and thus has a high cumulative reward, is good enough.

RL starts with a policy and iteratively changes it to get higher cumulative rewards. To determine what action is better, a value is given to each state with a value function V. This value function is typically defined as:

$$V(s) = E\Big[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s\Big] \tag{1}$$

Where r_t denotes the obtained reward in state s_t. The value of a state is determined by the possible reward that can be obtained from that state while following the policy. These rewards are discounted in time with discount factor γ to make sure that future rewards are taken into account but quick rewards are not overlooked as well. In Q-learning the Q-values are used instead, these values are defined as the possible reward that can be obtained from a state after taking an action and following the policy afterwards. The Q-function can easily be extracted from the earlier defined value function V:

$$Q(s,a) = E\Big[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a\Big] \tag{2}$$

The policy simply selects the action with the highest Q-value in every state. Thus to find the optimal policy, the real Q-values need to be found or at least be approximated. To do this, a Q-value can be updated with an experience in this state via the update rule defined as:

$$Q(s,a) = Q(s,a) + \alpha\Big(Y - Q(s,a)\Big) \tag{3}$$

This rule changes the current Q-value with the difference between this value and a target value Y calculated from the experience as follows:

$$Y = R(s) + \gamma max_{a'} Q(s', a') \tag{4}$$

The impact of this difference on the Q-value is determined by the learning rate α, which is smaller than one. By exploring the environment and training on the experiences, the policy becomes more optimal.

While exactly keeping track of every Q-value works in theory and on simple environments with a relative low amount of Q-values. More complex environments can have a very high amount of states, making it very difficult or even impossible to efficiently keep track of every Q-value. To overcome this, an approximator can be used to approximate the Q-value for a given state-action combination. In the Deep Q-Network algorithm [7] a neural network is used to predict the Q-values for every action a in a given state s. This not only removes the need to keep track of every Q-value separately, but also adds the ability of a neural network to generalise states.

Simply using a neural network combined with the update rule does not always work because it can become very unstable. Mostly the Q-values for a state are approximately the same, thus very small changes can have a big impact on the policy. To solve this instability as good as possible the DQN algorithm adds two extra components, namely iterative updates and experience replay. In iterative updates a second neural network, noted by Q_2, is added, this network is an exact copy of the first neural network, noted by Q_1, but it only gets updated with the first neural network every few steps. This network is then used to calculate the target Q-value Y:

$$Y = R(s) + \gamma max_{a'} Q_2(s', a') \tag{5}$$

This target is then used to update the first neural network:

$$Q_1(s,a) = Q_1(s,a) + \alpha\Big(Y - Q_1(s,a)\Big) \tag{6}$$

Because this secondary network is updated less, it removes the possibility of the Q-values to diverge because of a sequence of experiences around the same state. Thus making this algorithm more stable.

Experience replay adds a memory to the algorithm which keeps track of previous experiences. At each step the neural network then trains on a batch from this memory. This has multiple advantages which improve both stability and training speed. Because a batch is taken from the memory, the neural network can train on a more diverse set of states. Making the learning process more stable. By using this memory experiences can be used multiple times, limiting the need to visit certain state-action combinations multiple times. Thus speeding up the training time as e.g. states that are harder to reach do not need to be visited very often.

The DQN algorithm works fairly well and was already used to beat the best human players in some computer games. However this algorithm still suffers from instability after a long time of training. It was shown in [8] that the cause for this was overoptimism. The approximated Q-values become a lot higher than the real values after a long time of training. This overoptimism comes from the calculation of the target Q-value in the update rule, where the maximal Q-value is selected for the next state.

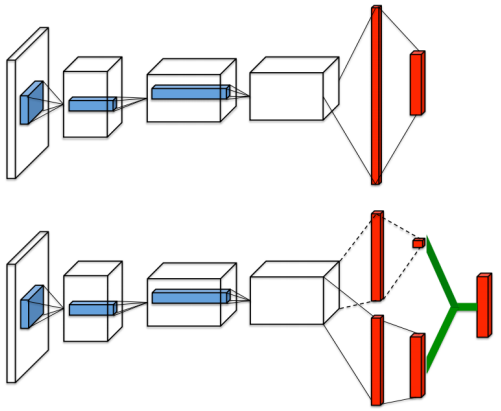

Fig. 1. Transition from a feed-forward neural network to the dueling architecture. Extracted from [9].

These Q-values are however estimated and by selecting the maximum Q-value a lot of times an overestimated value is going to be picked. This in turn has an effect on the states before, causing a chain reaction. This instability can be solved by improving the way the target is calculated:

$$Y = R(s) + \gamma Q_2(s', argmax_{a'} Q_1(s', a')) \tag{7}$$

This is called double Q-learning and the DQN algorithm becomes the Double DQN (DDQN) algorithm.

Another improvement that can be made is the architecture of the neural network. Typically a feed forward neural network was used to approximate the Q-value for each action for a given state as input. Using an architecture specifically designed for Q-learning can lead to a drastic increase in performance. One of these architectures is called the dueling architecture [9] and when used in the DQN or the DDQN algorithm these algorithms are respectively called the Dueling DQN and Dueling DDQN algorithm. This architecture receives the state as input and processes it as before, then the network is split in two parts as can be seen in figure 1. The first part estimates the value V of the state, while the second part estimates the advantage A of every action. Both these parts can then simply be combined to calculate the Q-values:

$$Q(s, a) = V(s) + A(s, a) \tag{8}$$

To be able to split the Q-value in V and A for backpropagation the advantages are normalised. This normalisation can be done in several different ways. In this thesis the normalisation is done with the average advantage:

$$Q(s, a) = V(s) + \big(A(s, a) - \frac{1}{|A|} \sum_a A(s, a)\big) \tag{9}$$

With this split the network can better differentiate between the values of states and the values of actions, making it easier to learn the correct Q-values.

III. Alternative memory structure

Experience replay has a lot of advantages, but not all experiences are interesting. In most cases only a handful of experiences are interesting, while the memory is oversaturated with other experiences. This results in the algorithm sometimes having trouble to find the right policy. To solve this, prioritised experience replay [15] can be used. In this version the experiences are ordered by some metric to train more on interesting experiences and forget the less interesting experiences. With Q-learning the difference between the target Q-value, that would be calculated from an experience, and the current Q-value could be used as a metric. However if the memory exists of thousands of experiences it becomes difficult to keep this ordering in an efficient way. Even the most efficient implementations of prioritised experience replay will decrease the computational performance, but as shown in [15] will boost the learning speed.

This prioritised experience replay can however be simplified by our knowledge of the environment. Beforehand it is known that some experiences carry vital information for the algorithm, but don't happen very often. We can therefore split the memory in multiple parts, with one or more small memories for these interesting experiences and one big memory for all other experiences. When selecting experiences for training we can force the algorithm to also train on some of these interesting experiences.

This way of organising the memory is not as good as prioritised experience replay, as it will not differentiate between different experiences in these memories. It is however a lot easier to use and doesn't take a lot of time performance-wise. In the following section we use our new way of prioritising experiences with the studied algorithms in two environments from the OpenAI Gym environment.

IV. OpenAI Gym

A. Cartpole

In cartpole the agent controls a block on which a stick has been mounted. By moving to the left or to the right the agent needs to keep the stick from falling for as long as possible. This environment is very simple and is one of the most well studied examples to apply RL algorithms. Because this environment is very simple it is also very easy to evaluate an algorithm on it.

In this thesis we use both the DQN and the DDQN algorithm to find a good policy for keeping the stick up. We also use our new way of prioritising experiences to get better results. The interesting experiences in this case are when the task fails, either by dropping the stick or by moving out of bounds. In this environment there can be maximum one of these interesting experiences per episode, while there can be a maximum of 500 experiences per episode. After 500 episodes the game is won and the environment is reset, in that case there is no experience of losing. A normal memory would be oversaturated with generic memories, causing the algorithm to learn a lot slower from its bad experiences. Over time situations where a certain action needs to be chosen to avoid losing, could even be forgotten. Thus, making clear that prioritising experiences is important.

When using neural networks it is important to prevent over-

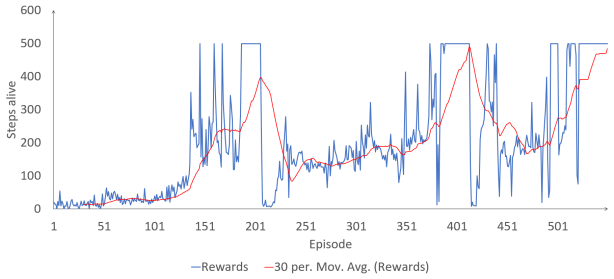

Fig. 2. Number of steps before losing in cartpole for DDQN without taking overtraining into account.

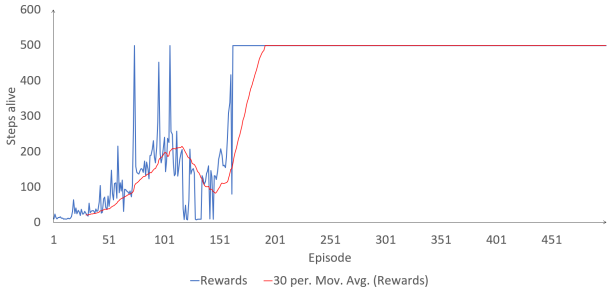

Fig. 3. Number of steps before losing in cartpole for DQN with optimal policy extraction.

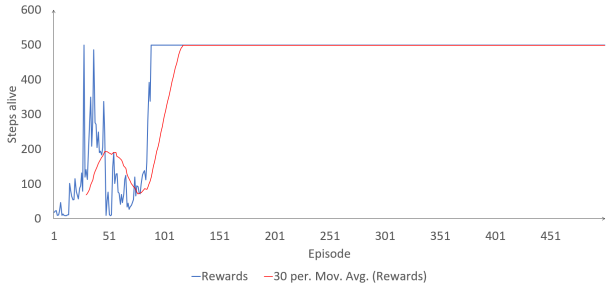

Fig. 4. Number of steps before losing in cartpole for DDQN with optimal policy extraction.

Fig. 5. Number of steps before losing in cartpole for DDQN with extra exploration.

training and this is also the case with the DQN algorithm and its variants. However it is difficult to predict and prevent overtraining in RL algorithms [16]. This importance is clearly visible when we don't take overtraining into account. In this case the algorithms learn to obtain the maximum score, but after a while the score drops. As can be seen in figure 2 for the DDQN algorithm. The score builds up to the maximum and stays there for some episodes before dropping down after which it builds back up. This degradation of the score can be explained by overtraining, as the optimal policy only visits a very few states.

What is done most of the time to prevent overtraining, is simply stopping the training when a good policy has been achieved. This can also be done here. The maximal number of steps an episode in cartpole can take is set to 500, therefore we can stop the learning process when this score has been obtained multiple times in a row. Our result for both the DQN and the DDQN algorithms can be seen in figures 3 and 4 respectively.

RL has the ability to adapt to a changing environment, but this ability is shut off by stopping the algorithm from learning from new experiences. In this environment this is not necessary, but in other environments it would be better to not stop the learning process.

An alternative way to prevent overtraining would be to explore other states as well, even when the optimal policy has been found. In this case we let the algorithm chose all actions randomly every 25 episodes. The result can be seen in figure 5, which shows that the policy remains optimal by exploring, even when the optimal policy has already been found.

This could also be achieved in some other ways where we keep some form of randomness over time. This could, for instance, be achieved by using the ϵ-greedy method with a higher minimal value for ϵ. In these tests we always used the ϵ-greedy method with a lower value, as a way to explore more in the beginning of the learning process and less in the end.

B. Beamrider

The discussed algorithms can also be used to play more complex computer games, such as Beamrider. In Beamrider the player controls a spaceship that has to get through multiple levels in which it needs to destroy a number of enemies. All while evading meteoroids, projectiles and the enemies themselves. To do this, the player can move the spaceship to the left, the right or remain still and has the ability to shoot. The frames from the game are used as states. We first preprocess these frames as to decrease the amount of calculations done by the neural network. This preprocessing starts by transforming the frame from color to grayscale. Next, the frame is resized to a standard, smaller size. Finally, the user information with the score, the level, etc. is cropped from the frame to limit the number of states.

We keep the reward function very simple. When the score in the game is increased the algorithm gets a reward of one. When the agent dies, a reward of minus one is given. In all other cases the given reward is zero. This shows that when using a RL algorithm to find a strategy in an environment only the good and/or bad results should be known. The algorithm can then find a strategy to get good results and to avoid bad results.

Again we use our new way of prioritising experiences. As the player has only a few lives, the interesting experiences are these where the player dies. This way we heavily prioritise the

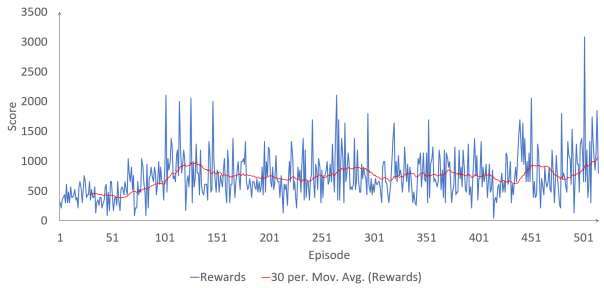

Fig. 6. Score achieved in each episode by the Dueling DDQN algorithm in Beamrider.

survival of the agent. It could be possible to also add the experiences where an enemy is killed to a separate memory. We did not do this, because there are already a lot of ways to increase the score and a small increase in score is not as important as staying alive.

Our best result in this computer game was with the Dueling DDQN algorithm and is shown in figure 6. Even though the scores are not as impressive as the state-of-the-art implementations and don't come near the best human scores, it shows that the algorithm is learning a decent policy. When examining the learned policy, it is very clear that the agent focuses on staying alive instead of gaining a higher score. Because we heavily prioritised the survival of the agent, the algorithm tries to take no risks. Aside from the main focus on survival instead of achieving scores, the reward function does not take into account the amount of points gained. There are bonus points that can be gained in certain levels which can increase the score drastically. The combination of these two reasons explains why our learned policies don't get higher scores.

When looking at the different learned policies, the influence of the introduced alternative for prioritised experience replay stands out the most. When comparing the results of each algorithm, it is not really clear what version of the DQN algorithm is better. This could be the case because the differences start to show after more training. What is more clear however, is the fact that the Dueling DDQN algorithm keeps increasing its score to the end, while the scores of other versions don't.

V. Controlling an industrial robot

The discussed algorithms can also be used for more complex tasks than playing computer games. In this thesis we used the DDQN algorithm, with and without the dueling architecture, to control the Denso vs060 industrial robotic arm with a Robotiq Hand-e gripper. We want this algorithm to learn a strategy for picking tasks in a simulated environment so it can later be extracted for use in a real robot.

First we build a modular framework with ROS in which we can use different algorithms, robots and tasks. The framework exists out of four main components that all communicate with each other. The first component is the model of the robot with controllers to give it commands to move different components of the robot. The second component gives preprocessed images from the scene taken by a camera. The third component manages objects in the environment. The final component is the algorithm used to control the robot.

We used the reward function from [13] and made some modifications to it. This reward function is more complex and exists out of multiple conditions. As long as the item has not been picked up, a small reward is given. This reward increases exponentially when the gripper gets closer to the object. When the object has been picked up, a higher reward that increases with the height of the object is given. Finally when the picked up item gets over a certain height a high reward of 1000 is given. This value has been chosen as the algorithm can already obtain a reward of multiple hundreds and we want this final reward to mark the end of a successful task. It is possible for the robot to get stuck in some positions, e.g. when pressing the gripper against the ground. Because it is important that the robot does not damage itself, a negative reward is given in that case. The earlier discussed alternative for prioritised experience replay is not used. We could use it for making sure that the arm does not damage itself, however this could also make the policy less likely to move the arm downwards as the arm could get stuck against the ground.

While training it became clear that the used learning rate has to be very low as to prevent the estimated Q-values to diverge. This however made the learning process very slow. When starting from the designated starting point the algorithm learned to move the arm a bit downwards after two days. After two more days of training the algorithm had learned to move the arm quicker downwards, but it still did not come close to the object.

When starting the arm directly above the object, the algorithm learned to almost pick up the object after a single day. The gripper mostly touches the object and moves or tilts it on its corners. When starting the arm in a position where it can immediately pick up the object, the algorithm effectively learned to pick up the object in less than a day of training. This shows that it is possible for the algorithm to learn to pick up the object. The manipulation of the gripper close to the object is very complex, thus making it very hard for the algorithm to quickly learn the exact strategy to do this.

Finally we combined the previous three starting positions. At the start of every episode one of these positions is chosen. We use our alternative method for prioritising experiences to save the experiences from each starting position in a separate memory. This way the algorithm can learn all three parts of the task at the same time. We also updated the used algorithm to the Dueling DDQN algorithm.

In one week of training, our setup was able to do 2000 episodes. The achieved reward in each episode is shown in figure 7. These scores do not show much improvement because of a high amount of exploration. The algorithm learned to move the gripper towards the object. When starting close to the object,

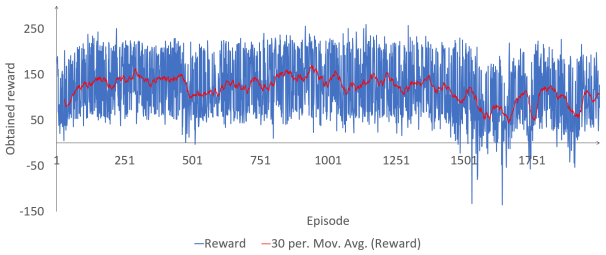

Fig. 7. Obtained reward per episode when randomly picking a starting position.

the gripper could be moved to a position where the object could be picked up. However, mostly the gripper missed the object and moved or tilted it. These are the same results we obtained for each starting position separately, but this time they are combined in a single policy. With the shown progress it is clear that, with more training, the algorithm will be able to consistently pick up the object.

VI. Conclusions

In this thesis we studied different versions of the DQN algorithm and evaluated them on two computer games from the OpenAI Gym framework. To solve these games we introduced an alternative way to prioritise the experiences. This alternative has less variety than prioritised experience replay, but is much simpler and easier to use. It also has a negligible effect on the number of calculations that need to be done in the learning process. While solving these games we also showed that it is important to keep exploring when the optimal policy has already been found, if we want the algorithm to continue learning.

In the second part of this thesis we set up a framework to train algorithms to control a robot and to make it manipulate objects in a simulated environment. We then used this framework to train an industrial robotic arm to pick up an object with the DDQN algorithm. Because of the slow learning process we were not able to find a policy which can consistently perform this task successfully. However, the algorithm already found a policy which lets the gripper move towards the object and comes close to picking up this object. With more training it should thus be possible to learn to consistently pick up the object.

References

[1] Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 2016.

[2] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis, "Mastering the game of Go with deep neural networks and tree search.," *Nature*, vol. 529, no. 7587, pp. 484–9, 2016.

[3] "OpenAI Five," 2018.

[4] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula, "Resource Management with Deep Reinforcement Learning," pp. 50–56, 2016.

[5] I. Arel, C. Liu, T. Urbanik, and A.G. Kohls, "Reinforcement learning-based multi-agent system for network traffic signal control," *IET Intelligent Transport Systems*, vol. 4, no. 2, pp. 128, 2010.

[6] Alexander Bernstein and Evgeny Burnaev, "Reinforcement Learning for Computer Vision and Robot Navigation," in *Machine Learning and Data Mining in Pattern Recognition*, vol. 7988, pp. 258–272. 2018.

[7] Volodymyr Mnih, Koray Kavukcuoglu, and David Silver, "Human-level control through deep reinforcement learning," *Nature*, 2015.

[8] Hado van Hasselt, Arthur Guez, and David Silver, "Deep Reinforcement Learning with Double Q-Learning," *AAAI*, pp. 2094–2100, 2016.

[9] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas, "Dueling Network Architectures for Deep Reinforcement Learning," , no. 9, 2015.

[10] Jeffrey Mahler and Ken Goldberg, "Learning Deep Policies for Robot Bin Picking by Simulating Robust Grasping Sequences," *Proceedings of the 1st Annual Conference on Robot Learning*, vol. 78, no. CoRL, pp. 515–524, 2017.

[11] Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, and Sergey Levine, "QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation," , no. CoRL, pp. 1–23, 2018.

[12] Sergey Levine, Peter Pastor, Alex Krizhevsky, Julian Ibarz, and Deirdre Quillen, "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection," *International Journal of Robotics Research*, vol. 37, no. 4-5, pp. 421–436, 2018.

[13] Stephen James and Edward Johns, "3D Simulation for Robot Arm Control with Deep Q-Learning," 2016.

[14] Martijn van Otterlo, "Markov Decision Processes: Concepts and Algorithms," , no. May, pp. 1–23, 2009.

[15] Tom Schaul, John Quan, Ioannis Antonoglou, David Silver, and Google Deepmind, "Prioritized Experience Replay," pp. 1–21, 2016.

[16] Chiyuan Zhang, Oriol Vinyals, Remi Munos, and Samy Bengio, "A Study on Overfitting in Deep Reinforcement Learning," pp. 1–25, 2018.