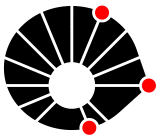

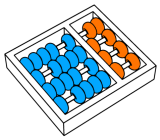

Atari-playing Robot

Renato Landim Vargas *Esther Luna Colombini*

Relatório Técnico - IC-PFG-17-18
Projeto Final de Graduação
2017 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Atari-playing Robot

Renato Landim Vargas * Esther Luna Colombini †

Abstract

Great results achieved by Deep Q-Networks on Atari games gathered a lot of attention from researchers in the past year. Despite this, not many research has been conducted on using these networks on real-world robots learning through high-dimensional sensory input. This work proposes ALE-Robot, which is an architecture that uses ALE (Arcade Learning Environment) and the V-REP environment to simulate a robot learning to play Atari directly from its camera and a game controller through Reinforcement Learning. Results have demonstrated that, when applied to real robotics architectures, longer training periods and re-shaping rewards functions are required. Moreover, more time and computational power is necessary to test the feasibility of the proposed approach over different hyperparameters.

1 Introduction

Creating autonomous agents that learn from interacting with the environment is usually mentioned as one of the goals of artificial intelligence (AI). Reinforcement learning (RL) is an AI area that does that by making the agents learn by themselves, externally providing just rewards for its actions on its current state. Recently, thanks to DeepMinds's Deep Q-Network paper [1], RL agents that learn visually from game environments have gathered a lot of attention from researchers. This high interest can be explained by how games are good environments to test algorithms, given its richness and complexity, and also by all the real-world applications of visual learning agents.

However, most studies rely on a scenario where the agent's action is executed on the environment and returns a new state without delay, and where the state has no noise. Considering environments where agents have to learn from a camera input and execute actions mechanically, these properties are mostly not true. This work simulates a robot on such environment, where the game screen is projected on the simulation and the actions are executed on a physical controller, and analyses the performance of previous proposed algorithms.

*Institute of Computing, Unicamp, 13083-852 Campinas, SP. ra118557@students.ic.unicamp.br
†Institute of Computing, Unicamp, 13083-852 Campinas, SP. esther@ic.unicamp.br

2 Deep Reinforcement Learning

In recent years, deep learning methods have been providing good results for high-dimensional sensory inputs. Deep Q-learning [1] was the first to apply these methods successfully on RL with sensory data. Convolutional networks trained with this approach are called Deep Q-Networks (DQN). DQN reached state-of-the-art results for many Atari games, without adjustments of the hyperparameters.

Following this, extensions built on top of DQN were proposed, and many reached new state-of-the-art results. The first of them was Double DQN (DDQN) [2]. DDQN uses Double Q-learning instead of traditional Q-learning. This modification results in much more stable and reliable learning.

For this work, DDQN is used because it is a good extension to DQN, without clear downsides. Other extensions are not used, as their additional complexity are not justifiable for a initial research on the topic.

2.1 DQN

On the original paper of DQN [1], the authors consider an agent performing actions on an environment $\mathcal{E}$, and receiving back observations and rewards for each action. In their case, the environment used was the Atari emulator. On every time-step, the agent selects an action a_t from a set of possible actions $\mathcal{A} = \{1, ..., K\}$ and performs this action on the emulator. The emulator updates its state and gives back an image x_t and a reward r_t. x_t represents the game screen and it is a vector of raw pixel values. r_t is the change on game score since the last time-step.

Because of the nature of the environment, it is impossible to fully understand the current state of the game just from x_t, so the authors consider a sequence of observations and actions $s_t = x_1, a_1, ..., a_{t-1}, s_t$. This definition of a states gives a Markov decision process (MDP), allowing the use of traditional methods of reinforcement learning methods for MDPs.

One of these methods is Q-learning, which can used to find an optimal policy for selection an action given the current state. This method tries to learn an action-value function Q that returns the expected reward of taking a given action on a given state and continuing to follow the same policy. When we have the optimal function Q^*, the optimal policy would be to always select the action that maximizes the expected value $r + \gamma\, Q^*(s', a')$. This can be represented by

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}}\Big[r + \gamma \max_{a'} Q(s', a')\Big]. \tag{1}$$

The reinforcement learning idea of this method is to have Q_i and update it iteratively, so that $Q_i \to Q^*$. On the paper, a function approximator Q is used, so that $Q(s, a; \theta) \approx Q^*(s, a)$. A neural network function approximator with weights θ is referred as Q-network. This network is then trained by minimizing a loss function, where the current value is $Q(s, a; \theta_i)$ and the target value is $y_i = \mathbb{E}_{s' \sim \mathcal{E}}\Big[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})\Big]$. A gradient descent is then performed based on the loss function to update the network weights.

For every time-step, the agent's experience is stored as $e_t = (s_t, a_t, r_t, s_{t+1})$ on a data-set $\mathcal{D} = e_1, ..., e_N$ called replay memory. A random batch of samples $e \sim \mathcal{D}$ are then drawn and used to perform a Q-learning update. Such technique is known as experience replay [3] and allows for more efficient utilization of the agent's experience.

In order to explore the space state efficiently, it is used an ϵ-greedy strategy that selects a random action with probability ϵ or follows the greedy strategy with probability $1 - \epsilon$. Since updating the network with all the history s_t is impractical, a function ϕ produces a limited-size history to be used.

Summarizing all these techniques, the paper proposes the Algorithm 1.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode = 1, M **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for t = 1, T **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = \max_a Q(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_t, a_t, r_t, \phi_{t+1})$ from $\mathcal{D}$
        $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta^-) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
        Every a given number of steps, update target weights as $\theta^- = \theta$
    end for
end for

Note that the target values use a second target network that it is the same as the online network, but have different weights. Every a given number steps, the target network is updated with the current weights of the online one. This greatly improves the performance of the algorithm [2].

2.2 DDQN

On a new paper [2], the same authors of the DQN, proposes the DDQN. The target used by DQN at given time-step t is,

$$Y_t^{\text{DQN}} = R_t + \gamma \max_a Q(\phi_{t+1}, a; \theta_t^-). \tag{2}$$

The max operator on equation 2 uses the same values to select and also to evaluate an action, which makes it more likely to overestimate values. For a better visualization, the equation can be rewritten as

$$Y_t^{\text{DQN}} = R_t + \gamma\, Q(\phi_{t+1}, \underset{a}{\text{argmax}}\ Q(\phi_{t+1}, a; \theta_t^-); \theta_t^-). \tag{3}$$

To prevent an overestimation of values, the selection is decoupled from the evaluation, through a method called Double Q-learning [4]. The DDQN version of the former equation is then given by

$$Y_t^{\text{DDQN}} = R_t + \gamma\, Q(\phi_{t+1}, \underset{a}{\text{argmax}}\ Q(\phi_{t+1}, a; \theta_t); \theta_t^-). \tag{4}$$

The difference between them rely on using the online network to choose the actions and the target network to evaluate it. The authors show that this approach reduces overoptimism of the DQN and generally finds better policies. Other than the target values, the algorithm is the same as Algorithm 1.

3 Proposed System

The system created on this work consists of three parts: a simulator, an emulator and a learning agent.

The simulator simulates an environment where the robot interacts with a game controller and can see the game screen with its camera. The chosen simulator was the Virtual Robot Experimentation Platform (V-REP) by Coppelia Robotics [5].

The emulator emulates an Atari game, performing given actions and returning the game screen and score. The chosen emulator was the Arcade Learning Environment (ALE) [6].

The agent tries to learn how to play the games on the emulator through the robot on the simulation. For this it uses a DDQN [2].

A simplified overview of the system is given by Figure 1. We also implemented a simpler architecture without the robot, which can be seen on Figure 2. This was used to replicate the previous results of the DDQN [2] and to serve as base of comparison. For simplification, we call the proposed architecture ALE-Robot and the simpler one ALE-Only.

3.1 V-REP

V-REP [5] is a versatile robotics simulator with realistic physics. In its environment, controllers can be written in many programming languages, interacting with the simulator through a remote API and, as proposed in this project, the Python remote API. By default, it also provides many real-world robots to be used. For the purpose of this project, the provided humanoid NAO robot was chosen.

In the scene constructed for this work, we have three main elements: the robot, the controller and a projector screen. The only element not provided by V-REP was the controller, which was built manually, as it is depicted if Figure 3. It has fours keys and one fire button so that it would be possible to play all Atari games. Each button works by detecting collisions with other objects and, once it detects one, it changes its color and sets a integer flag indicating its pressed state, which is then later read by the remote API.

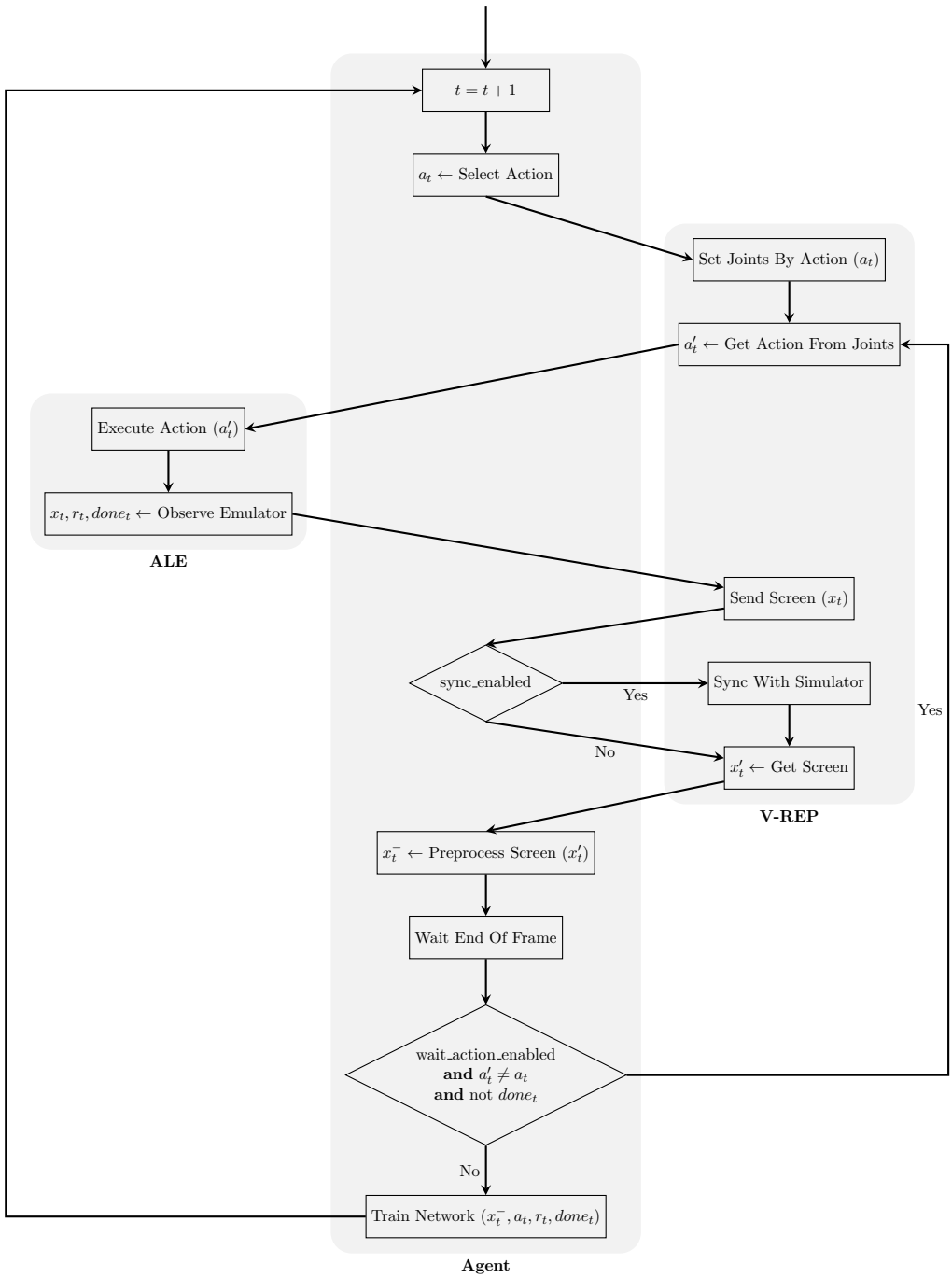

Figure 1: Overview of proposed architecture (ALE-Robot). Interactions between the agent and the simulator are highlighted on the V-REP section, and between the agent and the emulator, on the ALE section. The section Agent highlights actions on the agent itself.

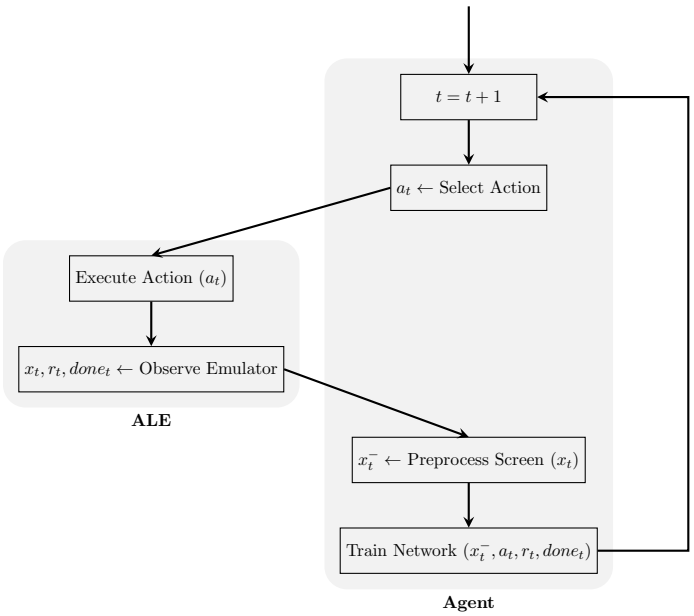

Figure 2: Overview of simpler architecture (ALE-Only) used for experiments without the robot (directly with the emulator). Interactions between the agent and the emulator are highlighted on the ALE section. The section Agent highlights actions on the agent itself.

The NAO robot is placed on a chair with its body in a fixed position, so it will not move and interfere with the training. In front of it, there is a desk with the controller on top of it. The robot is adjusted so that it can reach and press all the buttons moving only its shoulder and elbow joints. This positioning can be seen on Figure 4.

Lastly, there is a projector screen as shown on Figure 5. On this figure, the *projectorVisionSensor* pop-up is displaying the image that the agent is sending to the simulator. Notice that we had to add a black border around the image because V-REP repeats the outermost pixels, giving us a black background around the image on the projector. The *NAO_vision1* pop-up is displaying what is being seen by the robot though its camera, which is being accessed by the agent.

Communication with V-REP is made through the remote API. By default, none of the commands sent to the simulator wait for a reply. As shown on Figure 1, we have a *sync_enabled* variable which enables or not syncing between the agent and V-REP.

When disabled, the simulation runs on real-time and it is not guaranteed that, when sending new commands, older ones will have even been received by the simulator. When enabled, the simulation does not run on real-time. It works by sending a blocking command followed by a trigger to V-REP, making sure all previous commands are received and a new simulation step starts. So the speed of the simulation is dependent on how fast the hardware can run one step.

Normally V-REP uses socket communication for its remote API, but tests showed that this process can be very slow. For faster communication, a GitHub project called

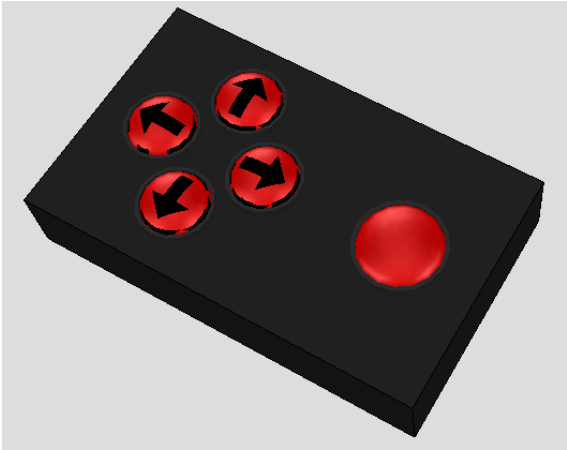

Figure 3: Custom controller with four arrow keys and one fire button.

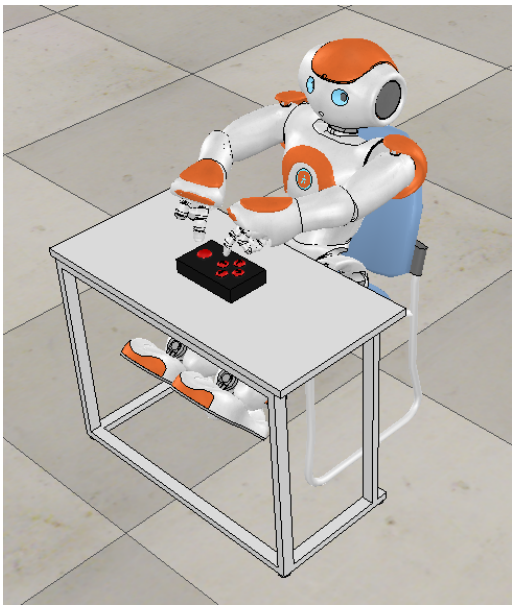

Figure 4: NAO robot positioned on the scene.

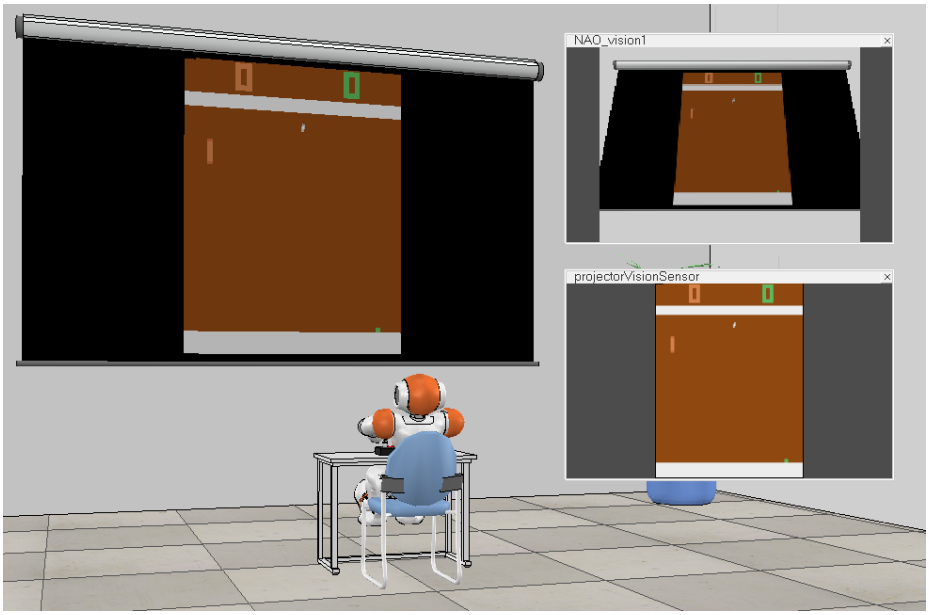

Figure 5: Projector screen. *projectorVisionSensor* displays the image sent to V-REP and *NAO_vison1* displays the camera vision.

vrep_remote_api_shm [7] was used. This project is a modification of the API that adds support to shared memory communication, resulting in dramatically less delay on sending and receiving messages.

3.2 ALE

ALE [6] is an Atari emulator projected to be used as a learning environment of AI agents. It was used by the DQN paper and it has been used by many papers since then.

The emulator provides a set of valid actions which can be used by the agent. When receiving one of those actions, the emulator runs one frame and updates its internal state. This update is fast, so the game basically runs as fast as actions are sent. When training directly with the emulator, this behavior is desired, since it implies faster training. However, for the proposed architecture, it is also important to maintain emulation and simulation running at the same pace.

In the case of synchronization with V-REP disabled, we have an internal timer that tries to make sure the game is running at real-time speed. This is seen by the *Wait End Of Frame* block on Figure 1. When enabling synchronization, the emulation will run at the same speed as the simulation, so it is important to calibrate the simulator time step correctly.

ALE also provides functions to get the current game screen, current total score and whether it reached a terminal state or not. These are used by the agent to observe the emulator state after it performs an action.

3.3 Agent

The agent is responsible for learning how to play the games using a DDQN. The library chosen for implementing the network was TensorFlow [8] running on top of Keras [9]. It was programmed using Python and some of its implementation was inspired by or directly adapted from the GitHub projects Simple DQN [10] and keras-rl [11].

An overview of the ALE-Robot architecture is given on Figure 1. For a given time t the agent first selects an action a_t. For every action, it has a predefined set of positions for the robot's joints which performs this action. This positions are then sent to V-REP. Then, the agent reads integer flags indicating whether the buttons of the controller are pressed or not. So it translates the combination of pressed buttons into the corresponding action a'_t. a'_t is executed on the emulator, returning its new state information, x_t (game screen), r_t (change on game score) and $done_t$ (whether it reached terminal state or not).

A black border is added to x_t and the image is sent to V-REP. In case of synchronization being enabled, we wait for the next simulation step. This also makes sure that we will have the most recent screen when reading the camera. After that or in case synchronization is disabled, the camera is read, giving x'_t.

On x'_t, the first step of pre-processing is extracting the game screen. We first find the projector screen by identifying the biggest black parallelogram on the screen. This is followed by cropping the biggest parallelogram that isn't black on the projector screen, which is the game screen. We project this parallelogram on a rectangle with the size of the

original game screen. The second step of pre-processing is the same used on [1], which is scaling the image to 84×84 pixel and converting it into gray-scale, resulting in x_t^-.

Then, we wait the appropriate time to synchronize emulation and simulation. Another customization option is given by *wait_action_enabled*. If enabled, the agent will read a'_t again until $a'_t = a_t$ or $done_t$ is *true*. This has the effect of making sure that training is only performed after the robot finished executing the action. Finally, training is performed and a new step starts.

In the ALE-Only architecture shown by Figure 2, there is no robot or simulation, so the action a_t is performed directly on the emulator. Also, only the second step of pre-processing is applied directly to x_t. *sync_enabled* and *wait_action_enabled* do not have any effect either.

The implementation is available at `https://github.com/renattou/ALE-Robot`.

4 Experiments

The network model used on all experiments are the same used on [1] and [2]. The first convolution layer convolves the input with 32 filters of size 8 and stride 4. The second layer has 64 layers of size 4 and stride 2. The third convolution layer has 64 filters of size 3 and stride 1. After the convolutional layers, there is a fully-connected hidden layer of 512 units. These layers are activated by Rectifier Linear Units (ReLu). The final layer is a fully-connected linear layer, outputting the Q-values.

Mean squared error is the function used to calculate loss. The network uses RMSprop as optimizer with learning rate of 0.00025, ρ of 0.95 and ϵ of 0.01. The discount rate of future rewards γ is set to 0.99. The exploration policy is a ϵ-greedy policy, with ϵ decreasing linearly from 1 to 0.1 over the first 1M steps.

The experience replay memory is limited to 1M samples, with older ones being discarded. Every state of the game is represented by a history of 4 frames. Every 4 steps, the network is trained with random minibatches of size 32 from the replay memory. And every 10000 steps, the target network is updated. We trained on epochs of 250K steps for 200 epochs, totalizing 50M steps. After each epoch, we tested the agent on 125K steps. Because of time constraints, most experiments were stopped before they finished 200 epochs.

As for ALE, frame skip is set to 4, so the emulator executes the same action 4 times for every action taken. So to run the game at real-time, we have to make sure it runs at 15 frames per second (FPS), which results on 60 FPS when we take frame skip into account. To ensure this, the agent waits until at least approximately 66.7ms have passed since the last frame. ALE's *repeat action probability* is set to 0, with randomness being provided by perform at most 30 random action when restarting a game. *Color averaging* is set to *true*.

On V-REP, we set the timestep to 66.7ms, so that we can synchronize it with the emulator in case *sync_enabled* is set to *false*.

All experiments were performed using the game Pong. This game was chosen for being one of the easiest game to learn, but training with other games would be trivial. The first experiment was performed using the ALE-Only architecture. This enables us to compare results of the proposed architecture with it. The other three experiments are all performed on ALE-Robot, but with different combinations of *sync_enabled* and *wait_action_enabled*.

The hardware used was a machine with az Intel® Core™ i7-7700 @ 3.60GHz, a GeForce GTX 1080 and 16GB of RAM.

5 Results

Results of training on the ALE-Only architecture are displayed on Figure 6. Since the first epoch it is already possible to see improvements on the average reward and average Q value and very few epochs show decrement of these values. After only 50 epochs, the agent learns a quasi-optimal policy, reaching 21 points on most plays, which is the maximum score. From this point until the 200 epochs, results remain mostly the same. Running all epochs took roughly 5 days. These results are similar to the DQN [1] and DDQN [2] papers.

The first experiment with the ALE-Robot used *sync_enabled = false* and *wait_action_enabled = true*. Waiting the action to be performed by the robot made the training very slow, because every step on the network took many steps on the simulator. This effect was worsened by synchronization being disabled, since the delay in the communication implied in 3 to 4 steps until the action was executed and detected by the agent. The results are shown on Figure 7. Approximately 2 epochs were ran per day. After 15 epochs and 7 days of experiments, the average reward of the policy learned was worse than the random policy and the average Q value was stopping improvement, so training was terminated.

The next experiment used *sync_enabled = false* and *wait_action_enabled = false*. Without having to wait for actions to be executed by the robot, training was slightly faster, with approximately 3 epochs running per day. However, results did not improve, as it can be seen on Figure 8. The average reward after 17 epochs and 5 days was worse than the random policy average. The average Q value only got worse with every epoch, starting to stabilize towards the ends. Without many signs of improvement, training was interrupted.

The final improvement on ALE-Robot used *sync_enabled = true* and *wait_action_enabled = false*. Running on synchronization with V-REP allows for running the simulation at the maximum speed that the hardware can since we do not have to run the simulation on real-time anymore. On the hardware used for training, we could run approximately 4 epochs per day. After 25 epochs and 7 days, training was stopped. From the Figure 9, we can see that results were similar to the two previous experiments.

6 Conclusion

Results of the ALE-Only experiment shows that the implementation of DDQN is working as expected. In spite of this, all three ALE-Robot experiments did not provide good results.

The two unsynchronized experiments were inherently more difficult, because the agent had also to learn the delays in communication besides learning how to play the game. This is also not desired because the agent would learn to play too specifically for V-REP and the available hardware, making it more difficult to transfer it to a real robot or even another machine. Waiting for the actions to complete could also make learning more difficult, because two consecutive observed screens could potentially be multiple steps away, possibly breaking the MDP.

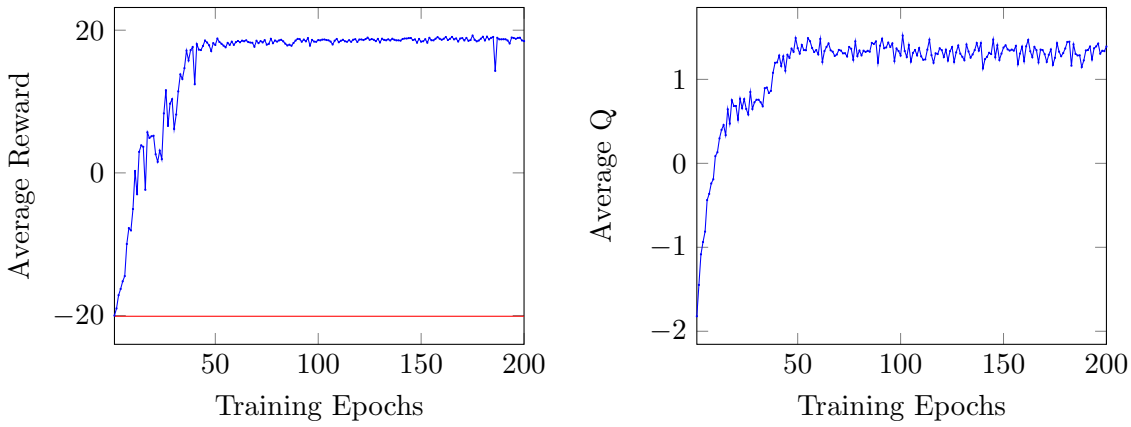

Figure 6: Average reward and Q values when training on ALE-Only. 200 epochs were ran. The red line represents the average reward of the random policy.

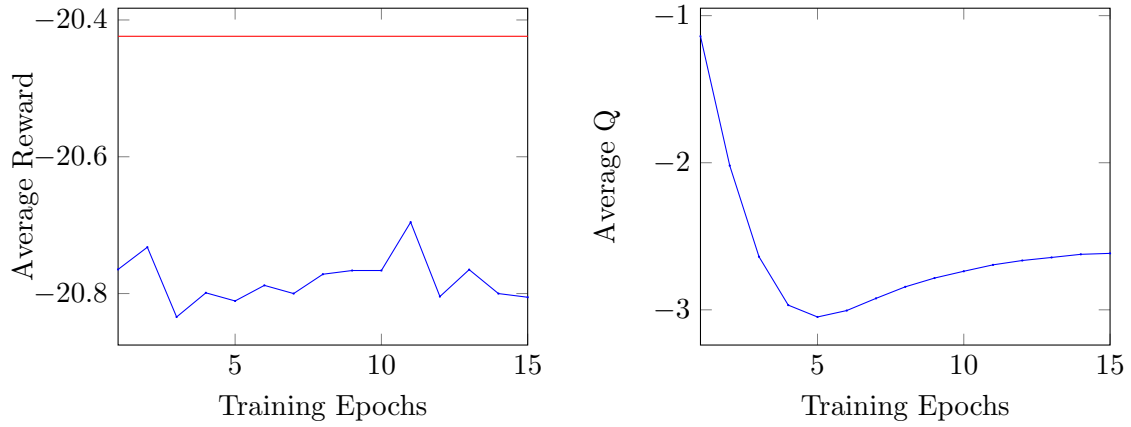

Figure 7: Average reward and Q values when training on ALE-Robot with $sync\_enabled = false$ and $wait\_action\_enabled = true$. 15 epochs were ran. The red line represents the average reward of the random policy.

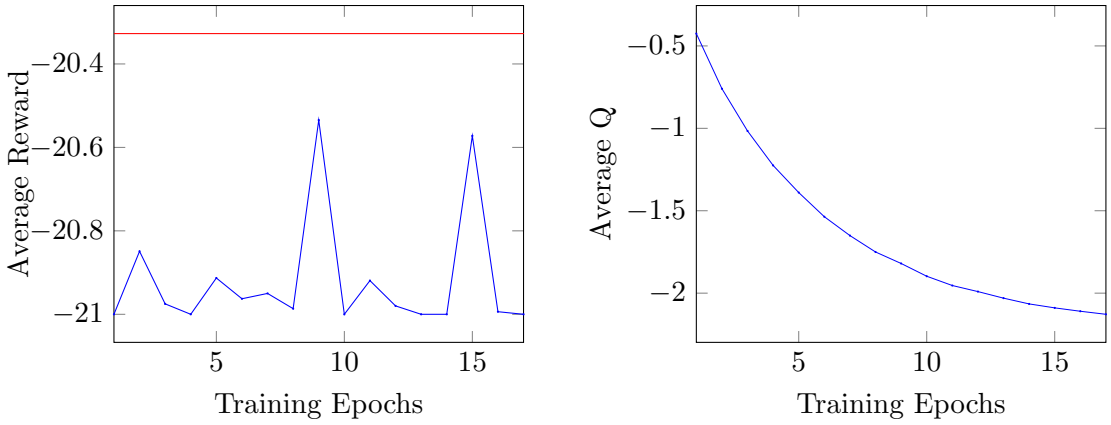

Figure 8: Average reward and Q values when training on ALE-Robot with $sync\_enabled = false$ and $wait\_action\_enabled = false$. 17 epochs were ran. The red line represents the average reward of the random policy.

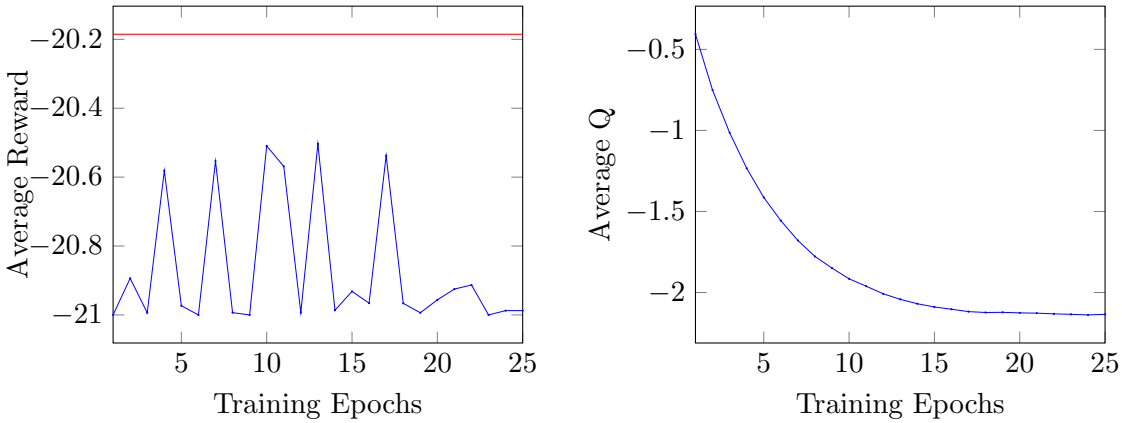

Figure 9: Average reward and Q values when training on ALE-Robot with $sync\_enabled = true$ and $wait\_action\_enabled = false$. 25 epochs were ran. The red line represents the average reward of the random policy.

Given these considerations, the apparent best solution would be running synchronously and without waiting for actions to be performed. Another advantage of this approach is that we can run the simulation as fast as the hardware can handle. One of the main difficulties of testing the proposed architecture is the speed of training, which is mainly held back because of the simulation. So given sufficiently good hardware, training and testing changes in the architecture could be much faster.

Although looking promising, this last approach also did not have good results. Given the time constraints, we had to terminate the training after a few epochs, although it lasted 7 days. So it is possible that, with sufficient time, this and the previous two experiments could have better results.

7 Future Work

Firstly, it is important to test ALE-Robot for longer periods of time. Considering the last synchronous approach, this could be feasible if more computational power was available, such as in a cluster. This would also open the possibility of testing different values of the hyperparameters. Another option is to study how to modify the model of the network to better handle delays in the results of its actions.

Then, the next step would be transferring the intelligence to a real-world robot. The applications for this method are many, but a simple one is creating a game-playing companion robot.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[2] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[3] Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon School of Computer Science, 1993.

[4] Hado V. Hasselt. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010.

[5] M. Freese E. Rohmer, S. P. N. Singh. V-REP: a versatile and scalable robot simulation framework. In *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013. `http://www.coppeliarobotics.com/`. Version 3.4.0.

[6] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial*

Intelligence Research, 47:253–279, jun 2013. https://github.com/mgbellemare/Arcade-Learning-Environment.

[7] Benjamin Navarro. vrep_remote_api_shm. https://github.com/BenjaminNavarro/vrep_remote_api_shm, 2017.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. https://www.tensorflow.org/, 2015. Version 1.4.0.

[9] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015. Version 2.1.1.

[10] Tambet Matiisen. Simple dqn. https://github.com/tambetm/simple_dqn, 2015.

[11] Matthias Plappert. keras-rl. https://github.com/matthiasplappert/keras-rl, 2016.