CMPE 492

A Comparative Study of Simulators For Sim-to-Real Deep Reinforcement Learning

Midterm Progress Report

Mertcan Özkan

Advisors:

Emre Uğur - Suzan Ece Ada

08.04.2023

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Broad Impact

This project aims to compare the accuracy of two physics-based simulators, namely MuJoCo and Pybullet, for a sim-to-real deep reinforcement learning transfer task. As mentioned in the survey [1] , collecting data to train a policy for a robot in the real world can be very costly and inefficient. Moreover, further safety risks can arise during the learning progress where the robot might perform an unwanted action that can danger the environment and people around it. Therefore, problems posed by these issues limit or prevent the possibility of training a robot in the real world. That is the reason why using simulations for training of the robots is very important for safety. Simulations present a safe environment for the robot to discover possible actions within the environment and also provide a faster way of collecting data. But using simulations can bring new problems as well. As explained in [1],using simulations can also create a gap between the performance of the robot in the simulated environment and the real-world application. In this aspect, it is important to analyse different simulators and understand the difficulties they can create when transforming the policies trained on the simulations to real world. This project aims to evaluate the polices trained on different simulators on a real UR10 robot.

1.2. Ethical Considerations

As pointed in [2], the ethics of AI and robotics is a very young field within applied ethics, with significant dynamics. The main ethical concerns in AI can rise from privacy or misinformation concerns which revolve around issues on collection of data by artificially intelligent systems and misuse of the collected data. For the scope of this project these concerns do not pose any problems since data collected by the robot in the simulation do not pose any privacy issues. However, for the second part of the project, the trained policies on the simulations will be tested on a real

UR10 robot, which can raise some safety concerns for the testing. These concerns are explained in the risk analysis part. As well as comparing the accuracy of two physics-based simulators, this project also provides insights on problems that can arise when transferring and using a model that was trained on the simulation on the real robots and ways to overcome these difficulties.

2. PROJECT DEFINITION AND PLANNING

2.1. Project Definition

As mentioned in the broad impact part, aim of this project is to compare the accuracy of MuJoCo and Pybullet, for a sim-to-real deep reinforcement learning transfer task. A UR10 robot will be designed in both simulators. Then, a policy will be trained for each simulator using a policy gradient algorithm, namely Deep Q-Learning with one and two hidden layers. The robot will be trained for a reacher task. Finally, a comparative evaluation study of the policies trained in different simulators will be conducted on the real UR10 robot.

Definition of the Task
The task is a simple objective that involves a goal position. The goal position is a randomly generated position on the x-y axis within the reach of the robot's gripper. For each episode of the simulation, the robot's aim is to reach this randomly generated the goal position in every episode. For each step of the episodes in the simulation, the robot will be receiving a reward based the distance between the goal position and robot's gripper position. The rewards will get higher as robot get closer to the position.

2.2. Project Planning

2.2.1. Project Time and Resource Estimation

Project will be separated into two parts: First, the Ur10 robot will be simulated for the task described in the project definition part using Deep Q-Learning algorithm in MuJoCo and Pybullet simulators, then trained policies for the UR10 robot will be evaluated. Then, for the second part of the project, a comparison will be made on the policies trained in the simulators using a real UR10 robot. The midterm report will be covering the first part of the project that is simulating the task environment of the

robot; which will involve defining the state, action and rewards for the policies. Th final report will also involve the results of the test conducted on the real robot and challenges encountered during the transfer from sim-to-real.

2.2.2. Success Criteria

For the first part of the project, the main aim is to be able to simulate the UR10 robot successfully and training the robot successfully such that the robot reaches randomly generated goal positions and collects as much rewards as possible. Since aim of the reinforcement algorithms is to maximize the rewards the agent gets over time, the running average for the last two hundred episodes of cumulative reward graphs will be created to assess the success of the polices over four thousand episodes. For the second part, the models trained in the simulations will be tested on the real UR10 robot. To achieve this aim, a new environment that functions in the same way as the environment used in the simulations will be created to communicate with the robot. Then the simulators will be compared for a given case.

2.2.3. Risk Analysis

For the second part of the project, the trained policies on simulators on Mujoco and Pybullet will be tested on a real UR10 robot. Which can give rise many risks during testing. There can be sensor corruptions on the data that the robot will be receiving that can cause the robot to act in an unexpected way. Another safety risk is that the policies that were trained on the simulators might have errors and can cause the robot to act differently than expected.

3. RELATED WORK

There are several studies on Sim-to-Real Deep Reinforcement Learning Tasks. [1] presents a survey for Deep Reinforcement Learning for Robots. In the paper, the fundamental background behind sim-to-real transfer in deep reinforcement learning is covered. Then an overview of the main methods used are given these are namely; domain randomization, domain adaptation, imitation learning, meta-learning and knowledge distillation. [3] demonstrates a single sim-to-real approach for the 4-DoF robot arm. In [4] sample-efficient deep bayesian reinforcement learning and a model-free deep reinforcement learning is compared from both simulations and real-world experiments. In the survey [5] benefits and applications of different physic simulators for different use-cases are evaluated and these simulators are revived based on their performance.

4. METHODOLOGY

Reinforcement learning (RL) is a machine learning method where the main aim is to make the agent learn the best behaviour in a given environment. The agent receives a feedback from the environment after performing an action. The feedback is a reward value that determines if taking that action was beneficial for reaching determined the goal state. Therefore, in general, the agent is able to perceive and interpret the results of the actions it took within the environment and it is expected to learn good a policy based on these feedbacks by trial and error. Also, the rewards given as feedbacks to the agent come from a reward function that is determined according to the goal position before the beginning of the training phase. It is also important to note that action taken in a given state can also influence the rewards received from later states as well.

Algorithms Used

DQN (Deep Q-Network) is a reinforcement algorithm that relies on a Q-function ,that is a state-action value function of a policy, which is used to find an expected return of rewards from a state if an action 'a' is taken for a state 's' then the same policy is followed for the next state resulting from that action until the end of the episode. To estimate the Q-values of actions, a neural network is trained to as a function approximator. [6] The same method is followed for training the agent given in [6] but with a small modification. Instead of clipping the loss function, the gradient is clipped between -1 to 1 using Huber loss function.

5. REQUIREMENTS SPECIFICATION

The required behaviour expected from the agent is to find the optimal policy to reach the goal given environment and the goal position. The goal position can be arbitrarily chosen but it should be within the reach of the agents gripper. Providing

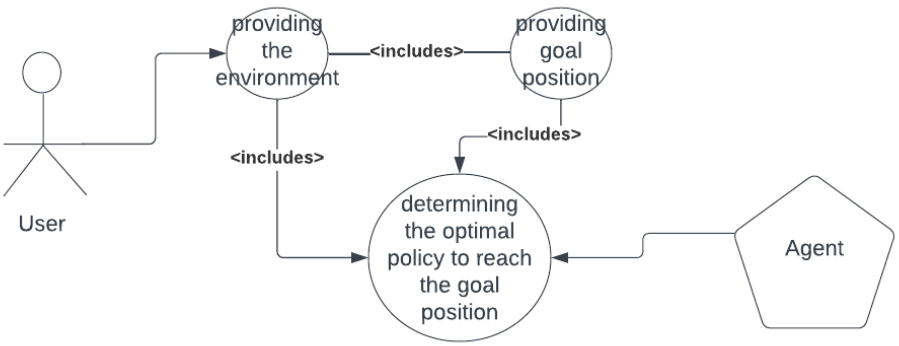

Figure 5.1. Agent use case

the environment means that the user should give the agent a viable reward function and the action-space of the agent should be determined in such a way that agent should be able to reach the goal position within fifty steps.

6. DESIGN

6.1. Information Structure

ER Diagram for reinforcement deep q-network can be seen below.

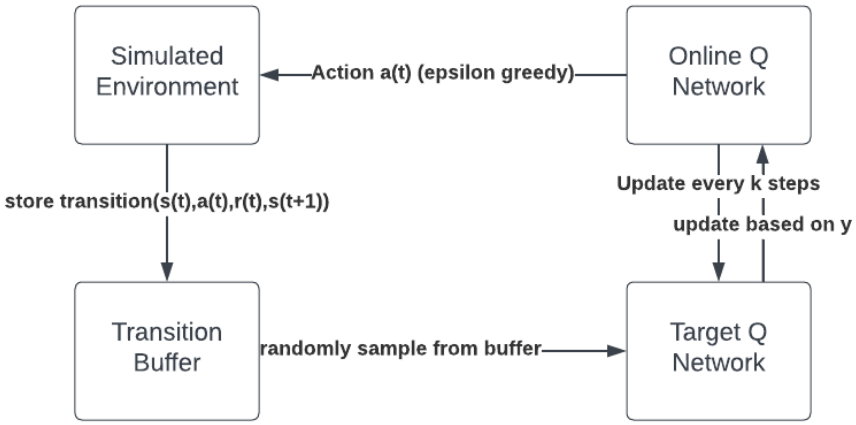

Figure 6.1. ER Diagram of dqn

6.2. Information Flow

Activity diagram for reinforcement learning can be seen below.

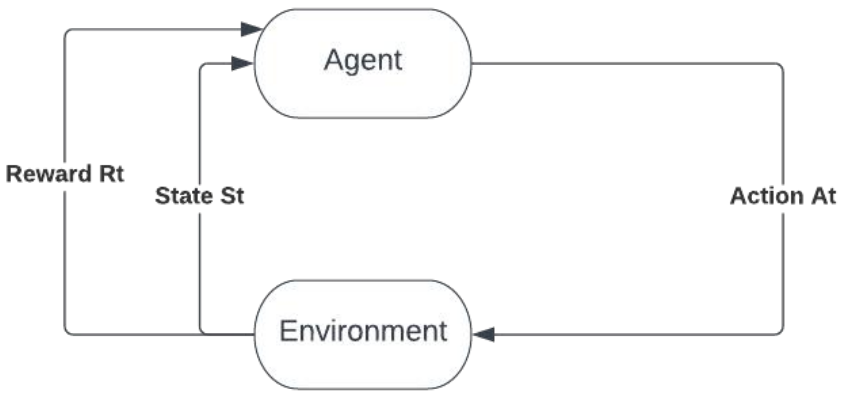

Figure 6.2. Activity diagram for reinforcement learning

6.3. System Design

Module diagram for deep q network can be seen below.

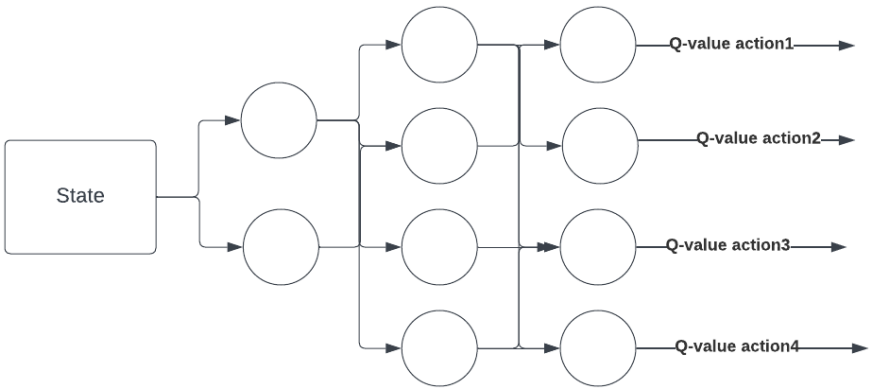

Figure 6.3. Module diagram for deep q network

7. IMPLEMENTATION AND TESTING

7.1. Implementation

In this project, dqn algorithm was used to train a policy for the environment. This algorithm is implemented using the machine learning framework called Pytorch[1] . Initially, during the midterm report submissions, the action-space for the robot consisted of four actions; right, left, up and down, which represents the robots grippers movement on the cartesian plane , but then the actions were redefined. You can check figure 7.1 for the new actions and read the "Action Space" part under environment configuration. These actions correspond a coodinate in the x-y plane which is added to the position of the gripper to move the robot. Two neural networks consisting of one hidden layer and two hidden layers are trained for the dqn with activation function ReLu. During the training, agent performs random actions for one hundred episodes and collects experiences from the environment which consists of (old, action,reward, done, new) pairs. Then the target network and the q network are trained for four thousand episodes in both Mujoco and Pybullet simulators. For simulating the environment, a modified version of Alper Ahmetoğlu's code was used. [2] [3] The hyper-parameters used for the training can be seen below. Parameter "UPDATE FREQ" determines the number of step between each update of the Q network while "TARGET NETWORK UPDATE FREQ" determines the number of steps for each update on the target network. In the midterm report, it can be seen that proximal policy optimization(PPO) was also used for training but later a dqn network with two hidden layers was used and trained instead of the PPO algorithm.

Environment Configurations

The Reward Function

[1]More details on Pytorch: `https://pytorch.org/`
[2]For Mujoco simulation: https://github.com/alper111/mujoco$_m$*enagerie*
[3]For Pybullet simulation: https://github.com/alper111/DeepSym

As mentioned in the project definition part, the environment used is a simple environment that only consists of goal position that is randomly generated at the start of each episode of the simulations. The reward function is defined by one over the l2-norm of the distance between the goal position and the position of the robots gripper:

$$f(X) = 1/(||X||_2) \mid \forall x_i \in X, x_i = goal\_position_i - gripper\_position_i$$

The terms goal_position and gripper_position are two dimensional array that represent the position of the goal and the gripper in the Cartesian plane. Therefore, the average reward received by the agent in an episode increases if the agent manages to reach the goal position in as few steps as possible.

Termination and Truncation Conditions

In the simulation, an episode can end in two different ways. The first way is truncation which means that the robot has reached the maximum number of steps allowed with in a episode. When the robot reaches this limit, environment is reset even if the goal position is not reached. The second way for an episode to end is termination, the episode ends when the gripper position reaches the goal position. Since the predefined actions for the robot is discrete, it might be hard for the robot to set its gripper position to the desired goal position exactly, therefore, a goal threshold was determined for the environment. When the distance between the goal position and the gripper position is very small, meaning the distance is within the preset goal threshold, the episode is terminated. In the first part of the project this threshold value was set to 0.01. Therefore, models trained in the midterm progress report had the threshold value of 0.01.

Environment Configuration Updates During Real Robot Testing

The Reward Function

The reward formula given above was used for the training of the models during the first

part of the project and resulting reward graphs were added to the midterm report of the project. However, during testing; a problem was detected for the reward function given above: Instead going to directly to the goal position, the robot choose to come close the goal position but it did not enter the goal position as expected which would have terminated the episode, it basically goes around the goal position from a distance. After examination, we realized that going to the goal position and terminating is not very advantageous for the robot since the episode terminates early. Instead, the robot comes to a close distance to the goal position and decides to wait there to gain more and more rewards in each episode. To solve this problem, the reward function was updated as follows:

$$f(X) = -1 * ||goal\_position - gripper\_position||_2$$

In the first reward function, a positive reward was generated in each episode which was one over the distance between gripper and goal position, in the updated version, the reward function generates a negative result which is the distance between goal position and gripper position multiplied with minus one. With the updated reward function robot always receives a negative reward value at each step so it is forced to reach the goal position as fast as possible. This modification solves the problem described above. The models were trained again with the updated reward function in the simulations for testing.

Termination Condition

As mentioned above, the goal threshold used for training during the midterm report was set to 0.01. During the testing of the models, it was seen that the threshold value might prevent an episode to terminate on time. It was seen that the data coming from the real UR10 robot has some noise in it. For instance, even when the robot is just standing still, the gripper positions sent by the robot can vary up to 0.04 in terms of magnitude in some cases. This causes the robot to move several times on the goal position to terminate since it tries to get to a smaller distance but it fails several times because of the noise problem of the received gripper position. It finally terminates

when it receives a gripper position that is closer to the goal position after several tries on the goal position. To solve this issue, the threshold value was increased to 0.02, which eliminates most of the extra unnecessary steps taken by the robot during testing. The models were trained again with this new threshold value.

Action Space

As mentioned above, the models trained for the midterm report had an action space that consisted of four actions: up down right an left. During testing, it was discovered that in the code used for training the predefined actions were wrongly defined which caused the robot to zigzag when it tried to reach the goal position. The actions were redefined so that there are now eight directions that correspond to movement in the Cartesian space. The models were trained for the newly defined action space for the testing. The newly defined actions can be seen below.

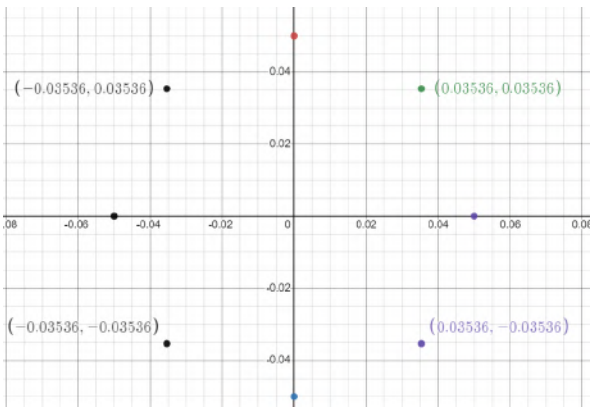

Figure 7.1. Action space of the environment

7.2. Testing

Two methods are used to test whether the training of the policies are done correctly. To test if the models for different algorithms are built correctly, the same models are first used to train a policy on a simpler and well-known environment to check if the given model can produce the expected results. Gym's 'Cartpole' environment is chosen for the testing which corresponds to the version of the cart-pole problem described by Barto, Sutton, and Anderson in [7]. The action-space of the environment only consists of two actions, left and right and the observation space of the environment consists of

Table 7.1. Hyper-parameters used for Dqn

	Header 1
GAMMA	0.99
BATCH SIZE	32
BUFFER LENGTH	10000
EPSILON	1.0
MIN EPSILON	0.1
EPSILON DECAY	0.001
EPSILON DECAY ITER	100
LEARNING RATE	0.0005
UPDATE FREQ	4
TARGET NETWORK UPDATE FREQ	1000
N ACTIONS	4

four variables, namely "Cart Position", "Cart Velocity", "Pole Angle" and "Pole Angular Velocity". The goal is to prevent the pole from falling down as long as possible and the +1 reward is received by the agent for every step taken until the pole falls down.

Secondly, training of the models for the environment and the goal function of the UR10 robot in the simulations described in the project definition and methodology parts, cumulative reward graphs are generated during training process for every two hundred episodes. The cumulative reward graph for the policies trained can be seen in the results part. As explained in the implementation part, environment configuration was changed after the progress report. The graphs are for the newly trained models in the updated environment.

Testing on Real UR10 Robot

Finally, the policies trained in the simulations are used for the real UR10 robot. An

environment file that behaves in the same way in the simulation environment was created as described in the environment part. The only difference is that now there will be four goal positions in the environment meanwhile simulated environment only had one goal position. The robot receives these goal positions one by one. When the robot reaches the first goal position, the next goal position is given instead of termination, the episode only ends when all four goal positions are traversed by the robot or the maximum time steps is reached. These goal positions are predefined and are not randomly generated. The environment used for the real robot can be seen below. Keep in mind that this environment was not used for the simulations so the figure below is only used to show the environment configuration used to test the trained models on the real UR10 robot. The implementation and problems encountered when testing on the real UR10 robot is given in the deployment part.

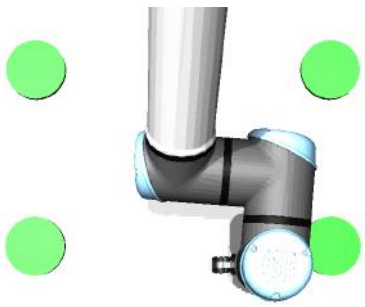

Figure 7.2. Representation of the environment used on the real UR10 robot

7.3. Deployment

For the part two of the project, the policies trained on the simulations will be deployed on the real UR10 robot. A new environment was designed for interacting with the robot using the models trained in the simulations. This environment basically works in the same way like the environment in the simulations as defined in the implementation part. Therefore, as in the simulation, the model receives the goal position, the gripper position and a reward value and produces the same action values as defined in the implementation part. Then the generated action a values is used to move the relative position of the robots gripper.For interacting with the robot, Muhammet Hatipoğlu's code was used. [4] To get the value of the grippers position,

[4]https://github.com/ocg2347/UR10$_{d}$*ockerized/blob/master/scripts/gamepad*$_{c}$*ontrol.py*

"get_current_state" method was used, which returns the position of the robots arm in terms of Cartesian coordinates in the pose matrix. Another method used is the "MoveArm" method, for computing target position, we simply add the action value to the current position of the robot. Then this method is used to move the robot to the target position.

In the simulation environment, a new episode starts when we reach the goal position or maximum number of steps is taken by robot. Then, another episode is rendered where the robots gripper starts from the same starting position always. Therefore, in the real UR10 robot, it is also necessary to start from the same starting position also when a new episode starts. To solve this issue, a new method is added to the environment file called "return_to _pos". Every time the robot takes a step, the joint values of the robot are saved in a file called "traj.txt". Therefore; at the end of the episode, we will have the entire trajectory that the robot follows in the episode saved in the file "traj.txt". When the episode ends, the values in this file are executed in the reverse order which makes the robot follow the entire trajectory in the reverse order, therefore we return to the starting position before the next episode starts.

Controller Configuration

To run the environment code, the following configuration is followed: First, "program robot" button is selected form the first figure, then load program is selected from the second figure. The "ros driver" is selected from the shown list. To initialize the robot, the on button and the start button is selected. To connect the computer to controller, ip of the computer is given to the controller from program-externaş control section in the fifth figure. You can check your computers ip by running the "ifconfig" command from our terminal. Pressing the "move" button opens the screen shown in the sixth figure. Here you can select the "home" button and choose the "auto" button to move the robot safely back to its original position before closing the controller. To move the robot to the starting position of the episodes, a file called "init.txt" is used. This file contains the values of the joints of the robot that were recorded when moving the robot manually from the the orignal position down to the starting position of the robot. This file basically moves the gripper down so that it points to the ground.

Figure 7.3. UR10 controller screens

Problems Faced

After creating the environment file described above, dqn models are loaded to a file called ”dqn.py” which basically uses the environment file to interact with the robot. Actions generated by the model is also fed back to the environment using the ”step”

function in the environment file. When using the models trained in the simulation, several problems occurred.

The Noisy Data

As explained in the implementation part, the data coming from the robot is noisy therefore, the gripper position coming from the robot can change even the robot does not move. However, such a problem does not exist in the simulation environment since there is no noise in the simulation. This can also cause the robot to end up in a slightly different position than expected after executing an action. The models are expected to move to the goal position even in this noisy environment and choose actions accordingly. Therefore, models with higher robustness is expected to perform better when such unexpected changes occur.

Defining the task space

As explained above, we rely on the data coming from the robot which gives the Cartesian coordinates of the tip of the robot. Then the robot moves in the task space based on the actions generated. In order to make sure that robot does not behave unexpectedly while performing the actions, the task space limits have to determined for the real UR10 robot. This means that robot is not allowed pass certain limits so that it does not try to go to a position that is unreachable by the robot. Also it was seen that when the robot tries to reach a position that is close to the physical limits, the robot can move in high speeds and make very sharp turns.

To determine the task space, the gamepad_controller code was used in the Muhammet Hatipoğlu's code. This code basically allows the user to move the robots gripper through a controller. To determine the safe task space limits, this controller was used to move around the robots arm and determine the limits that are safe for the robot. The following limits were found for the x-y coordinates:

$$x \in [-1.1, -0.60] y \in [-0.2, 0.50]$$

Another problem arises when the newly created task space is used. The simulations

that were used to train the models have different task space limits than the real robots task space limits. Therefore, the simulations task limits were readjusted for the real robots task space limits. Then the models were retrained for the new task space values. The codes implemented can be seen in the footer below.[5]

[5]https://github.com/mertcan-ozkan/cmpe492

8. RESULTS

The running average graph for deep q-network for Mujoco can be seen below. After four thousand episodes, it can be seen the policy converges for randomly generated goal positions. The results of the tests on the real UR10 robot can be seen below in

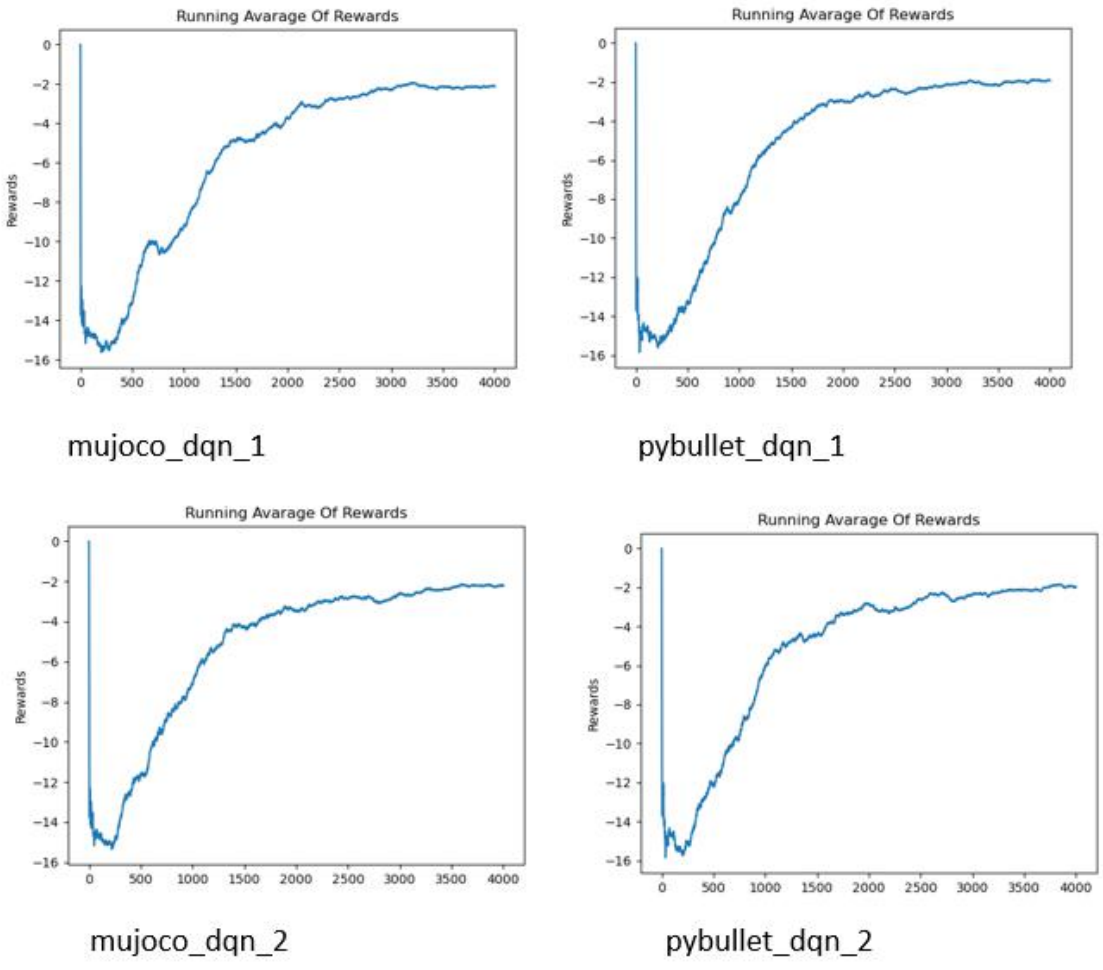

Figure 8.1. Running Avarage Graphs for deep q-network with one and two hidden layers

figure 8.2. All four models were tested for the given environment for fifteen episodes. At each episode, robot traverses all four goal positions, when the episode ends; total steps taken by robot (episode steps), total reward collected by the robot within the

episode (rewards) and time it takes for robot to complete the episode is recorded.

	Episode steps(mean)	Episode steps(std)	Rewards (mean)	Rewards (std)	Episode time (mean)	Episode time (std)
Dqn – 1 layer (mujoco)	38.2000	1.5144	-6.8116	0.3878	19.0944	0.7452
Dqn – 1 layer (pybullet)	34.4000	0.4899	-6.2309	0.1544	17.1942	0.2379
Dqn – 2 layer (mujoco)	34.3333	1.1926	-6.5041	0.1744	17.1609	0.5875
Dqn – 2 layer (pybullet)	34.2000	1.3266	-6.4511	0.1991	17.0942	0.6548

Figure 8.2. Results of the tests on the real UR10 robot

Below is shown the UR10 robot in Mujoco and Pybullet simulations.

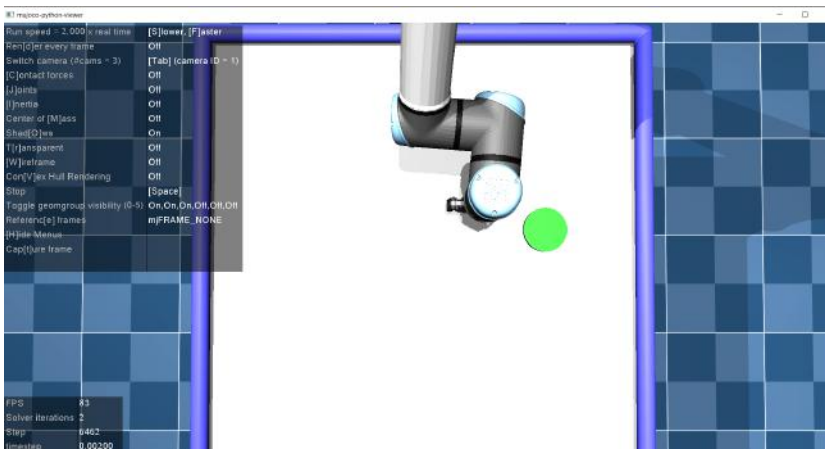

Figure 8.3. UR10 in Mujoco

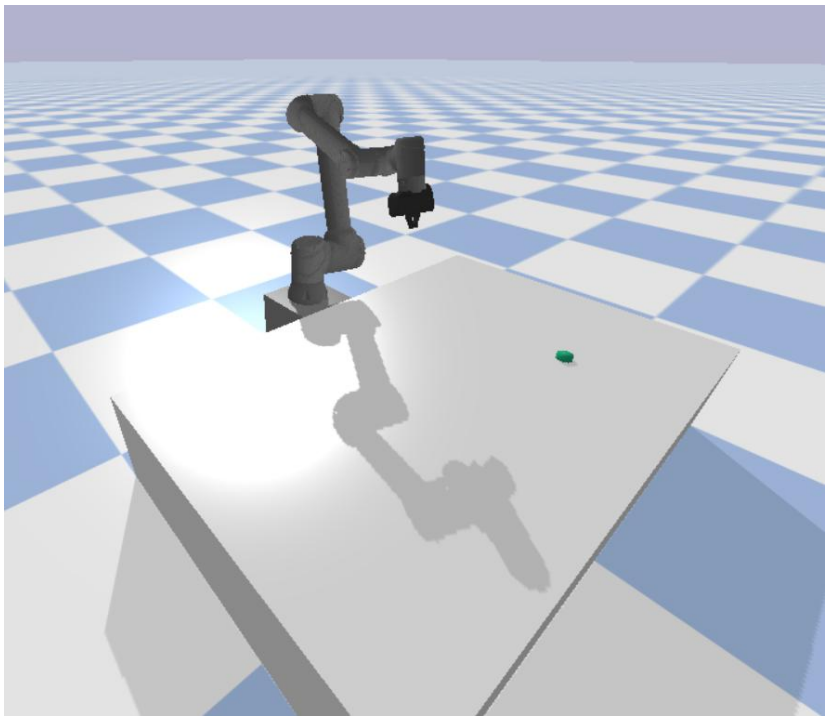

Figure 8.4. UR10 in Pybullet

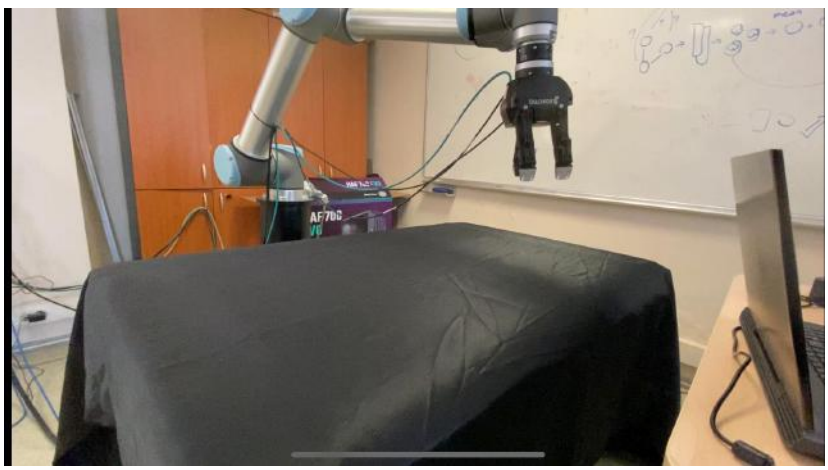

Figure 8.5. testing in real UR10

9. CONCLUSION

In this study, reinforcement learning algorithm, deeq q learning is implemented in the simulations Pybullet and Mujoco for the UR10 robot. These policies are then tested by checking the rewards that they were able to collect after the training. Also,correctness of the algorithms were also checked using a simpler well-known environment in the Gym library called 'CartPole'. From the results, it can be seen that both trained models for the both simulations produces results that can vary between episodes even though the positions of the goals or the staring position of the robot does not change between episodes. One of the main reasons for such a difference comes from the noisy data coming from the UR10's sensors. This noise can cause robot to end up in a slightly different position than in the simulated environment. In this noisy environment, it can be seen that algorithms trained with Pybullet gets higher rewards ,completes the episodes in lesser steps and in a shorter time period.On the other hand, it seems that dqn with an additional layer does not provide clear additional advantage in terms of robustness. The same experiment can be performed to see if adding more than one additional hidden layer has an effect on robustness.

Future Work

The task the robot is expected to perform is a simple problem. The same experiment can be repeated for a more complex task. For example, the new task can be modified so that there will be an additional object in the environment and the aim will be to push the object to the goal position.For the same task, an additional algorithm can be used such as proximal policy optimization (PPO).


BOĞAZİÇİ ÜNİVERSİTESİ
1863

REFERENCES

1. Zhao, W., J. P. Queralta and T. Westerlund, "Sim-to-Real Transfer in Deep Reinforcement Learning for Robotics: a Survey", *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 737–744, IEEE, 2020.

2. Müller, V. C., *Ethics of Artificial Intelligence and Robotics*, Metaphysics Research Lab, Stanford University, 2021.

3. Lin, Y., J. Lloyd, A. Church and N. F. Lepora, "Tactile Gym 2.0: Sim-to-real Deep Reinforcement Learning for Comparing Low-cost High-Resolution Robot Touch", , 2022.

4. Huang, J., Y. Zhang, F. Giardina and A. Rosendo, "Trade-off on Sim2Real Learning: Real-world Learning Faster than Simulations", *2022 8th International Conference on Control, Automation and Robotics (ICCAR)*, pp. 95–100, 2022.

5. Collins, J., S. Chand, A. Vanderkop and D. Howard, "A Review of Physics Simulators for Robotic Applications", *IEEE Access*, Vol. 9, pp. 51416–51431, 2021.

6. Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg and D. Hassabis, "Human-level control through deep reinforcement learning", *Nature*, Vol. 518, No. 7540, pp. 529–533, Feb. 2015.

7. Barto, A. G., R. S. Sutton and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-13, No. 5, pp. 834–846, 1983.