Deep RL Arm Manipulation Project

Author: Roberto Zegers Rusche
Date: March 15h, 2019

Introduction

The main objective of this project is to get a robotic arm to perform a task by the means of using Deep Reinforcement Learning. This will serve to demonstrate the skills, competences and knowledge in this field gained by the author through the Udacity's "Robotics Software Engineer Nanodegree Program". In order to be carried out, this project is implemented in a Gazebo simulated environment using a C++ API to run a RL Agent provided as part of the classroom material.

To demonstrate the RL pipeline two different goals will be considered separately:

- GOAL 1: Any part of the robot arm touches the can (with at least 90% accuracy over 100 runs)
- GOAL 2: The gripper base touches the can (with at least 80% accuracy over 100 runs)

The scope of this work includes setting the reward function and fine tuning hyperparameters in order to accomplish both of these goals successfully.

Background

Reinforcement Learning is an area of Machine Learning, concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward [1]. As encountered in nature, human or animal can learn by interacting with their environment, evaluate the effect, and adapt to respond to changes to ultimately achieve their goals. Reinforcement Learning is an area of machine learning which utilizes such learning machinery. It aims to find an optimal policy to achieve the goal by interacting with the environment in absence of explicit teachers [2]. In that sense RL is a completely different way of looking at a problem than conventional programming. In traditional programming all steps needed to solve a task are explicitly provided to the system. They are comprised of a limited set of instructions and therefore limit how much a program can do to solve tasks or problems. In contrast, due to their nature, reinforcement learning algorithms can perform better in more ambiguous, real-life environments rather than from the limited options of explicitly programmed software [3]. Deep Reinforcement Learning (reinforcement algorithms that incorporate deep learning) improves the capabilities even further. In Deep Reinforcement Learning, image recognition is used to recognize an agent's state; e.g. the joint positions of a robot arm, or the road in front of an autonomous vehicle. And given that image that represents a state, a convolutional network can rank all actions possible and find the best to perform in that given state.

Joint Control and Reward Functions

Joint control

An arbitrary rule is needed to decide on how the action array translates to movement (joint increase or decrease) of the robotic arm. In its current implementation, two choices for controlling joints are possible: either control the velocity of joints or control the position of joints.

Controlling the position of joints was chosen, as it seemed more adequate as the task at hand requires accurately touching an object of interest, independently from the time required, so positional accuracy is more important than speed.

```
// Set joint position based on whether action is even or odd
float joint = 0.0;
if ( action % 2 == 0)
{
    if(DEBUG){printf("action is even \n");}
    joint = ref[action/2] + actionJointDelta;
}
else
{
    if(DEBUG){printf("action is odd \n");}
    joint = ref[action/2] - actionJointDelta;
}
```

Reward functions

In order to get the robotic arm to archive goal nr.1 a reward system was put in place that issues a positive reward every time a collision is detected by the can. Because the reward is triggered in the event that any object collides with the can, it issues a reward regardless of if the can was touched by the robot arm or the robot gripper. This reward function is triggered as soon as the onCollisionMsg callback function is executed and it also ends the training episode.

```
if (GOAL == 1)
    {
    // check any collision with the can
    bool anyCollisionCheck = (strcmp(contacts->contact(i).collision1().c_str(),
COLLISION_ITEM) == 0);
    if (anyCollisionCheck)
        {
            rewardHistory = REWARD_WIN;
            newReward   = true;
            endEpisode = true;
            return;
        }
    }
```

In order to get the robotic arm to archive goal nr.2, the reward system was modified to checks for a collision only on the gripper base. Tests were conducted verify that a collision of the gripper with the ground plane does not trigger a reward. If that would have been the case the check condition would need to be expanded with the condition to check both for a collision detected by the gripper base and a collision detected by the can.

```
else if (GOAL == 2)
    {
    // check any collision with the gripper base
    bool gripperCollisionCheck = (strcmp(contacts-
>contact(i).collision2().c_str(), COLLISION_POINT) == 0);
    if (gripperCollisionCheck)
        {
            rewardHistory = REWARD_WIN;
            newReward  = true;
            endEpisode = true;
            return;
        }
    }
```

Also note that in both reward functions after a collision is detected the reward is issued and a new episode is started. This is very important, because otherwise the simulation would allow the robotic arm to touch the object, move away from it, touch it again and continue to get more and more rewards. It would exhibit this behavior because its entire purpose is to get the highest reward possible [4].

In addition, the algorithm alerts in the case the gripper hits the ground. This is done using the grippers bounding box attribute. In the event of such a collision, a negative reward is issued and a new episode is started.

```
bool checkGroundContact = (gripBBox.min.z <= groundContact) ? true : false;
// check if the gripper is hitting the ground or not using its bounding box
if(checkGroundContact)
{
    if(DEBUG){printf("GROUND CONTACT, EOE\n");}
    // Reward for robot gripper hitting the ground
    rewardHistory = REWARD_LOSS;
    newReward     = true;
    endEpisode    = true;
}
```

Finally an interim reward was set, based on the distance of the gripper to the object. Instead of computing the smoothed moving average of the delta of the distance to the goal as suggested in the lesson, the maximum distance registered from the gripper base to the can was kept on a variable. This maximum distance was then used to issue a proportional reward that is zero when the gripper is farthest away from the can and increases the closer the gripper gets to the can.

```
const float distDelta  = lastGoalDistance - distGoal;
// get max distance to goal
const float maxDistGoal = (maxDistGoal >= distGoal) ? maxDistGoal : distGoal;
// Compute reward based on distance from goal
if (distGoal < lastGoalDistance)
{  // is closer to goal
    if (distGoal > groundContact) // has not reached goal
    {   // the closer the higher the reward
        rewardHistory = REWARD_WIN * ((maxDistGoal - distGoal) / maxDistGoal);
        newReward     = true;
        endEpisode    = false;
    }
```

Note: Because the onCollisionMsg callback function is executed whenever a collision is detected, the code implementing the interim reward based on the distance to the goal does not include a check for the case when the gripper touches the object of interest (example: `else if (distGoal <= groundContact)`).

The interim reward also verifies if the gripper is farther away from the goal (last distance to goal is greater or equal to the last distance to goal registered). In such a case a full negative reward is issued. See code snippet below:

```
else if (distGoal >= lastGoalDistance)
{ // is farther away from goal
        rewardHistory = REWARD_LOSS;
        newReward     = true;
        endEpisode    = false;
}
```

In both cases the interim reward function does not start a new episode, as the robot arm has to continue moving closer to the object of interest.

Also note that an additional condition verifies if the episode timeout has been reached. In that case a full negative reward is issued and a new episode is started.

Reward parameters

The value for positive rewards was set to 2.0f while the value for the negative rewards was set to -1.0f. This value might appear low, but in rewards the absolute value is not important only the relative value is. If all rewards (good and bad) are increased or decreased equally, nothing changes really. The optimizer tries to minimize the loss (maximize the reward), that means it is interested only in the delta between values (the gradient), not their absolute value.

Hyperparameters

Initial parameter values

First the algorithm was tested when configured with the minimum possible configuration work. The aim was to observe the results and then start fine tuning the values one by one.

Initial parameters values

```
#define INPUT_WIDTH   512
#define INPUT_HEIGHT  512
#define OPTIMIZER "RMSprop"
#define LEARNING_RATE 0.1f
#define REPLAY_MEMORY 10000
#define BATCH_SIZE 8
#define USE_LSTM true
#define LSTM_SIZE 32
```

As optimizer RMSprop, Adam, AdaGrad are possible options. RMSprop was chosen arbitrarily to concentrate on the tuning of other hyperparameters. USE_LSTM was set to true to take advantage of capturing a much richer latent representations and long term dependencies. The learning rate can be set between 0 and 1. Setting it to 0 means that the Q-values are never updated, hence nothing is learned. Setting a high value such as 0.9 means that learning can occur quickly. I started with a value of 0.1 as it is low value which is better as I remember from my previous experience with deep neural networks.

Further fine tuning of hyperparameters

Due to an out of memory error (/opt/pytorch/torch/lib/THC/generic/THCStorage.cu), the initial input value to the DQN (512 x 512) was reduced to 64 x 64.

Increasing the batch size allows to include more past observations and train them together. Changing it from 8 to 64 was key to increase the performance and have some sporadic training runs able to attain the project goal's values. For instance one training run for Goal nr 1 resulted in 90 percent accuracy in just 80 runs. Unfortunately these results were infrequent events and could not be obtained repeatedly in a long term sustainable manner. These results suggest that the high variance en the results could be caused to a large extent by the initialization values, which are random.

By increasing the memory capacity, changing the LSTM_size parameter, the long short-term memory (LSTM) was able to capture much richer long term dependencies.

Finally the learning rate was increased to 0.2f and the Epsilon decay in epsilon greedy was reduced from 200 to 30 and then to 15, which caused a performance success breakthrough, as the algorithm was not depending so much on the random initialization values to archive the required goal repeatedly on a consistent basis.

Final parameters values

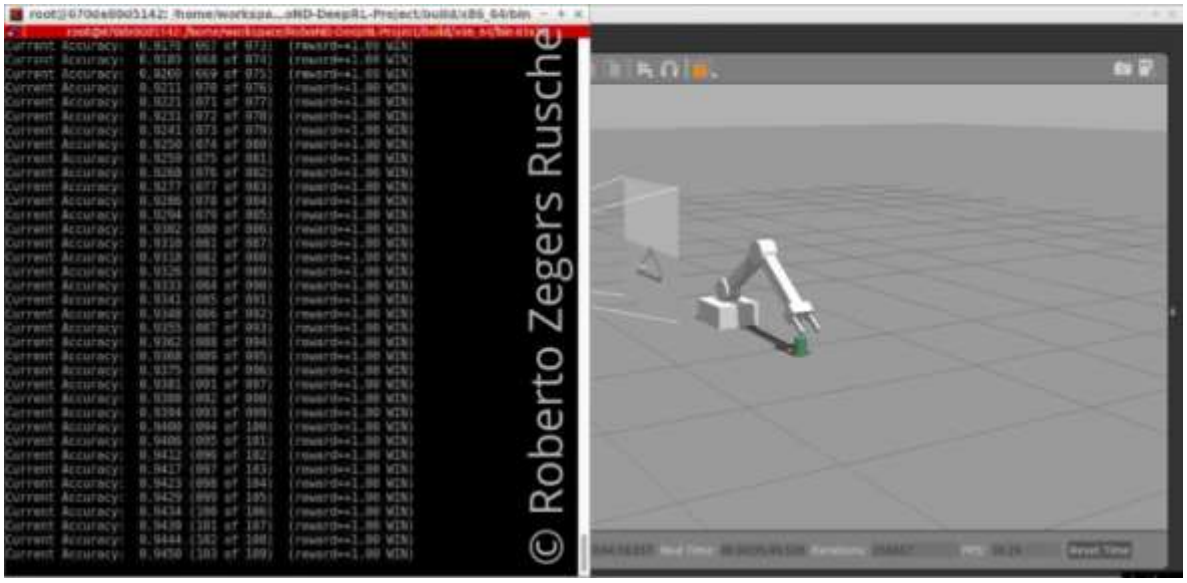
```
// Define DQN API Settings
#define INPUT_CHANNELS 3
#define ALLOW_RANDOM true
#define DEBUG_DQN false
#define GAMMA 0.9f
#define EPS_START 0.9f // epsilon_start of random actions
#define EPS_END 0.01f // epsilon_end of random action (initial value: 0.5f)
#define EPS_DECAY 15 // exponential decay of random actions(initial value: 200)
#define NUM_ACTIONS 6

// Tune hyperparameters
#define INPUT_WIDTH   128 // initial value: 512
#define INPUT_HEIGHT  128 // initial value: 512
#define OPTIMIZER "RMSprop" // RMSprop, Adam, AdaGrad, None
#define LEARNING_RATE 0.2f // initial value: 0.0f
#define REPLAY_MEMORY 10000
#define BATCH_SIZE 64 // initial value: 8
#define USE_LSTM true // initial value: false
#define LSTM_SIZE 256 // initial value: 32
```

Results

Running the RL network with the minimum configuration work did not show satisfactory results. CPU and/or GPU showed signs of being under heavy workload. Gazebo's real time factor was observed ranging from 60 to 80%. At times the robot arms seemed stuck at one position and an end of episode was triggered by the timeout. Also on each training run, an out of memory at /opt/pytorch/torch/lib/THC/generic/THCStorage.cu error occurred and the program finished abruptly. This was observed to happen anywhere between episode 50 and 110. With minimum configuration work, the arm movement was jiggly and the learning appeared to be very slow. It was also observed that the robot arm seemed to enter what appeared to be either winning or losing streaks. For instance with Goal set to 2, 89 episodes were run before getting the out of memory error message, obtaining 2 WINS and 87 LOSSES, resulting in 3% accuracy.

To get rid of the out of memory message and speed up performance, the size of the pixel array (Hyperparameters width and height) was set to 64 just as in the "Fruit Game". As a result Gazebo's Real Time Factor increased to a value fluctuating between 90 and 95 percent. With Goal set to 1, 75% current accuracy was achieved within 100 runs. After observing a 400 episode run, accuracy seemed only to increase very slowly and converge around an average value of 82 percent.

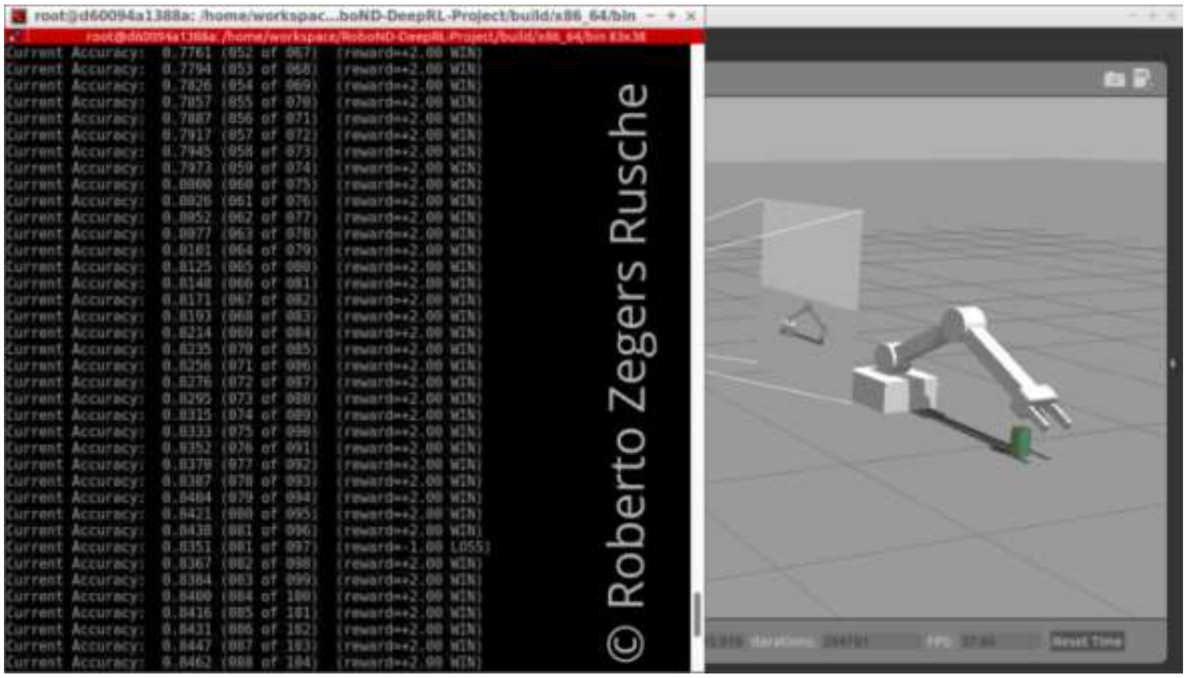


```
root@670de80d5142: /home/workspa...oND-DeepRL-Project/build/x86_64/bin
Current Accuracy:  0.9178 (067 of 073)  (reward=+1.00 WIN)
Current Accuracy:  0.9189 (068 of 074)  (reward=+1.00 WIN)
Current Accuracy:  0.9200 (069 of 075)  (reward=+1.00 WIN)
Current Accuracy:  0.9211 (070 of 076)  (reward=+1.00 WIN)
Current Accuracy:  0.9221 (071 of 077)  (reward=+1.00 WIN)
Current Accuracy:  0.9231 (072 of 078)  (reward=+1.00 WIN)
Current Accuracy:  0.9241 (073 of 079)  (reward=+1.00 WIN)
Current Accuracy:  0.9250 (074 of 080)  (reward=+1.00 WIN)
Current Accuracy:  0.9259 (075 of 081)  (reward=+1.00 WIN)
Current Accuracy:  0.9268 (076 of 082)  (reward=+1.00 WIN)
Current Accuracy:  0.9277 (077 of 083)  (reward=+1.00 WIN)
Current Accuracy:  0.9286 (078 of 084)  (reward=+1.00 WIN)
Current Accuracy:  0.9294 (079 of 085)  (reward=+1.00 WIN)
Current Accuracy:  0.9302 (080 of 086)  (reward=+1.00 WIN)
Current Accuracy:  0.9310 (081 of 087)  (reward=+1.00 WIN)
Current Accuracy:  0.9318 (082 of 088)  (reward=+1.00 WIN)
Current Accuracy:  0.9326 (083 of 089)  (reward=+1.00 WIN)
Current Accuracy:  0.9333 (084 of 090)  (reward=+1.00 WIN)
Current Accuracy:  0.9341 (085 of 091)  (reward=+1.00 WIN)
Current Accuracy:  0.9348 (086 of 092)  (reward=+1.00 WIN)
Current Accuracy:  0.9355 (087 of 093)  (reward=+1.00 WIN)
Current Accuracy:  0.9362 (088 of 094)  (reward=+1.00 WIN)
Current Accuracy:  0.9368 (089 of 095)  (reward=+1.00 WIN)
Current Accuracy:  0.9375 (090 of 096)  (reward=+1.00 WIN)
Current Accuracy:  0.9381 (091 of 097)  (reward=+1.00 WIN)
Current Accuracy:  0.9388 (092 of 098)  (reward=+1.00 WIN)
Current Accuracy:  0.9394 (093 of 099)  (reward=+1.00 WIN)
Current Accuracy:  0.9400 (094 of 100)  (reward=+1.00 WIN)
Current Accuracy:  0.9406 (095 of 101)  (reward=+1.00 WIN)
Current Accuracy:  0.9412 (096 of 102)  (reward=+1.00 WIN)
Current Accuracy:  0.9417 (097 of 103)  (reward=+1.00 WIN)
Current Accuracy:  0.9423 (098 of 104)  (reward=+1.00 WIN)
Current Accuracy:  0.9429 (099 of 105)  (reward=+1.00 WIN)
Current Accuracy:  0.9434 (100 of 106)  (reward=+1.00 WIN)
Current Accuracy:  0.9439 (101 of 107)  (reward=+1.00 WIN)
Current Accuracy:  0.9444 (102 of 108)  (reward=+1.00 WIN)
Current Accuracy:  0.9450 (103 of 109)  (reward=+1.00 WIN)
```

Image 1: Goal 1, first training to successfully achive 90% accuracy in less than 100 runs

For Goal 2, the required accuracy could also be archived, but not at all times. In 1 out of 4 training runs the network did not converge and scored more LOSSes than WINs. Sometimes the network would converge and in occasions it would not.

After increasing LTSM from 32 to 256 we observed that the robot arm was able to recover from bad starts (for instance situation with 4 LOSSes out of 5 runs) and train successfully.



```
root@d60094a1388a: /home/workspac...boND-DeepRL-Project/build/x86_64/bin
root@d60094a1388a: /home/workspace/RoboND-DeepRL-Project/build/x86_64/bin 83x38
Current Accuracy:  0.7761 (052 of 067)  (reward=+2.00 WIN)
Current Accuracy:  0.7794 (053 of 068)  (reward=+2.00 WIN)
Current Accuracy:  0.7826 (054 of 069)  (reward=+2.00 WIN)
Current Accuracy:  0.7857 (055 of 070)  (reward=+2.00 WIN)
Current Accuracy:  0.7887 (056 of 071)  (reward=+2.00 WIN)
Current Accuracy:  0.7917 (057 of 072)  (reward=+2.00 WIN)
Current Accuracy:  0.7945 (058 of 073)  (reward=+2.00 WIN)
Current Accuracy:  0.7973 (059 of 074)  (reward=+2.00 WIN)
Current Accuracy:  0.8000 (060 of 075)  (reward=+2.00 WIN)
Current Accuracy:  0.8026 (061 of 076)  (reward=+2.00 WIN)
Current Accuracy:  0.8052 (062 of 077)  (reward=+2.00 WIN)
Current Accuracy:  0.8077 (063 of 078)  (reward=+2.00 WIN)
Current Accuracy:  0.8101 (064 of 079)  (reward=+2.00 WIN)
Current Accuracy:  0.8125 (065 of 080)  (reward=+2.00 WIN)
Current Accuracy:  0.8148 (066 of 081)  (reward=+2.00 WIN)
Current Accuracy:  0.8171 (067 of 082)  (reward=+2.00 WIN)
Current Accuracy:  0.8193 (068 of 083)  (reward=+2.00 WIN)
Current Accuracy:  0.8214 (069 of 084)  (reward=+2.00 WIN)
Current Accuracy:  0.8235 (070 of 085)  (reward=+2.00 WIN)
Current Accuracy:  0.8256 (071 of 086)  (reward=+2.00 WIN)
Current Accuracy:  0.8276 (072 of 087)  (reward=+2.00 WIN)
Current Accuracy:  0.8295 (073 of 088)  (reward=+2.00 WIN)
Current Accuracy:  0.8315 (074 of 089)  (reward=+2.00 WIN)
Current Accuracy:  0.8333 (075 of 090)  (reward=+2.00 WIN)
Current Accuracy:  0.8352 (076 of 091)  (reward=+2.00 WIN)
Current Accuracy:  0.8370 (077 of 092)  (reward=+2.00 WIN)
Current Accuracy:  0.8387 (078 of 093)  (reward=+2.00 WIN)
Current Accuracy:  0.8404 (079 of 094)  (reward=+2.00 WIN)
Current Accuracy:  0.8421 (080 of 095)  (reward=+2.00 WIN)
Current Accuracy:  0.8438 (081 of 096)  (reward=+2.00 WIN)
Current Accuracy:  0.8351 (081 of 097)  (reward=-1.00 LOSS)
Current Accuracy:  0.8367 (082 of 098)  (reward=+2.00 WIN)
Current Accuracy:  0.8384 (083 of 099)  (reward=+2.00 WIN)
Current Accuracy:  0.8400 (084 of 100)  (reward=+2.00 WIN)
Current Accuracy:  0.8416 (085 of 101)  (reward=+2.00 WIN)
Current Accuracy:  0.8431 (086 of 102)  (reward=+2.00 WIN)
Current Accuracy:  0.8447 (087 of 103)  (reward=+2.00 WIN)
Current Accuracy:  0.8462 (088 of 104)  (reward=+2.00 WIN)
```

Image 2: Goal 2, first successful training run to obtain 80 percent accuracy in less than 100 runs

However we also observed that for Goal Nr.2, the robot was often barely missing the can and hitting the ground instead. WE decide to increase the image resolution from 64 x 64 to 128 x 128 to resolve this issue.

Finally lowering the Epsilon Decay value showed a decisive improvement on the repeatability of training results, causing successive independent training runs to produce successful outcomes in a sustainable, long-term stable manner.

Goal 1: Under the final set of parameters accuracy was several times permanently above 90% after reaching an episode that could be any number in between of 20 and 80. At episode 100, 89 to 94 percent accuracy was registered.

Goal 2: Under the final set of parameters, several times starting anywhere between episode 35 and 100, the accuracy kept permanently above 80%. At 100 runs 80% to 96% accuracy was reached.

During the development of this project it was not necessary to try a different reward system, as expected results were obtained. The only adjustment in this regard was to increase the WIN reward from the initial value of 1.0 to 2.0 to provide a higher incentive to the robotic arm to pursue a WIN.

Future Work

In future research, it will be important to investigate the fact that the interim reward is set to incentive the gripper to move closer to the object of interest. As a result the robot arm tries always to touch the can with the gripper even when the goal is set to 1 (the reward is issued when any part of the arm collides with the can). This causes the arm to do not overreach because this would increase the distance to the object and issue a negative reward when it does. One way that could reconcile both objectives, namely incentive the arm to get closer without punishing the arm for overreaching its target, could be to incentivize the movement of the third joint closer to the can instead of incentivizing the gripper to get closer to the can.

Future work might address the fact that the negative reward issued when moving away from the target is not proportional to the distance but instead it is a full negative reward. A reward that would be proportional to the distance the gripper moves away, could yield new strategies that could increases the probability of the gripper to touch the can.

An interesting topic for further expanding the scope of this work would be not only changing the position of the object o interest, as suggested by one of the project challenges, but also to randomly change the objects shape, color and orientation.

Last but not least, future work could aim to replicate our results on a real robot in a real environment. In that scenario new challenges such as real time constraints or smooth exploration to avoid mechanical wear and tear on the robot are expected.

Reproduce the work

To get started, first clone this repository:

https://github.com/rfzeg/RoboND-DeepRL-Project.git

Then fix the 'make' compilation error 'IGN_MASSMATRIX3_DEFAULT_TOLERANCE not defined' by installing libignition-math2-dev:

```
$ sudo apt-get install libignition-math2-dev
```

Next, the package needs to be build using make, so once in the project folder run:

```
$ cd build
$ make
```

Then run the gazebo-arm.sh script inside the /x86_64/bin folder:

```
$ cd x86_64/bin
$ ./gazebo-arm.sh
```

The RL agent and the reward functions are to be defined in gazebo/ArmPlugin.cpp.

References

[1] Wikipedia "Reinforcement learning ", https://en.wikipedia.org/wiki/Reinforcement_learning, last visited: 2019-03-15

[2] Website of the Machine Learning & Robotics Lab, University Stuttgart, https://ipvs.informatik.uni-stuttgart.de/mlr/teaching/reinforcement-learning-ss16/, last visited: 2019-03-15

[3] "A Beginner's Guide to Deep Reinforcement Learning", https://skymind.ai/wiki/deep-reinforcement-learning, last visited: 2019-03-15

[4] Deep Reinforcement Learning Models: Tips & Tricks for Writing Reward Functions https://medium.com/@BonsaiAI/deep-reinforcement-learning-models-tips-tricks-for-writing-reward-functions-a84fe525e8e0, last visited: 2019-03-15