UDACITY PROJECT: DEEP RL ARM MANIPULATION

STEPHAN HOHNE

1. Introduction

This is a solution of the Deep RL Arm Manipulation project as part of the Robotics Nanodegree. The goal of the project is to set up and optimize a reinforcement learning framework. A Deep Q-Network learns to control a robotic arm in a simulated Gazebo environment. The reinforcement learning agent is given the task of touching a target object.

Objective 1: Have any part of the robot arm touch the object of interest, with at least a 90% accuracy for a minimum of 100 runs.

Objective 2: Have only the gripper base of the robot arm touch the object, with at least a 80% accuracy for a minimum of 100 runs.

The project tasks were to implement parts of the ArmPlugin.cpp file, see section 2. The reward function was to be designed (section 2.1), and the DQN hyper-parameters were to be tuned (section 2.2).

The network training along with the simulation runs were performed on the Udacity Project Workspace with enabled GPU.

2. Simulations

Before running simulation experiments, the following parts of the ArmPlugin.cpp file were implemented.

- The subscription to camera and collision topics.
- The creation of the DQN agent using learning hyperparameters defined as constants in the file.
- The definition of collision checks to determine which robot arm link touched the target object.
- Both velocity control and position control of the robot were implemented.
- Criteria for ground contact of the arm were defined, see listing 1. The z-coordinates of the gripper link bounding box were used. The threshold ground level was set at $z = 0.05$ meters to accommodate for the inaccuracy of the distance calculation.

```
// define the z value for  ground contact
const float groundContact = 0.05f;

// define condition for ground contact
const bool checkGroundContact =
          ( gripBBox.min.z <= groundContact
          || gripBBox.max.z <= groundContact  );
```

FIGURE 1. Criteria for ground contact of the gripper base link.

Several experimental runs of the simulation were performed, testing different versions of the reward function and tuning the DQN parameters.

2.1. **Reward Function Design.** A reward system was set up in order to teach the robotic arm how to reach for the target object. Positive and negative rewards are defined as numeric constants `REWARD_WIN` and `REWARD_LOSS`. A win is issued for touching the prop with any link (objective 1) or the gripper base link (objective 2). A loss is issued when the arm touches the ground or when the length of the episode exceeds 100 steps in time. The occurrence of any of these events will end the episode.

Since positive rewards from touching the prop by the gripper link are very sparse, an interim reward function was designed. Its purpose is to guide the robot arm towards the target. The relevant measure is the distance d_t between gripper and target object. For consecutive steps $t-1$ and t, the difference $\Delta_t = d_{t-1} - d_t$ was calculated. In order to obtain a smooth reward function, the moving average of the recorded Δ values was computed, see listing 2.

```
const float distDelta  = lastGoalDistance - distGoal;

avgGoalDelta  = (avgGoalDelta * ALPHA) + (distDelta * (1.0f - ALPHA));
```

FIGURE 2. Code for computing the delta of the distance to goal and the corresponding smoothed moving average. The value of the alpha parameter is a constant that was tuned in the experiments.

A linear function of the `avgGoalDelta` function is proportional to the average speed towards the goal and should be a good candidate for the interim reward function, see for example 6. The parameter values found in the experiments are shown in table 3.

2.1.1. *Objective 1.* Velocity control of the robot arm was enabled during the simulation runs. Two versions of a reward function were tested, see figure 3 for the source code. The first version is a linear function proportional to the moving average. The second version issues different rewards, depending on the sign of `avgGoalDelta`. The positive reward is the same as in version 1, the negative reward is proportional to the goal distance.

Date: May 29, 2018.

Experiment	1	2	3
Reward function, see 3	version 1	version 2	version 2
Best run			
Accuracy / episodes	0.9694/98	0.9466/131	0.9800/100

TABLE 1. Experimental runs for objective 1. See figure 8 for an image showing the result of experiment 3.

```
// version 1
rewardHistory = INTERIM_REWARD * avgGoalDelta;

// version 2
rewardHistory = avgGoalDelta > 0
        ? INTERIM_REWARD * avgGoalDelta
        : REWARD_LOSS * distGoal;
```

FIGURE 3. Implementation of two versions of the reward function. The value of the reward loss is −10.0, and the value of the interim reward is 10.0, see table 3.

Both versions of the interim reward function met objective 1, see table 1 and section 3. The fixed configuration of the DQN agent used during the experimental runs is shown in figure 7a.

2.1.2. *Objective 2.* The reward parameters and agent control were adjusted following advice from students on Slack community channel.

- The velocity control was switched to position control.
- The value of the moving average `ALPHA` was lowered from the initial guess of 0.9 to values in the range $[0.4, 0.5]$.

In order to explore alternatives to the linear reward function, the effect of an exponential function of the distance of the form

$$\gamma\left(\texttt{distGoal}\right) = \exp\left(-\beta\,\texttt{distGoal}\right) \tag{1}$$

with $\beta \in \mathbb{R}$ was studied. An exponential factor with $\beta = -1.0$ was applied to the reward function, see listing 4. Since the range is $\gamma \in [0, 1]$, the reward should be damped with increasing distance, making gripper positions close to the target object more desirable. In the experimental runs the arm accelerated towards the target objective and failed to accomplish the task.

```
rewardHistory = exp(-1.0f * distGoal) * INTERIM_REWARD * avgGoalDelta;
```

FIGURE 4. Implementation of the exponential damping factor 1 as part of the interim reward function.

The exponential 1 with $\beta = -1.0$ was also used in the calculation of the moving average as a replacement of the constant alpha parameter, see listing 5. The intention was to give less weight to distance deltas when the arm is close to the target object. This should accommodate for the inaccuracy in the computation of small distance deltas. In the experimental runs the arm slowed down to a halt before reaching the target objective and failed to accomplish the task. It is left to future work to investigate the use of exponential reward functions further.

```
const float gamma = exp(-1.0f * distGoal);
avgGoalDelta  = (avgGoalDelta * gamma) + (distDelta * (1.0f - gamma));
rewardHistory = 40.0f * avgGoalDelta;
```

FIGURE 5. Implementation of the exponential damping factor 1 in the computation of the moving average.

In subsequent experiments, a linear reward function was used again, see listing 6. A negative offset was introduced to punish standstill or the moving away from the target. The parameters INTERIM_REWARD and INTERIM_OFFSET represent the slope and the offset, respectively. These parameters, together with the alpha parameter of the moving average were tuned experimentally, see table 2. During the experiments, REWARD_WIN was set to 20.0, and REWARD_LOSS was set to -20.0. The configuration of the DQN agent used during the experimental runs is shown in table 2 and figure 7b. Objective 2 was met using the linear reward function.

```
// linear reward function
rewardHistory = INTERIM_REWARD * avgGoalDelta - INTERIM_OFFSET;
```

FIGURE 6. Code for the interim reward function as a linear function of the moving average. The parameters INTERIM_REWARD and INTERIM_OFFSET are constants that were tuned in the experiments.

2.2. DQN Hyperparameter Tuning.

The size of the input was set to 64×64, since the camera image sent to the agent is of that size. The long-short-term memory was enabled by setting LSTM constant to true. The RMSprop and the Adam optimizer [1] were tested. RMSprop was used during the simulation runs for objective 1, Adam was used in the reward function design experiments for objective 2. When going from objective 1 to objective 2, the batch size was increased from 32 to 256, see 7. The LSTM size was increased from 64 to 256, which should allow the agent to memorize more complex moves.

During the simulation runs for objective 2 (see table 2), the start value of the exploration parameter EPS_START was tuned between the values 0.9 and 0.7. The end value of the exploration parameter EPS_END was tuned between the values 0.05 and 0.02. The learning rate was tuned between the values 0.08 and 0.1. The performance of each configuration was determined using the accuracy output during the simulations.

Experiment	1	2	3
LEARNING_RATE	0.1	0.08	0.1
BATCH_SIZE	32	256	256
EPS_START	0.9	0.7	0.7
EPS_END	0.05	0.02	0.02
INTERIM_REWARD	4.0	4.0	4.0
INTERIM_OFFSET	0.2	0.2	0.3
ALPHA	0.5	0.5	0.4
Best run			
Accuracy / episodes	0.8070/715	0.8125/352	0.8202/228

TABLE 2. Parameter tuning for achieving objective 2. The linear reward function 6 was used.

```
// Define DQN API Settings
#define INPUT_CHANNELS 3
#define ALLOW_RANDOM true
#define DEBUG_DQN false
#define GAMMA 0.9f
#define EPS_START 0.7f
#define EPS_END 0.02f
#define EPS_DECAY 200

// Define Learning Hyperparameters
#define INPUT_WIDTH    64
#define INPUT_HEIGHT   64
#define OPTIMIZER "RMSprop"
#define LEARNING_RATE 0.1f
#define REPLAY_MEMORY 10000
#define BATCH_SIZE 32
#define USE_LSTM true
#define LSTM_SIZE 64
```

(A) Experiments for objective 1.

```
// Define DQN API Settings
#define INPUT_CHANNELS 3
#define ALLOW_RANDOM true
#define DEBUG_DQN false
#define GAMMA 0.9f
#define EPS_START 0.7f
#define EPS_END 0.02f
#define EPS_DECAY 200

// Define Learning Hyperparameters
#define INPUT_WIDTH    64
#define INPUT_HEIGHT   64
#define OPTIMIZER "Adam"
#define LEARNING_RATE 0.1f
#define REPLAY_MEMORY 10000
#define BATCH_SIZE 256
#define USE_LSTM true
#define LSTM_SIZE 256
```

(B) Experiment 3 for objective 2.

FIGURE 7. DQN configurations used during the reward function experiments.

3. RESULTS

Both objectives were achieved. The best performing runs were observed with the linear reward function 6, while the exponential reward function did not meet the objectives.

The largest positive effect on the agents performance was observed when lowering the value of the moving average `ALPHA` parameter from 0.9 to 0.4, see section 2.1. This finding emphasizes the importance of designing a smooth reward function.

The best performing configurations for each objective are shown in table 3. The number of episodes it took the agent to reach the threshold accuracy varied between ≈ 200 and ≈ 700, depending on the initial conditions of the simulation run. During the simulation runs, the accuracy improved slowly over time. For example, after taking screenshot 9 the accuracy of that run went up to over 83% after ≈ 260 episodes.

Objective	1	2
`REWARD_WIN`	10.0	20.0
`REWARD_LOSS`	−10.0	−20.0
`INTERIM_REWARD`	10.0	4.0
`INTERIM_OFFSET`	n. a.	0.3
`ALPHA`	0.9	0.4
Best run		
Accuracy / episodes	0.9800/100	0.8202/228
Image	8	9

TABLE 3. Values of reward parameters for the best performing simulation runs, achieving objectives 1 and 2.

4. Future Work

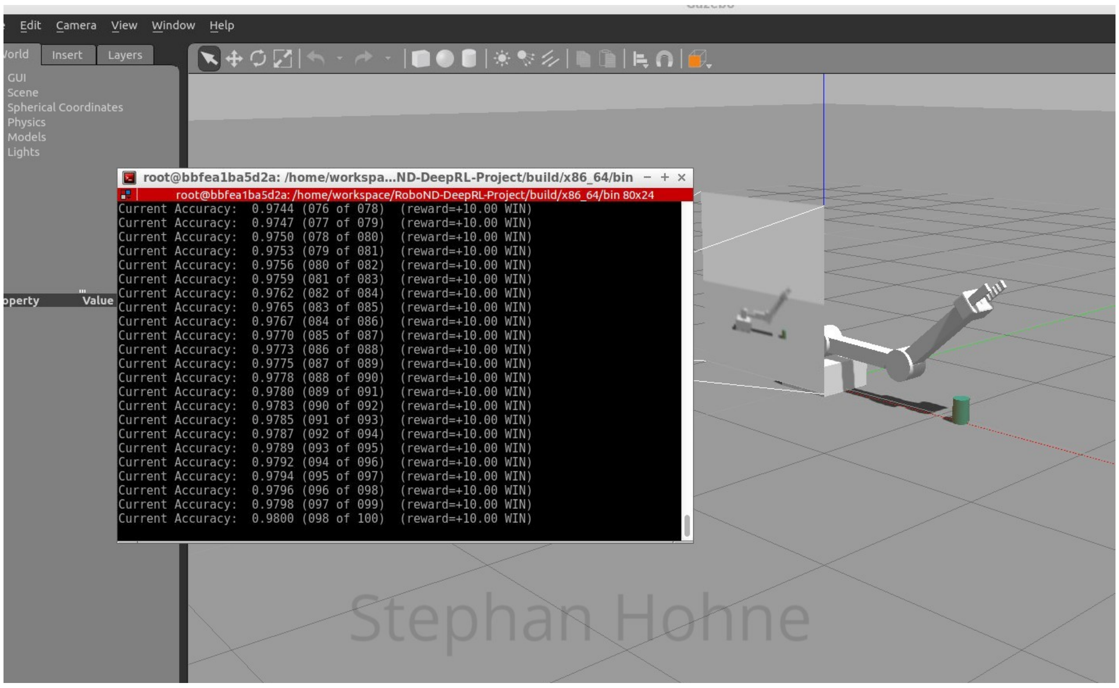
The reinforcement-learning framework can be improved further by:

- Tuning the remaining DQN hyperparameters like the discount factor, size of replay memory, and epsilon decay rate.
- Investigating whether the exponential damping function 1 can be improved to create a reward function able to achieve the objectives.

Additionally, the plan is to let the project run on the Jetson TX2. It will also be interesting to see how well the framework generalizes to controlling a robotic arm with enhanced capabilities and more degrees of freedom.

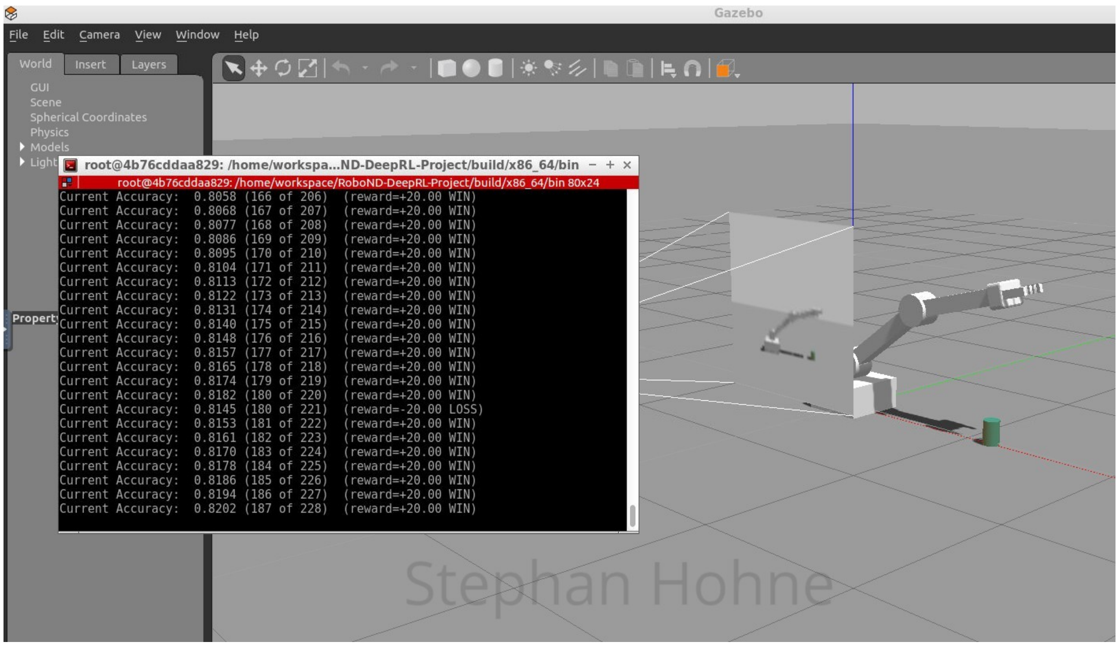
Addendum, May 2019. The project has been improved as follows.

- The project is running on the Jetson TX2 and can be used with either RoboND-DeepRL-Project or jetson-reinforcement.
- The code for the additional challenge of increasing the arm's reach is implemented. The revolute joint in the base link is activated, for implementation



```
root@bbfea1ba5d2a: /home/workspa...ND-DeepRL-Project/build/x86_64/bin
root@bbfea1ba5d2a: /home/workspace/RoboND-DeepRL-Project/build/x86_64/bin 80x24
Current Accuracy:  0.9744 (076 of 078)  (reward=+10.00 WIN)
Current Accuracy:  0.9747 (077 of 079)  (reward=+10.00 WIN)
Current Accuracy:  0.9750 (078 of 080)  (reward=+10.00 WIN)
Current Accuracy:  0.9753 (079 of 081)  (reward=+10.00 WIN)
Current Accuracy:  0.9756 (080 of 082)  (reward=+10.00 WIN)
Current Accuracy:  0.9759 (081 of 083)  (reward=+10.00 WIN)
Current Accuracy:  0.9762 (082 of 084)  (reward=+10.00 WIN)
Current Accuracy:  0.9765 (083 of 085)  (reward=+10.00 WIN)
Current Accuracy:  0.9767 (084 of 086)  (reward=+10.00 WIN)
Current Accuracy:  0.9770 (085 of 087)  (reward=+10.00 WIN)
Current Accuracy:  0.9773 (086 of 088)  (reward=+10.00 WIN)
Current Accuracy:  0.9775 (087 of 089)  (reward=+10.00 WIN)
Current Accuracy:  0.9778 (088 of 090)  (reward=+10.00 WIN)
Current Accuracy:  0.9780 (089 of 091)  (reward=+10.00 WIN)
Current Accuracy:  0.9783 (090 of 092)  (reward=+10.00 WIN)
Current Accuracy:  0.9785 (091 of 093)  (reward=+10.00 WIN)
Current Accuracy:  0.9787 (092 of 094)  (reward=+10.00 WIN)
Current Accuracy:  0.9789 (093 of 095)  (reward=+10.00 WIN)
Current Accuracy:  0.9792 (094 of 096)  (reward=+10.00 WIN)
Current Accuracy:  0.9794 (095 of 097)  (reward=+10.00 WIN)
Current Accuracy:  0.9796 (096 of 098)  (reward=+10.00 WIN)
Current Accuracy:  0.9798 (097 of 099)  (reward=+10.00 WIN)
Current Accuracy:  0.9800 (098 of 100)  (reward=+10.00 WIN)
```

FIGURE 8. Simulation run that achieved objective 1. See experiment 3 in table 1.



```
root@4b76cddaa829: /home/workspa...ND-DeepRL-Project/build/x86_64/bin
root@4b76cddaa829: /home/workspace/RoboND-DeepRL-Project/build/x86_64/bin 80x24
Current Accuracy:  0.8058 (166 of 206)  (reward=+20.00 WIN)
Current Accuracy:  0.8068 (167 of 207)  (reward=+20.00 WIN)
Current Accuracy:  0.8077 (168 of 208)  (reward=+20.00 WIN)
Current Accuracy:  0.8086 (169 of 209)  (reward=+20.00 WIN)
Current Accuracy:  0.8095 (170 of 210)  (reward=+20.00 WIN)
Current Accuracy:  0.8104 (171 of 211)  (reward=+20.00 WIN)
Current Accuracy:  0.8113 (172 of 212)  (reward=+20.00 WIN)
Current Accuracy:  0.8122 (173 of 213)  (reward=+20.00 WIN)
Current Accuracy:  0.8131 (174 of 214)  (reward=+20.00 WIN)
Current Accuracy:  0.8140 (175 of 215)  (reward=+20.00 WIN)
Current Accuracy:  0.8148 (176 of 216)  (reward=+20.00 WIN)
Current Accuracy:  0.8157 (177 of 217)  (reward=+20.00 WIN)
Current Accuracy:  0.8165 (178 of 218)  (reward=+20.00 WIN)
Current Accuracy:  0.8174 (179 of 219)  (reward=+20.00 WIN)
Current Accuracy:  0.8182 (180 of 220)  (reward=+20.00 WIN)
Current Accuracy:  0.8145 (180 of 221)  (reward=-20.00 LOSS)
Current Accuracy:  0.8153 (181 of 222)  (reward=+20.00 WIN)
Current Accuracy:  0.8161 (182 of 223)  (reward=+20.00 WIN)
Current Accuracy:  0.8170 (183 of 224)  (reward=+20.00 WIN)
Current Accuracy:  0.8178 (184 of 225)  (reward=+20.00 WIN)
Current Accuracy:  0.8186 (185 of 226)  (reward=+20.00 WIN)
Current Accuracy:  0.8194 (186 of 227)  (reward=+20.00 WIN)
Current Accuracy:  0.8202 (187 of 228)  (reward=+20.00 WIN)
```

FIGURE 9. Simulation run that achieved objective 2. See experiment 3 in table 2.

details see table 4. The arm can rotate about its base within the joint limits. This adds a third degree of freedom to the configuration space.

- The code for object randomization is implemented. In the method `ArmPlugin::updateJoints()`, the call to `ResetPropDynamics()` is replaced with `RandomizeProps()`. At the beginning of an episode the target object is placed randomly within the arm's reach, see figure 10 for implementation details.

Arm rotation	Enabled	Disabled
`ArmPlugin.cpp`	`#define LOCKBASE false`	`#define LOCKBASE true`
Tube model pose in `gazebo-arm.world`	`[0.75 0.75 0 0 0 0]`	`[1.15 0 0 0 0 0]`

TABLE 4. Project configurations for toggling the revolute joint in the arm's base link.

```
pose.pos.x = randf(0.02f, 0.30f);
pose.pos.y = 0.0f;
pose.pos.z = 0.0f;
```

(A) Arm rotation disabled.

```
pose.pos.x = randf(0.35f, 0.45f);
pose.pos.y = randf(-1.5f, 0.2f);
pose.pos.z = 0.0f;
```

(B) Arm rotation enabled.

FIGURE 10. Limits for random initialization of target object pose in `PropPlugin::Randomize()`.

REFERENCES

[1] Diederik P. Kingma and Jimmy Ba, *Adam: A Method for Stochastic Optimization*, CoRR, 2014, arXiv abs/1412.6980