

Project: Deep RL Arm Manipulation

Rastri Dey

I. INTRODUCTION

Deep Reinforcement Learning is a paradigm shift in the world of robotics. A Deep RL agent is initialized with raw sensor inputs and defined goal outputs fitting a given application. The agent thereby figures out the best way to achieve its goal through recursive training of trial and error iterations. In this paradigm the concept of acquiring raw sensor data and post process it for planning and control with internal states and measurement uncertainty are all replaced with a Deep Neural Network. A Deep Neural Network learns all its actions based on the input states through a process of continuous interaction with the environment. Based on the Neural Network's output, the environment feedbacks a reward (can be referred as stimulus), that's delivered immediately after the agent's behavior to make the behavior more likely to happen in future. The environmental states are the context provided to the agent for choosing intelligent actions. The main objective of a DQN agent is to perform actions to maximize the cumulative reward received, from several iterations of feedback. With every iteration, the Neural Network model to map state-action pair for the agent improves, enhancing the agent's response towards its environment.

This project explores the science of reinforcement learning through two tasks, given to an Arm Manipulator DQN agent. The task goals are:

1. To have any part of the robot arm touch the object of interest, with at least a 90 percent accuracy for a minimum of 100 runs.
2. To have only the gripper base of the robot arm touch the object, with at least a 80 percent accuracy for a minimum of 100 runs.

Both the tasks are performed in a gazebo world with the main objective of accomplishing the defined goals through a deep learning agent. The RL agent for the robotic arm learns to manipulate its joints to reach and touch the object placed in its vicinity through multiple reward feedbacks from its interaction with the Gazebo environment.

II. REWARD FUNCTIONS

The environment built to accomplish the robotic tasks consists of a Gazebo world with the following components:

- The robotic arm with a gripper attached to it.
- A camera sensor, to capture images to feed into the DQN.
- A cylindrical object or prop.

The robotic arm model constructed in the gazebo-arm.world file, calls upon a gazebo plugin called ArmPlugin, responsible for creating the DQN agent and training it to learn, to touch the prop. The DQN agent and its learning parameters are defined in the ArmPlugin.cpp file, located in the gazebo folder in the project directory. The reward functions are created and assigned in the ArmPlugin.cpp file in the following categories:

- **REWARD WIN** - The value for positive reward. The REWARD WIN variable is set to +100. For the respective objectives of, 'robotic arm touching the prop' and 'only the gripper base touching the prop', the arm manipulator is rewarded a +100 value to make the robotic behavior more likely towards touching the prop for all its future episodes.
- **REWARD LOSS** - The value for negative reward. The REWARD LOSS variable is set to -100. For both tasks, the agent is penalized for hitting the ground by a constant value, which is the lowest value that the agent earns to discourage itself from repeatedly performing the action. Moreover, the robot is also penalized with a REWARD LOSS, after a 100 frames of indecision or touching anything else than the object of interest in the given environment.
- **rewardHistory** – The variable that stores the reward values in terms of REWARD WIN, REWARD LOSS or INTERIM rewards. The arm manipulator accuracy till the current episodic iteration is calculated on the basis of this variable.

a. Issue a reward based on collision between the arm and the object:

Under the callback function onCollisionMsg in the ArmPlugin.cpp file, a check condition is defined to compare if the particular links of the arm with their defined collision elements are colliding with the COLLISION ITEM. Based on the first objective, for any part of the robotic arm touching the prop, a REWARD WIN of +100 is assigned to rewardHistory, failing at which a REWARD LOSS of -100 is assigned. Similarly, a REWARD WIN of +100 is assigned when the arm manipulator touches the prop using the gripper base alone in case of second objective, and the same way as the first objective a REWARD LOSS of -100 is assigned, otherwise. In case of second objective, the goal of the robotic arm to touch

the prop narrows down to only gripper, making it further challenging for the robot to learn the strategy that will earn it a positive reward. The robotic arm through multiple trial and error iterations over several episodes, learns the strategy to maximize this cumulative reward. Furthermore, the robotic arm also gets penalized with a REWARD LOSS for any collision with itself to direct its decisions towards the prop alone, enhancing its decision strategy even further.

b. Issue a reward for robot gripper hitting the ground:

In Gazebo API, the function called `GetBoundingBox()` calculates the minimum and maximum values of x, y and z axes of the box defining the gripper. From these variables, it is checked whether the gripper z axis lies below the threshold defining the ground contact, following which a REWARD LOSS, is assigned to `rewardHistory` to penalize the robot arm from hitting the ground.

c. Issue a reward for episode timeout:

The robot is further penalized with a REWARD LOSS, at the end of a certain episode after 100 frames, if the arm manipulator is unable to make a decision for assigning a certain reward. This also encourages the robotic arm for faster convergence towards its goals.

The aforementioned REWARD functions are the terminal rewards based on the actions taken by the arm manipulator at a given state calling for the end of the episode. This reward strategy though directed for attaining its stated objectives, are largely uninformative as the robot is unaware of its directions towards attaining its goals. In this way the robot needs to try out varying control forces to the joint manipulator before it learns the intended way to reach near the goal. Hence, this kind of reward function suffers with the problem of sparse rewards and might take indefinite time through random joint control forces before the robot learns the apt strategy through rigorous interaction. At certain circumstances, this trial and error might not ensure any convergence at all if the control forces applied are never towards the intended direction. To tackle this problem, there requires some intermediate reward feedback that will define the strategy to direct the arm manipulator towards its intended goal during the course of a certain episode.

d. Issue an interim reward based on the distance to the object:

In `ArmPlugin.cpp`, a function called `BoxDistance()` calculates the distance between two bounding boxes. Using this function, the distance between the arm and the object is calculated and the manipulator is assigned an INTERIM REWARD based on the function of this distance. The reward is a smoothed moving average, or SMMA of the delta of the distance to the goal. The smoothed moving average calculates the distance between the gripper to the goal, wherein the main idea is to trigger the gripper towards the object by a gradual feedback of

incrementing rewards as the arm approaches the prop. These rewards, allows the agent for maximizing its prevailing intermediate rewards towards the object of interest while removing the otherwise deviations, thus helping the agent learn faster. The moving average towards the goal is calculated by giving weight biases to current delta distance and the previous moving average to the goal. The function is defined as follows:

$$\text{avgGoalDelta} = (\text{avgGoalDelta} * \alpha) + (\text{distDelta} * (1 - \alpha));$$

Here, α is the constant bias ranging between 0 to 1, given a smaller value of 0.4 to give more weightage to the current distance from the gripper to the object while not neglecting the previous moving average distances completely. This smoothed moving average is further amplified with a reward gain from which a constant offset is removed and fed back to the `rewardHistory`.

$$\text{rewardHistory} = \text{avgGoalDelta} * \text{REWARD_GAIN} - \text{Interim_offset};$$

Here, the `REWARD_GAIN` and `Interim_offset` are 5 and 0.5 respectively. These reward feedbacks guide the agent to its goal during a certain episode till a terminal reward based on the corresponding objective is achieved. The INTERIM REWARDS for both the objectives are set in similar ways in the project. The terminal rewards of REWARD WIN or REWARD LOSS for any part of the robotic arm touching the object or the gripper alone touching the object, decides the final strategy in the training phase of the DQN agent.

- `newReward` – This variable is set ‘true’ when a new reward is issued in terms of REWARD WIN or REWARD LOSS or INTERIM REWARD. At the default state or during initialization `newReward` is set as ‘false’. At the end of an iteration, the reward indicator is again reset to ‘false’.
- `endEpisode` – The variable that determines if the episode is over or not. `endEpisode` is set ‘false’ at initialization and gets converted to ‘true’ when either a terminal reward is assigned or the episode has reached its maximum limits for the number of frames. The variable is reset to ‘false’ at the beginning of every episode.

Overall, all the parameters for both the objectives in terms of reward functions are set similar with just a difference in the approach of collision check while assigning the positive maximum reward. The reward function is constructed such that a minimal change in the `ArmPlugin.cpp` file is sufficient for the RL agent to train, for both the objectives. This although eases the user to handle the RL agent, might not be the case always. As for instance, more challenging tasks may call for varying reward functions and complex parameters suited to the task requirements.

Joint Control –

In the given project, the robotic arm manipulator is having three non-fixed joints. At every iteration the RL agent requires to take an action to manipulate its joints such that the arm reaches and touches the object of interest. The control forces required at the joints can be provided through either position control or velocity control. In each of the controller, the joint positions or joint velocities are taken as actions in response to the rewards achieved, which then becomes the state for the next controller action required. As the reward functions defined in the project are based on the distances and position of the gripper with respect to the object, the joint manipulator is controlled with POSITION CONTROLLER for both the tasks. The position control although generates unrealistic sudden changes to the robotic arm owing to its direct position outputs to the joints, is simpler to handle in a simulated environment.

III. HYPERPARAMETERS

The DQN hyperparameters are tuned for the RL agent to obtain the required accuracy assigned for the given tasks. Alongside reward function and joint controller, hyperparameters play a crucial role in the training of a reinforcement learning agent. The hyperparameters used in the project, are as explained below:

- The captured images from the camera state is condensed to a smaller input size to reduce complexity and help with convergence. The width and height is thereby reduced to a size of 64 X 64. Such smaller square size image, also optimizes the GPU operations.

```
#define INPUT_WIDTH 64
#define INPUT_HEIGHT 64
```

- The batch size for the neural networks has also been reduced to a value of 32, to further enhance the GPU processing and speed up the operations.

```
#define BATCH_SIZE 32
```

- In regards to the choice for an effective optimizer, RMSProp is given priority over Adam, SGD or others. RMSProp resulted in a faster convergence within 100 runs for both the tasks in the given simulation environment. Moreover with the values tuned for the rest of the hyperparameters, RMSProp is found to be the best suited.

```
#define OPTIMIZER "RMSprop"
```

- The learning rate for the training of neural networks is set to a value of 0.1 for objective 1 and 0.01 for objective 2. A lower learning rate ensures a better training and improved performance to the learning agent. As the objective 2 is more challenging, a reduced learning rate is given for better training within a few number of runs. A

value of 0.1 also ensures meeting the objective though at a delayed episode.

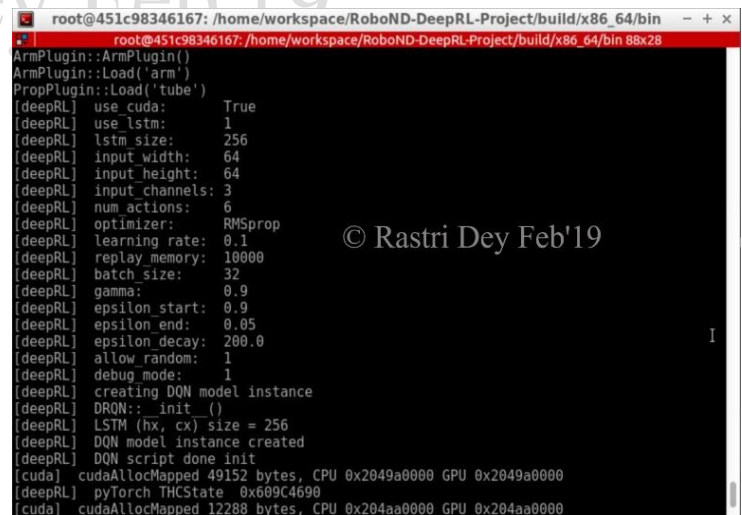
```
#define LEARNING_RATE 0.1f ..... Objective 1
#define LEARNING_RATE 0.01f ..... Objective 2
```

- Replay memory is a cyclic buffer of bounded size that holds the transitions observed recently, allowing to reuse the data later. By sampling from the data randomly, the transitions that build up a batch are decorrelated which greatly stabilizes and improves the DQN training procedure. This value is left unchanged at 10000.

```
#define REPLAY_MEMORY 10000
```

- Lastly, the LSTM architecture is used to keep track of both, the long-term memory and the short-term memory, where the short-term memory is the output or the prediction. This is a mean to keep track of the previous camera states instead of a single camera frame in the network internal memory. Utilizing LSTM as part of the network helps the RL agent learn better by tracking the useful past history alongside the recent predictions. The size of the data defines the requirement for the size of LSTM.

```
#define USE_LSTM true
#define LSTM_SIZE 256
```



```
root@451c98346167: /home/workspace/RoboND-DeepRL-Project/build/x86_64/bin - + x
root@451c98346167: /home/workspace/RoboND-DeepRL-Project/build/x86_64/bin 88x28
ArmPlugin::ArmPlugin()
ArmPlugin::Load('arm')
PropPlugin::Load('tube')
[deepRL] use_cuda: True
[deepRL] use_lstm: 1
[deepRL] lstm_size: 256
[deepRL] input_width: 64
[deepRL] input_height: 64
[deepRL] input_channels: 3
[deepRL] num_actions: 6
[deepRL] optimizer: RMSprop
[deepRL] learning_rate: 0.1
[deepRL] replay_memory: 10000
[deepRL] batch_size: 32
[deepRL] gamma: 0.9
[deepRL] epsilon_start: 0.9
[deepRL] epsilon_end: 0.05
[deepRL] epsilon_decay: 200.0
[deepRL] allow_random: 1
[deepRL] debug_mode: 1
[deepRL] creating DQN model instance
[deepRL] DRQN::init()
[deepRL] LSTM (hx, cx) size = 256
[deepRL] DQN model instance created
[deepRL] DQN script done init
[cuda] cudaAllocMapped 49152 bytes, CPU 0x2049a0000 GPU 0x2049a0000
[deepRL] pytorch THCState 0x609C4690
[cuda] cudaAllocMapped 12288 bytes, CPU 0x204aa0000 GPU 0x204aa0000
```

Fig. 1. Hyperparameters for Objective 1

```

root@5da43d938957: /home/workspace/RoboND-DeepRL-Project/build/x86_64/bin - + x
root@5da43d938957: /home/workspace/RoboND-DeepRL-Project/build/x86_64/bin 8x26
ArmPlugin::ArmPlugin()
ArmPlugin::Load('arm')
PropPlugin::Load('tube')
[deepRL] use_cuda: True
[deepRL] use_lstm: 1
[deepRL] lstm_size: 256
[deepRL] input_width: 64
[deepRL] input_height: 64
[deepRL] input_channels: 3
[deepRL] num_actions: 6
[deepRL] optimizer: RMSprop
[deepRL] learning_rate: 0.01
[deepRL] replay_memory: 10000
[deepRL] batch_size: 32
[deepRL] gamma: 0.9
[deepRL] epsilon_start: 0.9
[deepRL] epsilon_end: 0.05
[deepRL] epsilon_decay: 200.0
[deepRL] allow_random: 1
[deepRL] debug_mode: 1
[deepRL] creating DON model instance
[deepRL] DRQN::init()
[deepRL] LSTM (hx, cx) size = 256
[deepRL] DON model instance created
[deepRL] DON script done init
[cuda] cudaAllocMapped 49152 bytes, CPU 0x2049a0000 GPU 0x2049a0000

```

Fig. 2. Hyperparameters for Objective 2

Fig. 1 and Fig. 2, represents the hyperparameters used for training the RL agent in case of objective 1 and objective 2 respectively. The hyperparameters are tuned similarly for both the objectives defined in the project, except for learning rate values.

IV. RESULTS

The defined hyperparameters and reward functions resulted in high accuracy and convergence rate for both the objectives. Over 90% accuracy is obtained for both the tasks in less than 200 episodes. The arm at the beginning undergoes some fluctuations but with a few learning episodes, from 100 onwards or even before at certain circumstances, starts converging towards maximizing its cumulative reward.

- a. **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.**

To achieve the task of hitting the prop with any part of the robotic arm, the arm must manipulate it's joints at certain angles to receive the corresponding reward to get trained for the appropriate joint positions required to hit the prop. The robotic arm manipulator during a certain episode, exerts controller forces at its joints, which is driven by the position controls from the controller output. In the beginning, the robotic arm explores all the joint positions and gets penalized for the arm configuration that doesn't meet the criteria. This includes the instances of hitting the ground or fluctuating randomly mid-air for more than a 100 frames. The interim rewards guides the robotic arm towards decreasing its distance from the prop. Once the robotic arm hits the prop and achieves a REWARD WIN, it tries to increase the accuracy by repeating the joint actions in a repeatable fashion towards the prevailing joint positions. The configuration with respect to Objective 1, is learnt quickly by the arm and its repeated consistently for the upcoming episodes to increase the accuracy. The robotic arm starts converging around 45 episode and produces over 90% accuracy. This is represented

in Fig 3. The arm manipulator continues to increase its accuracy and as can be seen in Fig. 4 and Fig. 5, the robotic arm achieves an accuracy of 98% for over 100 runs.

```

root@5cdddbf8855: /home/workspa...ND-DeepRL-Project/build/x86_64/bin - + x
root@5cdddbf8855: /home/workspa...ND-DeepRL-Project/build/x86_64/bin 80x25
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9574 (045 of 047) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9583 (046 of 048) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9592 (047 of 049) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9600 (048 of 050) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9608 (049 of 051) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9615 (050 of 052) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9623 (051 of 053) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9630 (052 of 054) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9636 (053 of 055) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9643 (054 of 056) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]

```

Fig. 3. Early convergence of RL agent in Objective 1: 95% accuracy at 47 run

```

root@5cdddbf8855: /home/workspa...ND-DeepRL-Project/build/x86_64/bin - + x
root@5cdddbf8855: /home/workspa...ND-DeepRL-Project/build/x86_64/bin 80x25
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9792 (094 of 096) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9794 (095 of 097) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9796 (096 of 098) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9798 (097 of 099) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9800 (098 of 100) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9802 (099 of 101) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9804 (100 of 102) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9806 (101 of 103) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9808 (102 of 104) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9810 (103 of 105) (reward=+100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.9811 (104 of 106) (reward=+100.00 WIN)

```

Fig. 4. Accuracy for RL agent in Objective 1: 98% for more than 100 runs

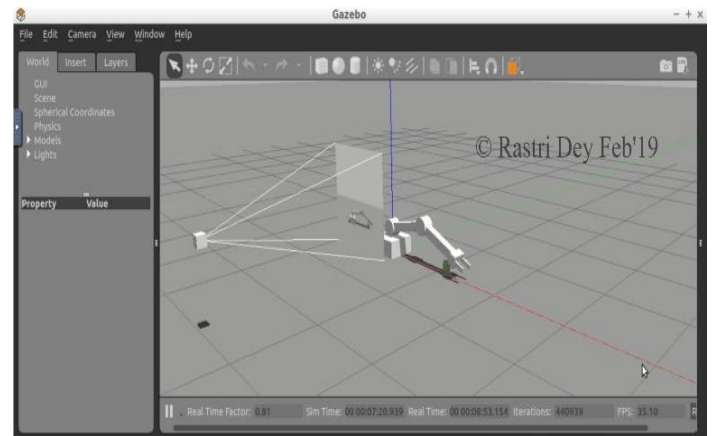


Fig. 5. Robotic arm hitting the prop in Objective 1

- b. *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 second task required a finer control to execute the joint positions satisfying the goal objective. The interim rewards defined, would drag the arm nearer to the object but only a satisfying collision with the gripper would earn the RL agent a REWARD WIN. The repetitive transitions while moving towards the object but touching only with the gripper might lead in multiple fluctuations while the arm approaches the prop. The appropriate joint positions near the object is learnt through continuous interaction for maximizing the rewards. The arm itself, would often hit the object or the ground while gaining negative rewards and penalizing itself for its actions. Through multiple episodes though, the arm learns to touch the prop with the gripper alone. The position controller, outputs the corresponding joint positions, giving the arm an arc shaped look that keeps hitting the prop through the learnt configuration in order to keep gaining rewards. The arm manipulator achieves an accuracy of over 80% satisfying the criteria within 150 episodes. Fig. 6 and Fig. 7 showcases the end results of the objective fulfilled, satisfying the criteria of the robotic manipulator hitting the prop with the gripper alone for more than a 100 runs while achieving more than 80% accuracy.

```

root@5da43d938957: /home/workspace/RoboND-DeepRL-Project/build/x86_64/bin - + x
root@5da43d938957: /home/workspace/RoboND-DeepRL-Project/build/x86_64/bin 87x24
Collision between[tube::tube link::tube collision] and [arm::gripperbase::gripper_link]
Current Accuracy: 0.8014 (113 of 141) (reward+=100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::gripperbase::gripper_link]
Collision between[tube::tube link::tube collision] and [arm::gripperbase::gripper_link]
Current Accuracy: 0.8028 (114 of 142) (reward+=100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Collision between[tube::tube link::tube collision] and [arm::gripperbase::gripper_link]
Collision between[tube::tube link::tube collision] and [arm::gripperbase::gripper_link]
Current Accuracy: 0.8042 (115 of 143) (reward+=100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::gripperbase::gripper_link]
Collision between[tube::tube link::tube collision] and [arm::gripperbase::gripper_link]
Collision between[tube::tube link::tube collision] and [arm::gripperbase::gripper_link]
Collision between[tube::tube link::tube collision] and [arm::gripperbase::gripper_link]
Current Accuracy: 0.8056 (116 of 144) (reward+=100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::gripperbase::gripper_link]
Collision between[tube::tube link::tube collision] and [arm::gripperbase::gripper_link]
Current Accuracy: 0.8069 (117 of 145) (reward+=100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Collision between[tube::tube link::tube collision] and [arm::link2::collision2]
Current Accuracy: 0.8014 (117 of 146) (reward=-100.00 LOSS)
Collision between[tube::tube link::tube collision] and [arm::gripperbase::gripper_link]
Collision between[tube::tube link::tube collision] and [arm::gripperbase::gripper_link]
Current Accuracy: 0.8027 (118 of 147) (reward+=100.00 WIN)
Collision between[tube::tube link::tube collision] and [arm::gripperbase::gripper_link]

```

Fig. 6. Accuracy for RL agent in Objective 2: 80% for more than 100 runs

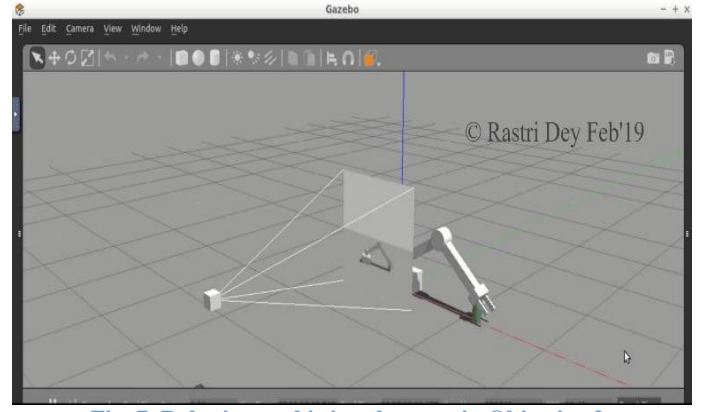


Fig. 7. Robotic arm hitting the prop in Objective 2

It can be observed that the RL agent performs exceedingly well in objective 1 giving 98% accuracy and starts converging at 90% within 50 runs. Though this might not be the case always and it might take more episodes to converge in certain scenarios. Still in general, most of the times, the arm manipulator attains at least 90% accuracy within 100 runs. In objective 2, on the other hand the arm manipulator attains 80% accuracy within 150 runs and 90% accuracy within 250 runs. Overall, the objectives stated in the project are achieved with a good performance meeting the well bounding constraints within a few number of episodes.

V. CONCLUSION & FUTURE WORK

The robotic arm manipulator meets the expectations while getting trained under a reinforcement learning architecture. Reinforcement Learning can be used to find a good solution to a problem by focusing on goal rather than the steps to reach the goal. With RL, the industrial robots can train themselves to perform new tasks, saving the requirement for programming techniques exploiting engineering resources. Thus, RL in long term can save a lot of money, effort and resources in terms of business perspective, while outperforming in the allotted tasks through self-learning from its surrounding environment. This encourages for making the RL agent's system even more intelligent and efficient for robust control actions. The environment provided in the project is comparatively a simpler world. The RL agent in real time would face highly dynamic and complex world which requires for testing the adaptability of the robot in the underlying RL architecture, for instance, in an RL agent for self-driving car in a real city traffic conditions. A well-engineered reward system could prevent the robot from indecisions, for example, the results in the current project can be improved with better reward functions that allows the robotic arm for getting near the prop while touching only with a particular part of the robotic arm in task 2. Also, the RL agent could be trained through penalizing for exceeding the control forces or sudden jerks at the joints to restrict it under the physical limitations of a real world robot. A real world robot cannot sustain the sudden position changes that comes as an output from a position controller, penalizing such actions would create more realistic robots. The reward function could further be enhanced with different moving

average techniques such as Simple Moving Averages (SMA), Exponential Moving Averages (EMA), Linear Weighted Moving Averages etc., acquiring more effective interim rewards. The hyperparameters could also be tuned with a different optimizer such as Adam, AMSGrad, SGD etc. The controller used in the project, is a position controller which might not be feasible in a real robotic system owing to its sudden changes in positions out limiting the arm's physical constraints of movement. This can be tried out with velocity control instead or, could be entirely replaced with a better configuration, that directly controls the arm's joint angles. A path could be planned for the arm's trajectory to reach the prop and the joint angles could be calculated in reference to the designed trajectory. This provides a direct control of the robot for the assigned task while limiting its risks for touching the ground or keep swinging mid-air. Such kind of system would result in intelligent actions while enhancing the convergence rate at a greater extent.

VI. REFERNECES

[1] Udacity, Robotics Software Engineer Nanodegree Program Term2: Project Deep RL Arm Manipulation, Udacity, 2019.

Rastri Dey Feb'19