Robotics Deep Learning Arm Manipulation

Amay Yadav

Abstract—In this project I have applied Deep Reinforcement Learning algorithm to train the robotic arm to touch the target object in a simulated environment. There are two primary objectives of this project: (1) Have any part of the robot arm touch the object of interest, with at least a 90% accuracy and (2) Have only the gripper base of the robot arm touch the object, with at least a 80% accuracy. This paper will demonstrate how those two goals have been successfully achieved using simulated Gazebo environment.

Index Terms—Robot, Udacity, Robotic arm manipulation, Deep Reinforcement Learning.

1 Introduction

Reinforcement learning (RL) enables a robot to automatically discover an optimal behavior through trial-and-error interactions with its environment. For instance if we train a Robotic Arm to catch a ball then the RL algorithm wil learn over time to make observations of dynamic variables specifying ball position and velocity and then will figure out how to move it's own joints and position to catch the ball effectively. The training of this RL system will capture the states of the system providing a complete observation of the Robotic Arm Kinematics and statistic for predicting future observations. The actions available to the robot might be the torque sent to motors or the desired accelerations sent to an inverse dynamics control system. A function is called the policy that generates the motor commands that in effect is actuates the action of catching based on the incoming ball and current internal arm position also called observation or the current state of the arm. A RL algorithm is meant to compute a policy that optimizes the long term sum of rewards [1].

In this project we are not trying to catch a ball but have the Robot touch the target object. We are going to use PyTorch DQN which is a great library for deep reinforcement learning for problems like these where we take sensor input and produce actions. PyTorch supplied us with an API which we can just call and not worry about underlying details just like abstraction. The PyTorch library is in Python but the low level code is leverages C to pass memory objects between the users application which makes the library faster and optimized. The performance can be further improved by using GPU acceleration [2]. For this Deep RL Arm Manipulation project, the goal is to create a DQN agent and define reward functions to teach a robotic arm to touch target object.

2 Background

In Reinforcement Learning (RL), an agent tries to maximize the accumulated reward over its life-time. In an episodic setting, where the task is restarted after each end of an episode, the objective is to maximize the total reward per episode. If the task is ongoing without a clear beginning and end, either the average reward over the whole life-time or a discounted return (i.e., a weighted average where distant rewards have less influence) can be optimized. In such reinforcement learning problems, the agent and its environment may be modeled being in a state (S) and can perform an action (A), each of which may be members of either discrete or continuous sets and can be multidimensional [1].

Classical reinforcement learning approaches are based on the assumption that we have a Markov Decision Process consisting of:

- Set of states S
- Set of actions A.
- Rewards R.
- Transition probabilities T that capture the dynamics of a system. Transition probabilities (or densities in the continuous state case) T(s,a,s) = P(ss,a) describe the effects of the actions on the state.

Deep-Q-Networks (DQN) uses the previous scenarios to keep training the neural networks and find the approximation of policy function. In this project, the power of DQN is leveraged with a classical arm manipulation problem. The two objectives of this project consist are as follows:

1) Have any part of the robot arm touch the object of interest, with at least a 90% accuracy.
2) Have only the gripper base of the robot arm touch the object, with at least a 80% accuracy..

This project is completed on Jetson TX2. The environment is simulated on Gazebo. There are three main components are:

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

3 Reward Functions

The reward of a Deep Q-Network (DQN)is mapped to the action. Action for the robotic arm can be the control of the joint which can be measured through position. Usually DQN can use velocity or position but in this case we have use position as the reward funciton and the step by step rewardHistory function are both based on the distance from target. As the objective is to touch the target object we can

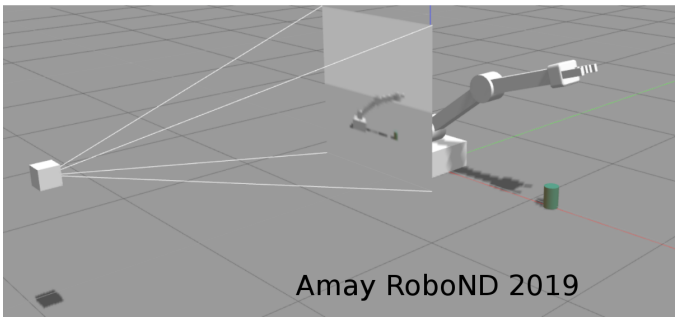

Fig. 1. Robot Deep RL Arm Manipulation.

calculate in each step how much farther away are we from it and use the position for calculating our reward.

The reward system has been designed to train the robotic arm to reach its primary objective i.e to touch the target object. Reward functions defined are simple they are geared towards success i.e if the arm touches the object of interest then we add the numerical REWARD value. Positive and negative rewards are defined as numeric constants REWARD_WIN and REWARD_LOSS. A win is when task is accomplished and REWARD_WIN is added to the overall output. A loss is when the arm when the arm touches the ground or when the length of the episode exceeds 100 steps at which point we start over and we add (subtract as its negative in value) REWARD_LOSS from the total reward. The occurrence of any of these events will end that particular episode.

Its obvious that we can't solely rely on the aforementioned reward functions because we need to guide the arm towards the object too because positive rewards from touching the prop by the gripper is rare. We need to define an interim reward function. Its purpose is to guide the robot arm towards the target. The relevant measure is the distance distDelta between the gripper and target object between consecutive steps. We need to normalize the value of avgGoalDelta and for that we multiply to avgGoalDelta by 0.4f and distDelta by 0.6f to get a normalized value of avgGoal. avgGoalDelta is nothing by the difference of avgGoal in consecutive steps.

$$distDelta = lastGoalDistance - distGoal \quad (1)$$

$$avgGoal = 0.4(avgGoalDelta) + 0.6(distDelta) \quad (2)$$

$$rewardHistory = (4 * avgGoalDelta) - 0.3 \quad (3)$$

A avgGoalDelta is a very good measure for rewardHistory as it captures the difference between current position and the goal in a linear time based fashion. The coefficients for all the equations are all from trial and error.

4 Hyperparameters

Most of the hyperparameters that were provided in the code were slightly modified for better results. The Long Short-Term Memory (LSTM) architecture was enabled. The input height and width was decreased from 512x512 to 64x64. The Adam optimizer was used to achieve both the objectives. The batch size was initially set to 8 which is very low and it was cranked up-to 32 for objective 1 and then to 256 for objective 2 for better results. The LSTM size was increased from 64 to 256 so that the agent can memorize more information and calculate better rewards the next time around. During the simulation runs for objective 2. The EPS_START value was tuned down from 0.9 to 0.8 and the EPS_END value was tuned from 0.05 to 0.02. The learning rate was set to 0.1 and the left untouched as this is optimal value. All the changed hyperparameters are listed below.

TABLE 1
Hyperparameters

Hyperparameters	Goal 1	Goal 2
GAMMA	0.9	0.9
EPS_START	0.8	0.8
EPS_END	0.02	0.02
EPS_DECAY	200	200
INPUT_WIDTH	64	64
INPUT_HEIGHT	64	64
OPTIMIZER	Adam	Adam
LEARNING_RATE	0.1	0.1
REPLAY_MEMORY	10000	10000
BATCH_SIZE	32	256
LSTM_SIZE	64	256
REWARD_WIN	20	20
REWARD_LOSS	-20	-20

5 Result

The reward functions was able to achieve both the objectives without a lot of changes. The linear time-based reward function was the best metric. The accuracy reached in both the cases is listed below

1) The accuracy for goal 1 is around 0.95.
2) The accuracy for goal 1 is around 0.90.

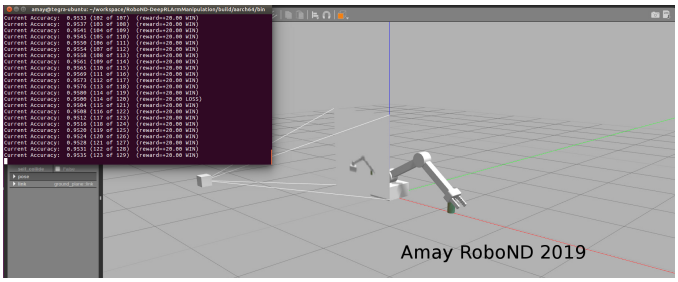

Fig. 2. Goal 1 with reward output in console and Gazebo output.

6 Discussion

As you can see the results that are achieves are quite over the expectation of 90% for goal 1 and 80% for goal 2 for a minimum of 100 runs which is not bad at all a. The number of episodes it took the agent to reach the threshold accuracy varied between 200 and 600. As is expected from a RL algorithm the accuracy improved slowly over time. The rewardHistory equation took a bit tuning but otherwise it was smooth. Initialilly the coefficient for rewardHistory was set to 2 and it was taking too many runs so it was increased to 8 and then down to 4.

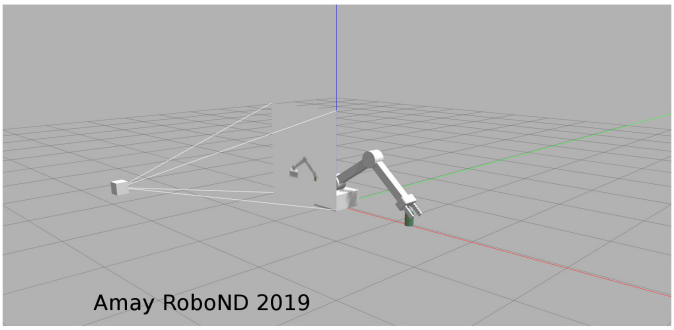

Fig. 3. Goal 2 Gazebo output.

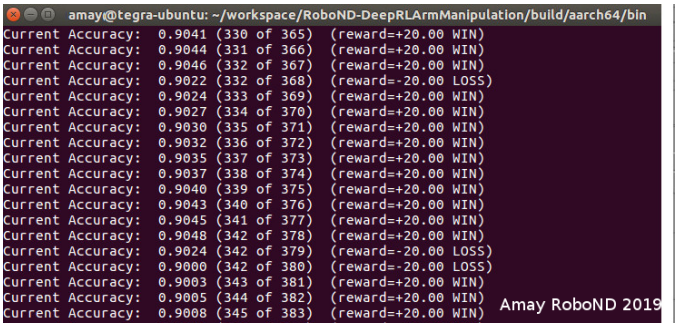

Fig. 4. Goal 2 with reward output in console.

7 Conclusion / Future work

The two primary objectives of the project were achieved. We can further improve the RL framework and we can keep tuning the DQN hyperparameters. We can also invest in preprocessing the images and probably look at Kinematics of the arm.

References

[1] A. G. B. Richard S. Sutton, *Reinforcement Learning: An Introduction*. The MIT Press, 2015.
[2] lecturers, *Udacity RoboticsND lectures*. Udacity, 2019.