Deep RL Arm Manipulation

Naruhiko Nakanishi

Abstract—A DQN agent is created and reward functions are defined to teach a robotic arm. Two primary objectives are carried out and achieved. One objective is to have any part of the robot arm touch the object of interest, with at least a 90 percent accuracy. The other objective is to have only the gripper base of the robot arm touch the object, with at least a 80 percent accuracy. The hyperparameters are tuned well also.

Index Terms—Robot, Deep RL, DQN agent, reward, hyperparameter.

1 Introduction

The PyTorch DQN demonstrates how deep reinforcement learning works for problems that take sensor input and produce actions.

A library format that can integrate with robots and simulators is important to successfully leverage deep learning technology in robots. In addition, robots require real-time responses to changes in their environments, so computation performance matters.

An API (application programming interface) in C/C++ provides an interface to the Python code written with PyTorch, but the wrappers use Pythons low-level C to pass memory objects between the users application and Torch without extra copies. By using a compiled language (C/C++) instead of an interpreted one, performance is improved, and speeded up even more when GPU acceleration is leveraged. [1]

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 carry out two primary objectives.

1.1 Objective 1

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.

1.2 Objective 2

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.

2 Simulations

2.1 Project Environment

This project is completed on Jetson TX2.

Fig.1 shows a robot of Deep RL Arm Manipulation. The environment uses the C++ API and Gazebo.

In the project repo, the gazebo-arm.world file in /gazebo/ can be located. There are three main components to this gazebo file, which define the environment:

- 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.

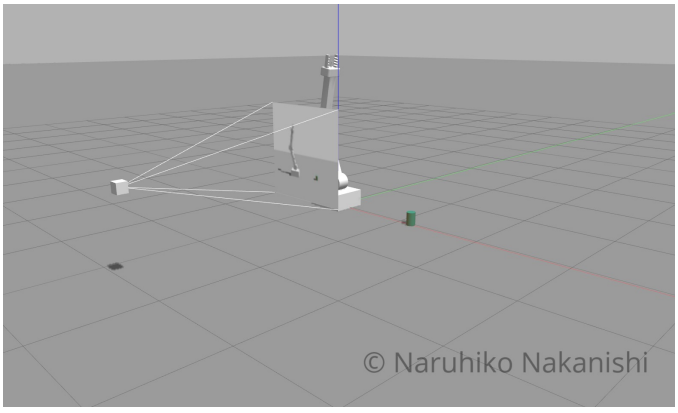

Fig. 1. Deep RL Arm Manipulation Robot

One of the tasks for the project is to create an RL agent for the robotic arm that will learn to manipulate its joints to reach and touch the object placed in its vicinity. The arm has three non-fixed joints excluding the gripper.

2.2 Arm Plugin

The robotic arm model, found in the gazebo-arm.world file, calls upon a gazebo plugin called the ArmPlugin. This plugin is responsible for creating the DQN agent and training it to learn to touch the prop. The gazebo plugin shared object file, libgazeboArmPlugin.so, attached to the robot model in gazebo-arm.world, is responsible for integrating the simulation environment with the RL agent. The plugin is defined in the ArmPlugin.cpp file, also located in the gazebo folder.

The ArmPlugin.cpp file takes advantage of the C++ API covered earlier. This plugin creates specific constructor and member functions for the class ArmPlugin defined in ArmPlugin.h. [1]

3 Tasks

Since no DQN agent has been defined, the arm isn't learning anything and has no input to control it.

Once the gazebo environment loads up, the robotic arm, a camera sensor, and an object are observed in the environment. The gazebo arm falls to the ground after a short while, and the terminal continuously displays the following message (fig.2):

ArmPlugin - failed to create DQN agent

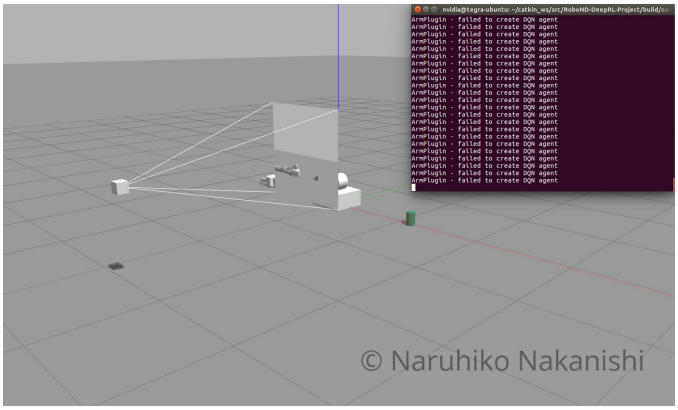

Fig. 2. The robot arm falling to the ground

The ArmPlugin.cpp file has specific TODOs or tasks listed out for the Project. Each of those tasks is completed in the following order.

3.1 Task 1. Subscribe to camera and collision topics.

The nodes corresponding to each of the subscribers have already been defined and initialized. The subscribers in the ArmPlugin::Load() function are created.

3.2 Task 2. Create the DQN Agent

The DQN agent is created using the Create() function from the dqnAgent Class, in ArmPlugin::createAgent().

3.3 Task 3. Velocity or position based control of arm joints

In ArmPlugin::updateAgent(), there are two existing approaches to control the joint movements.

- Velocity Control
- Position Control

For this project, position type of joint control is implemented because the rewards are based on the distance to the object.

3.4 Reward Functions

Reward functions are created and assigned. There are a few important variables in relation to rewards.

- rewardHistory - Value of the previous reward, you can set this to either a positive or a negative value.
- REWARD WIN or REWARD LOSS - The values for positive or negative rewards, respectively.
- newReward - If a reward has been issued or not.
- endEpisode - If the episode is over or not.

REWARD WIN is set to 10 (20 for objective 2) with REWARD LOSS to -10 (-20 for objective 2).

3.5 Task 4. Reward for robot gripper hitting the ground

In Gazebos API, there is a function called GetBoundingBox() which returns the minimum and maximum values of a box that defines that particular object/model corresponding to the x, y, and z axes.

Using the above, it is checked if the gripper is hitting the ground or not, and an appropriate reward is assigned. The bounding box for the gripper, and a threshold value have already been defined in the ArmPlugin::OnUpdate() method.

3.6 Task 5. 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. Then, this distance is used to calculate a reward.

The reward is a smoothed moving average of the delta of the distance to the goal. It is calculated as:

average delta = (average delta * alpha) + (dist * (1 - alpha));

3.7 Task 6. Issue a reward based on collision between the arm and the object.

This reward function help with the first part of the objectives.

In the callback function onCollisionMsg, a check condition is defined to compare if particular links of the arm with their defined collision elements are colliding with the COLLISION ITEM or not. Then the appropriate rewards are assigned.

3.8 Test the Project

At this stage, the project is tested.

Once the environment/gazebo is loaded, it is noticed that the arm trying to move but it might not be learning.

3.9 Task 7. Tuning the hyperparameters

The list of hyperparameters is provided in ArmPlugin.cpp file, at the top. The hyperparameters are tuned to obtain the required accuracy.

Fig.3 shows the hyperparameters for object 1.

The hyperparameters are set via trial and error. OPTIMIZER Adam that generally performs better than RMSProp is chosen. LEARNING RATE is 0.1 with REPLAY MEMORY at 10000. BATCH SIZE is set to 32. LSTM is used USE LSTM true with LSTM SIZE 64.

Fig.4 shows Robot arm manipulation for object 1.

The image of the terminal displays the accuracy for when any part of the arm is touching the object and the robot arm in action.

```
[deepRL]  use_cuda:       True
[deepRL]  use_lstm:       1
[deepRL]  lstm_size:      64
[deepRL]  input_width:    64
[deepRL]  input_height:   64
[deepRL]  input_channels: 3
[deepRL]  num_actions:    6
[deepRL]  optimizer:      Adam
[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:     0
[deepRL]  creating DQN model instance
[deepRL]  DRQN::__init__()
[deepRL]  LSTM (hx, cx) size = 64
[deepRL]  DQN model instance created
[deepRL]  DQN script done init
[cuda]  cudaAllocMapped 49152 bytes, CPU 0x101340000 GPU 0x101340000
[deepRL]  pyTorch THCState  0x68F55070
[cuda]  cudaAllocMapped 12288 bytes, CPU 0x101540000 GPU 0x101540000
ArmPlugin - allocated camera img buffer 64x64  24 bpp  12288 bytes
[deepRL]  nn.Conv2d() output size = 800
```


Fig. 3. Object 1 Parameters

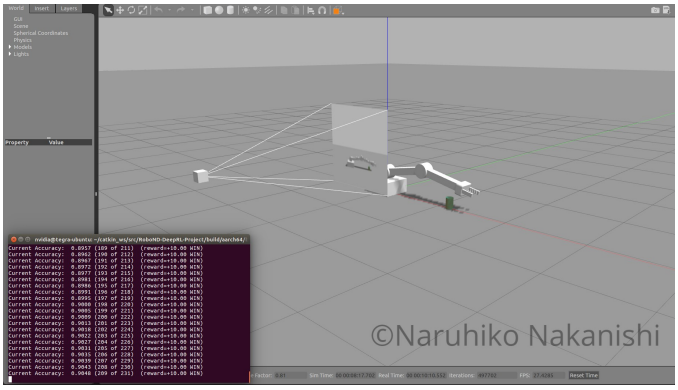

Fig. 4. Robot arm manipulation for object 1

```
[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:      Adam
[deepRL]  learning rate:  0.1
[deepRL]  replay_memory:  10000
[deepRL]  batch_size:     256
[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:     0
[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 0x101340000 GPU 0x101340000
[deepRL]  pyTorch THCState  0x3CF559F0
[cuda]  cudaAllocMapped 12288 bytes, CPU 0x101540000 GPU 0x101540000
ArmPlugin - allocated camera img buffer 64x64  24 bpp  12288 bytes
[deepRL]  nn.Conv2d() output size = 800
```


Fig. 5. Object 2 Parameters

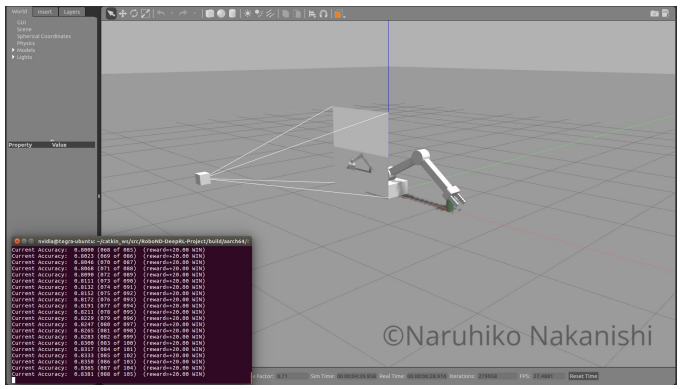

Fig. 6. Robot arm manipulation for object 2

3.10 Task 8. Issue a reward based on collision between the arms gripper base and the object.

The collision check defined in task 6 is modified to check for collision between the gripper base and the object. And then, the appropriate reward is assigned.

The project is run again, and it is observed how the agent performs. The rewards or the hyperparameters are tuned to obtain the required accuracy.

Fig.5 shows the hyperparameters for object 2.

The hyperparameters are set via trial and error. OPTIMIZER Adam is chosen. LEARNING RATE is 0.1 with REPLAY MEMORY at 10000. BATCH SIZE is set to 256. LSTM iss used USE LSTM true with LSTM SIZE 256.

Fig.6 shows Robot arm manipulation for object 2.

The image of the terminal displays the accuracy for when only the gripper base is touching the object and the robot arm in action.

4 Results/ Discussion

A DQN agent is created and reward functions are defined to teach a robotic arm to carry out two primary objectives.

After the hyperparameters are tuned, the objective 1 is achieved. The objective is to have any part of the robot arm touch the object of interest, with at least a 90 percent accuracy.

And then, after another tuning, the objective 2 is achieved. The objective is to have only the gripper base of the robot arm touch the object, with at least a 80 percent accuracy.

The hyperparameters are tuned well to obtain the both of the required accuracies.

Overall the agent performs well to achieve the objectives. However, it sometimes happens that the agent fails to learn about what to do initially, then it takes much time to achieve a higher accuracy, especially for the objective 2.

5 Conclusion/ Future Work

A DQN agent is created and reward functions are defined to teach a robotic arm. Two primary objectives are carried out and achieved. The hyperparameters are tuned well also.

For future work, the approach to improve the current results would be tuning the remaining hyperparameters such as optimizer, size of replay memory size, and epsilon decay rate.

In addition, to improve the results, the interim reward would use other type of moving average than smoothed moving average such as simple moving average, exponential moving average, and linear weighted moving average.

References

[1] Udacity, *Robotics Software Engineer Nanodegree Program Term2 Lesson25: Deep RL Manipulator*. Udacity, 2018.