Deep Reinforcement Learning: Robotic Arm Manipulation

This project is based on the Nvidia open source project "jetson-reinforcement" developed by Dustin Franklin. The goal of the project is to create a DQN agent and define reward functions to teach a robotic arm to carry out two primary objectives:

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

Setup

The DQN agent was trying to manipulate a robotic arm to reach certain object in the environment using a camera feed. The arm and the object were simulated in gazebo, the DQN agent was written using Pytorch and the interface between them was made using a plugin that allowed to use C++ to interface between the two and to achieve high utilization of hardware resources.

Code Details

The project was done using Gazebo simulated environment with a C++ plug-in to interface Gazebo Simulated Robot to the DQN agent. To achieve the objectives required, the following tasks were completed:

- Subscribe to camera and collision topics published by Gazebo.
- Create the DQN Agent and pass all required parameters to it.
- Define position control function for arm joints.
- Penalize robot gripper hitting the ground.
- Interim Reward/Penalize based on the arm distance to the object.
- Reward based on collision between the arm and the object.
- Tuning the hyperparameters.
- Reward based on collision between the arm’s gripper base and the object.

Next sections will explain in detail how the required tasks were achieved.

Nodes Creation

In **Armplugin.cpp**, inside the method ArmPlugin::Load(), subscriber nodes were defined to facilitate camera communication and collision detection through Gazebo:

```
// Create our node for camera communication

cameraNode->Init();

cameraSub = cameraNode->Subscribe("/gazebo/" WORLD_NAME "/camera/link/camera/image",
&ArmPlugin::onCameraMsg, this);

// Create our node for collision detection

collisionNode->Init();

collisionSub = collisionNode->Subscribe("/gazebo/" WORLD_NAME "/" PROP_NAME
"/link/my_contact", &ArmPlugin::onCollisionMsg, this);
```

Define the Agent

In the method ArmPlugin::Create(), the DQN Agent was created using the parameters defined at the top of **Armplugin.cpp**:

```
// Create DQN Agent

agent = dqnAgent::Create(INPUT_WIDTH, INPUT_HEIGHT, INPUT_CHANNELS, DOF*2, OPTIMIZER,
LEARNING_RATE, REPLAY_MEMORY, BATCH_SIZE, GAMMA, EPS_START, EPS_END, EPS_DECAY, USE_LSTM,
LSTM_SIZE, ALLOW_RANDOM, DEBUG_DQN);
```

Define Position Control

The output of the DQN agent is mapped to actions of the robotic arm. The method ArmPlugin::updateAgent() is responsible for executing the action selected by the DQN agent. Two types of joint control were implemented: velocity control and position control. Both are dependent upon whether the issued action is an odd or even number. Odd-numbered actions decrease the joint position/velocity, while even-numbered actions increase the joint position/velocity. Additionally, the current delta of the position/values is used:

```
// Increase or decrease the joint position based on whether the action is even or odd

const int jointIdx = action / 2;

// Set joint position based on whether action is even or odd.

float joint = ref[jointIdx] + actionJointDelta * ((action % 2 == 0) ? 1.0f : -1.0f);
```

Reward Function

The reward function is a crucial component in guiding the DQN agent through the process of learning to manipulate the arm. After every end of episode (EOE), a particular reward is issued, depending on the triggering event.

REWARD_LOSS is issued when the current episode exceeds 100 steps or when the arm contacts the ground.

The REWARD_LOSS issued when the arm contacts the ground is detected using the GetBoundingBox() function from the Gazebo API which provides the min/max XYZ values of the arm. Comparing this with the Z value of the ground, collisions can be detected.

```
if(gripBBox.min.z <= groundContact || gripBBox.max.z <= groundContact)

{

    // set appropriate Reward for robot hitting the ground.

    …

    …

}
```

REWARD_WIN is issued when any part of the arm contacts the object (Objective 1) and when only the gripper contacts the object (Objective 2).
The reward function for arm and object collision first checks for a collision between any part of the arm and the object. Depending on which objective was being attempted, the macro GRIPPER_ONLY controls whether the arm-object collision or gripper-object collision is a win.

If GRIPPER_ONLY is not defined, objective 1 is achieved, and the arm-object collision is considered a win.

If GRIPPER_ONLY is defined, objective 2 is achieved, and the arm-object collision is considered a loss by default. However, if that part happens to be the gripper, the collision issues a reward for our agent.

```
// Comment The following line to achieve object 1. If it's left defined, objective 2 (only the gripper-object) collision is considered a win.

#define GRIPPER_ONLY

…

…

// onCollisionMsg

void ArmPlugin::onCollisionMsg(ConstContactsPtr &contacts)

{

#ifdef GRIPPER_ONLY

    if((strcmp(contacts->contact(i).collision1().c_str(), COLLISION_ITEM) == 0) &&
    strcmp(contacts->contact(i).collision2().c_str(), COLLISION_POINT) == 0 && !testAnimation)

#else

if((strcmp(contacts->contact(i).collision1().c_str(), COLLISION_ITEM) == 0))

#endif
```

Interim Reward

To reduce the necessary training time, an interim reward function was implemented which would help guide the arm towards the object. The reward is proportional to the change in distance between the arm and the object. If the arm is moving away from the object, the interim reward is negative. Conversely, if the arm is moving towards the object, the interim reward is positive. The raw values cause the arm to have a jerking movement, so they are smoothed by

taking a moving average. Finally, due to the agent's tendency to remain still, a time penalty was introduced to force the agent to finish as quickly as possible.

```
// The current distance to the goal
const float distGoal = BoxDistance(gripBBox,propBBox);

// The current change in distance from the last position
const float distDelta  = lastGoalDistance - distGoal;

// Smoothing the values using a moving average
avgGoalDelta  = (avgGoalDelta * ALPHA) + (distDelta * (1.0f - ALPHA));

// Time penalty to reduce stand-still and completion time
rewardHistory = REWARD_INTERIM * avgGoalDelta - TIME_PENALTY;
```

Hyperparameters

Below are all the relevant parameters used. Between both objectives.

In Q-Learning, epsilon-Greedy is a typical exploration method used to encourage an agent to explore more of the state-action space. The larger the value, the more frequently a random action is chosen instead of one with the highest q-value. `EPS_START` defines the initial value. **0.9f** ensures the agent is exposed to many state-action spaces. `EPS_END` defines the final value. A **0.05f** value was used for it. Although it is better to zero it as this environment is static and unlikely to change. However, that was not used here. `EPS_DECAY` defines the rate by which the value decays from initial value to final one over time. It was set to 200.

```
// Define DQN API Settings

#define INPUT_CHANNELS 3

#define ALLOW_RANDOM true

#define DEBUG_DQN true

#define GAMMA 0.9f

#define EPS_START 0.9f

#define EPS_END 0.05f

#define EPS_DECAY 200
```

The input dimensions were set to match that of the image captured by the camera: 64 x 64 pixels. "RMSprop" was used as the optimizer. LSTM was set to **true** to enable the agent to consider previously encountered states. LSTM_SIZE was set to **256**, increasing the agent's ability to consider more complex moves. A LEARNING_RATE of **0.1f** was sufficient. Finally, the BATCH_SIZE was set to 32

```
// Tune the following 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 256
```

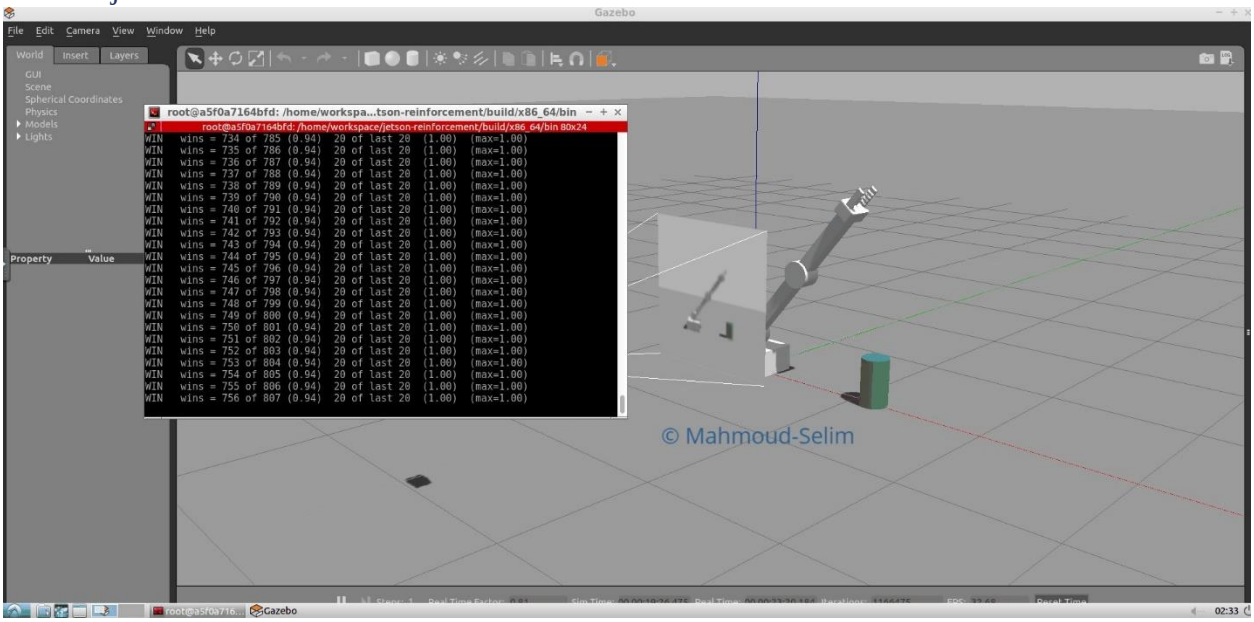
REWARD_INTERIM was the reward incrementally issued to lead the agent towards the object. REWARD_WIN was the winning reward, and REWARD_LOSS was the loss reward. They were set to +500 and -500 respectively.

```
// Define Reward Parameters

#define REWARD_WIN   500.0f

#define REWARD_LOSS -500.0f

#define REWARD_MULTIPLIER 10.0f
…

…

// Issue an interim reward based on the distance to the object

const float distGoal = BoxDistance(gripBBox, propBBox); // compute the reward from distance to
the goal

// issue an interim reward based on the delta of the distance to the object

if( episodeFrames > 1 )

{

    const float distDelta  = lastGoalDistance - distGoal;

    const float movingAvg  = 0.9f;

    const float timePenalty  =  0.50f;


    // compute the smoothed moving average of the delta of the distance to the goal

    avgGoalDelta  = (avgGoalDelta * movingAvg) + (distDelta * (1.0f - movingAvg));

    rewardHistory = avgGoalDelta * REWARD_MULTIPLIER - timePenalty;

    newReward     = true;

}
```

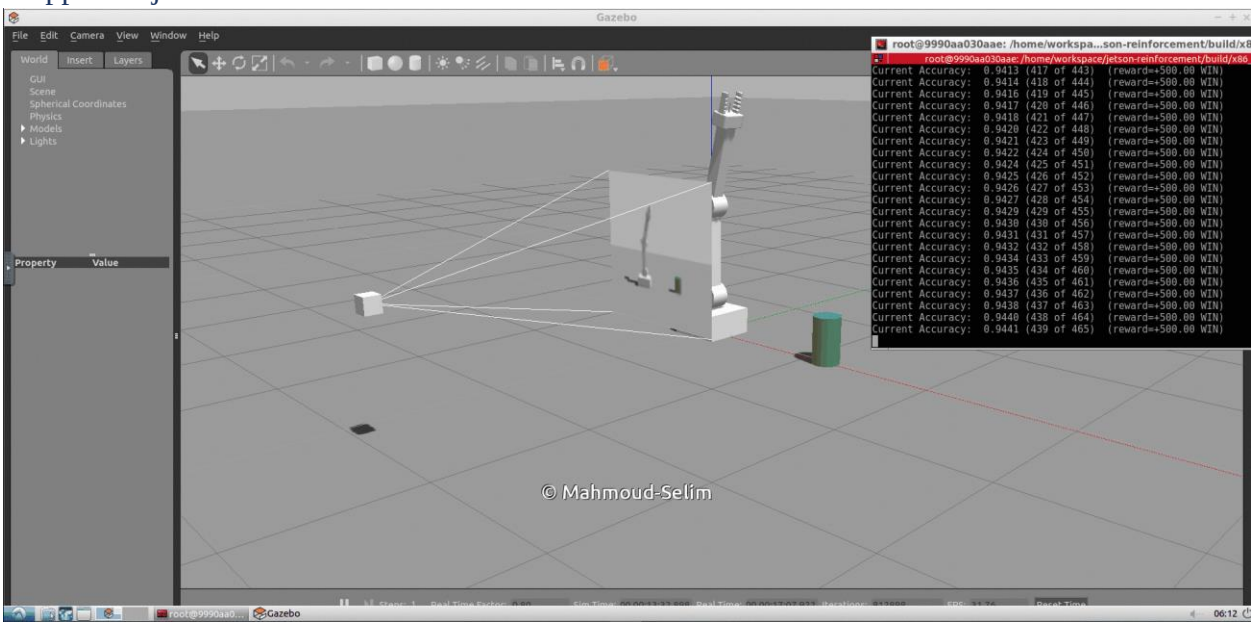
Results

Both objectives were achieved and accuracy higher than 90% were achieved for both of them.

Arm-Object Collision



Gripper-Object Collision



Future Work

With enough GPU time, the performance of the DQN agent could be improved by employing a more extensive approach to tuning. Each set of parameters could be run multiple times to decrease the effect of varying initial episodes. Also, the base locking could be removed and the desired accuracy would be achieved.