Deep RL Arm Manipulation

Miguel A. Colmenares Barboza

Abstract — **The project presented in this document focuses on implementing a Deep Q-Learning Network (DQN) agent developed by Dustin Franklin in the "Jetson-Reinforcing" project of Nvidia and the compensation functions to teach a robotic robot to perform two main objectives:**

- **Click to see the photo of the robotic touch loot, with a minimum of 90% accuracy and a minimum of 100 runs.**
- **Click on the base of the robot's boot clamp to touch the object, with at least 80% accuracy for a minimum of 100 runs.**

Both objectives are achieved by adjusting the values of rewards and parameters in the code.

Index Terms — Robotics, IEEEtran, Reinforcement Learning, DQN**.**

I. INTRODUCTION.

Deep reinforcement learning is at the forefront of artificial intelligence research. Many experts see it as a path towards General Artificial Intelligence. In this project, an artificially intelligent agents is created that learns from the interaction with their environment, the collection of experience and a reward system with a deep reinforcement learning (deep RL). Using end-to-end neural networks that translate raw pixels into actions, the agent trained in RL will be able to exhibit intuitive behavior and perform complex tasks.

II. REWARDS FUNCTIONS.

The most relevant configuration or programming is centered on the `ArmPlugin.cpp` file located in the `/gazebo` folder. The `ArmPlugin.cpp` file is a complement to which the robotic arm model shown in the simulation uses. This add-on is responsible for creating the DQN agent and training it to learn to play the accessory. The shared object file of the gazebo complement `libgazeboArmPlugin.so`, attached to the robot model in `gazebo-arm.world`, is responsible for integrating the simulation environment with the RL agent.

In the file `ArmPlugin.cpp` a series of tasks are programmed, exposed below.

✱ Subscribe to camera and collision topics:
The nodes corresponding to each one of the subscribers have already been defined and initialized. Subscribers are only created in the `ArmPlugin::Load()` function.

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

collisionSub = collisionNode -> Subscribe
("/gazebo/arm_world/tube/tube_link/my_contact",
&ArmPlugin::onCollisionMsg, this);
```

The next set of tasks is based on the creation and assignment of reward functions based on the required objectives.

✱ **Create the DQN agent:**
By consulting the API instructions to create the agent using the `Create()` function of the dqnAgent class, in the `ArmPlugin::createAgent()` function.

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

✱ **Define a control based on the position of the arm joints:**
The output of the DQN assigns a particular action, which, for this project, is the control of each articulation for the robotic arm. In `ArmPlugin::updateAgent()`, there are two existing approaches to controlling joint movements, an approach based on the speed of the arm joints and another approach based on the position of the arm joints.

In this project, we chose to use the position-based approach. The position is a value that simply changes and the speed is a value that varies based on the zero and the change of the value is more frequent and can be difficult to capture. For this reason, the control based on the position supposed to be a simple solution.

```
float joint = (action %2 == 0) ? ref[action/2] +
actionJointDelta : ref[action/2] - actionJointDelta;

if( joint < JOINT_MIN )

        joint = JOINT_MIN; // #define JOINT_MIN -0.75f

if( joint > JOINT_MAX )

        joint = JOINT_MAX; // #define JOINT_MAX  2.0 f

ref[action/2] = joint;
```

✱ Reward: if the robot clamp touches the ground.
There is a function called `GetBoundingBox ()` that returns the minimum and maximum values of a table that defines that particular object / model corresponding to the x, y, and z axes.

Using this function, it is verified if the clamp is touching the ground or not, and a reward is assigned.

```
// Get the bounding box for the gripper
const math::Box& gripBBox = gripper->GetBoundingBox();
const float groundContact = 0.05f;

bool checkGroundContact = (gripBBox.min.z <=
                        groundContact) ? true : false;

        if(checkGroundContact)
                { rewardHistory = REWARD_LOSS * 20;
                  newReward     = true;
                  endEpisode    = true; }
```

✱ Provisional reward based on the distance to the object.
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.

```
if(!checkGroundContact)
{
const float distGoal = BoxDistance
                                (gripBBox, propBBox);

if( episodeFrames > 1 )
        {
        const float distDelta =
                lastGoalDistance - distGoal;
        const float alpha = 0.2f;
                // 0.1f; [80%] : 0.2f; [90%]

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

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

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

        if (avgGoalDelta > 0)
                {
                if(distGoal > 0.0f){
        rewardHistory = REWARD_WIN * avgGoalDelta;}
                else if (distGoal == 0.0f){
        rewardHistory = REWARD_WIN * 10.0f;}
                }
        else {
        rewardHistory = REWARD_LOSS * distGoal;
                }
        newReward = true;
        }
lastGoalDistance = distGoal;
}
```

✱ Reward based on the collision between the arm and the object.
The callback function `onCollisionMsg`, can verify certain collisions. Specifically, it will define a verification condition to compare whether the particular links of the arm with its defined collision elements are colliding with the `COLLISION_ITEM` or not.

`//Objetivo [90%]`

```
bool collisionCheck =
((strcmp(contacts->contact(i).collision1().c_str
(), COLLISION_ITEM) == 0) ||
(strcmp(contacts->contact(i).collision2().c_str(
), COLLISION_ARM) == 0)) ? true : false;
```

`//Objetivo [80%]`

```
bool collisionCheck =
(strcmp(contacts->contact(i).collision2().c_str(
), COLLISION_POINT) == 0) ? true : false;
```

Depending on the objective one is looking for, one of the two collision checks is used. After determining if the arm collided with the object, a reward is assigned.

```
if (collisionCheck){

        rewardHistory = REWARD_WIN * 10;
        // REWARD_WIN*10 [90%] : REWARD_WIN*100 [80%];

        newReward  = true;
        endEpisode = true;
        return;
}
```

III. HYPERPARAMETERS.

A list of hyperparameters is provided in the `ArmPlugin.cpp` file, at the top. The following hyperparameters are configured to obtain the precision required for the objective [80%] and [90%].

`INPUT_WIDTH 64 y INPUT_HEIGHT 64:`

The size of the image captured by the camera is reduced.

`OPTIMIZER "RMSprop":`

This type of optimizer is a good option for recurrent neural networks.

`LEARNING_RATE 0.0f - [80%] 0.15f , [90%] 0.2f:`

This parameter determines how quickly the arm learns to touch the object.

`REPLAY_MEMORY 10000:`

10000 Repetitions of experience allow the agent to learn from previous memories, which accelerate learning and break undesirable temporal correlations.

BATCH_SIZE - [80%] 32 , [90%] 16:

Defines the number of samples that propagate through the network.

USE_LSTM true:

Define the decision to use LSTM (Long short-term memory).

LSTM_SIZE 256:

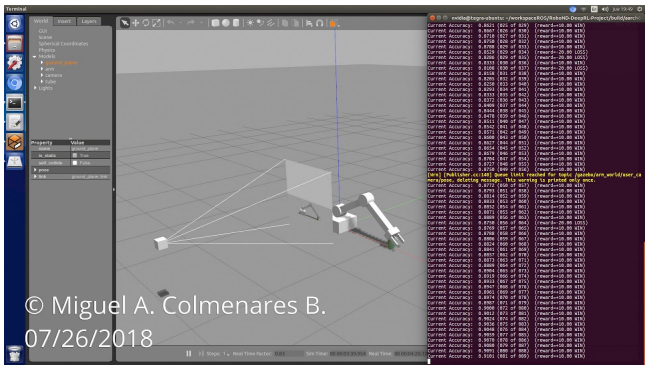
Defines the size of the memory unit.

REWARD_WIN 1.0f - REWARD_LOSS -1.0f:

By convention we worked with these reward values.

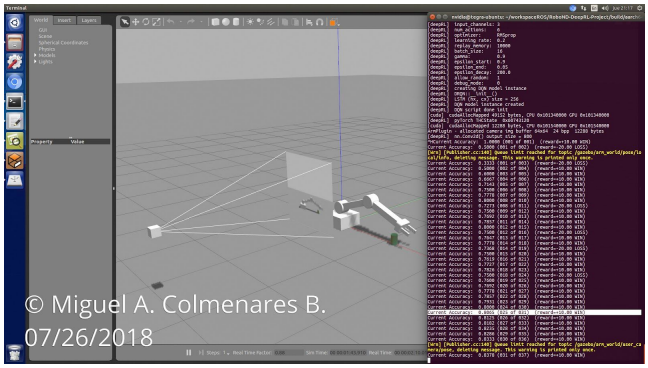
IV RESULTS.

Objective 1
In this objective, the robotic arm must touch an object (green cylinder) with a precision of 90% in a minimum of 100 runs. The following are the successful results:



In race number 88, after successfully touching the object 80 times, an accuracy of 90.91% was obtained.

Objective 2
In this objective, the robotic arm clamp must touch an object (green cylinder) with an accuracy of 80% in a minimum of 100 runs. The following are the successful results:



In race number 31, after successfully playing the object on 25 occasions, an accuracy of 80.61% was obtained.

V FUTURE WORK.

This project was very interesting and it is an area of neural networks that attracts my attention. In essence the objectives were achieved with the required precision. For the goal of 90% accuracy I would like to continue testing the code with other configurations to get better results. Mainly to get the arm to learn faster. With the objective of 80% accuracy I am satisfied with the results. Achieve an accuracy of 80% in 30 attempts, for me they are very good results.

In addition to this project I want to develop the other examples of the Nvidia repository and continue researching on reinforced learning.