UdaCity Term 2 Deep Reinforcement Learning Arm Manipulation Project

Abstract

This project needs to use Deep Reinforcement Learning technology to train an Robotic Arm Manipulation. The goal 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 a 90% accuracy.
2. Have only the gripper base of the robot arm touch the object, with at least a 80% accuracy.

For both objectives, the results were presented in this article.

Introduction

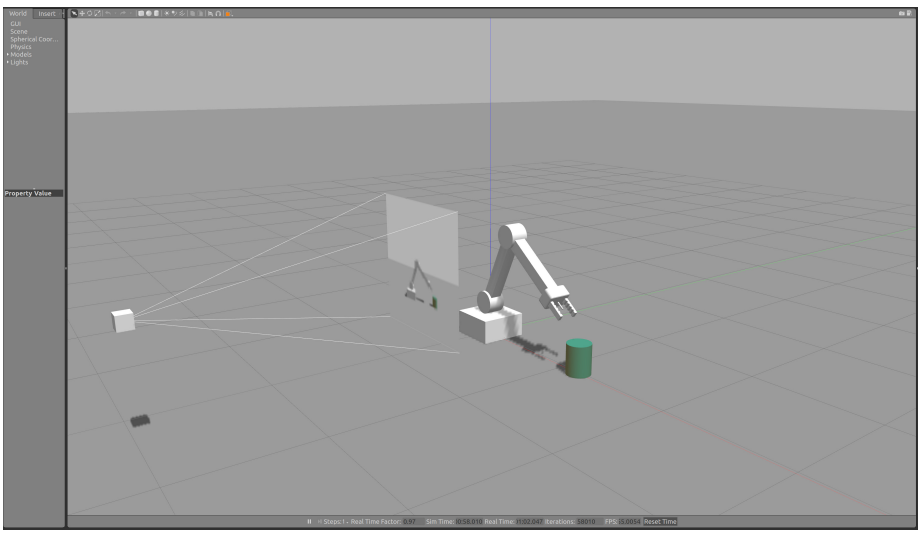
This project used a Gazebo environment to build a 3D Robotic Arm Manipulator model and used C++ language to develop the Plugin code.

There are three main components to this gazebo file, which define the environment:

I. The robotic arm with a gripper attached to it.

II. A camera sensor, to capture images to feed into the DQN.

III. A cylindrical object or **prop**.



Arm Plugin

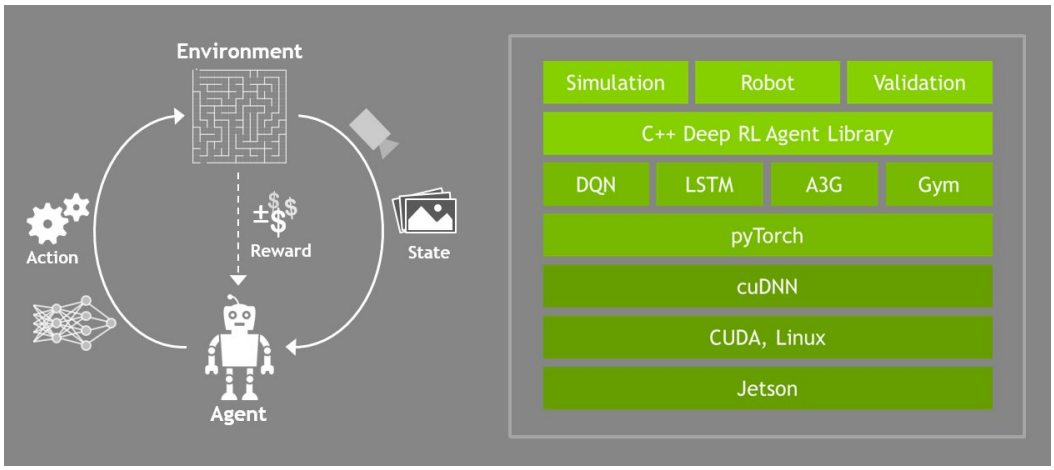
The robotic arm model, found in the gazebo-arm.world file, calls upon a gazebo plugin called the ArmPlugin. This plugin is responsible for creating the DQN agent and training it to learn to touch the prop. The gazebo plugin shared object file, libgazeboArmPlugin.so, attached to the robot model in gazebo-arm.world, 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.

Background / Tasks

The project template from UdaCity is based on the Nvidia open source project "jetson-reinforcement" developed by Dustin Franklin[1].

The deep reinforcement learning works for problems that take sensor input and produce actions. However, to successfully leverage deep learning technology in robots, it needs to move to a library format that can integrate with robots and simulators. In addition, robots require real-time responses to changes in their environments, so computation performance matters. So an API (application programming interface) in C/C++ is introduced in this project.



API stack for Deep RL (from Nvidia)

Tasks:

Complete each of those tasks in the following order ArmPlugin.cpp file

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 needs to be created.

   For the camera node -

   - **Node** - cameraNode
   - **Topic Name** - /gazebo/arm_world/camera/link/camera/image
   - **Callback Function** - ArmPlugin::onCameraMsg (this should be a reference parameter)
   - **Class Instance** - refer to the same class, using the "this" pointer or keyword.

   For the contact/collision node -

   - **Node** - collisionNode
   - **Topic Name** - /gazebo/arm_world/tube/tube_link/my_contact
   - **Callback Function** - ArmPlugin::onCollisionMsg (this should be a reference parameter)
   - **Class Instance** - refer to the same class, using the "this" pointer or keyword.

2. Create the DQN Agent.

   Refer to the API instructions to create the agent using the Create() function from the dqnAgent Class, in `ArmPlugin::createAgent()`. The variable names defined at the top of the file to this function call need to pass in. Refer to the constructor in project_folder/c/dqnAgent.cpp for the same.

3. Velocity or position based control of arm joints.

   The DQN output is mapped to a particular action, which, for this project, is the control of each joint for the robotic arm. In `ArmPlugin::updateAgent()`, there are two existing approaches to control the joint movements.

   Velocity Control - The current value for a joint's velocity is stored in the array `vel` with the array lengths based on the number of degrees of freedom for the arm. If selecting this control strategy, the variable `velocity` based on the current joint velocity and the associated delta value, `actionVelDelta`.

   ```
   float velocity = ref[action / 2] + actionVelDelta * ((action % 2 == 0) ? 1.0f :
   -1.0f);
   ```

   Position Control - The current value for a joint's position is stored in the array `ref` with the array lengths based on the number of degrees of freedom for the arm. If selecting this control strategy, the variable `joint` based on the current joint velocity and the associated delta value, `actionJointDelta`.

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

4. Reward for robot gripper hitting the ground.

   In Gazebo's API[2], 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 function to check if the gripper is hitting the ground (REWARD_LOSS) or not(REWARD_WIN), and assign an appropriate reward. The bounding box for the gripper, and a threshold value have already been defined in the `ArmPlugin::OnUpdate()` method.

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, calculate the distance between the arm and the object. Then, use this distance to calculate an appropriate reward as well.

   TIP: One recommended reward is a smoothed moving average of the delta of the distance to the goal. It can be calculated as -

   ```
   average_delta  = (average_delta * alpha) + (dist * (1 - alpha));
   rewardHistory = tanh(avgGoalDelta)*REWARD_WIN*REWARD_MULTIPLIER;
   ```

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

   In the callback function `onCollisionMsg`, it checks for certain collisions. Specifically, and needs to define a check condition to compare if particular links of the arm with their defined collision elements are colliding with the `COLLISION_ITEM` or not. The function already contains some code with this task.

7. Tuning the hyper parameters

   The list of hyper parameters that need to tune is provided in `ArmPlugin.cpp` file, at the top.

They are also listed below:

```
#define INPUT_WIDTH   512
#define INPUT_HEIGHT  512
#define OPTIMIZER "None"
#define LEARNING_RATE 0.0f
#define REPLAY_MEMORY 10000
#define BATCH_SIZE 8
#define USE_LSTM false
#define LSTM_SIZE 32
```

Tune the above hyper parameters to obtain the required accuracy.

8. Issue a reward based on collision between the arm's gripper base and the object.

   For this task of the project, modify the collision check defined in task 6, to check for collision between the gripper base and the object. And then, assign the appropriate reward.

   ```
   bool collisionGripperCheck = (strcmp(contacts->contact(i).collision2().c_str(),
   COLLISION_POINT) == 0);
   ```

The hyper parameter setting and turning:

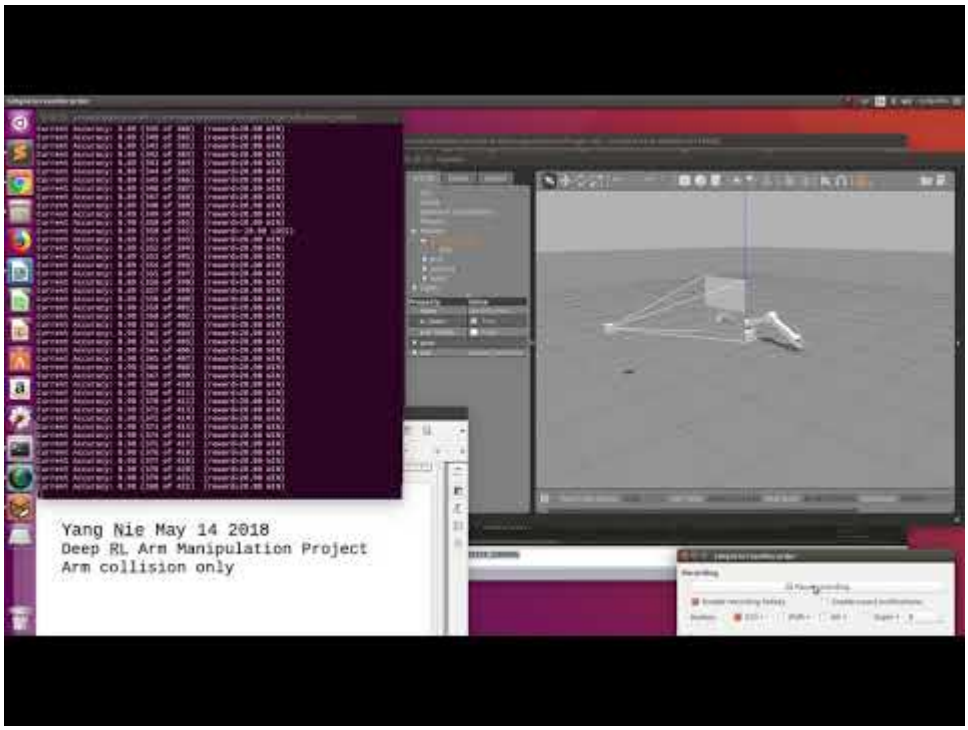
```
#define GAMMA 0.9f
#define EPS_START 0.9f
#define EPS_END 0.01f
#define EPS_DECAY 200

// Set Input image size to 64 x 64 to improve train performance without impacting too
much accuracy
#define INPUT_WIDTH   64
#define INPUT_HEIGHT  64

// Use RMSprop as optimzer method
#define OPTIMIZER "RMSprop"

// Set REPLAY_MEMORY as large value to improve the train performance
#define REPLAY_MEMORY 20000

// Set Batch size to small value to speed train,
#define BATCH_SIZE 32
#define USE_LSTM false
#define LSTM_SIZE 256

// Set reward Win and Loss values as 20
#define REWARD_WIN  20.0f
#define REWARD_LOSS -20.0f
// Set reword multiper to 12 to enlarge reward in training
#define REWARD_MULTIPLIER 12.0f
```

Results

After implementing the code and adjusting the hyper parameters, the network achieved the required objectives.

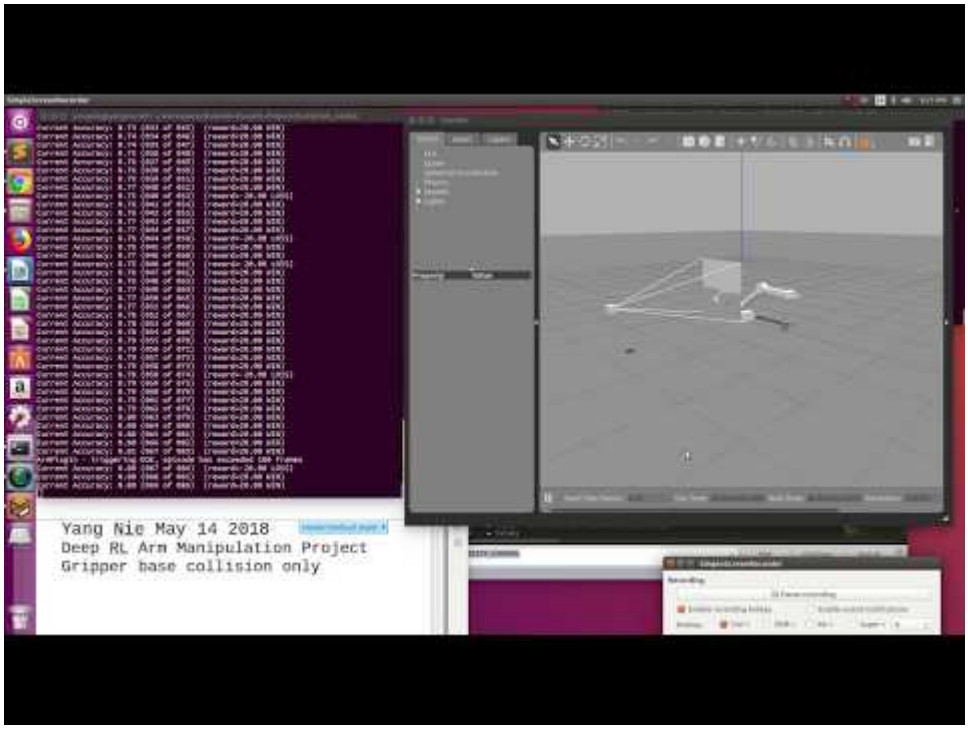
1. The end objective of the robot arm touching the object with at least a 90% accuracy for a minimum of 100 runs.

Video link for Arm testing (https://youtu.be/vDM-J5Cjrjo)



2. The end objective of the arm's gripper base touching the object with at least a 80% accuracy for a minimum of 100 runs.

Video link for Gripper base testing (https://youtu.be/8szf34IS1WE)



Item Name	Gripper Base	Any part of Arm
Touched prop	137	359
Total Run	160	401
Successful Percentage	86%	90%

Conclusion

Both objectives: the robot arm touching the object with at least a 90% accuracy for a minimum of 100 runs and the arm's gripper base touching the object with at least a 80% accuracy for a minimum of 100 runs were achieve in reasonable iterator. The DQN agent received the camera images and sensor data and trained the network well and took the correct actions on robot arm.

Future Work

1. The LSTM method was not used in both results because there was so many Loss when starting train, one of future work is to spend more time to adjust different hyper parameters to make LSTM working.
2. Trying Project Challenges in this project:

```
  A. Object Randomization
    In the project, so far, the object of interest was placed at the same location,
throughout. For this     challenge, the object will instantiate at different
locations along the x-axis.
  B. Increasing the Arm’s Reach
    In this challenge, the object’s starting location will be changed, and the arm
will be allowed to rotate about its base instead of disabling to restrict the arm’s
reach to a specific axis.
  C. Increasing Arm’s Reach with Object Randomization
    It will build on top of the previous challenge.
```

3. Deploying the model and package on Jetson TX2 board and testing them again.

References

[1] Dustin Franklin, jetson-reinforcement "https://github.com/dusty-nv/jetson-reinforcement" 2018

[2] Gazebo, API Gazebo "http://osrf-distributions.s3.amazonaws.com/gazebo/api/2.2.1/classgazebo_1_1physics_1_1Model.html#a27cf5a6ec66f6a5c03ebf05ff9592c9a" 2018