Deep RL Arm Manipulation Project Starter Code

Abhay Luna

The objective of this project is to create a Deep Q-Learning Network (DQN) agent and define reward functions to teach a robotic arm to carry out two primary objectives.

How to run the program with your own code

Go to Desktop and open a terminal.

For the execution of your own code, we head to the Project Workspace. For this setup, RoboND-DeepRL-Project is the name of the active workspace. If your workspace name is different, change the commands accordingly.

If you do not have an active workspace, you can create one. You can launch it by running the following commands first.

```
cd /home/workspace/
mkdir RoboND-DeepRL-Project
```

Clone the required repositories to the `/home/workspace/RoboND-DeepRL-Project/` folder

```
cd /home/workspace/RoboND-DeepRL-Project/
git clone https://github.com/Abhaycl/RoboND-DeepRL-2P4.git
```

Build the project:

```
cd /home/workspace/RoboND-DeepRL-Project/build
make
```

For run the project open a terminal:

```
cd /home/workspace/RoboND-DeepRL-Project/build/x86_64/bin
./gazebo-arm.sh
```

The summary of the files and folders within repo is provided in the table below:

File/Folder	Definition
c/*	Folder that contains all the C code of different functions.
cuda/*	Folder that contains all the Cuda code of different functions.
docs/*	Folder that contains all the project documentation or reference images.
gazebo/*	Folder that contains all the main files of the project.
lua/*	Folder that contains all the Lua code of different functions for DQN.
python/*	Folder that contains all the Python code of different functions for DQN.
samples/*	Folder that contains all the examples of previous laboratories catch and fruit.
tools/*	Folder that contains all the DeepRL tools.
utils/*	Folder that contains all the C/C++ wrapper Linux utilities for NVIDIA Jetson TX1/TX2 - camera, HID, GStreamer, CUDA, OpenGL/XGL.
misc_images/*	Folder containing the images of the project.
CMakeLists.txt	Contains the System dependencies that are found with CMake's conventions.
CMakePreBuild.sh	Contains the script that automatically run from CMakeLists.txt.
LICENSE.md	Contains the Nvidia software license.
README.md	Contains the project documentation.
README.pdf	Contains the project documentation in PDF format.
README_udacity.md	Is the udacity documentation that contains how to configure and install the environment.

Steps to complete the project:

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.

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Abstract

C++ API (application programming interface)

C++ API provides an interface to the Python code written with PyTorch, but the wrappers use Python's low-level C to pass memory objects between the user's 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.

Gazebo Arm Plugin

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. This plugin creates specific constructor and member functions for the class `ArmPlugin` defined in `ArmPlugin.h`.

You can refer to the documentation for more details on the above：

- Subscribers in Gazebo
- Gazebo API

Introduction

This project aims to create a Deep Q-Learning Network (DQN) and, with it, train a robotic arm to meet certain objectives. The Robotic Arm used for this project is simulated on Gazebo and run and trains (learns) on the Nvidia Jetson TX2 Deep Reinforcement Learning platform. This project is based on the Nvidia open source project "jetson-reinforcement" developed by Dustin Franklin.

This project mainly leverages an existing DQN that gets instantiated with specific parameters to run the Robotic Arm. For Deep Reinforcement Learning to occur, reward functions and hyperparameters must be defined. To test the capabilities of DQN, two objectives were established:

- a. Have any part of the robot arm touch the object of interest, with at least a 90% accuracy for a minimum of 100 runs.
- b. Have only the gripper base of the robot arm touch the object, with at least an 80% accuracy for a minimum of 100 runs.

Reward Functions

To achieve both objectives, rewards functions were configured slightly different as you'll see in the following.

Objective 1

To meet objective 1, which is to have the robotic arm collide with the target object at any point, the approach taken was to use "velocity control" instead of position.

```
#define VELOCITY_CONTROL true
```

Hyperparameters

To achieve the first objective, to train the robotic arm to touch the target at any point, the following hyperparameters were used:

Hyperparameter	Value	Reason/Intuition/Comments
INPUT_WIDTH	64	Reduced width since didn't believe a higher one was required.
INPUT_HEIGHT	64	Reduced height since didn't believe a higher one was required.
OPTIMIZER	RMSprop	Started with RMSprop and SGD but looking at forums found that Adam also worked well. It did.
LEARNING_RATE	0.05f	Iterated from 0.05f to 0.02f.
REPLAY_MEMORY	10000	Stayed the same as the lesson suggested.
BATCH_SIZE	32	Thought it required deeper learning than 8 batches.
USE_LSTM	true	Improves learning from past experiences.
LSTM_SIZE	256	Seemed like 32 was not working so iterated until 256.
REWARD_WIN	10	Value of reward when the objective is reached.
REWARD_LOSS	-10	Value of reward when the objective isn't reached.

The code added specifically for objective 1 is:

- Issue a reward based on collision between the desired arm part and the object.

```
if((strcmp(contacts->contact(i).collision1().c_str(), COLLISION_ITEM) == 0))
{
    rewardHistory = REWARD_WIN;
    newReward = true;
    endEpisode = true;
    return;
}
else
{
    // Give penalty for non correct collisions
    rewardHistory = REWARD_LOSS;
    newReward = true;
    endEpisode = true;
}
```

- Set appropriate Reward for robot hitting the ground.

```
bool checkGroundContact = false;
if( gripBBox.min.z < groundContact ){checkGroundContact = true;}

if( checkGroundContact )
{
    printf("GROUND CONTACT, EOE\n");
    rewardHistory = REWARD_LOSS * 20.0f;
    newReward = true;
    endEpisode = true;
}
```

- Issue an interim reward based on the distance to the object. A `distPenalty` reward is added to encourage arm approach the object quickly for the intermediary reward.

```
if( !checkGroundContact )
{
    const float distGoal = BoxDistance(gripBBox, propBBox); // Compute the reward
from distance to the goal

    if(DEBUG){printf("Distance('%s', '%s') = %f\n", gripper->GetName().c_str(),
prop->model->GetName().c_str(), distGoal);}

    if( episodeFrames > 1 )
    {
        const float distDelta = lastGoalDistance - distGoal;
        const float alpha = 0.2f;

        // Compute the smoothed moving average of the delta of the distance to the
goal
        avgGoalDelta = (avgGoalDelta * alpha) + (distDelta * (1.0f - alpha));

        float distPenalty = (1.0f - exp(distGoal));
        float distGoalx = gripBBox.max.x - propBBox.max.x;

        if(avgGoalDelta > 0)
            {rewardHistory = REWARD_WIN * 0.5f - distGoal * 0.5f + distPenalty;}
        else
            {rewardHistory = avgGoalDelta + distPenalty;}
        newReward = true;
    }
    lastGoalDistance = distGoal;
}
```

Objective 2

To meet objective 2, which is to have the robotic gripper base collide with the target object, the approach taken was to use “position control” instead of velocity.

```
#define VELOCITY_CONTROL false
```

Hyperparameters

To achieve the second objective, to train the robotic gripper base collide with the target object, the following hyperparameters were used:

Hyperparameter	Value	Reason/Intuition/Comments
INPUT_WIDTH	64	Reduced width since didn't believe a higher one was required.
INPUT_HEIGHT	64	Reduced height since didn't believe a higher one was required.
OPTIMIZER	Adam	Started with Adam and SGD but looking at forums found that RMSprop also worked well. It did.
LEARNING_RATE	0.01f	Iterated from 0.01f to 0.0f.
REPLAY_MEMORY	10000	Stayed the same as the lesson suggested.
BATCH_SIZE	128	Thought it required deeper learning than 8 batches.
USE_LSTM	true	Improves learning from past experiences.
LSTM_SIZE	16	Seemed like 256 was not working so iterated until 16.
REWARD_WIN	1	Value of reward when the objective is reached.
REWARD_LOSS	-1	Value of reward when the objective isn't reached.

DQN API Settings

Parameter	Value	Reason/Intuition/Comments
EPS_END	0.01f	Is reduced to 0.01 so that the robot arm is more stable when it finish learning.

Note: `EPS_DECAY` can be reduced if your reward function is appropriate. The premises is your robot should find the right way within this epochs. This can save lots of time when you try to save your final result.

The code added specifically for objective 2 is:

- Check if there is collision between the arm and object, then issue learning reward.

```
    if((strcmp(contacts->contact(i).collision1().c_str(), COLLISION_ITEM) == 0))
    {
        if((strcmp(contacts->contact(i).collision2().c_str(), COLLISION_POINT) == 0))
        {
            rewardHistory = REWARD_WIN * 10.0f + (1.0f - (float(episodeFrames) /
float(maxEpisodeLength))) * REWARD_WIN * 100.0f;
            newReward = true;
            endEpisode = true;
            return;
        }
        else
        {
            // Give penalty for non correct collisions
            rewardHistory = REWARD_LOSS;
            newReward = true;
            endEpisode = true;
        }
    }
```

- Issue an interim reward based on the distance to the object.

```
    if( !checkGroundContact )
    {
        const float distGoal = BoxDistance(gripBBox, propBBox); // Compute the reward
from distance to the goal

        if(DEBUG){printf("Distance('%s', '%s') = %f\n", gripper->GetName().c_str(),
prop->model->GetName().c_str(), distGoal);}

        if( episodeFrames > 1 )
        {
            const float distDelta = lastGoalDistance - distGoal;
            const float alpha = 0.9f;

            // Compute the smoothed moving average of the delta of the distance to the
goal
            avgGoalDelta = (avgGoalDelta * alpha) + (distDelta * (1.0f - alpha));
            float distPenalty = (1.0f - exp(distGoal));

            if(avgGoalDelta > 0.01)
                {rewardHistory = (REWARD_WIN + distPenalty * 0.1f) * 0.1f;}
           else
               {rewardHistory = REWARD_LOSS - distGoal * 2.0f;}
           newReward = true;
        }
        lastGoalDistance = distGoal;
    }
```

Results

As seen in (Fig. 1 y Fig. 1a & Fig 2), both objectives were achieved. Meeting percentage accuracy for both scenarios right after 100 runs were executed.

Objective 1

Objective 1 was relatively simple to train, the robotic arm intuitively started learning better as hyperparameters were tuned and the reward values were increased. At almost each iteration of testing, at the beginning before learning the objective, the arm started colliding with the ground, seeming to break multiple times. Increasing the negative rewards for ground collisions didn't dramatically improve this behavior. This could definitely be a problem for real-life solution. However, this behavior did not happen as much on the last iteration, and learned fairly quickly, which is why it became the final version for this objective.

It was noted that it kept trying to explore other ways to tackle the problem, obtaining multiple solution approaches along the way. Ultimately, the latest version of the code for objective 1 managed to get to 90% accuracy almost right after 100 runs.

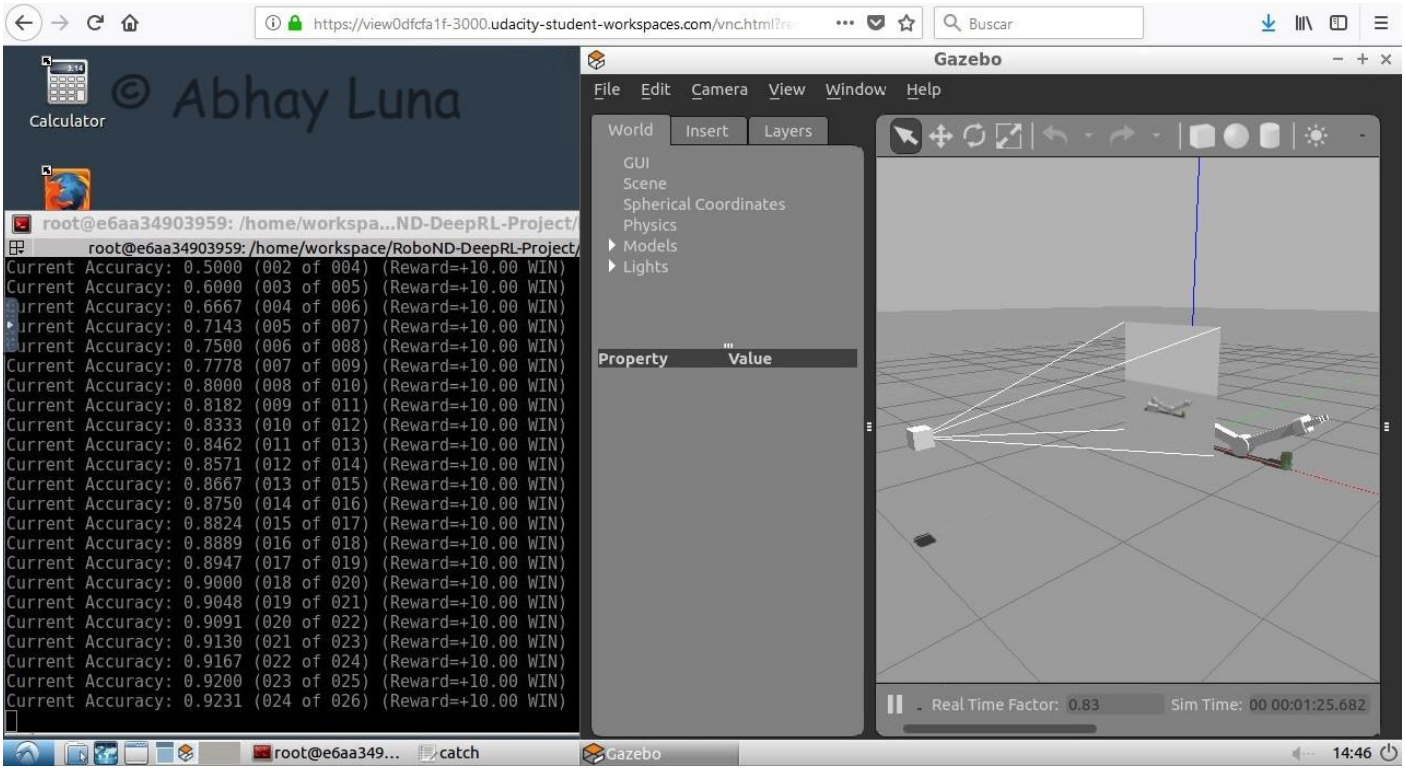


Figure 1.- Arm collision with target before at least 100 runs.

To demonstrate that it's a consistent learned behaviour and not just a fluke, the image below shows the result after at least 100 runs.

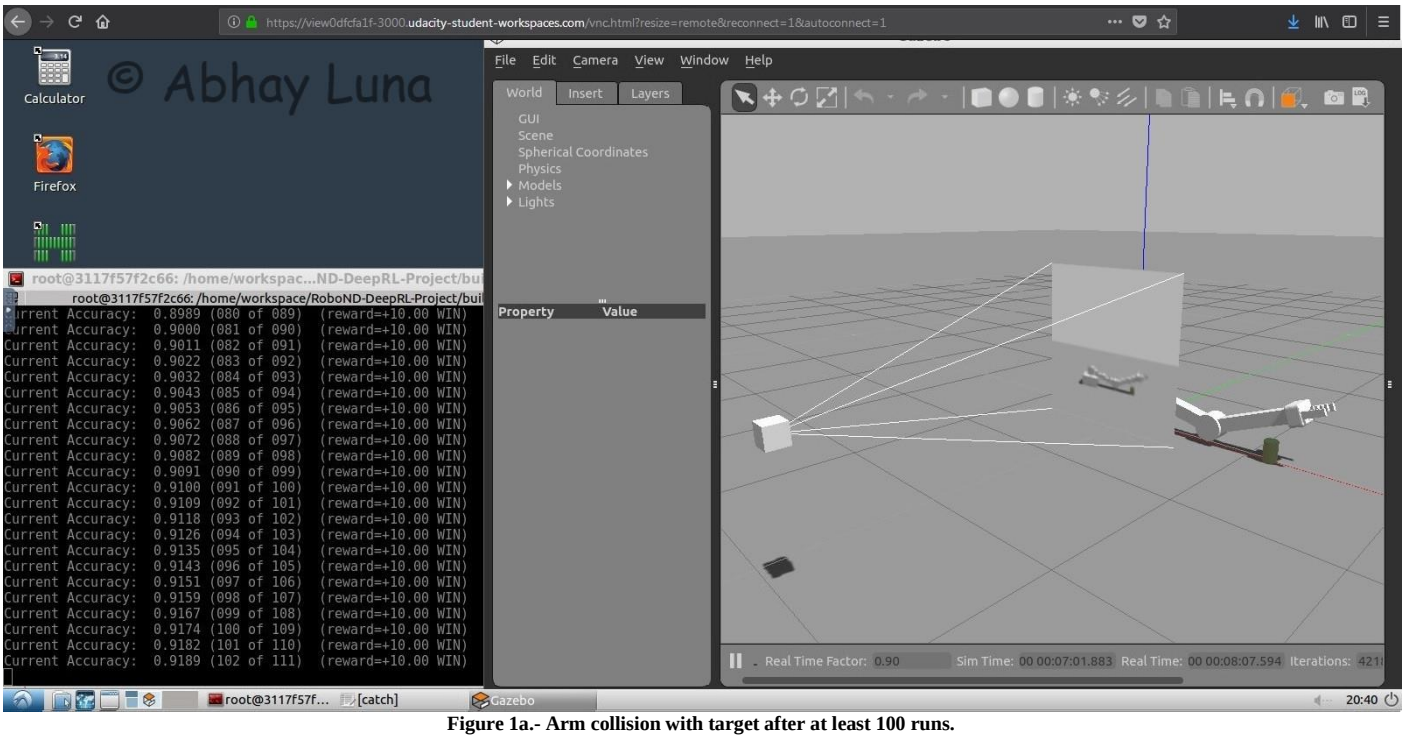


Figure 1a.- Arm collision with target after at least 100 runs.

Task	Total Runs	Accuracy
Arm	18	90.00%
Arm	100	91.74%

Objective 2

Normally, for objective 2, required more accuracy in its behavior. This was dramatically harder to train than objective 1. After iterating and testing many times, by tuning hyperparameters, it was noticed that the agent was trying to explore too much. However, there are so many approaches that can be taken to touch the target with just one point. For that reason, the DQN API Settings were changed to decrease exploration slightly compared to objective 1. This helped on teaching the robotic arm agent to perform a solution approach quicker. Also, the fact that the learning rate was decreased and the batch size increased, seemed to have helped the agent to train and learn deeper than objective 1.

However, as with objective 1, the agent was also displayed hitting the ground hard enough to sometimes break, at least in the initial stages of the iteration. This a problem that could be catastrophic for a real robot (which normally costs a lot of money to make). This should be explored further.

Nonetheless, for the purpose of this project, the robotic arm exceeded the minimum accuracy of 80% reaching up to 86% accuracy after almost 200 runs.

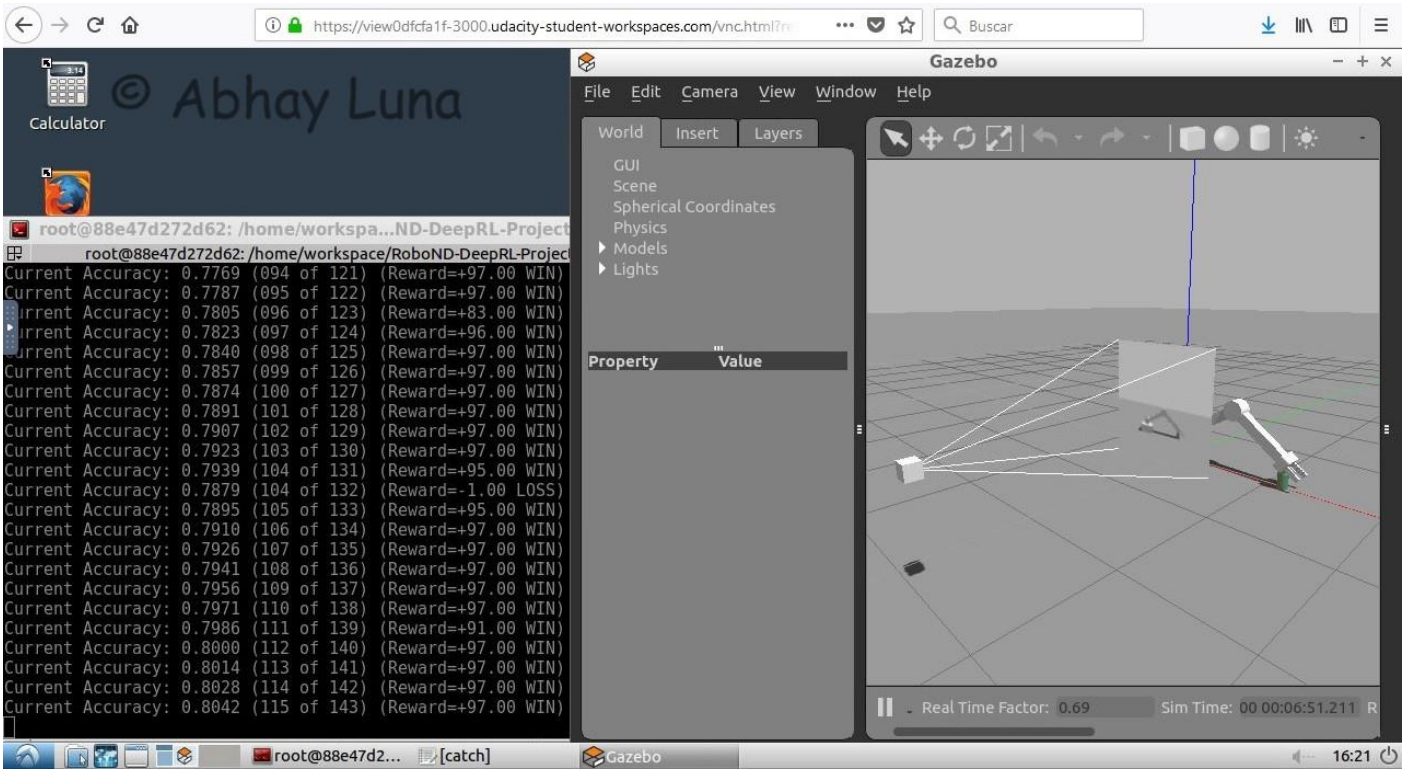


Figure 2.- Gripper collision with target.

Task	Total Runs	Accuracy
Gripper base	112	80.00%

Challenges

Challenge1: Object Randomization

Two set this challenge1 environment, the following steps:

1. In `PropPlugin.cpp`, redefine the prop poses in `PropPlugin::Randomize()` to the following:

```
pose.pos.x = randf(0.35f, 0.45f);
pose.pos.y = randf(-1.5f, 0.2f);
pose.pos.z = 0.0f;
```

2. In `ArmPlugin.cpp`, replace `ResetPropDynamics();` set in the method `ArmPlugin::updateJoints()` with `RandomizeProps();`

The object will instantiate at different locations along the x-axis.

- Use a larger `INPUT_WIDTH` and `INPUT_HEIGHT` value to distinguish object more clear. For all the cases it is used by default the output of the Gazebo camera that is only 64x64.

```
#define INPUT_WIDTH  64
#define INPUT_HEIGHT 64
```

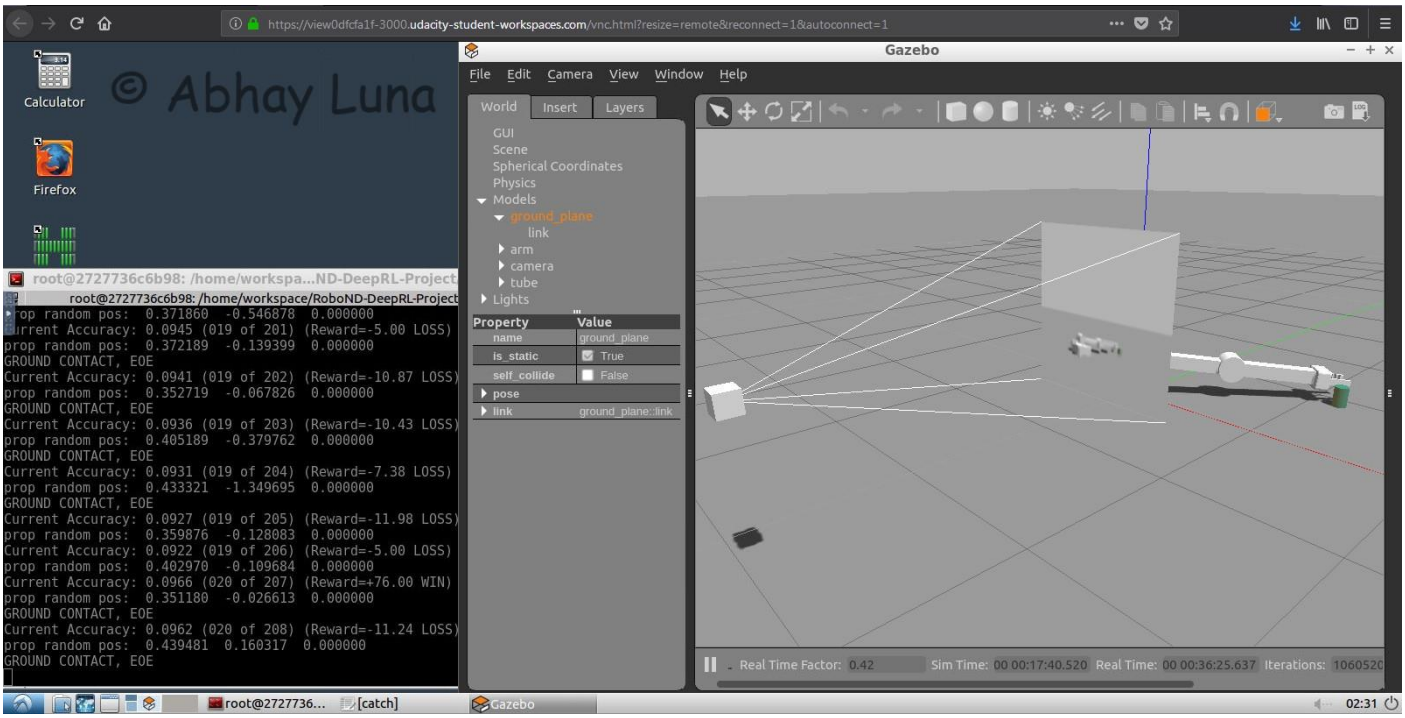
- Revise the intermediary reward, give more penalty for arm stop

```
    if( !checkGroundContact )
    {
        const float distGoal = BoxDistance(gripBBox, propBBox); // Compute the reward
from distance to the goal

        if(DEBUG){printf("Distance('%s', '%s') = %f\n", gripper->GetName().c_str(),
prop->model->GetName().c_str(), distGoal);}

        if( episodeFrames > 1 )
        {
            const float distDelta = lastGoalDistance - distGoal;
            const float alpha = 0.9f;

            // Compute the smoothed moving average of the delta of the distance to the
goal
            avgGoalDelta = (avgGoalDelta * alpha) + (distDelta * (1.0f - alpha));
            float distPenalty = (1.0f - exp(distGoal));

            if(avgGoalDelta > 0.01)
                {rewardHistory = (REWARD_WIN + distPenalty * 0.1f) * 0.1f;}
            else
                {rewardHistory = REWARD_LOSS - distGoal * 2.0f;}
            newReward = true;
        }
        lastGoalDistance = distGoal;
    }
```

3. In `gazebo-arm.world`, modify the tube model's pose to `[0.75 0.75 0 0 0 0]`.

4. In `ArmPlugin.cpp`, set the variable `LOCKBASE` to `false`.

A better result is foreseeable with larger input width and height value.



Challenger image.

Future Work

I would like to improve on the hyperparameter tuning for the first objective to achieve almost 100% accuracy results. Obtaining a higher accuracy and faster might be another improvement to investigate, as in real-life, having to wait long periods for robots to learn how to perform tasks might be slightly more inconvenient than in a simulation. Furthermore, cannot emphasize enough the ground collision issue that must be mitigated for this to be a useful real-life solution. It must never collide with the ground, or at least not with that amount of force. Perhaps a constraint can be developed and implemented to shut down the arm right before colliding with the ground.