Introduction

The work presented here trains a robotic arm manipulator using Deep Q-learning Neural Network as the agent. The gazebo simulation environment define the environment as follows:

- 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.

The RL task is episodic and the goal is to train the robotic arm manipulator to:

- At first have any part of the robot arm touch the object of interest
- Then have only the gripper base of the robot arm touch the object

Background

Deep Reinforcement Learning for Robotics is a paradigm shift. The basic idea is to start with raw sensor input, define a goal for the robot, and let it figure out, through trial and error, the best way to achieve the goal. In this paradigm, perception, internal state, planning, control, and even sensor and measurement uncertainty, are not explicitly defined. All those traditional steps between observations (the sensory input) and actionable output are learned by a neural network. Some basic characteristics of RL are given below:

- The reinforcement learning (RL) framework is characterized by an agent learning to interact with its environment.
- At each time step, the agent receives the environment's state (the environment presents a situation to the agent), and the agent must choose an appropriate action in response. One time-step later, the agent receives a reward (the environment indicates whether the agent has responded appropriately to the state) and a new state.
- All agents have the goal to maximize expected cumulative reward, or the expected sum of rewards attained over all time steps.

Episodic vs. Continuing Tasks

- A task is an instance of the reinforcement learning (RL) problem.
- Continuing tasks are tasks that continue forever, without end.
- Episodic tasks are tasks with a well-defined starting and ending point.
    - In this case, we refer to a complete sequence of interaction, from start to finish, as an episode.
    - Episodic tasks come to an end whenever the agent reaches a terminal state.

The Reward Hypothesis

The hypothesis says that, all goals can be framed as the maximization of (expected) cumulative reward.

Rewards

Looking at the goals to the problem it can be said that after the manipulator ‘learns’ to reach out the object, the task is complete. Therefore, by definition, the task is an episodic task.
The following reward values are updated from the predefined values given in the project. If the robot acts in favour of the goal, it is rewarded with a high value(reward_win). However, if it acts against the favour of the goal, it is penalised with a high value as well(reward_loss).

```
#define REWARD_WIN  300.0f //0.0f
#define REWARD_LOSS -300.0f //-0.0f
```

Following are the list of the rewards used for this project:

1. Reward based on collision between the arm and the object
2. Reward based on collision between the arm’s gripper base and the object.
3. Reward for robot gripper hitting the ground
4. Interim reward based on the distance to the object

Reward based on collision between the arm and the object &
Reward based on collision between the arm's gripper base and the object.

To sense the collision between the robot arm and the object a contact sensor is used. The callback function onCollisionMsg() checks whether the collision happened between the arm tube and the object or between gripper and the object. If any of the robot part touches the object, it is rewarded, otherwise penalised.

Code snippet:

```
if((strcmp(contacts->contact(i).collision1().c_str(),COLLISION_ITEM)==0)
            || (strcmp(contacts->contact(i).collision2().c_str(),COLLISION_POINT)==0))
            {
                  printf("\n collision reward_win \n");
                  rewardHistory = REWARD_WIN;
                  newReward  = true;
                  endEpisode = true;
                  return;
            }
            else{ printf("\ncollision reward_loss");
                  rewardHistory=REWARD_LOSS;
                  newReward = true;
                  endEpisode =false;      }
```

Reward for robot gripper hitting the ground

Whether the robot touched the ground or not is determined using GetBoundingBox() function which returns the minimum and maximum values of a box that defines that particular object/model corresponding to the x, y, and z axes. The z value returned for the gripper is compared against a threshold 0.05(groundContact) to determine if the gripper touched the ground. If it did touch the ground, it is penalised and the episode is called off to give it a fresh start to try from beginning.

Code snippet:

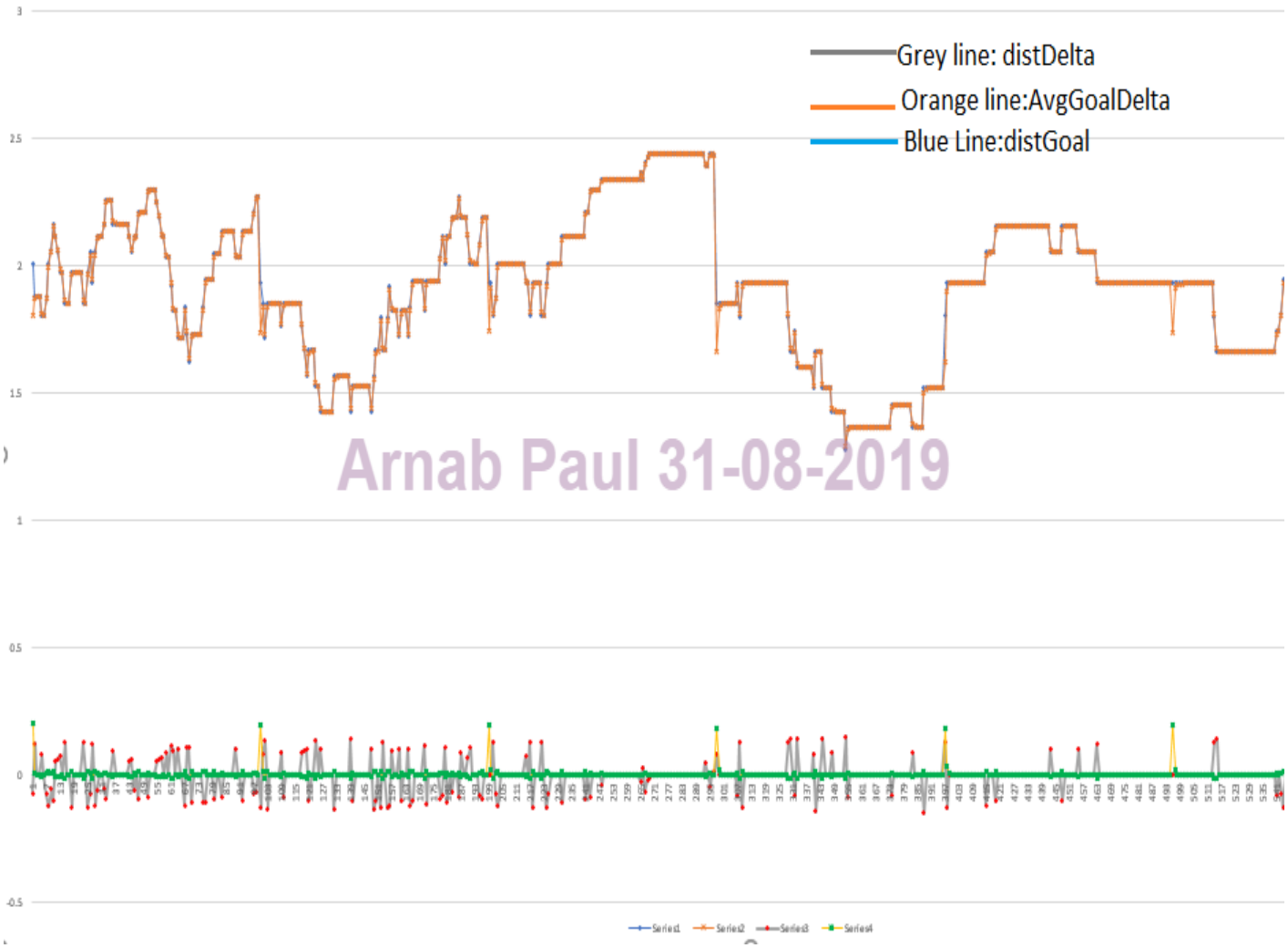
```
if((gripBBox.min.z <= groundContact)||(gripBBox.max.z <= groundContact))
            {     if(DEBUG){printf("GROUND CONTACT, EOE\n");}
                  rewardHistory = REWARD_LOSS;
                  newReward     = true;
                  endEpisode    = true;
            }
```

Improvements

It is understood that the robot will act more towards the goal if it is encouraged for getting closer to the object. Hence an interim reward is assigned if the robot got closer to the object but did not touch the ground. The reward is chosen to be different than the other rewards and essentially a higher value than the reward_win:
#define REWARD_INTERIM 400.0f
The function called "BoxDistance()" calculates the distance between two bounding boxes, that represent the pose of the gripper and the target object. Then a reward is issued based on the moving average of the delta of the distance to the object using a discount parameter-alpha (0.1). Three different parameters are calculated for this purpose:

```
distGoal = BoxDistance(propBBox, gripBBox);
avgGoalDelta = (avgGoalDelta * alpha) + (distGoal * (1 - alpha));
distDelta = lastGoalDistance - distGoal;
```

By plotting these three parameters over time, it is observed that the value of avgGoalDelta gets closer to distDelta as the robot approaches to the object. This allowed to decide the condition for interim reward which is shown in the code snippet below.



Interim reward based on the distance to the object

Code snippet:

```
{
      const float distGoal = BoxDistance(propBBox, gripBBox);
      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;
             avgGoalDelta  = (avgGoalDelta * alpha) + (distGoal * (1 - alpha));
             if(avgGoalDelta <= distDelta){
               printf ("Near to object reward\n");
               rewardHistory = REWARD_INTERIM;
               newReward     = true;
               endEpisode    = false; } }
lastGoalDistance = distGoal;}
```

Hyperparameters

For the purpose of this project the following hyperparameters are chosen. The effect of the hyperparameters are described below.

```
#define INPUT_WIDTH   256
#define INPUT_HEIGHT  256
#define OPTIMIZER "RMSprop"
#define LEARNING_RATE 0.01f
#define REPLAY_MEMORY 1000
#define BATCH_SIZE 16
#define USE_LSTM true
#define LSTM_SIZE 256
```

- **Input Height** and **Width** determines the size of the camera frame image file to process each time during training the neural network. Usually a square image is chosen in order to facilitate padding. Starting size was 512x512 and it was found that higher value helped improving accuracy of the RL. However, higher input pixel size required more memory and computationally higher cost. Hence optimal size was set to 256x256
- Two **optimizers** was available for DQN training: RMSProp and ADAM. The central idea of RMSprop is to keep the moving average of the squared gradients for each weight. And then the gradient is divided by square root the mean square. *Adam* vary the learning rate for each individual weight depending on the training process. The RMSprop optimizer were found more efficient than adam for this project.
- A **learning rate** was changed from 0.00 to 0.01 which helped converge to global minimum faster.
- **Replay memory** stores the transitions that the agent observes, allowing us to reuse this data later. By sampling from it randomly, the transitions that build up a batch are decorrelated. It has been shown that this greatly stabilizes and improves the DQN training procedure. For the default replay memory (10000) the process quickly overflowed. Finally, a replay memory of 1000 and **batch size** of 16 helped improved the training.
- For the project, every camera frame, at every simulation iteration, is fed into the DQN and the agent then makes a prediction and carries out an appropriate action. Using **LSTMs** as part of that network, one can train network by taking into consideration multiple past frames from the camera sensor instead of a single frame. An LSTM size of 256 improved accuracy of the training.

Advantages/Disadvantages

The RL agent created for this task uses C/C++ API to solve RL problems. It has several advantages:

- Leverages the power of GPU acceleration to speed up RL agent training
- Provides popular RL algorithms in library form that can be integrated with robots and simulators
- Increases execution performance by using C/C++ compilation instead of interpreted python scripts

One of the drawbacks for this method is that it takes long time to train. A high replay memory quickly destabilize the training process.

Results

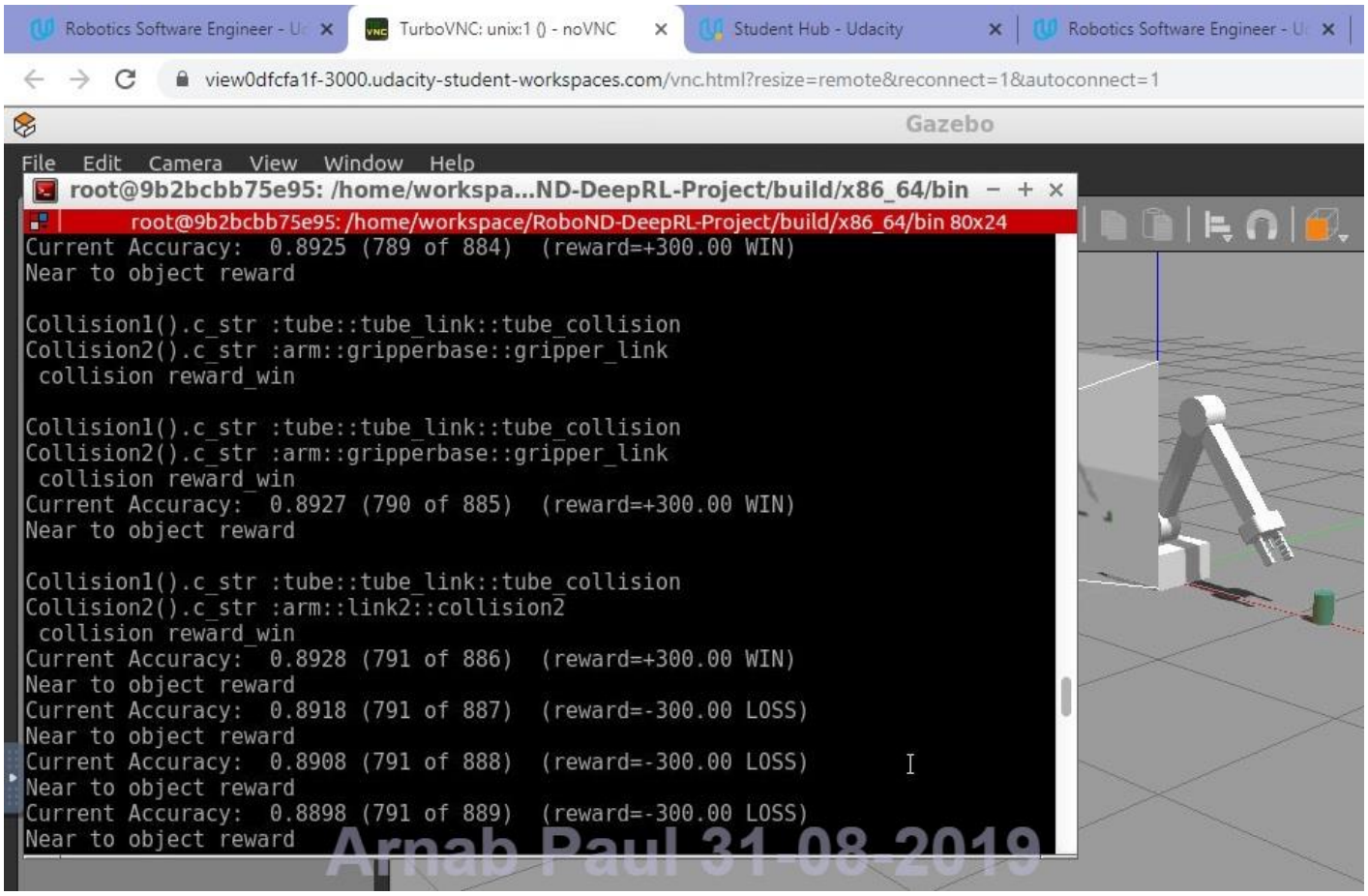


Fig 1. Robot accuracy for robot arm touching the object of interest

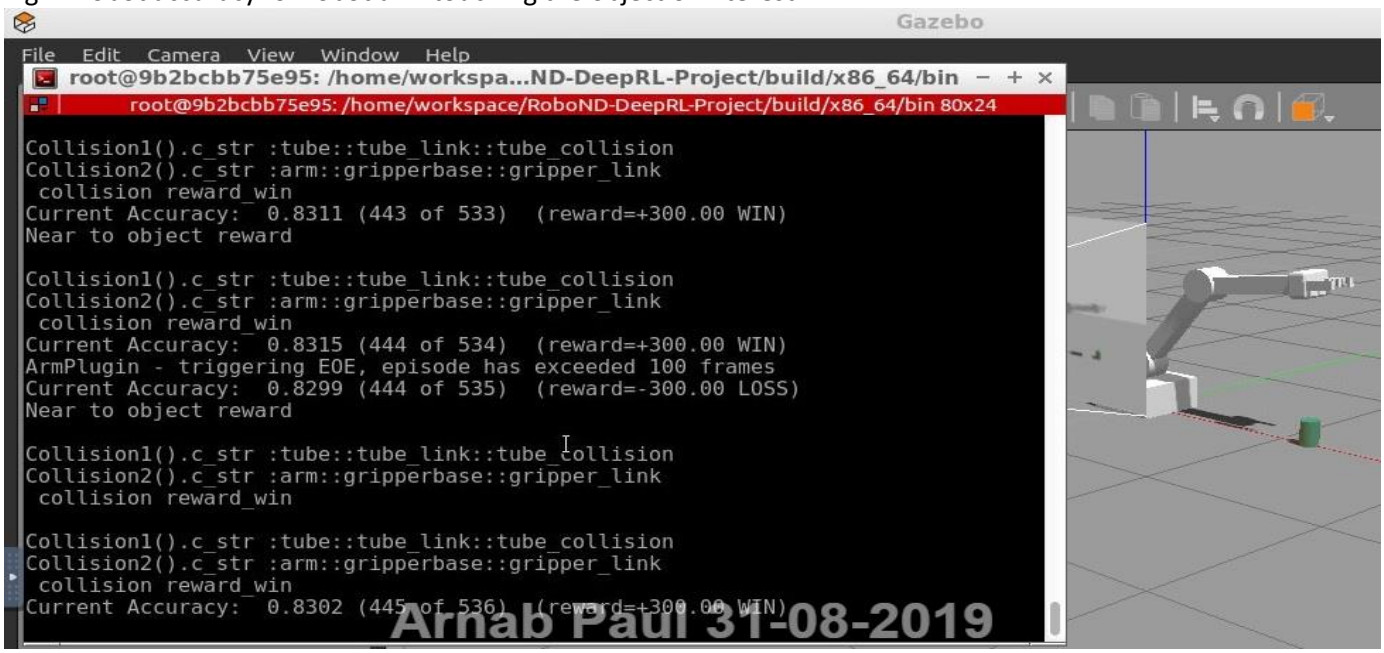


Fig 2. Robot accuracy for only gripper link touching the object of interest

Conclusion

The result shows that there are still scope of improving accuracy of the training. Tuning the hyperparameters improving the reward functions can improve the accuracy more.