Deep RL Arm Manipulator Project

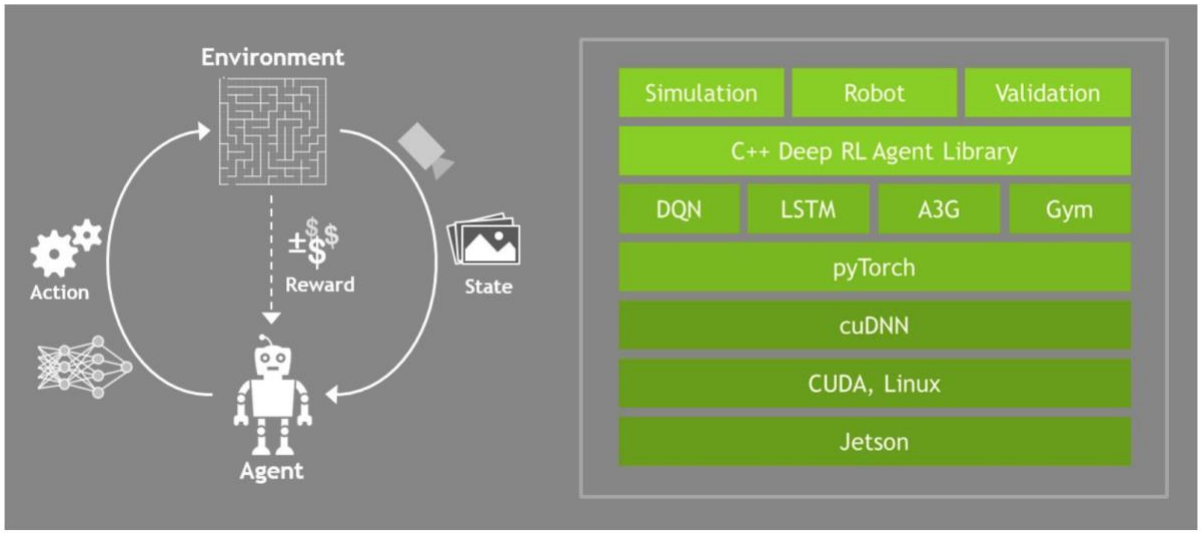
Abstract – The goal of Deep RL Arm Manipulator Project is to train Deep Q Network (DQN) agent by defining reward points that would teach a robotic arm in a simulated Gazebo environment to achieve two primary goals:

- Have any part of the robot arm touch the target object with at least 90% accuracy
- Have only the gripper base of the robot arm touch the target object with at least 80% accuracy

Introduction –
Deep Reinforcement Learning is an evolving field of research in robotics. This project uses Deep RL to train a robotic arm agent hit a target object with high accuracy. The project was simulated in Gazebo environment interfacing with C++ API with DQN agent written in PyTorch. Agent was tuned with positive and negative rewards depending upon it accomplishments. This project was ran on Nvidia TX2.

Architecture –
C++ API provides and to the Python code written in PyTorch, the wrappers use Python's low-level C to pass memory objects between the application and Torch without extra copies. Using a complied language like (C/C++) improves performance as compared to Python. GPU in TX2 helped improve the seed further.

Architecture diagram



Reference – [DRL GPU library from NVIDIA]

Project Explanation –
Gazebo-arm.world file in /gazebo/ directory has three components to it

- Three joints Robotic arm with a gripper
- Camera sensor, to capture images to feed into DQN
- Cylinder object or prop

The robotic arm model, found in gazebo-arm.world calls gazebo plugin "ArmPlugin". This plugin is responsible for creating DQN agent and training it to learn to touch the prop. The gazebo plugin shared object libgazeboArmPlugin.so in the gazebo-arm.world integrates the simulated environment with RL agent. This plug in creates constructor of ArmPlugin ArmPlugin.h.

ArmPlugin::Load() creates and initializes nodes that subscribe to camera and contact sensor of the object.

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

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

Callback function which is the second parameter is called when a new message is received. Class instanstans(this) is used when a new message is received.

Refer to the documentation for further information

- **Subscribers in Gazebo**
- **Gazebo API**

ArmPlugin::onCameraMsg()
This is a callback function for the camera subscriber. It takes the message from the camera topic, extracts the image and saves it which is passed to the DQN

ArmPlugin::onCollisionMsg()
This is a callback function for the object's contact sensor. This function is used to test whether the contact sensor called my_contact, defined for the object in gaxebo-arm.workd, observes a collision with another element/model. This function is also used to define a reward function based on collision.

ArmPlugin::createAgent()
Creates an DQN agent

```
#define REWARD_WIN  1.0f
#define REWARD_LOSS -1.0f

#define INPUT_WIDTH 64
#define INPUT_HEIGHT 64
#define OPTIMIZER "RMSprop"
#define LEARNING_RATE 0.01f
#define REPLAY_MEMORY 10000
#define BATCH_SIZE 512
#define USE_LSTM true
#define LSTM_SIZE 256

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

ArmPlugin::updateAgent()
Agent needs to take action for every frame it receives. Since DQN agent is discreate, the network selects an output for each frame. The output value is mapped to a specific action to control the arm.

updateAgent() method, receives the action value from DQN, and decides to action to be taken.

There are two possible ways to control the arm joint

- Velocity control

The current velocity of the joints were available in an array named “vel” whose size same as number of degree of freedom. “actionVelDel” tunable parameter is used update the joint velocity through the vel array

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

velocity of each joint is updated depending upon the orientation of the current velocity value and by a magnitude of actionValDelta

- Position control

The current position of the joints were available in an array named “ref” whose size same as number of degree of freedom. “actionJointDel” tunable parameter is used update the joint position through the ref array

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

position of each joint is updated depending upon the orientation of the current position and by a magnitude of actionJoinDelta.

ArmPlugin::OnUpdate()
This method issues rewards and train the DQN. It is called on every simulation iteration and used to update the robot joints, issue end of episode (EOE) and interim rewards. The parameters of the API and plugin are reset at EOE.

Hyperparameter Tuning to accomplish first objective –
Have any part of the robot arm touch the target object with at least 90% accuracy

```
#define INPUT_WIDTH   64
#define INPUT_HEIGHT  64
#define OPTIMIZER "RMSprop"
#define LEARNING_RATE 0.01f
#define REPLAY_MEMORY 10000
#define BATCH_SIZE 512
#define USE_LSTM true
#define LSTM_SIZE 256
```

Screen size was reduced to 64x64 to reduce memory foot print. RMSprop optimizer was used for this objective. Learning rate was set to 0.01, results did not converge on decreasing them. BATCH_SIZE was set to 512 and LSTM_SIZE was set to 256. There wasn't much tuning required except for changing the screen size to accomplish this objective.

Velocity control was switched on with min and max value of -0.2 and 0.2

Velocity control
```
// Turn on velocity based control
#define VELOCITY_CONTROL true
#define VELOCITY_MIN -0.2f
#define VELOCITY_MAX 0.2f
```

No Collison
A high negative reward was given for no collision, is was to discourage it from doing it.

```
        // episode timeout
        if( maxEpisodeLength > 0 && episodeFrames > maxEpisodeLength )
        {
                rewardHistory = REWARD_LOSS * 1000.0f;
                newReward     = true;
                endEpisode    = true;
        }
```

Ground Contact
Distance between gripper bounding box and propBox is calculated. Z mn and max of the gripper was verified not to be in the rand of floor coordinates.

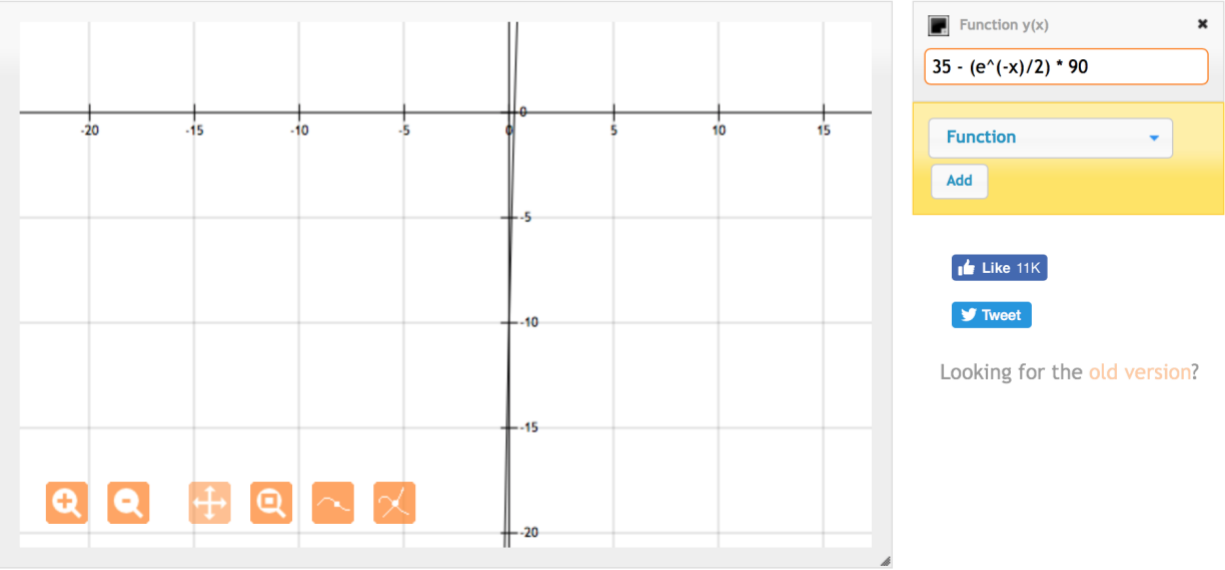
A high negative value is provided to discouraged.

```
const float groundContact = -0.5f;
const float distGoal = BoxDistance(gripBBox, propBBox); //distance to the goal

bool checkGroundContact = (gripBBox.min.z <= groundContact ||
gripBBox.max.z <= groundContact);

if(checkGroundContact)
{

    if(DEBUG){printf("GROUND CONTACT, EOE\n");}

    printf("GROUND CONTACT, EOE\n");
    rewardHistory = REWARD_LOSS * lastGoalDistance * 600.0f;
    newReward     = true;
    endEpisode    = true;
}
```

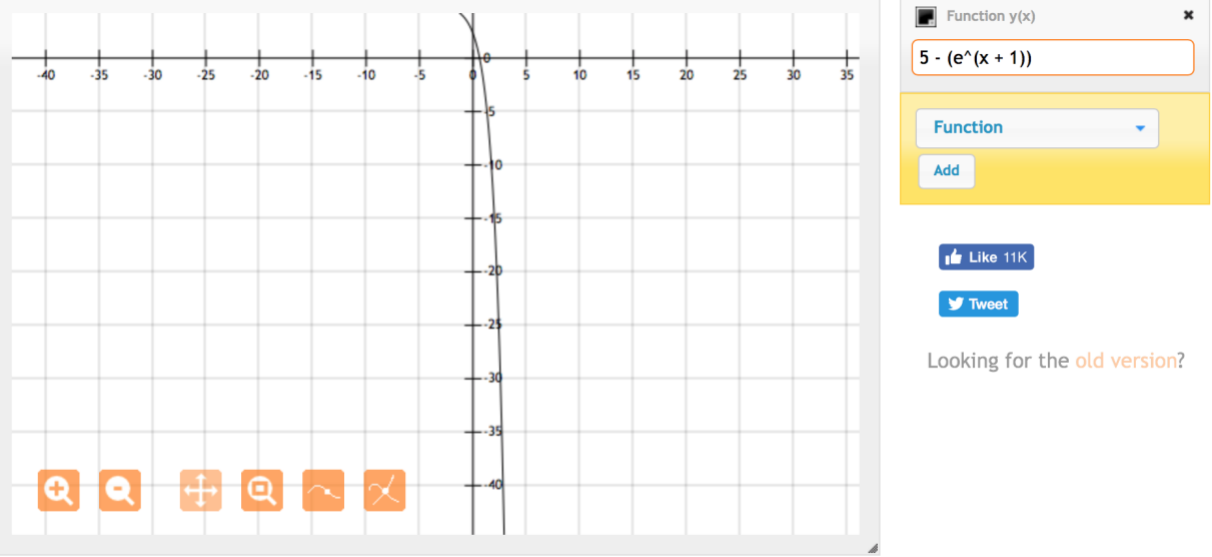
Intermediate award
Intermediate award is used to smoothen the arm movement.
RewardHistory was defined as a factor of gripper distance from target and avgGoalDelta which is positive when the arm moves clockwise and negative otherwise. The idea was to provide a high negative value the further it is from the target and also provide a high negative value when the arm moves anticlockwise. Both these were made as close to zero when the arm approaches target but not very high positive value.

Below are the functional graphs

avgGoalDeltaMultiplier ranges between -2 to +2. Below equation limit it to a small negative and positive value



Distance from the prop varies from 5 to 0. The reward is a high negative value for 5 and approaches a small positive value as the gripper gets close to the prop, that is o distance from target




```
if(!checkGroundContact)
{
	if( episodeFrames > 1 )
	{
		const float distDelta  = lastGoalDistance - distGoal;
		const float alpha = 0.4f;
		const float distDeltaMultiplier = 1.0f;

		// compute the smoothed moving average of the delta of the distance to the goal
		avgGoalDelta  = (avgGoalDelta * alpha) + (distDelta * (1.0f - alpha));
		const float avgGoalDeltaMultiplier = (exp(-1.0f * avgGoalDelta) / 2.0f) * 90.0f;

		const float distDeltaReward = 5.0f + exp(1.0f + distGoal) * distDeltaMultiplier *
REWARD_LOSS;
		const float avgGoalDeltaReward = 35.0f + avgGoalDeltaMultiplier *
REWARD_LOSS;

		//printf("rewardHistory: %f\n", rewardHistory);
		rewardHistory = (distDeltaReward + avgGoalDeltaReward);
		newReward = true;
	}
	lastGoalDistance = distGoal;
}
```

Award for arm or gripper hitting the prop

Collision between arm/gripper with the prop is checked and given a very high value for encouraging it.

```
bool armCollisionCheck = (strcmp(contacts->contact(i).collision2().c_str(), COLLISION_ARM)
== 0);
bool gripperCollisionCheck = (contacts->contact(i).collision2().find("arm::gripper") !=
std::string::npos);

if (armCollisionCheck)
{
        rewardHistory = REWARD_WIN * 1500.0f;
}

else if (gripperCollisionCheck)
{
        rewardHistory = REWARD_WIN * 5000.0f;
}

newReward  = true;
endEpisode = true;
```

Hyperparameter Tuning to accomplish second objective –

Have only the gripper base of the robot arm touch the target object with at least 80% accuracy

Second was much challenging to accomplish and lot of changes in the policy were to be made.

```
#define INPUT_WIDTH   64
#define INPUT_HEIGHT  64
#define OPTIMIZER "Adam"
#define LEARNING_RATE 0.01f
#define REPLAY_MEMORY 10000
#define BATCH_SIZE 128
#define USE_LSTM true
#define LSTM_SIZE 512
```

Screen size was reduced to 64x64 to reduce memory foot print. Adam Optimizer was used which proved to be better learning algorithm as compared to RMSprop in this case. Learning rate was set to 0.01, results did not converge on decreasing them. BATCH_SIZE was set to 512. Changing BATCH_SIZE, LSTM_SIZE, REPLAY_MEMORY did not show any better results.

There were many attempts to tune these parameters.
REPLAY_MEMORY was tired with 2000. LSTM_SIZE was attempted with 512, 1024.
BATCH_SIZE was set to 1024 but none of them showed better results.

Velocity range of the arm was reduced between -0.08 to 0.08 to have better control

```
#define VELOCITY_MIN -0.08f //-0.2f
```

#define VELOCITY_MAX 0.08f //0.2f

The range of negative and positive rewards had to be much precise. A high negative reward led to high no collision End of Episode (EOE) and a low negative reward did not discourage enough from restricting the event.

No Collision –

A moderate negative reward was given for no collision

```
// episode timeout
if( maxEpisodeLength > 0 && episodeFrames > maxEpisodeLength )
{
	printf("ArmPlugin - triggering EOE, episode has exceeded %i frames\n",
maxEpisodeLength);
	nNoCollision++;

	rewardHistory = REWARD_LOSS * 10.0f;
	newReward     = true;
	endEpisode    = true;
}
```

Ground contact –
High negative reward was given for ground contact discouraging it completely

```
if(checkGroundContact)
{
	nfloorCollision++;
	rewardHistory = REWARD_LOSS * distGoal * 600.0f;
	newReward     = true;
	endEpisode    = true;
}
```

Intermediate award –
Intermediate award was first tried with the same exponential intermediate award as of first objective however the score did not go beyond 50%. This was due to over influence of the policy and limiting the learning curve of the agent. After many trails and alterations, a simple intermediate award policy was created which let the agent learn of its own.

A reward of avggoaldelta was given to the agent to smoothen up the movement. Alpha factor was also keep to the minimum to keep the change in small steps. Alpha value tried from 0.9 through 0.1 and 0.2 gave the best results.

```
if( episodeFrames > 1 )
```

```
{
        const float distDelta  = lastGoalDistance - distGoal;

        const float alpha = 0.2f;
        avgGoalDelta  = (avgGoalDelta * alpha) + (distDelta * (1.0f - alpha));
        rewardHistory = avgGoalDelta;
        newReward = true;
}
```

Reward for Gripper/Arm collision with prop –
A high positive reward was given on hitting the prop with the gripper which was factor of current episodeframe to maximum episode frame. This encouraged the gripper to hit the prop as quickly as possible.

A moderate negative reward was provided on hitting the prop with arm which is a factor of avgGoalDelta the small delta. Increasing this value made the arm hesitant to try hitting the prop

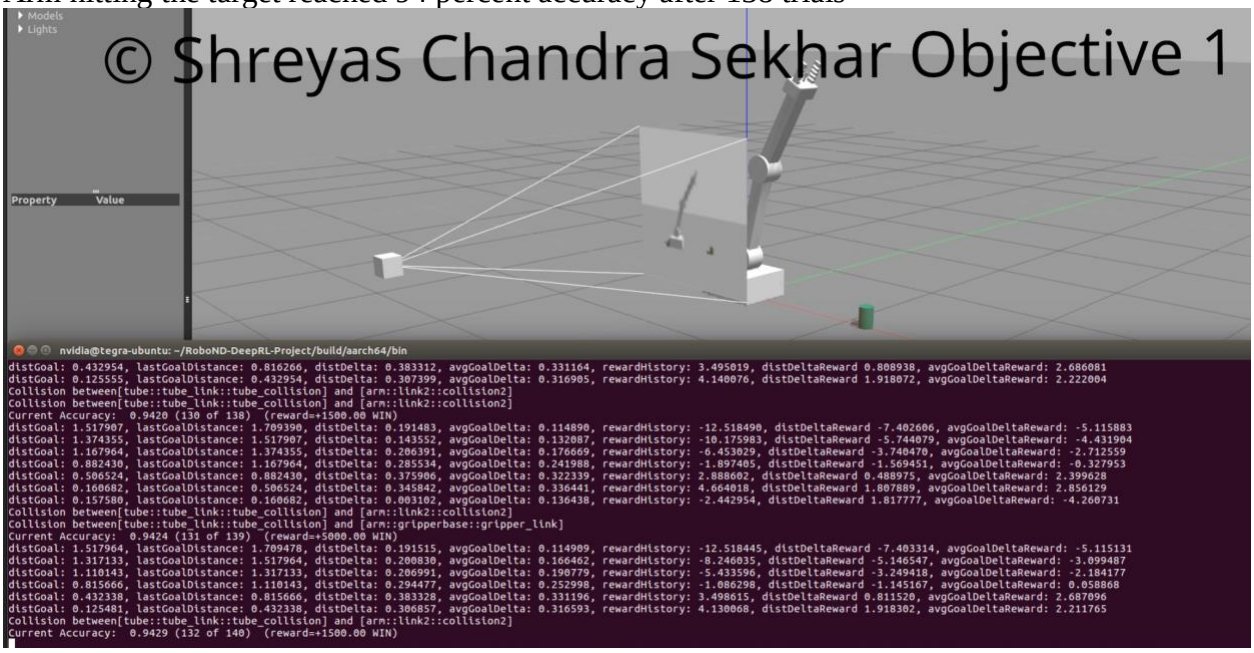
The key factor for a winning strategy is make the distance between the prop and gripper as zero. This is monitored by variable "distGoal" which gets updated on every OnUpdate call. To avoid confusions this was initialized to -1
float distGoal = -1.0f;

It was noticed that there were few occasions when no positive rewards were provided on zero distance between the gripper and prop. To address this issue, distGoal was made as a global variable and was checked for zero distance between gripper and prop. This was considered as WIN.

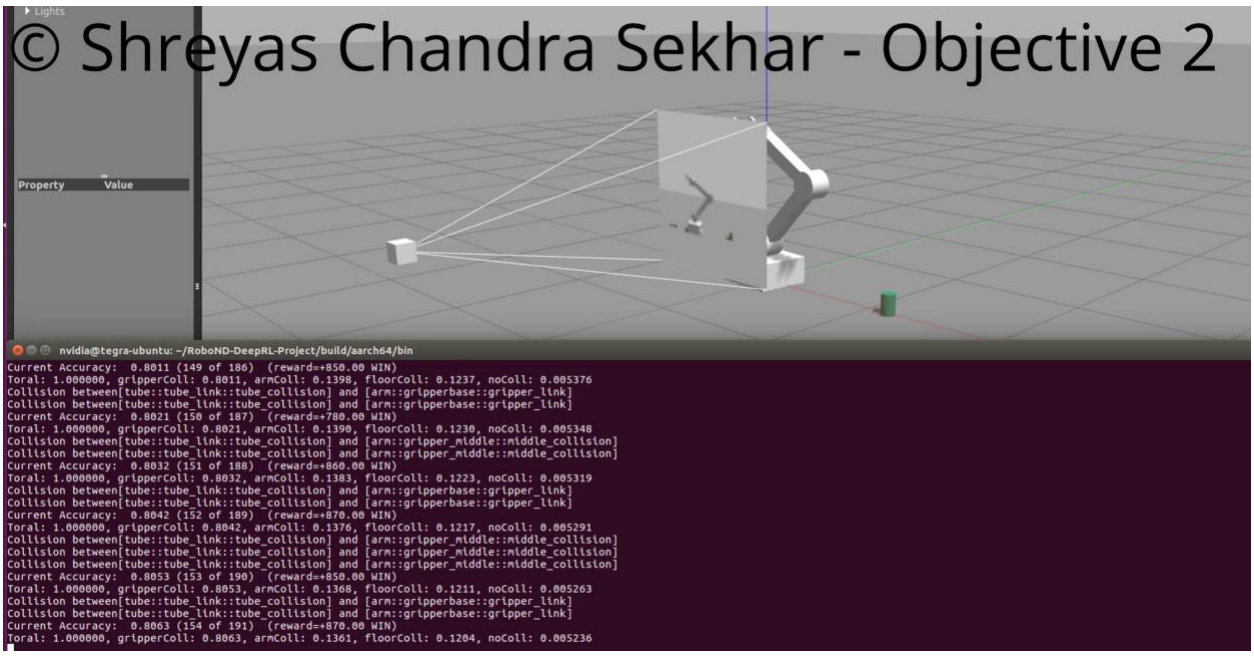
```
if (gripperCollisionCheck || distGoal == 0.0f)
{
        rewardHistory = REWARD_WIN * (1.0f - float(episodeFrames) /
float(maxEpisodeLength) * 1000.0f);
        newReward  = true;
        endEpisode = true;
}
else if (armCollisionCheck)
{
        nArmTargetCollision++;

        rewardHistory = REWARD_LOSS * 2.0f + avgGoalDelta;
        newReward  = true;
        endEpisode = true;
}
```

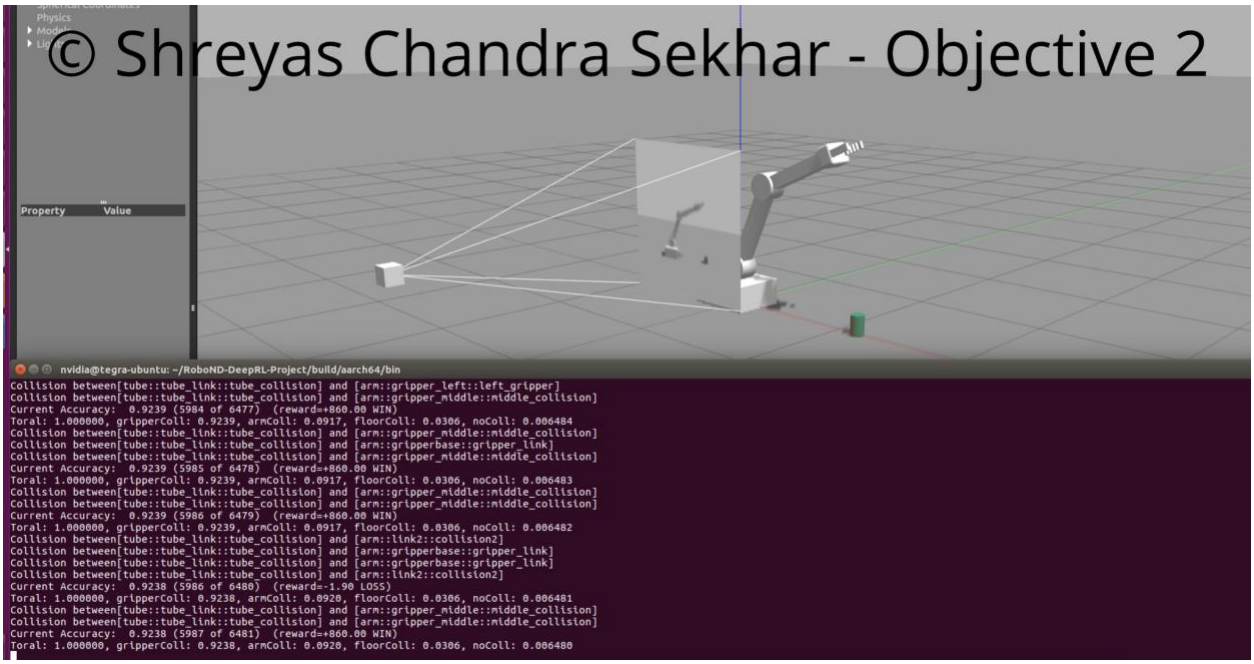
Results –
Both the objectives were met. The arm initially tried learning and the success rate increases after few attempts.

Objective 1
Arm hitting the target reached 94 percent accuracy after 138 trials



Objective 2
Gripper hitting the target reached 80 percent after 186 trials and 92 percent after 6481 trials





Future work –
I would like to work on the additional challenges mentioned in the project.

- Object Randomization: Object instantiate at different locations along the x-axis
- Increasing the Arm's Reach: Modify arm base as rotate instead of fixed. This will lead to change in object's starting location and the arm will be allowed to rotate about its base
- Increasing Arm's Reach with Object Randomization