Project: Deep RL Arm Manipulation

Author: Sim Kae Wanq

Introduction

This article describes the content of Udacity Robotics Software Nanodegree – Deep RL Arm Manipulation. The goal of this project was creating a DQN agent and defining reward system in order to teach a robotic arm to carry out two objectives in a simulated environment, gazebo. The two objectives are:

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

Implementation

The Armplugin.cpp is creating the DQN agent and training robotic arm to learn to touch the prop. This section lists the implementation of Armplugin.cpp.

ArmPlugin::Load()

In this function, camera node and collision node are created and subscribed to respective topics.

```
void ArmPlugin::Load(physics::ModelPtr _parent, sdf::ElementPtr /*_sdf*/)
{
    …
    // Create our node for camera communication
    cameraNode->Init();
    cameraSub = cameraNode->Subscribe("/gazebo/" WORLD_NAME "/camera/link/camera/image",
&ArmPlugin::onCameraMsg, this);

    // Create our node for collision detection
    collisionNode->Init();
    collisionSub = collisionNode->Subscribe("/gazebo/" WORLD_NAME "/" PROP_NAME
"/tube_link/my_contact", &ArmPlugin::onCollisionMsg, this);
    …
}
```

ArmPlugin::createAgent()

In this function, the DQN agent is created by providing the hyperparameters to the constructor. The number of possible actions is defined as 2x the number of joints (DOF) as they can rotate both clockwise and counter-clockwise.

```
bool ArmPlugin::createAgent()
{
     …
     // Create DQN Agent
     agent = dqnAgent::Create(INPUT_WIDTH, INPUT_HEIGHT, INPUT_CHANNELS, DOF*2,
                              OPTIMIZER, LEARNING_RATE, REPLAY_MEMORY, BATCH_SIZE,
                              GAMMA, EPS_START, EPS_END, EPS_DECAY,
                              USE_LSTM, LSTM_SIZE, ALLOW_RANDOM, DEBUG_DQN);
     …
}
```

ArmPlugin::onCollisionMsg()

In this function, once a collision is detected I've calculated the episodic reward. The reward is positive in cases where the gripper collides with the prop tube and negative otherwise.

```
void ArmPlugin::onCollisionMsg(ConstContactsPtr &contacts)
{
    …
    for (unsigned int i = 0; i < contacts->contact_size(); ++i)
    {
        …
        //Objective 1
        /*
        rewardHistory = REWARD_WIN;
        newReward  = true;
        endEpisode = true;
        return;
        */

        //Objective 2
        // Check if there is collision between the arm and object, then issue learning reward
        // If grip colides with tube
        bool collisionCheck = (strcmp(contacts->contact(i).collision2().c_str(), COLLISION_POINT)
== 0);

        if (collisionCheck)

        {
            rewardHistory = REWARD_WIN;
            newReward  = true;
            endEpisode = true;
            return;    // multiple collisions in the for loop above could mess with win count
        }
        else {
            rewardHistory = REWARD_LOSS;
            newReward  = true;
            endEpisode = true;
        }
    }
}
```

ArmPlugin::updateAgent()

This function updates the arm joints based on the action chosen by the DQN Agent. Depending of the joint control method the position of the joints can be incremented by a small position or a velocity delta

```
bool ArmPlugin::updateAgent()
{
    …
#if VELOCITY_CONTROL
        // if the action is even, increase the joint position by the delta parameter
        // if the action is odd,  decrease the joint position by the delta parameter
        float velocity = vel[action/2] + actionVelDelta * ((action % 2 == 0) ? 1.0f : -1.0f);
        …
#else
        // Set joint position based on whether action is even or odd.
        float joint = ref[action/2] + actionJointDelta * ((action % 2 == 0) ? 1.0f : -1.0f);
        …
#endif
        …
}
```

ArmPlugin::OnUpdate()

This function calculates the interim reward used to train the DQN Agent. A smoothed moving average of the delta of distance to the goal is used.

```
void ArmPlugin::OnUpdate(const common::UpdateInfo& updateInfo)
{
    …
    // Issue an interim reward based on the distance to the object
    const float distGoal = BoxDistance(gripBBox, propBBox);

    // issue an interim reward based on the delta of the distance to the object
    if( episodeFrames > 1 )
    {
        const float distDelta  = lastGoalDistance - distGoal;
        const float alpha  = 0.9f;
        const float timePenalty  =  0.50f;

        // compute the smoothed moving average of the delta of the distance to the goal
        avgGoalDelta  = (avgGoalDelta * alpha) + (distDelta * (1.0f - alpha));
        rewardHistory = avgGoalDelta * REWARD_MULTIPLIER - timePenalty;
        newReward     = true;
    }

    lastGoalDistance = distGoal;

    …
}
```

Reward Function

In objective 1, to train the robotic arm to touch the tube with any part:

1. REWARD_WIN is given when any part of the robotic arm collide with the tube.
2. REWARD_LOSS is given when arm hit the ground
3. REWARD_LOSS is given when episode end, but arm touch nothing
4. Interim reward proportional to the distance between arm and tube is given

Reward system of Objective 2 is very similar to objective 1, except that REWARD_LOSS is given when collision happen on non-gripper base part, in order to train the arm to touch the tube with only gripper base:

1. REWARD_WIN is given when gripper base collide with the tube.
2. REWARD_LOSS is given when other part of arm collide with the tube
3. REWARD_LOSS is given when arm hit the ground
4. REWARD_LOSS is given when episode end, but arm touch nothing
5. Interim reward proportional to the distance between arm and tube is given

Hyperparameters

This section brief the hyperparameter used for DQN. Code below shows the hyperparameter set for objective 1 and objective 2:

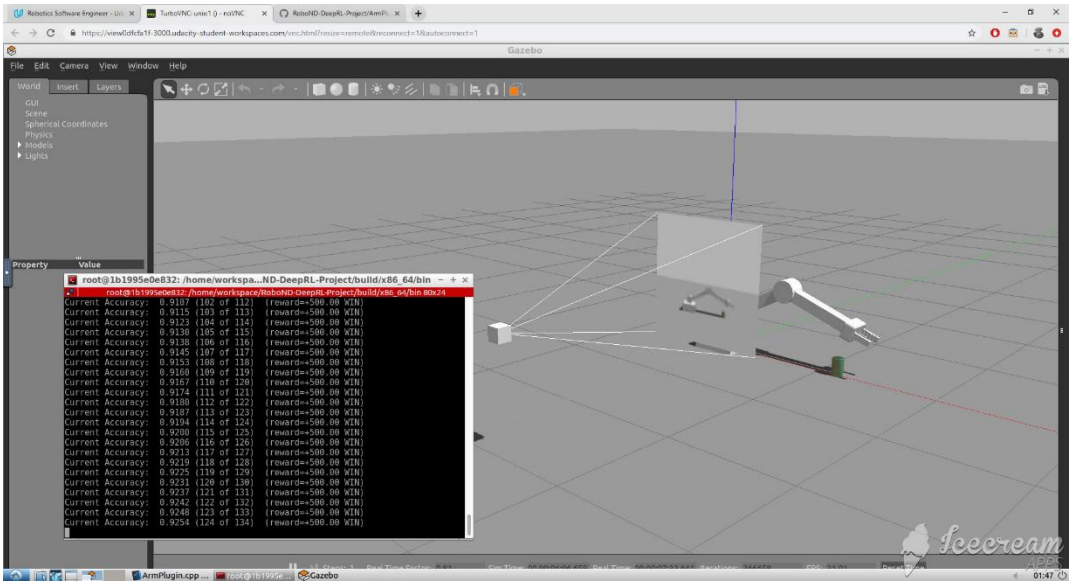
```
#define INPUT_WIDTH   64
#define INPUT_HEIGHT  64
#define OPTIMIZER "RMSprop"
#define LEARNING_RATE 0.01f
#define REPLAY_MEMORY 10000
#define BATCH_SIZE 32
#define USE_LSTM true
#define LSTM_SIZE 256
```

- Input image size is smallest as 64x64 ease the convergence

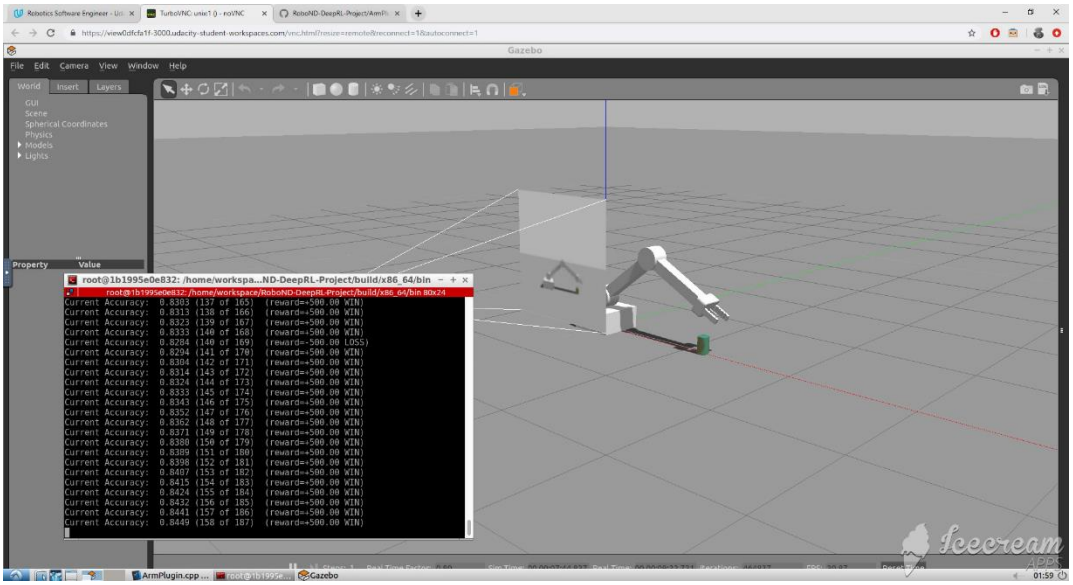
- “RMSprop” optimizer is usually a good choice for recurrent neural network
- Small learning rate and larger batch size is used
- Enable the LSTM architecture to keep track of the previous states in the network internal memory

Results

For objective 1, 90% of accuracy was achieved after 100+ of episodes.



For objective 2, 80% of accuracy was achieved after 140+ of episodes.



Future Work

In future, it’s good to improve the interim reward system and position update to make the robotic arm has a smoother movement. Besides, instead of touching the tube, should try to make the arm learn to

grab the object. Then, finally deploy it in actual world to enhance the performance of arm manipulator in learning and doing dynamic task.