Udacity Manipulator Arm Report

Ussama Naal

May 2019
v1.0

1 Introduction

This write-up describes the steps taken to train a Robot Arm Manipulator to grab a static object using DQN techniques. We start first by describing the implementation and changes to the supplied code, then we list the current configuration of parameters used during training. Finally, we present the results and closing statements.

2 Implementation

2.1 Subscribe to camera and collision topics

```
void ArmPlugin::Load(physics::ModelPtr _parent, sdf::ElementPtr /*_sdf*/)
{
    ...
    cameraSub = cameraNode->Subscribe(
        "/gazebo/arm\_world/camera/link/camera/image",
        &ArmPlugin::onCameraMsg, this);
    ...
    collisionSub = collisionNode->Subscribe(
        "/gazebo/arm_world/tube/tube_link/my_contact",
        &ArmPlugin::onCollisionMsg, this);
    ...
}
```

First line gives us camera feed which is then passed to the deep neural network to process. Second line subscribes to arm contact or collision topic.

2.2 Create the DQN Agent

The following code snippet creates the dqnAgent:

```
bool ArmPlugin::createAgent()
{
    ...
    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 );
    ...
}
```

2.3 Velocity or position based control of arm joints

Within the ArmPlugin::updateAgent() method, we have the choice to control arm joints through velocity or by adjusting the position.

The following two listing shows the changes made to the code, based on whether the action value is odd or even we decide if the velocity or position change positive or negative. Additionally, I modified the code such that the new velocity or joint deltas updates previous state as increments or decrements.

```
bool ArmPlugin::updateAgent()
{
#if VELOCITY_CONTROL
    ...
    float velocity = actionVelDelta * (action % 2 == 0 ? 1 : -1);
    ...
    vel[action/2] += velocity;  // This was originally a direct assignment
    ...
#else
    ...
    float joint = actionJointDelta * (action % 2 == 0 ? 1 : -1);
    ...
    ref[action/2] += joint; // This was originally a direct assignment
    ...
#endif
}
```

2.4 Reward for robot gripper hitting the ground

First I start by impelementing a helper function for checking that whether the gripper hit the or not. The function accepts two parameters, first a bounding box describing the gripper, the other is merely floor level.

```
bool ArmPlugin::checkGroundContact(const math::Box& box, float floor) const
{
        return (box.min.z < floor);
}
```

We call this method within ArmPlugin::OnUpdate and if the condition is met we punish the robot with a negative rewards and start a new episode as shown below

```
void ArmPlugin::OnUpdate(const common::UpdateInfo& updateInfo)
{
        ...
        if(checkGroundContact(gripBBox, groundContact))
        {
                if(true){printf("GROUND CONTACT, EOE .................\n");}
                rewardHistory = REWARD_LOSS;
                newReward     = true;
                endEpisode    = true;
        }
        ...
}
```

2.5 Issue an interim reward based on the distance to the object

In this task we need to issue an appropriate reward to the robot as the arm gripper approaches the object. However, we shouldn't rely on the direct changes of distance as they are too noisey. Instead we compute a moving average of the changes. For that we start by declaring a circular buffer to hold the last 5 distance deltas to the object in the ArmPlugin object as follows:

```
class ArmPlugin : public ModelPlugin
    ...
    boost::circular_buffer<float> distDeltas;
```

Within ArmPlugin::OnUpdate method, we compute the distnce between the gripper and the arm gripper and the object using the supplied BoxDistance method. We compute the difference from last distance and push the value to the distDeltas buffer and compute the average of the last 5 samples and store in avgGoalDelta. We set the reward by multiplying avgGoalDelta by our defined REWARD_WIN constant. Details shown in the following listing:

```
void ArmPlugin::OnUpdate(const common::UpdateInfo& updateInfo)
{
        ...
    if(!checkGroundContact(gripBBox, groundContact))
    {
        const float distGoal =
            BoxDistance(gripBBox, prop->model->GetBoundingBox());

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

            // compute the smoothed moving average of the delta
            // of the distance to the goal
            distDeltas.push_back(distDelta);
            avgGoalDelta = 0;
            for (auto e : distDeltas)
                avgGoalDelta += e;
            avgGoalDelta /= distDeltas.size();

            rewardHistory = avgGoalDelta * REWARD_WIN;
            newReward     = true;
        }

        lastGoalDistance = distGoal;
    }
        ...
}
```

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

Earlier we subscribed to the "/gazebo/arm_world/tube/tube_link/my_contact" topic, which notifies us when a contact happen between the arm and another object. We a collision detected we check if the contact between the gripper and the object. If it is the case we issue a reward equivalent to REWARD_WIN constant and consider the episode done. Otherwise, if the contact was between another part of the arm and the object we issue a loss reward REWARD_LOSS and consider the episode done too. We consider the later case a loss and terminate the episode since the interaction between the arm body and the object results of the object being pushed away from its place.

```
void ArmPlugin::onCollisionMsg(ConstContactsPtr &contacts)
{
    ...
    successfulTouch = true; /* NEW VARIABLE */
```

```
    if (contacts->contact(i).collision1() == COLLISION_ITEM &&
                        contacts->contact(i).collision2() == COLLISION_POINT)
    {
        rewardHistory = REWARD_WIN;
        newReward   = true;
        endEpisode = true;
        return;
    }
    else
    {
        rewardHistory = REWARD_LOSS;
        newReward   = true;
        endEpisode = true;
    }
    ...
}
```

2.7 Track the number of successful touches

The ArmPlugin code already tracks the number of successful grabs, i.e when the target object is touched by the gripper. However, it doesn't track the number of successful touches i.e when the target object is touched by any part of the arm (collision2, gripper, ..). For that we add a new variable named successfulTouches and track and report its value on the console as follows:

```
void ArmPlugin::OnUpdate(const common::UpdateInfo& updateInfo)
{
...
    if (successfulTouch)
    {
        successfulTouches++;
        successfulTouch = false;          //reset
    }

    printf("Current Accuracy:  "
        "Touches = %0.4f (%03u of %03u), Grabs = %0.4f (%03u of %03u)"
        "(reward=%+0.2f %s)\n",
        float(successfulTouches)/float(totalRuns), successfulTouches, totalRuns,
        float(successfulGrabs)/float(totalRuns), successfulGrabs, totalRuns,
        rewardHistory, (rewardHistory >= REWARD_WIN ? "WIN" : "LOSS"));
...
}
```

3 Tuning

Table 1, lists all used parameters and their values with a brief explanation to the used values when a value has been changed.

I have tested the code with USE_LSTM set to true but for me this resulted in worse performance. This may need further investigation.

4 Results

Under current implementation and the selected parameter values the robot achieves about 70% successful touches and 55% successful grabs as it learns. Currently, it is not clear if it is possible to gain better results

Table 1: Parameters

Parameter	Value	Explanation
VELOCITY_CONTROL	false	Based on recommendations from one of the mentors
EPS_START	0.9f	Default
EPS_END	0.05f	Default
EPS_DECAY	200	Default
INPUT_WIDTH	512	Default
INPUT_HEIGHT	512	Default
OPTIMIZER	"RMSprop"	Possibly the only available option
LEARNING_RATE	0.1	Changed from zero
REPLAY_MEMORY	1000	Reduced from 10000 to avoid "out of memory" exceptions
BATCH_SIZE	8	Default
USE_LSTM	false	true value didn't produce good results for me
LSTM_SIZE	256	LSTM is not enabled so it doesn't matter
REWARD_WIN	100.0f	Needs to be high enough since it is multiplied by avgGoalDelta
REWARD_LOSS	-100.0f	Loss should amount to a similar value as win

if we let the agent run for more rounds. Every-time we re-run the simulation the agent resets and start to train from the zero state.

The following figure shows the results at around 100+ episodes.

Typically, once the robot learns and we should be able to disable or reduce the amount of randomness in agent and apply the optimal policy and that should give us better results. However, it is not clear to me how to make the optimal policy effective all the time from the code.

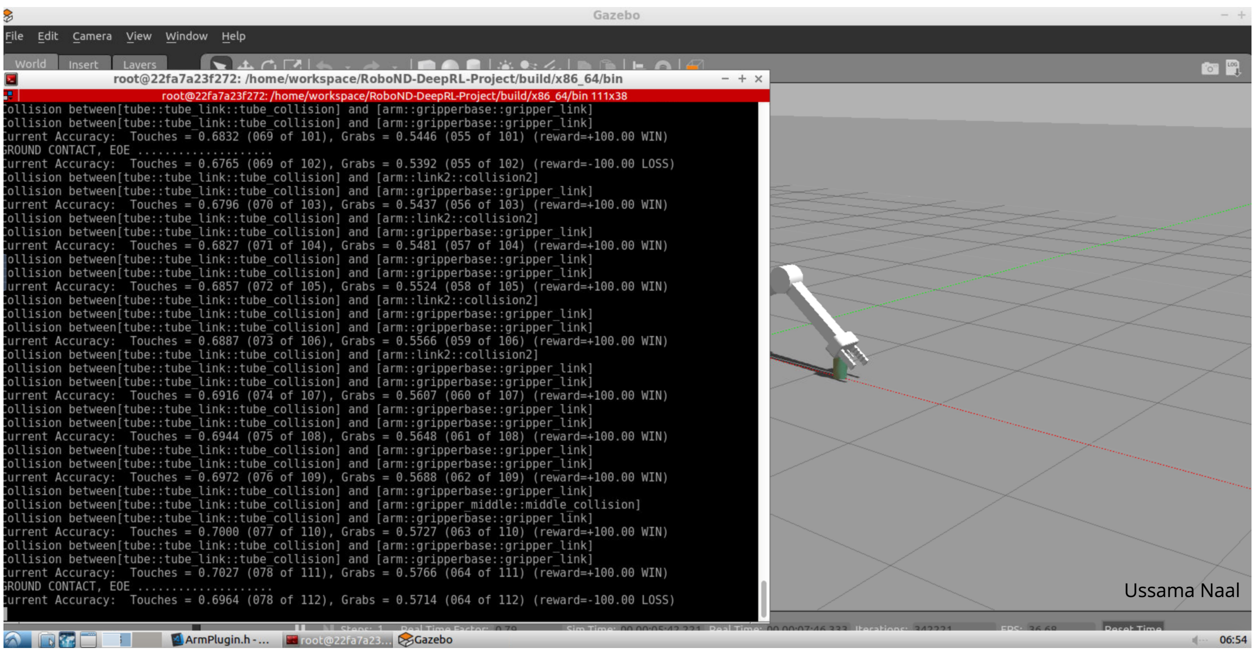

Figure 1: Best achieved result.