Udacity Manipulator Arm Report

Ussama Naal

May 2019
v3.0

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. With USE_LSTM enabled I have tested with LSTM_SIZE of 32, 64, 128, 256 and results were same. Almost 0% successful grabs. In addition to that, whenever USE_LSTM is enabled the simulation starts to misbehave quite often.

For the reward value, I have initially started with +100 and -100 for win and loss respectively. However, when I doubled these two values the agent was able to learn a bit faster and avoid making less mistakes.

Table 1: Initial 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	64	Reduced from 512 since gazebo camera output is only 64x64
INPUT_HEIGHT	64	Reduced from 512 since gazebo camera output is only 64x64
OPTIMIZER	"RMSprop"	Possibly the only available option
LEARNING_RATE	0.1	With few test this value seems reasonable
REPLAY_MEMORY	10000	Default
BATCH_SIZE	64	Increased from 8: produces more more consistent results
USE_LSTM	false	true value didn't produce good results for me (see below)
LSTM_SIZE	32	Default: LSTM is not enabled so it doesn't matter
REWARD_WIN	200.0f	Needs to be high enough since it is multiplied by avgGoalDelta
REWARD_LOSS	-200.0f	Loss should amount to a similar value as win

4 Results

4.1 Initiail Results

Under current implementation and the selected parameter values the robot achieves about 90% successful touches and 88% successful grabs within 130 episodes (Check Figure 1). It reaches easily 80% within just 100 iterations.

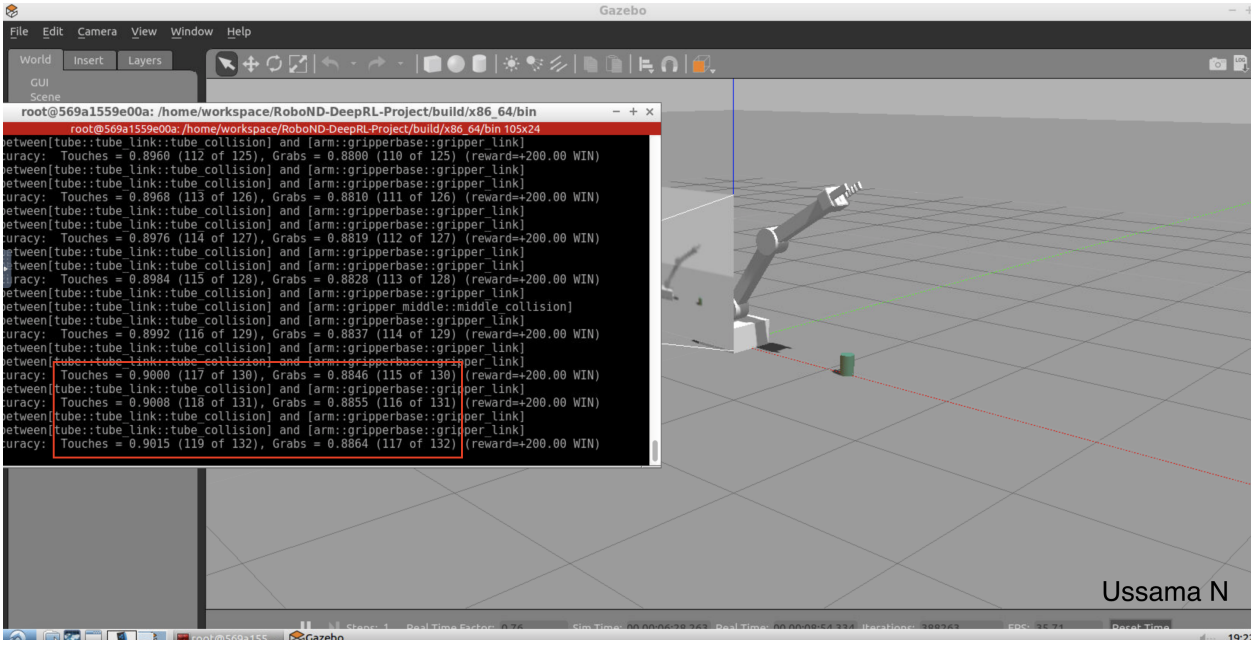

Figure 1: Initial achieved result.

Doubling the REWARD values for WIN and LOSS from previous tests, had a huge impact on the agent performance. This signals to the agent that the final results is the only that matters.

Some noticeable shortcomings is that the first few iterations greatly decides the final outcome. If the agent was able to touch or grab the object in the beginning, this results in a better chance of the agent achieving the required performance within a 100-150 runs. However, assuming the agent can do more runs then this wouldn't be an issue.

4.2 Improved Results

To make the arm gripper achieve 80+% accuracy within a 100 frames. I have made couple few changes to the code and tuned parameters appropriately.

In ArmPlugin::OnUpdate we made the following changes:

```
void ArmPlugin::OnUpdate(const common::UpdateInfo& updateInfo)
{
    ...
    const float a = 0.5;
    avgGoalDelta = a * avgGoalDelta + (1 - a) * distDelta;
    rewardHistory = (avgGoalDelta > 0.01 ? avgGoalDelta : 0.1 * REWARD_LOSS);
    newReward     = true;
        ...
}
```

What this change does it rewards the RL agent only when making significant steps towards the target (+0.01) otherwise the agent is rewarded negatively when it moves away or when the arm stay somewhat stationary.

Another change we implemented is when part of the arm hits the object we used to terminate the episode immediately in fear that the target moves from its place. This was not the cast all the time. The target object can potentially move away but only in fraction of the cases this happens. When the target object stays in place the arm can potentially proceed and succeed in grabbing the object. This allows us to gain more successful grabs. The following listing summarizes these changes

```
void ArmPlugin::onCollisionMsg(ConstContactsPtr &contacts)
{
    ...
    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 = false; /* Don't end episode, keep going */
    }
    ...
}
```

Since we removed the multiplication by REWARD_WIN for interim rewards. The REWARD_WIN and REWARD_LOSS had to be adjusted. Table 2 shows only updated parameters. Additionally, we lowered the learning rate from 0.1 to 0.01, this resulted in more expected outcome across different runs of the entire simulation.

Figure 2 shows the outcome of this run under the updated code and parameters. We can clearly see that the arm gripper was able to execute 83 successful grabs out of 100 runs.

Table 2: Updated Parameters

Parameter	Value	Explanation
LEARNING_RATE	0.01	Lowering this from previous makes the outcome consistent
BATCH_SIZE	32	Reduced from 64
REWARD_WIN	1.0f	Scales well with the new interim reward: avgGoalDelta
REWARD_LOSS	-1.0f	Consistent with REWARD_WIN

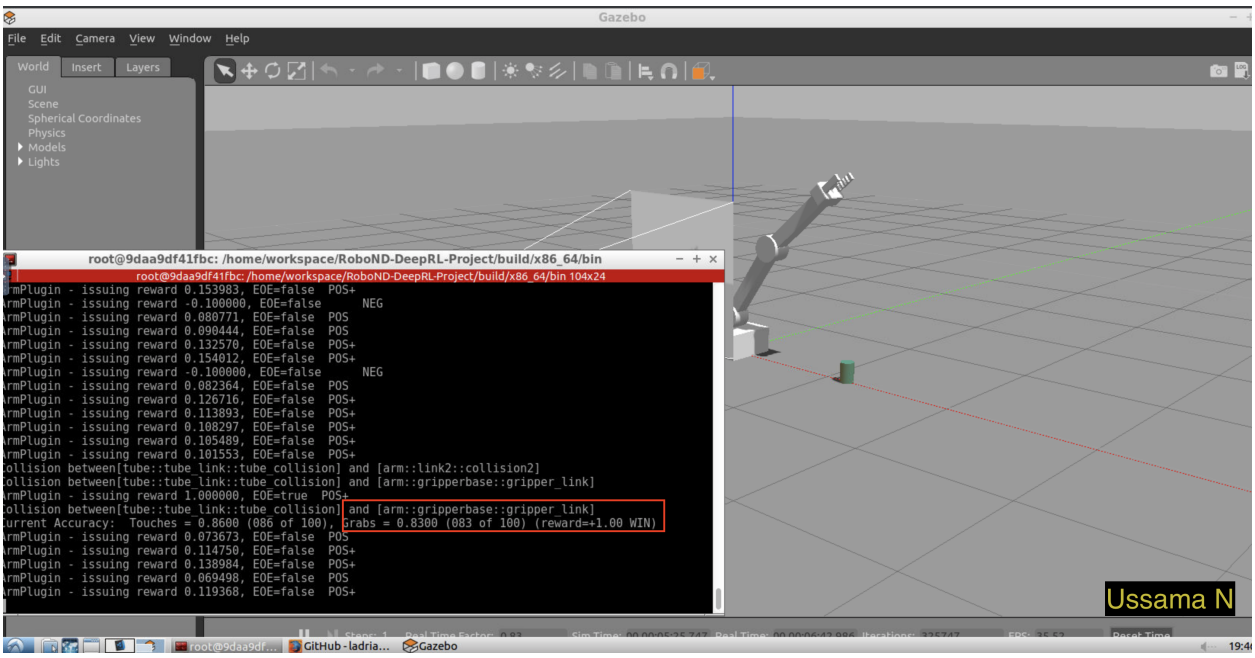

Figure 2: Best achieved result.

4.3 Conclusions

To improve the results even further, we can let the agent train for longer time such that it will have better chance of discovering the optimal policy. Once the robot learns, 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.

Another way to produce better results is to a grid search on the list of parameters.