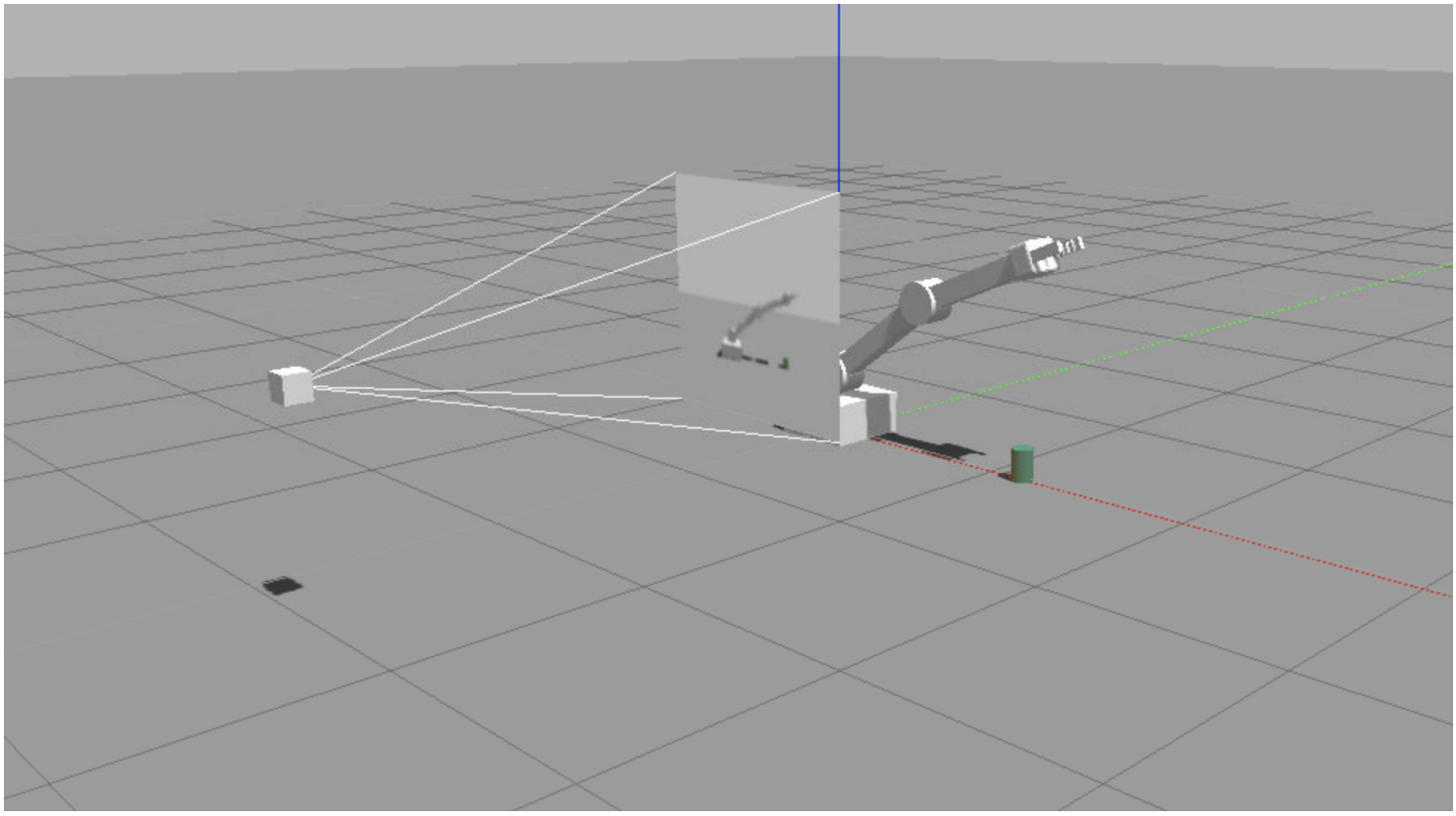
Fernando Jaruche Nunes
Dec 6 · 6 min read

Udacity Robotics ND Project 8–Deep RL Arm Manipulation



Robotic Arm simulated in ROS Gazebo

Introduction

In this article I'll go over my submission for the Udacity Robotics ND Project 8— Deep RL Arm Manipulation. The goal of this project is to create a DQN agent and define reward functions to teach a robotic arm to carry out two primary objectives in a simulated environment:

1. Have any part of the robot arm touch the object of interest, with at least a 90% accuracy.

2. Have only the gripper base of the robot arm touch the object, with at least a 80% accuracy.

I'll go over the implementation details of the agent below.

DQN Agent Implementation

The `ArmPlugin` plugin is responsible for creating the DQN agent and training it to learn to touch the prop. Below I'll go over the main changes to the original `ArmPlugin` methods for this project.

ArmPlugin::Load()

In this function I create the camera and collision nodes and subscribed them to the respective topics. Also ArmPlugin::OnUpdate() will be called for every world update.

```
// 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);


// Listen to the update event. This event is broadcast every
simulation iteration.
this->updateConnection =
event::Events::ConnectWorldUpdateBegin(boost::bind(&ArmPlugi
n::OnUpdate, this, _1));
```

ArmPlugin::createAgent()

I've then created the DQN agent 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.

```
// 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()

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:

```
// Check if there is collision between the gripper and
object, then issue learning reward
if((strcmp(contacts->contact(i).collision1().c_str(),
COLLISION_ITEM) == 0) &&
(strcmp(contacts->contact(i).collision2().c_str(),
COLLISION_POINT) == 0))
{
    rewardHistory = REWARD_WIN;
    newReward  = true;
    endEpisode = true;

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

ArmPlugin::updateAgent()

This method 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. I'll discuss the difference between the two in the next section.

```
#if VELOCITY_CONTROL
// Increase or decrease the joint velocity based on whether
the action is even or odd
// Set joint velocity based on whether action is even or
odd.
float velocity = vel[action/2] + actionVelDelta * ((action %
2 == 0) ? 1.0f : -1.0f);
```

```
(...)
#else
// Increase or decrease the joint position based on whether
the action is even or odd
const int jointIdx = action / 2;
// Set joint position based on whether action is even or
odd.
float joint = ref[jointIdx] + actionJointDelta * ((action %
2 == 0) ? 1.0f : -1.0f);
#endif
```

ArmPlugin::OnUpdate()

This method calculates the interim reward used to train the DQN Agent. I've used a smoothed moving average of the delta of the distance to the object and a time penalty.

```
const float distDelta = lastGoalDistance - distGoal;
const float movingAvg = 0.9f;
const float timePenalty = 0.50f;
```

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

Decision on Joint Control Methods

There are two joint control mechanisms available to control the arm joints: velocity or position based. I'll quickly discuss the differences between the two approaches as how each influenced in the DQN accuracy:

Velocity Based

The velocity based control takes into account the Arm physics where changes in velocity determine the robot position. This method produces movements that are way more realistic in contrast to controlling the joints position directly.

Position Based

The position based control is simpler in contrast as the Arm position is determined directly. While simpler to control this method produces unrealistic motion where position changes instantly producing accelerations that cannot be matched by a real robotic arm.

For the purpose of this project **I picked the position based control method** as it is simpler and led to more accurate results in simulation.

Reward Functions

I've created the following reward functions to address the objectives #1 and #2 respectively:

Reward Function #1—Have any part of the robot arm touch the object of interest

For the first objective I've created a reward function that rewards any collision between the arm and the tube, and penalizes when robot collides with itself or the ground.

I've also introduced an interim reward proportional to the distance to the objective looking to provide the agent a "trail of breadcrumbs" that creates a closer association between action and reward.

Finally, I've added a time penalty that guarantees the agent will reach the objective as quickly as possible. This factor punishes behaviors, such as wasting time by getting close to the tube and collecting as much interim rewards before touching it.

Here a summary of the reward function:

1. Any part of the Arm collides with the tube: +500
2. Any other form of collision (i.e. Arm colliding with itself): -500
3. Gripper hits the ground: -500
4. Interim reward based on the smoothed moving average of the delta of the distance to the object and a time penalty

Reward Function #2—Have only the gripper base of the robot arm touch the object

The Reward Function #2 is very similar to the one above and it has a slight modification to the collision reward. This function will only reward collisions between the gripper and the tube:

1. **Gripper (only) collides with the tube: +500**
2. **Any other form of collision (i.e. Arm colliding with itself or link2 collides with the tube): -500**
3. Gripper hits the ground: -500
4. Interim reward based on the smoothed moving average of the delta of the distance to the object and a time penalty

Hyperparameters

Below I'll describe my DQN hyperparameters and the reasoning behind the selection.

Image Input size

Keeping the state space smallest as possible helped with convergence:

```
#define INPUT_WIDTH 64
#define INPUT_HEIGHT 64
```

Training

I chose RMSProp as the optimizer as it is usually a good choice for recurrent neural networks.

```
#define OPTIMIZER “RMSprop”
```

I kept the learning rate unchanged but had I ran into convergence issues I would have tried to lower this value.

```
#define LEARNING_RATE 0.1f
```

Looking to keep batch gradient updates stable I chose to keep a pool of decorrelated replay memory transitions and increased the batch size:

```
#define REPLAY_MEMORY 10000
#define BATCH_SIZE 32
```

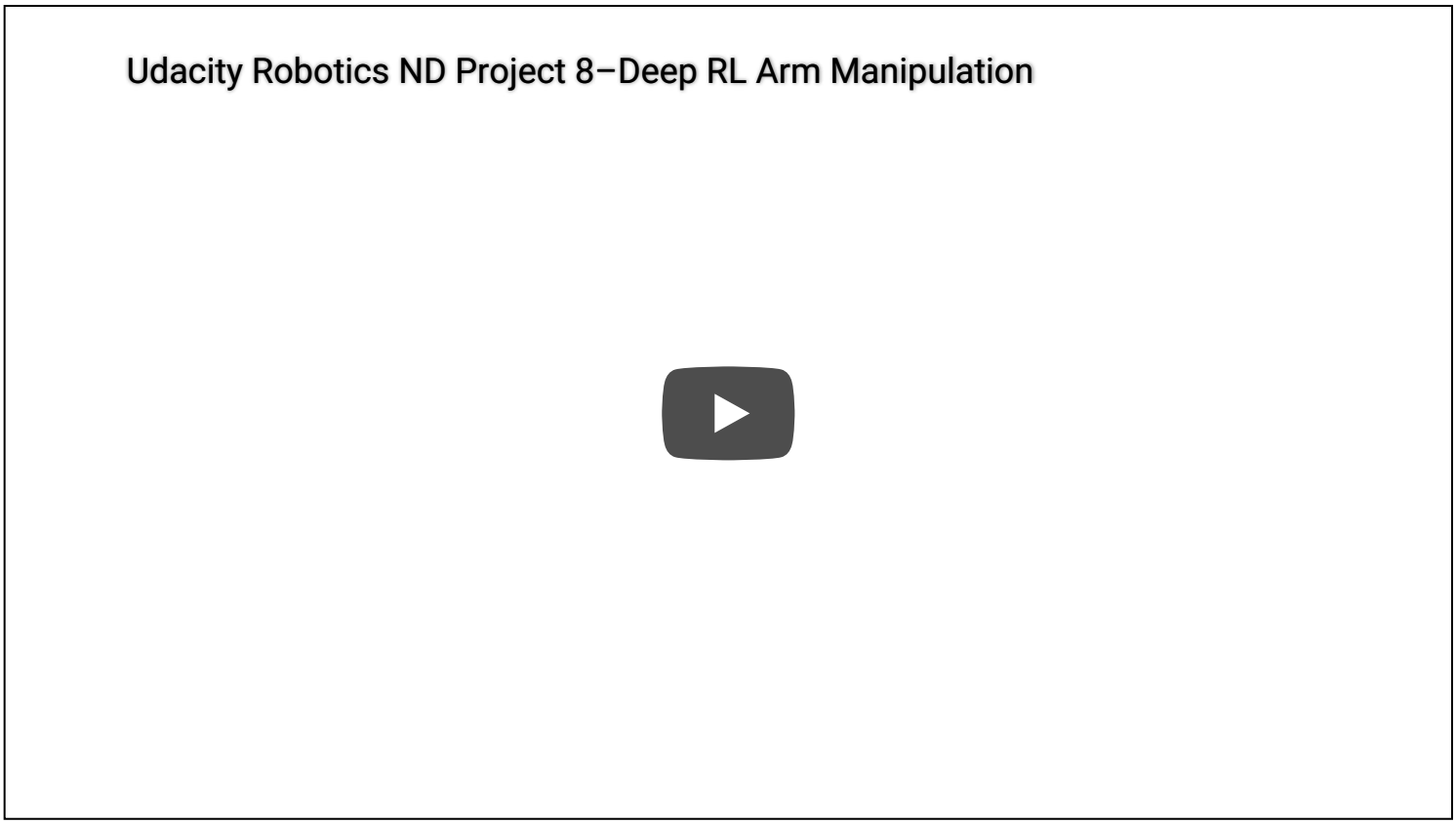
Network Architecture

I chose to use the LSTM architecture as a mean to keep track of the previous states in the network internal memory.

```
#define USE_LSTM true
#define LSTM_SIZE 256
```

Results

I'm pleased with the results. I got over 90% accuracy for both reward functions (see video below) in less than 200 episodes. The time penalty introduced last was crucial to have the Arm to complete the task quickly. Before introducing this factor the robot would get close to the tube and move back forth collecting as much interim reward as possible without ever touching the tube.



Results for Reward Functions #1 and #2 — Over 90% accuracy in both cases

The complete code can be found at my repo:

fjnunes/RoboND-DeepRL-Project

Contribute to fjnunes/RoboND-DeepRL-Project development by creating an account on GitHub.

github.com

Future Work

This was an extremely useful project. From all the RL examples I've seen so far this was the first that uses Gazebo as simulation environment. Thats really exciting as I can use the concepts from previous lessons to create new worlds in URDF and apply RL on top of them. I.e. I want to try to create a race track and have cars learning to complete a lap in the quickest time.

Also, if time permits, I would like to try to control the robotic arm in simulation with the velocity based method as it is more realistic and would allow me to test the resulting DQN in a real robot.