Deep RL Arm Manipulation Project

Douglas Teeple

Abstract—The goal to the Deep RL Arm Manipulation project is to create a DQN agent and define reward functions to teach a robotic arm to carry out two primary objectives:

1) Have any part of the robot arm touch the a designated object on the ground, with at least a 90% accuracy.
2) Have only the gripper base of the robot arm touch the designated object, with at least a 80% accuracy.

Index Terms—Robot, Mobile Robotics, Deep RL.

1 Introduction

Deep Reinforcement Learning is an active area of research in robotics. This project is a challenging Deep RL implementation using a gazebo environment and an arm plugin (*ArmPlugin.cpp*) that assigns rewards, either positive or negative, on repeated attempts by the robot to touch a cylinder in the gazebo world without touching the ground, to train the robot arm to reached the desired accuracy goal. The project was implemented on a Jetson TX2 board running Linux 16.04. The project sources reside in the git directory https://github.com/douglasteeple/DeepRL.

This figure shows the arm in the start position:

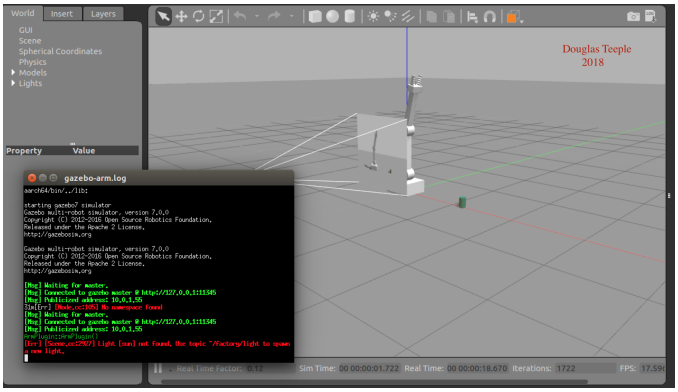

Figure 1. Start position of the arm.

The arm is in the vertical home position and the agent has not yet initialized.

2 Tasks

The following tasks were completed:

2.1 Tube Collision

1) Subscribe to camera and collision topics.
The task is to create the subscribers in the *ArmPlugin::Load()* function based on the following:
A) **cameraNode**
The camera node receives images from a camera in the scane, which are passed to the DQN agent. The first parameter is the topic name:

- /gazebo/arm_world/camera/link/camera/image

The second callback Function parameter is the reference parameter *ArmPlugin::onCameraMsg*, and the third is the class instance "*this*". The C++ code is as follows:

Listing 1. Camera Subcribe

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

B) **collisionNode**
The collision node receives collision messages from subscribed topics. The topic is the cylinder (called "tube" in the gazebo-arm world). The subscribed topic name is:

- /gazebo/arm_world/tube/tube_link/my_contact

The second argument is a reference to the callback Function called *ArmPlugin::onCollisionMsg* and the second a class instance, the "*this*" pointer.
The C++ code is as follows:

Listing 2. Collision Subscribe

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

2) Create the DQN Agent
In this task a call is made to the *Create()* function from the dqnAgent Class, in *ArmPlugin::createAgent()*. The parameters are defined as:

Listing 3. HyperParameters

```
// Define DQN API Settings

#define INPUT_CHANNELS 3
#define ALLOW_RANDOM true
#define DEBUG_DQN false
#define GAMMA 0.9f
#define EPS_START 0.9f
#define EPS_END 0.05f
#define EPS_DECAY 200
/*
/ Tune the following hyperparameters
*/
#define INPUT_WIDTH    64                  // TUNED from 512
#define INPUT_HEIGHT   64                  // TUNED from 512
#define INPUT_CHANNELS 3
#define OPTIMIZER "RMSprop"                // TUNED from "None"
#define LEARNING_RATE 0.2f                 // TUNED from 0.0
#define REPLAY_MEMORY 10000
#define BATCH_SIZE 32                      // TUNED from 8
#define USE_LSTM true                      // TUNED from false
#define LSTM_SIZE 32
```

The C++ code for the *createAgent* call is as follows:

Listing 4. Create an Agent Instance

```
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);
```

3) Velocity or position based control of arm joints The DQN output is mapped to a particular action which is the control of each joint for the robotic arm. In *ArmPlugin::updateAgent()*, there are two approaches to control the joint movements:

   a) Velocity Control - The current value for a joint velocity is stored in the array vel. The array lengths are based on the number of degrees of freedom for the arm. The variable *velocity* receives a value based on the current joint velocity and the associated delta value *actionVelDelta* which is a tunable parameter defined in the class initializer:

Listing 5. Velocity Action Delta

```
actionVelDelta    = 0.1f;
```

The C++ code is as follows:

Listing 6. Velocity

```
const int jointIdx = action / 2;

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

if( velocity < VELOCITY_MIN )
velocity = VELOCITY_MIN;

if( velocity > VELOCITY_MAX )
velocity = VELOCITY_MAX;

vel[jointIdx] = velocity;
```

If the action is even, increase the joint velocity by the delta parameter, if the action is odd, decrease the joint velocity by the delta parameter.

   b) Position Control - The current value for a joint position is stored in the array *ref*. The array size based on the number of degrees of freedom for the arm. For this project DOF is 3. The variable joint is modified based on the current joint velocity and the associated delta value defined in the class initializer *actionJointDelta*.

Listing 7. Action Joint Delta

```
actionJointDelta = 0.1f;
```

The C++ code is as follows:

Listing 8. Joint Position

```
const int jointIdx = action / 2;
float joint = ref[jointIdx] + actionJointDelta *
       ((action % 2 == 0) ? 1.0f : -1.0f);

// limit the joint to the specified range
if( joint < JOINT_MIN )
joint = JOINT_MIN;

if( joint > JOINT_MAX )
joint = JOINT_MAX;

ref[jointIdx] = joint;
```

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.

The important variables in relation to rewards are:

- rewardHistory - Value of the previous reward.
- REWARD_WIN - The value for positive rewards.
- REWARD_LOSS - The values for negative rewards.
- newReward - Whether a reward has been issued or not.
- endEpisode - Whether the episode is over or not.

1) Reward for robot gripper hitting the ground. The function *GetBoundingBox()* returns the minimum and maximum values of the x, y, and z axes.
Using *GetBoundingBox()*, a check is made to determine if the gripper hit the ground. If the gripper hits the ground a negative reward is issued.

Listing 9. Ground Collision

```
const math::Box& gripBBox = gripper->GetBoundingBox();
const float groundContact = 0.0f;
// compute the reward from distance to the goal
const float distGoal = BoxDistance(gripBBox, propBBox);
const bool checkGroundContact =
        (gripBBox.min.z <= groundContact
        || gripBBox.max.z <= groundContact);
if(checkGroundContact)
{
  rewardHistory = REWARD_LOSS;
  newReward     = true;
  endEpisode    = true;
  std::cout << colorTable[BoldYellow]
       << "Ground Contact, distance from goal: "
       << distGoal << ", REWARD: " << rewardHistory << " EOE"
    << " dist ground = " << distGround << colorTable[Reset_]
    << std::endl;
}
```

2) Issue an interim reward based on the distance to the object. The function "*BoxDistance()*" calculates the distance between two bounding boxes. Using this function, the distance between the arm and the object is calculated and the reward is assigned.
The reward is based on a smoothed moving average of the delta of the distance to the goal. It is calculated

as the tanh of the average distance to the goal times REWARD_WIN:

Listing 10. Smoothing Function

```
const float distDelta  = fabs(lastGoalDistance - distGoal);
// 10% current dist, 90% historical average
const float alpha = 0.1;
// compute the smoothed moving average of the delta
// of the distance to the goal
avgGoalDelta  = (distDelta * alpha) +
        (avgGoalDelta * (1 - alpha));
rewardHistory = tanh(avgGoalDelta)*REWARD_WIN*REWARD_MULTIPLIER;
```

The smoothing factor allocates 10% of the current value to the reward and 90% to the running average. The *tanh* function returns values from -1.0 to 1.0 based on the input value, so that the reward value is properly scaled.

2.2 Objective One Goal - Arm Collision

Collisions are detected in the callback function *onCollisionMsg*. The first part of the project detects collisions anywhere along the arm:

Listing 11. Arm Collision

```
for (unsigned int i = 0;
        i < contacts->contact_size();
    ++i)
{
  if( strcmp(contacts->contact(i).collision2().c_str(),
        COLLISION_GROUND) == 0 )        // i.e. skip ground contact
      continue;

  if(DEBUG2){std::cout << colorTable[BoldCyan]
        << "Collision between["
    << contacts->contact(i).collision1()
    << "] and [" << contacts->contact(i).collision2()
    << "]" << colorTable[Reset_] << std::endl;}
  bool collisionCheckArm = (
        strcmp(contacts->contact(i).collision1().c_str(),
    COLLISION_ITEM) == 0) &&
    (strcmp(contacts->contact(i).collision2().c_str(),
    COLLISION_POINT_ARM) == 0); // i.e. tube with arm

  if (collisionCheckArm) {
    if (DEBUG) {std::cout << "\033[1;36mCollision between "
        << contacts->contact(i).collision1() << " and "
        << contacts->contact(i).collision2() << "\033[0m"
        << std::endl;}
    rewardHistory = REWARD_WIN;
    newReward  = true;
    endEpisode = true;
  }
}
```

2.3 Objective Two - Gripper Collision

In the second part of the project, a reward is issued based on a collision between the arm's gripper and the object. The collision check is modified as follows:

Listing 12. Negative Reward for Tube Collision

```
bool collisionCheckArm = (
        strcmp(contacts->contact(i).collision1().c_str(),
    COLLISION_ITEM) == 0) &&
    (strcmp(contacts->contact(i).collision2().c_str(),
    COLLISION_POINT_ARM) == 0); // i.e. tube with arm

if (collisionCheckArm) {
        if (DEBUG) {std::cout << "\033[1;36mCollision between "
        << contacts->contact(i).collision1() << " and "
        << contacts->contact(i).collision2() << "\033[0m"
        << std::endl;}
        if (CHECKGRIPPER) {     // arm collision gets a negative reward
                rewardHistory = REWARD_LOSS/10.0;
                newReward  = true;
                endEpisode = true;
        } else {
                rewardHistory = (1.0f - (float(episodeFrames) /
                float(maxEpisodeLength))) * REWARD_WIN;
                rewardHistory = REWARD_WIN;
                newReward  = true;
                endEpisode = true;
        }
}
```

Collisions with the gripper are split into the various parts of the gripper. Note that the reward is increased by a factor of 4, since the gripper collision is much harder to achieve.

1) The Gripper Link Collision:

Listing 13. Gripper Link Collision

```
bool collisionCheckGripper = (
        strcmp(contacts->contact(i).collision1().c_str(),
    COLLISION_ITEM) == 0) &&
    (strcmp(contacts->contact(i).collision2().c_str(),
    COLLISION_POINT) == 0);      // i.e. tube with gripper
if (collisionCheckGripper) {
        if (DEBUG) {std::cout << "\033[1;32mCollision between "
        << contacts->contact(i).collision1() << " and "
        << contacts->contact(i).collision2() << "\033[0m"
        << std::endl;}
        rewardHistory = REWARD_WIN;
        newReward  = true;
        endEpisode = true;
}
```

2) The Gripper Base Collision:

Listing 14. Gripper Base Collision

```
collisionCheckGripper = (
        strcmp(contacts->contact(i).collision1().c_str(),
    COLLISION_ITEM) == 0) &&
    (strcmp(contacts->contact(i).collision2().c_str(),
    COLLISION_POINT_GRIP) == 0);         // i.e. tube with gripper
if (collisionCheckGripper) {
        if (DEBUG) {std::cout << "\033[1;32mCollision between "
        << contacts->contact(i).collision1() << " and "
        << contacts->contact(i).collision2() << "\033[0m"
        << std::endl;}
        rewardHistory = REWARD_WIN;
        newReward  = true;
        endEpisode = true;
}
```

3) The Gripper Middle Collision:

Listing 15. Gripper Middle Collision

```
collisionCheckGripper = (
        strcmp(contacts->contact(i).collision1().c_str(),
        COLLISION_ITEM) == 0) &&
        (strcmp(contacts->contact(i).collision2().c_str(),
        COLLISION_POINT_GRIP2) == 0);    // i.e. tube with gripper
if (collisionCheckGripper) {
        if (DEBUG) {std::cout << "\033[1;32mCollision between "
        << contacts->contact(i).collision1() << " and "
        << contacts->contact(i).collision2() << "\033[0m"
        << std::endl;}
        rewardHistory = REWARD_WIN;
        newReward  = true;
        endEpisode = true;
}
```

4) The Left Gripper Collision:

Listing 16. Left Gripper Collision

```
collisionCheckGripper = (
        strcmp(contacts->contact(i).collision1().c_str(),
        COLLISION_ITEM) == 0) &&
        (strcmp(contacts->contact(i).collision2().c_str(),
        COLLISION_POINT_GRIP3) == 0);    // i.e. tube with gripper
if (collisionCheckGripper) {
        if (DEBUG) {std::cout << "\033[1;32mCollision between "
        << contacts->contact(i).collision1() << " and "
        << contacts->contact(i).collision2() << "\033[0m"
        << std::endl;}
        rewardHistory = REWARD_WIN;
        newReward  = true;
        endEpisode = true;
}
```

5) The Right Gripper Collision:

Listing 17. Right Gripper Collision

```
collisionCheckGripper = (
        strcmp(contacts->contact(i).collision1().c_str(),
    COLLISION_ITEM) == 0) &&
    (strcmp(contacts->contact(i).collision2().c_str(),
    COLLISION_POINT_GRIP4) == 0);        // i.e. tube with gripper
if (collisionCheckGripper) {
        if (DEBUG) {std::cout << "\033[1;32mCollision between "
        << contacts->contact(i).collision1() << " and "
        << contacts->contact(i).collision2() << "\033[0m"
        << std::endl;}
        rewardHistory = REWARD_WIN;
        newReward  = true;
        endEpisode = true;
}
```

Here collision with any part of the gripper is given a very high positive reward, where collision with the arm is given a negative reward. This done to train the agent to collide with the gripper and not the arm itself.

2.4 Reward Functions

Reward functions are both positive and negative. Suitable values for base rewards that gave good results are:

Listing 18. Reward Values

```
#define REWARD_WIN  100.0f
#define REWARD_LOSS -100.0f
#define REWARD_MULTIPLIER 10.0f
\end{itemize}
```

The reward functions are as follows:

- Arm Contact Rewards
  If the goal is **gripper contact** then the reward for contacting the arm is **negative**:

  Listing 19. Negative Reward for Arm Contact

  ```
  rewardHistory = REWARD\_LOSS/10.0
  \end{itemize}
  ```

  Otherwise arm contact gives a positive reward:

  Listing 20. Positive Reward for Arm Contact

  ```
  rewardHistory = REWARD_WIN;
  newReward  = true;
  endEpisode = true;
  ```

- Objective Two Gripper Rewards
  For each of the parts of the gripper that make contact with the tube, the same positive reward is given:

  Listing 21. Positive Reward for Gripper Contact

  ```
  rewardHistory = REWARD_WIN;
  newReward  = true;
  endEpisode = true;
  ```

  Note that this is a deviation from the strict project goal, since it only included the gripper base collision. However I took the liberty of include the other parts of the gripper as any collision with any part of the gripper is a collision with the gripper. **NOTE: This change in the end was not necessary. The log shows that only the gripper_base collision occurred in the runs.**

- Timeout
  If no goal is reached within 100 iterations, a small negative reward is given:

  Listing 22. Positive Reward for Gripper Contact

  ```
  rewardHistory = REWARD_LOSS/10.0;
  newReward     = true;
  endEpisode    = true;
  ```

- Ground Contact
  If the gripper touches the ground a negative reward is given:

  Listing 23. Negative Reward for Ground Contact

  ```
  rewardHistory = REWARD_LOSS;
  newReward     = true;
  endEpisode    = true;
  ```

- Incremental Rewards For Moving Towards Goal
  Small incremental rewards are given as the arm moves away (negative) and closer to (positive) the goal:

  Listing 24. Rewards for Moving Towards Goal

  ```
  const float distDelta  = lastGoalDistance -
          distGoal;
  // 10% current dist, 90% historical average
  const float alpha = 0.1;

  // compute the smoothed moving average of the
  // delta of the distance to the goal
  avgGoalDelta  = (distDelta * alpha) +
          (avgGoalDelta * (1 - alpha));
  rewardHistory = tanh(avgGoalDelta)*REWARD_WIN*REWARD_MULTIPLIER;
  newReward     = true;
  ```

  The reward is calculated as the *tanh* of the delta distance to the goal multiplied by a constant multiplier to balance the other rewards. The *tanh* function converts the average values to numbers in the range from -1.0 to 1.0, which improves scaling.

2.5 Joint Control

Both types of joint control are implemented, and controlled by an #ifdef. It was found that *Velocity Control* gave the best results.

2.6 Hyperparameters and Tuning

The initial hyperparameters were:

Listing 25. Initial HyperParameters

```
#define INPUT_WIDTH   512
#define INPUT_HEIGHT  512
#define OPTIMIZER "None"
#define LEARNING_RATE 0.0f
#define REPLAY_MEMORY 10000
#define BATCH_SIZE 8
#define USE_LSTM false
#define LSTM_SIZE 32
```

These parameters were tuned as follows:

Listing 26. Initial Tuned HyperParameters

```
#define INPUT_WIDTH   64                  // TUNED from 512
#define INPUT_HEIGHT  64                  // TUNED from 512
#define INPUT_CHANNELS 3
#define OPTIMIZER "RMSprop"               // TUNED from "None"
#define LEARNING_RATE 0.2f                // TUNED from 0.0
#define REPLAY_MEMORY 10000
#define BATCH_SIZE 32                     // TUNED from 8
#define USE_LSTM true                     // TUNED from false
#define LSTM_SIZE 32
```

The *INPUT_HEIGHT* and *INPUT_WIDTH* were too large initially, causing poor performance. These were adjusted down to a more reasonable image size, without loss of accuracy.

The *OPTIMIZER* was changed from none to *RMSprop*. Possible common options are *Adam* or *RMSProp*, which are instances of adaptive learning rates to use in gradient-based optimization of parameters. RMSProp is the most common and gave good results.

The parameter use_LSTM was set to true to enable the LSTM agent.

The learning rate was set to 0.2 giving the best overall results and the batch size set to 32.

No other hyperparameters were changed.

3 Other Contributions

Scripts were written to aid in the training process. This launch script launches gazebo in an xterm window. This permits the ongoing process to be easily seen. It also saves a log of the results for debugging and tuning.

Listing 27. Launch Script

```
#!/bin/bash
#
# Launch gazebo-arm in an xterm window
#
xterm -e "cd ~/RoboND-DeepRL-Project/build/aarch64/bin;
         ./gazebo-arm.sh | tee ../../../gazebo-arm.log"
```

The ArmPlugin output was also enhanced with color coding and a running reward (for the incremental rewards) in the title bar:

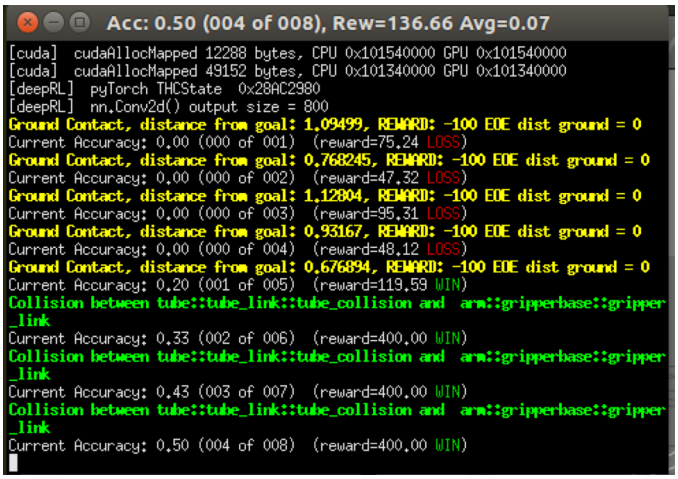

Figure 2. Xterm Window.

4 Results

The two primary objectives were achieved:

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

The first objective was relatively simple with the straightforward goal reward structure and with few iterations.

Objective one, 90% arm contact:

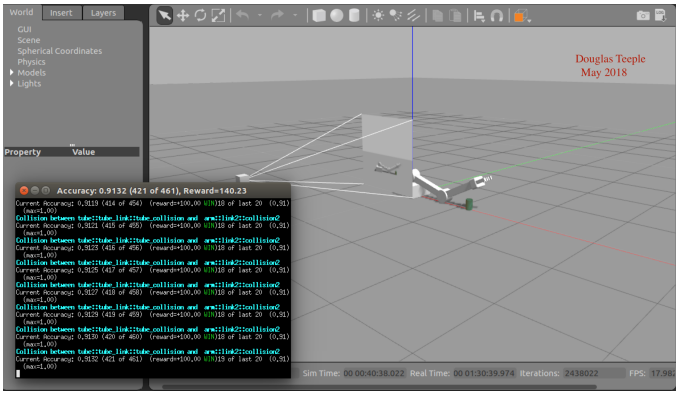

Figure 3. Arm Collisions 90% Success Achieved.

The second objective was much harder and took many more iterations to complete. It also required issuing a negative reward for contacting the arm.

Objective two, 80% gripper contact:

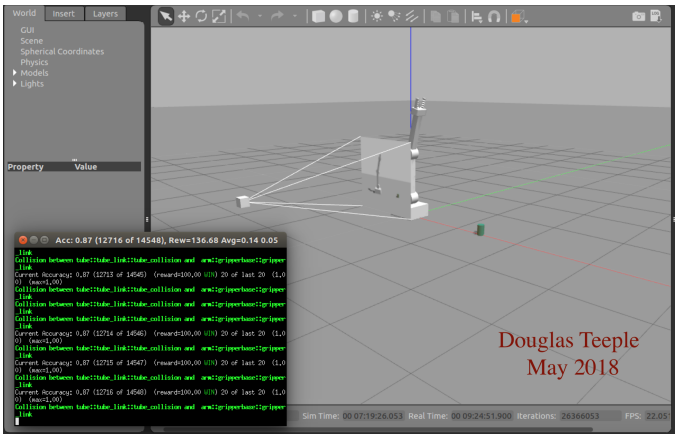

Figure 4. Gripper 87% Success Achieved.

Hyperparameter tuning was very difficult, because of the amount of randomness in reaching goals from random joint movements. It was difficult to tell if a hyperparameter change had any effect or the difference in behavior was just random. This could be remedied by using a repeatable stochastic sequence, although this was not done in this project.

5 Discussion

Training a DeepRL agent is a time consuming task, often taking many hours of computer time. Tuning parameters seems like black magic. The variability from run to run seems to far exceed the effect of any parameter changes. I once got these spectacular results:

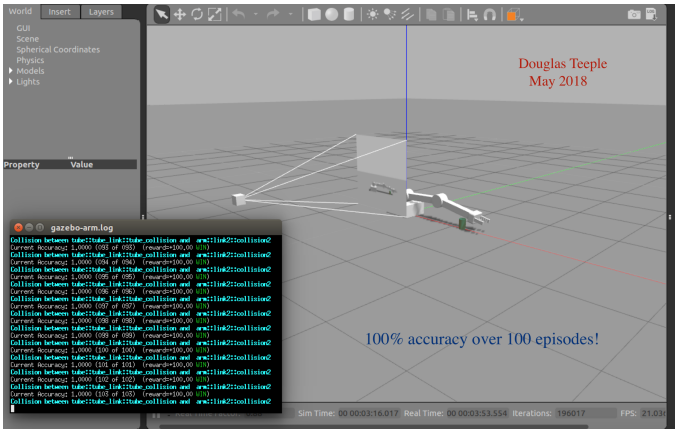

Figure 5. Arm Collisions 100% Success Achieved!

Tuning parameters such as LSTM size can give different results, some of them amusing and unexpected. In this case the agent learned that tossing and catching the cylinder in the air gave excellent results as the arm and gripper did not need to come anywhere near the ground and thus got no negative ground contact rewards:

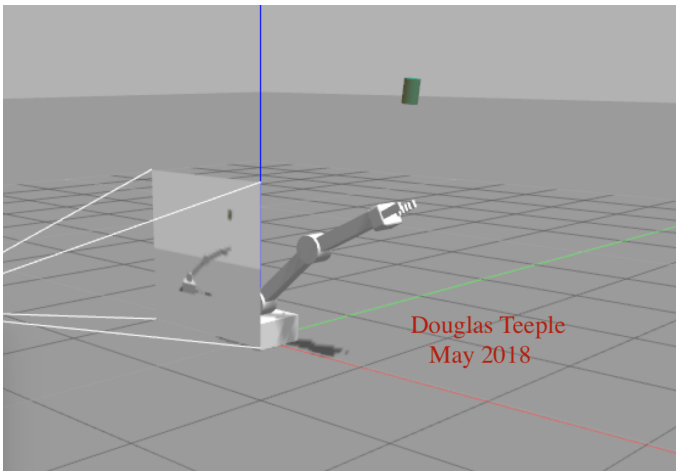

Figure 6. Playing Catch.

Note that although contact tests were implemented for other parts of the gripper, in fact, in looking at the log, no other contact than the gripper base occurred. Thus in fact the original requirement of 80% contact with the **gripper base** was met.

6 Conclusion / Future Work

The C++ API that was developed by Nvidia and optimized specifically for embedded platforms such as the TX2 performed extremely well on the native TX2 platform. The code takes advantage of the Jetson CUDA architecture and runs the episodes at a rate of about one per second.

Building the project on the Jetson platform was very simple and it performed well.

I thought about the randomness of the algorithm in this project and began to wonder if this randomness could be reduced or channeled by looking at Genetic Algorithms for Deep RL. The idea is that populations of living organism learn to adapt to their environment by a natural selection method. Perhaps something such as this could be used to give the DeepRL agent more guidance in choosing parameters to accomplish the goals. I would like to do more research in this area in the future.