Project: Deep RL Arm Manipulation - Thomas Hatting

1. Deep Reinforcement Learning in Robotics

In deep reinforcement learning (deep RL), artificial intelligent agents learn from interacting with their environment using a system of rewards. Using end-to-end neural networks raw pixels are translated into actions. The trained agents are capable of exhibiting intuitive behaviors and performing complex tasks. In many cases, the state space is complex and multi-dimensional, and it is an advantage to use neural networks, combined with GPU acceleration, to predict the best action. In deep reinforcement learning, the agents typically process 2D images and have thereby the ability to learn from vision. This is referred to as "pixels-to-actions" (Franklin 2018).

Figure 1 shows NVIDIA's API stack for deep reinforcement learning. The API provides an interface to the Python code written with PyTorch (a deep learning research platform), but the wrappers use Python's low-level C to pass memory objects between the user's application and Torch. By using a compiled language (C/C++) instead of an interpreted one (Python), performance is improved, and speeded up even further when GPU acceleration is used. This deep RL repository includes an interoperability library in C++ for integrating with Linux applications in robotics, simulation, and deployment to the field, and it also includes various examples of how to apply the API stack (Franklin 2018).

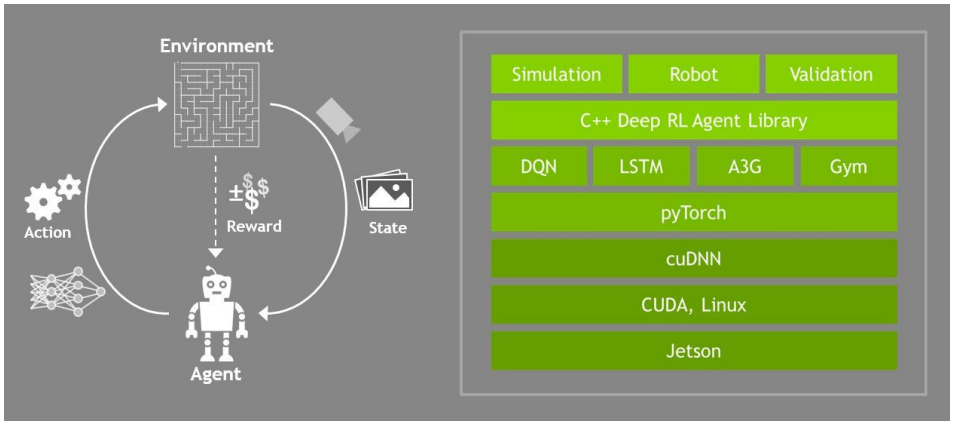

Figure 1: The API stack for deep reinforcement learning in robotics (Franklin 2018)

2. Project Environment and Gazebo Arm Plugin

The project has been implemented in Gazebo with above-mentioned C++ API. The project environment can found in file ***gazebo-arm.world*** in directory:

~/RoboND-DeepRL-Project/gazebo/

The environment uses URDF (Unified Robot Description Format), which is an XML format for representing a robot. There are three components to this file, which define the environment (Udacity 2018):

1) A robotic arm with an attached gripper
2) A camera sensor to capture images fed into the deep RL network
3) A cylindrical target object

The output in Gazebo is shown in figure 2 below.

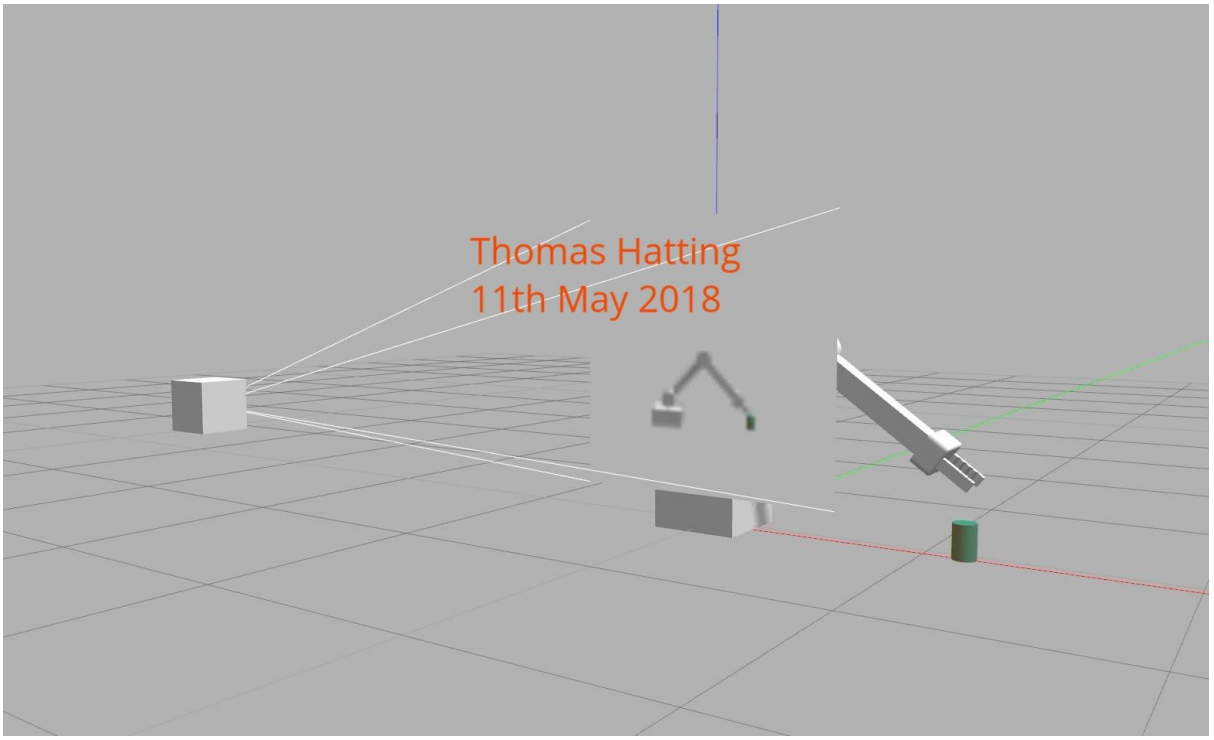

Figure 2: Project environment consisting of a robotic arm equipped with gripper, a camera sensor and a cylindrical object.

To achieve the necessary GPU acceleration, the Jetson TX2 hardware platform has been used. The Jetson TX2 accelerates deep neural network (DNN) architectures using the NVIDIA cuDNN and TensorRT libraries and provides support for recurrent neural networks (RNNs), long short-term memory networks (LSTMs) and reinforcement learning (NVIDIA 2017).

There are two objectives in this project (Udacity 2018):

1. Have any part of the robot arm touch the object with at least 90% accuracy
2. Have only the gripper base of the robot arm touch the object with at least 80% accuracy

The tasks performed to achieve this are described below.

3. Tasks Performed

In order to carry out the simulation, the tasks below have been performed. These all involve adding code to the C++ file, ArmPlugin.cpp, which is located in directory:

~/RoboND-DeepRL-Project/gazebo/

The required code changes have been determined by means of the documentation for the C++ API for deep reinforcement learning (Franklin 2018; Udacity 2018).

3.1 Subscription to Camera and Collision Topics

Subscription to the topics for camera and contact sensor for the object has been defined inside the member function ArmPlugin::Load() as follows:

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

collisionSub=collisionNode->Subscribe("/gazebo/arm_world/tube/tube_link/my_contact",&ArmPlugin::on
CollisionMsg,this);
```

where 'cameraNode' and 'collisionNode' are the subscriber nodes for camera and collision. The callback functions have been defined as pointers "&ArmPlugin::onCameraMsg" and "&ArmPlugin:onCollisionMsg" for the camera and collision nodes respectively.

3.2 Creation of DQN Agent

Inside member function ArmPlugin::createAgent() following command has been used to create a 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);
```

The various parameters passed to the Create() function are defined at the beginning of file ArmPlugin.cpp.

3.3 Position Control of Arm Joints

There are two ways to control the robot: velocity control or position control. It was decided to use positional control, as it seemed to lead to better results. It was done by setting parameter, VELOCITY_CONTROL, to 'false' at the beginning of file ArmPlugin.cpp. The joint position has been defined as follows:

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

where the joint's positional value is determined by an array 'ref' and an associated delta value 'actionJointDelta'. The length of the array 'ref' is based on the number of degrees of freedom for the robotic arm multiplied by 2, as the joint can have a positive or negative motion (the array therefore has a length of 3 x 2 = 6). The movement is determined by whether an action is even or odd. If the action is even, the joint position is increased by the delta value, and if the action is odd, the joint position is decreased by the delta value.

3.4 Rewarding Robot Gripper for Hitting the Ground

In Gazebo's API, there is a member function called GetBoundingBox() which returns the minimum and maximum values of a box defining the particular object, corresponding to the x-, y- and z-axes. This function is used to define the boolean variable 'checkGroundContact', where gripBBox is an instance of GetBoundingBox(). If this variable is 'true', the reward received by the learning agent, which is denoted by variable 'rewardHistory', is set to REWARD_LOSS thus penalizing the agent. Following commands have been included inside member function ArmPlugin::OnUpdate():

```
const float groundContact = 0.05f;
bool checkGroundContact= ( (gripBBox.min.z <= groundContact) || (gripBBox.max.z <= groundContact)
);

if(checkGroundContact)
{
        if(DEBUG){printf("GROUND CONTACT, EOE\n");}

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

3.5 Issuing an Interim Reward Based on the Distance to the Object

In file ArmPlugin.cpp, there is a function called BoxDistance() which calculates the distance between two bounding boxes. This function is used to calculate 'distGoal', which is the distance between the arm and the object. The reward received by the learning agent, which is denoted by variable 'rewardHistory', is then calculated as a smoothed moving average of the delta of the distance to the goal. Following has been inserted into member function ArmPlugin::OnUpdate():

```
const float distDelta  = lastGoalDistance - distGoal;
const float alpha = 0.5f;    //set to 0.9 for arm-to-object  & 0.5 for gripper-to-object
avgGoalDelta  = (avgGoalDelta*alpha) + (distDelta*(1.0f-alpha));
rewardHistory = avgGoalDelta;
newReward     = true;
```

For the first objective, arm touching object, parameter alpha was set to 0.9 thus putting more focus on immediate results. For the second objective, gripper base touching the object, the parameter alpha was set to 0.5 thus putting more focus on older results.

3.6 Issuing an Interim Reward Based on Collision between the Arm and the Object

The function ArmPlugin::onCollisionMsg() can be used to detect collisions. The boolean variable 'collisionCheck' has been defined inside this function to check for collisions between the arm and object. If this variable is 'true', the agent receives a positive reward. This is achieved by setting variable 'rewardHistory' to REWARD_WIN, as shown below:

```
bool collisionCheck = ( strcmp(contacts->contact(i).collision1().c_str(), COLLISION_ITEM) == 0 ) ;


if (collisionCheck)
{
        rewardHistory = REWARD_WIN;
        newReward  = true;
        endEpisode = true;

        return;
}
```

3.7 Tuning the Hyperparameters

The hyperparameters can be found at the beginning of file ArmPlugin.cpp. The parameters have been tuned with following values:

```
#define INPUT_WIDTH   64
#define INPUT_HEIGHT  64
#define OPTIMIZER "RMSprop"
#define LEARNING_RATE 0.1f  //set to 0.01 for arm-to-object  & 0.1 for gripper-to-object
#define REPLAY_MEMORY 10000
#define BATCH_SIZE 32
#define USE_LSTM true
#define LSTM_SIZE 256

#define REWARD_WIN  500.0f
#define REWARD_LOSS -500.0f
```

The input width and input height have been reduced from 512 to 64 in order to limit the memory and calculation requirements. The optimizer has been set to "RMSprop". The batch size has been increased from 8 to 32 to allow for larger batches. The Long-Term Short Memory (LSTM), which enables the system to learn from long-term dependencies, has been activated by setting the value 'USE_LSTM' to 'true' and has been given a size of 256. Furthermore, the parameter for positive reward, REWARD_WIN, has been given a value of +500, and the parameter for negative reward, REWARD_LOSS, has been given a value of -500.

The hyperparameters were the same for both objectives except for the learning rate. For the first objective, arm touching object, it was initially set to 0.1 but then reduced to 0.01. Reducing the learning rate appeared to increase the duration of the learning but also appeared to improve the model's accuracy in the long run. For the second objective, gripper base touching the object, the learning rate was kept at 0.1.

3.8 Issuing a Reward Based on Collision between the Arm's Gripper Base and Object

Later in the project, the boolean variable 'collisionCheck', which was located inside member function ArmPlugin::onCollisionMsg(), was redefined to account for collisions between the arm's gripper base and the object, as below:

```
bool collisionCheck = ( strcmp(contacts->contact(i).collision2().c_str(), COLLISION_POINT) == 0 );

if (collisionCheck)
{
        rewardHistory = REWARD_WIN;
        newReward  = true;
        endEpisode = true;

        return;
}
```

4. Results of the Simulations

All simulations were carried out on the Jetson TX2 hardware platform in order to achieve the necessary GPU acceleration. After the code changes were made to file ArmPlugin.cpp, as described in the previous section, the 'make' command was executed in directory **~/RoboND-DeepRL-Project/build/** in order to generate the model:

```
nvidia@tegra-ubuntu:~/RoboND-DeepRL-Project/build$ make
[ 38%] Built target jetson-utils
[ 50%] Built target jetson-reinforcement
Scanning dependencies of target gazeboArmPlugin
[ 52%] Building CXX object gazebo/CMakeFiles/gazeboArmPlugin.dir/ArmPlugin.cpp.o
[ 54%] Linking CXX shared library ../aarch64/lib/libgazeboArmPlugin.so
[ 54%] Built target gazeboArmPlugin
[ 58%] Built target gazeboPropPlugin
[ 64%] Built target gazeboGraspPlugin
[ 68%] Built target catch
[ 74%] Built target fruit
[ 80%] Built target deepRL-console
[ 84%] Built target deepRL-input
[ 88%] Built target gst-camera
[ 92%] Built target v4l2-console
[ 96%] Built target v4l2-display
[100%] Built target gl-display-test
nvidia@tegra-ubuntu:~/RoboND-DeepRL-Project/build$ cd aarch64/
```

To launch the simulation the Bash file, gazebo-arm.sh, was then executed in directory **~/RoboND-DeepRL-Project/build/aarch64/bin/**, as below:

```
nvidia@tegra-ubuntu:~/RoboND-DeepRL-Project/build/aarch64/bin$ ./gazebo-arm.sh
--
--
starting gazebo7 simulator
Gazebo multi-robot simulator, version 7.0.0
Copyright (C) 2012-2016 Open Source Robotics Foundation.
Released under the Apache 2 License.
http://gazebosim.org
--
--
ArmPlugin::ArmPlugin()
ArmPlugin::Load('arm')
PropPlugin::Load('tube')
```

```
[Err] [Scene.cc:2927] Light [sun] not found. Use topic ~/factory/light to spawn a new light.
[deepRL]  use_cuda:       True
[deepRL]  use_lstm:       1
[deepRL]  lstm_size:      256
[deepRL]  input_width:    64
[deepRL]  input_height:   64
[deepRL]  input_channels: 3
[deepRL]  num_actions:    6
[deepRL]  optimizer:      RMSprop
[deepRL]  learning rate:  0.1
[deepRL]  replay_memory:  10000
[deepRL]  batch_size:     32
[deepRL]  gamma:          0.9
[deepRL]  epsilon_start:  0.9
[deepRL]  epsilon_end:    0.05
[deepRL]  epsilon_decay:  200.0
[deepRL]  allow_random:   1
[deepRL]  debug_mode:     0
[deepRL]  creating DQN model instance
[deepRL]  DRQN::__init__()
[deepRL]  LSTM (hx, cx) size = 256
[deepRL]  DQN model instance created
[deepRL]  DQN script done init
[cuda]  cudaAllocMapped 49152 bytes, CPU 0x102a00000 GPU 0x102a00000
[deepRL]  pyTorch THCState  0x44F4E9E0
[cuda]  cudaAllocMapped 12288 bytes, CPU 0x102c00000 GPU 0x102c00000
ArmPlugin - allocated camera img buffer 64x64  24 bpp  12288 bytes
[deepRL]  nn.Conv2d() output size = 800
Current Accuracy:  0.0000 (000 of 001)  (reward=-500.00 LOSS)
Current Accuracy:  0.5000 (001 of 002)  (reward=+500.00 WIN)
--
--
```

The results of the simulations are described below.

The simulation carried out for the first scenario with the arm touching the object is shown in *figure 3*. The learning rate was initially set to 0.1 but then reduced to 0.01 to achieve a higher accuracy. Also, the parameter alpha, mentioned earlier for the smoothed moving average, was set to 0.9. The accuracy reached a value higher than 90%.

The simulation carried out for the second scenario with gripper base touching the object is shown in *figure 4*. The learning rate was set to 0.1. Also, the parameter alpha, mentioned earlier, was reduced from 0.9 to 0.5 to put more emphasis on previous data. The accuracy reached a value higher than 80%.

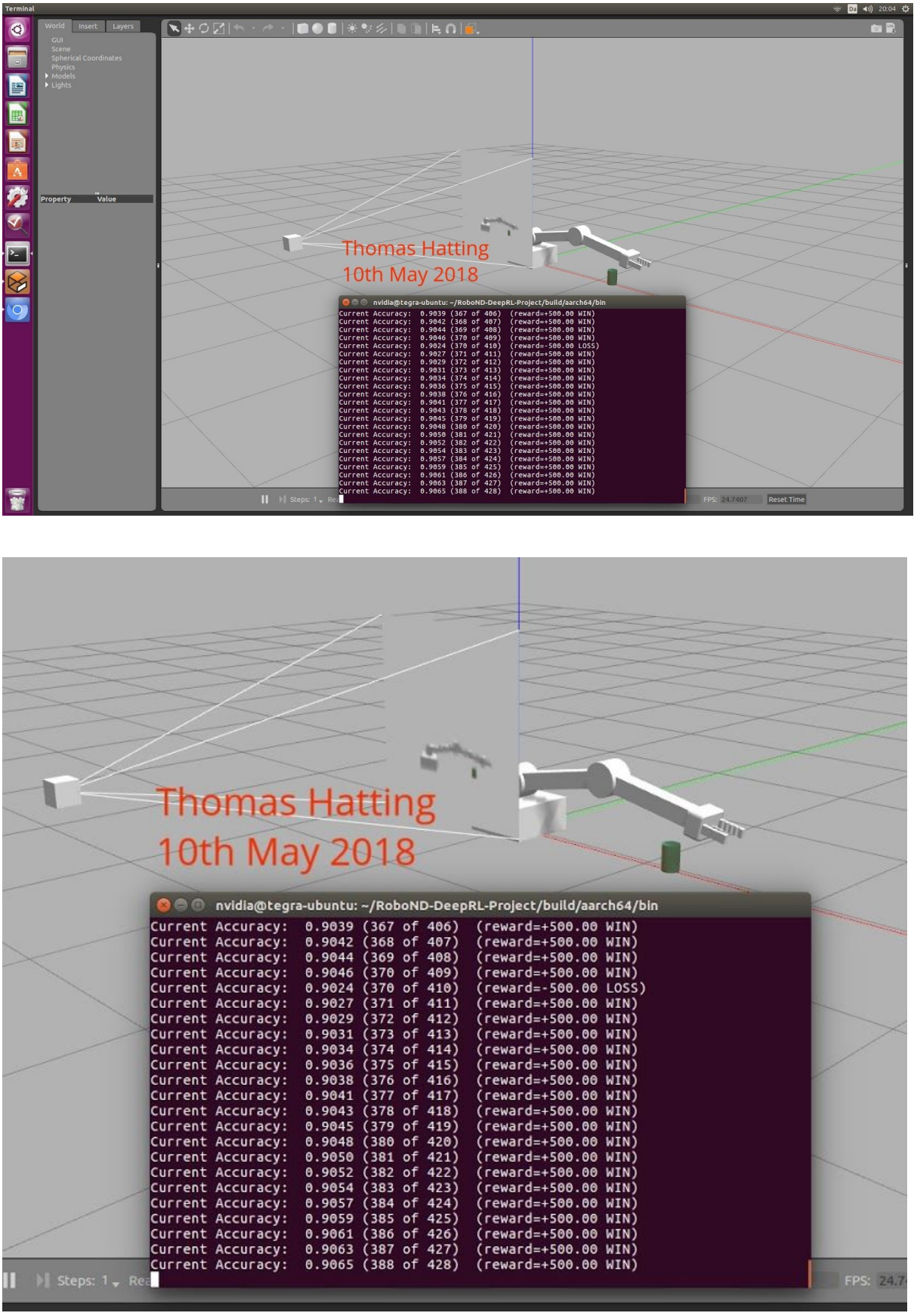

Figure 3: Screenshot and close-up of simulation of the robot arm (upper-section) touching the object. The accuracy reached a value higher than 90%. The simulation was carried out on the Jetson TX2.

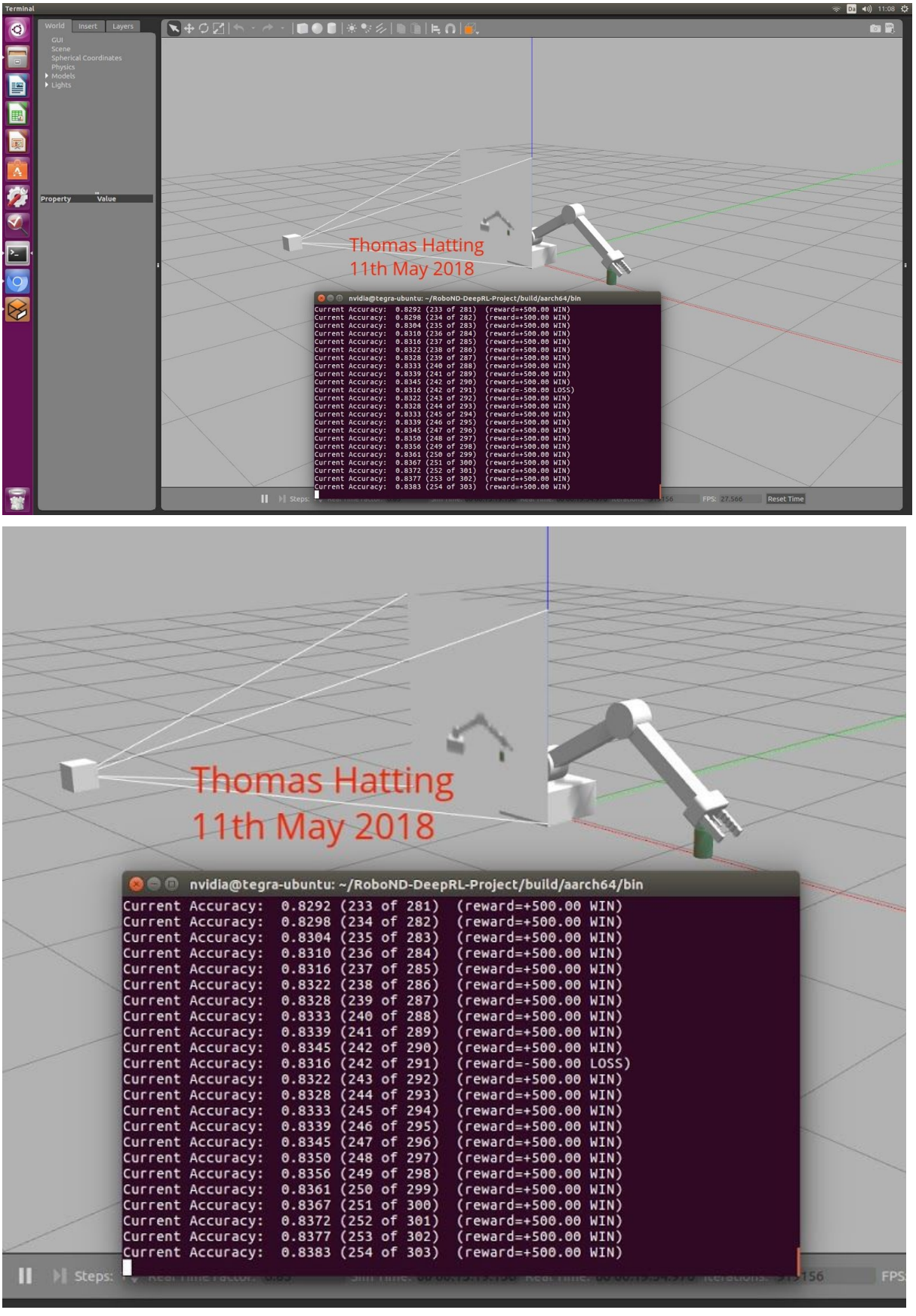

Figure 4: Screenshot and close-up for simulation for gripper base touching the object. The accuracy reached a value higher than 80%. The simulation was carried out on the Jetson TX2.

5. Conclusion and Future Work

This project looks at deep reinforcement learning in robotics using a C++ API that integrates linux applications in robotics with the PyTorch package for deep learning. The project environment consisted of a robotic arm with gripper, a camera sensor and a cylindrical object. The environment was implemented in Gazebo. To achieve the necessary GPU acceleration, the project was built on the Jetson TX2 hardware platform.

The required accuracy was achieved for all objectives (see figures 3 & 4), however there were a number of issues. The simulations had to be run a couple times to achieve these results. If the robot arm got off to a "bad start" and had a large number of random misses at the beginning of the simulation, the learning agent had difficulties optimizing the model to reach the necessary accuracy. However, after running the simulation a few times, the learning agent managed to achieve the required accuracy. So although the reward function worked, it could be further improved to achieve a more consistent learning process.

Future work could include defining a more complex reward function to improve the reinforcement learning process, possibly based on research papers. Also, the work carried out in this project could be extended to a hardware project involving a real robotic arm, controlled by the Jetson TX2 platform. Also, it would be interesting to look at some of the other examples provided in the C++ API. In particular, the library contains a scenario for navigating a rover in Gazebo and another scenario for a bipedal walker (continuous learner) that can be implemented in the OpenAI Gym environment.

References

Franklin, D. (2018) "Deep Reinforcement Learning in Robotics". Available online: https://github.com/dusty-nv/jetson-reinforcement

NVIDIA (2017) "NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge". Available online: https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/

Udacity (2018) , Classroom Material, Term 2, Sections 20-25 + P, Robotics Software Engineer Nanodegree Program, May 2018.