UDACITY RoboND

Deep RL Arm Manipulation

Requirements

1.Create a DQN agent and define reward functions to teach a robotic arm
2.Have the arm perform two primary objectives:

- Have any part of the robot arm touch the object of interest, with at least a 90% accuracy.
- Have only the gripper base of the robot arm touch the object, with at least an 80% accuracy.

Reward functions

Reward Types

Based on the goals of the project, there should be three different kinds of rewards:

1. Positive rewards for wins:
   "Win" is defined differently under two different scenarios, namely, either "any part" of the arm touching the tube, or more specifically, only the middle link of the gripper touching the tube.
2. Negative rewards for losses:
   "Loss" is defined by such a situation when any part the arm touches the ground, or there are over 100 episode frames, which is considered timeout.
3. Interim rewards during arm movements:
   Except for the above two ending rewards, an interim reward must be defined to encourage the arm to try to decrease the distance between the tube and itself.

In the above definition, a very subtle but vital point that demands attention is that the touching between the ground plane and the tube does not count and needs to be filtered out from the messages. The training was unsuccessful in the beginning because of this very reason.

Reward Values

By convention, the winning reward is defined as 1 while the losing reward is defined as -1. The interim reward is defined as the smoothed moving average of the delta of the distance from the tube to the target part of the arm, whether it is any part or just the gripper base.

Control Type

Position control is used in this project. For each joint, there are 3 actions to take, and the value is defined by the variable actionJointDelta. Here the default value is not changed.

Hyperparameters

As seen in the code, these are the hyperparameters after tweaking:

```
#define INPUT_WIDTH 64
#define INPUT_HEIGHT 64
#define OPTIMIZER "RMSprop"
#define LEARNING_RATE 0.005f
#define REPLAY_MEMORY 20000
#define BATCH_SIZE 64
#define USE_LSTM true
#define LSTM_SIZE 128
```

Compare the values with the provided ones:

```
#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
```

For a starter, I choose to use RMSprop as the optimizer and a learning rate of 0.01f, following the "catch" example. I first set the arm up to touch the tube with "any part" of it. After making all the necessary coding, I found my arm is not learning, and the response is very slow, and sooner or later I would get a "failed to alloc cuda buffer" error.

The parameters need to be tweaked. First of all, the input was 512 x 512, which is comparatively large. The printed information in the terminal reveals that the images taken by the camera are way smaller, 64 x 64 to be precise. All the extra sizes were wasting the processing power. So the input size was changed to be the same 64 x 64. Furthermore, the REPLAY_MEMORY was doubled to

20000, and the BATCH_SIZE was increased to 64, to make the arm learn from more historical data.

The arm starts to learn a bit, appears to be leaning towards the object, sometimes very close. However, the process seems to be random. The training process was repeated many times, and the performance is volatile. Sometimes the arm can touch the tube on the first try, while other times it would just shake left and right without even trying to get closer.

It seems like the arm is not learning well from its past success. To address this, LSTM was enabled, and the cell size increased to 128.

Finally, the arm appears to be learning normally. A problem that was still haunting was that it could not keep the accuracy. It would keep touching the tube if it succeeded for one time, but if it ever decided to move away, it would keep doing that too. It seems like the interim reward is not tuned correctly, and the arm was relying a little too much on the historical data than on the current one.

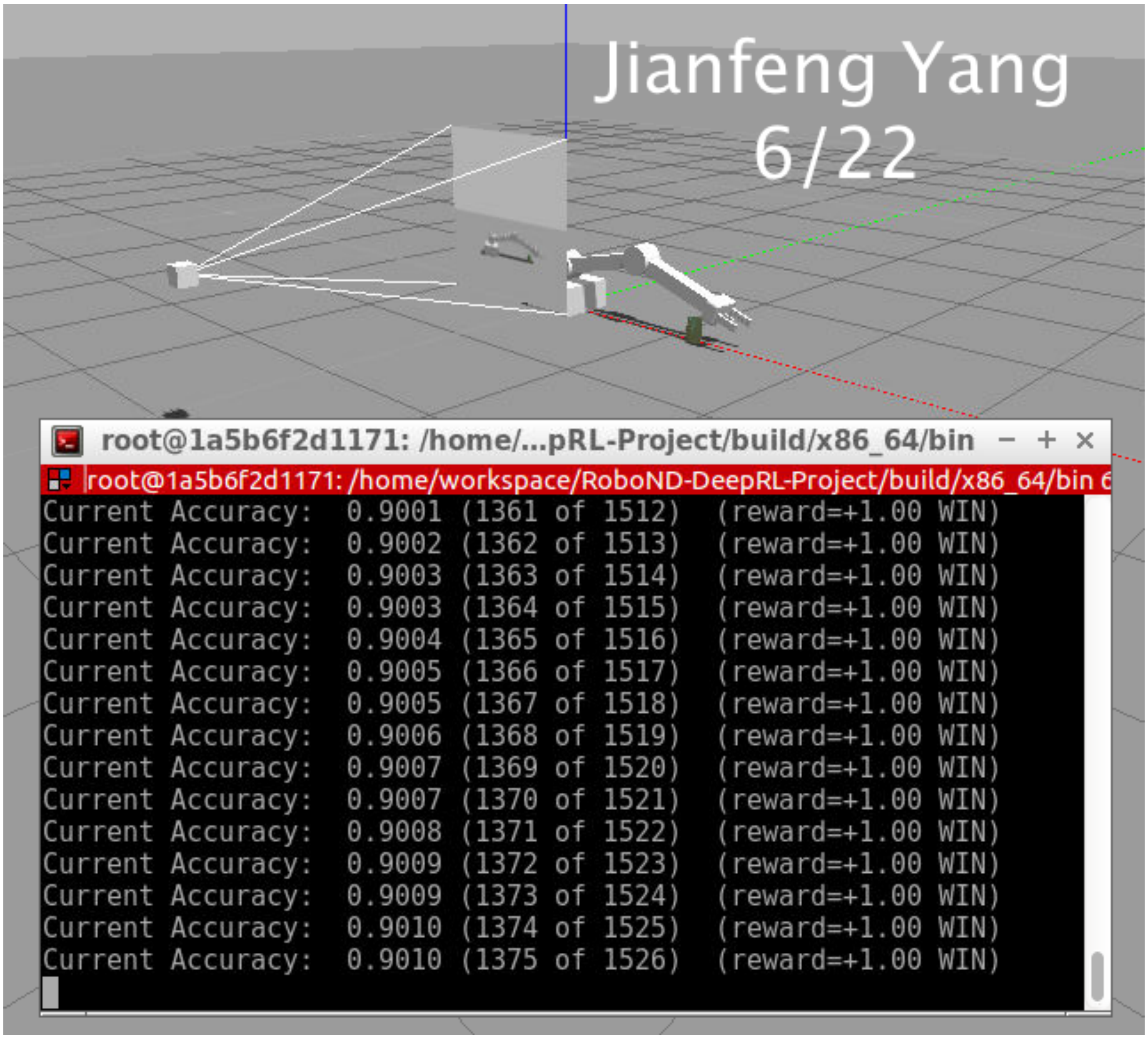
Thus, the alpha was lowered, and it did help in stabilizing the accuracy. A few different numbers were tried, and the learning rate was also lowered to 0.001, and finally, the accuracy reached 90%.

For the second objective, no significant changes were made. The only thing I noticed at the end of the previous training is that the training time is very long: needs to run over 1500 times to reach the 90% accuracy.

Considering that the second objective should be harder, it may take even longer time. So the learning rate was increased to 0.005.

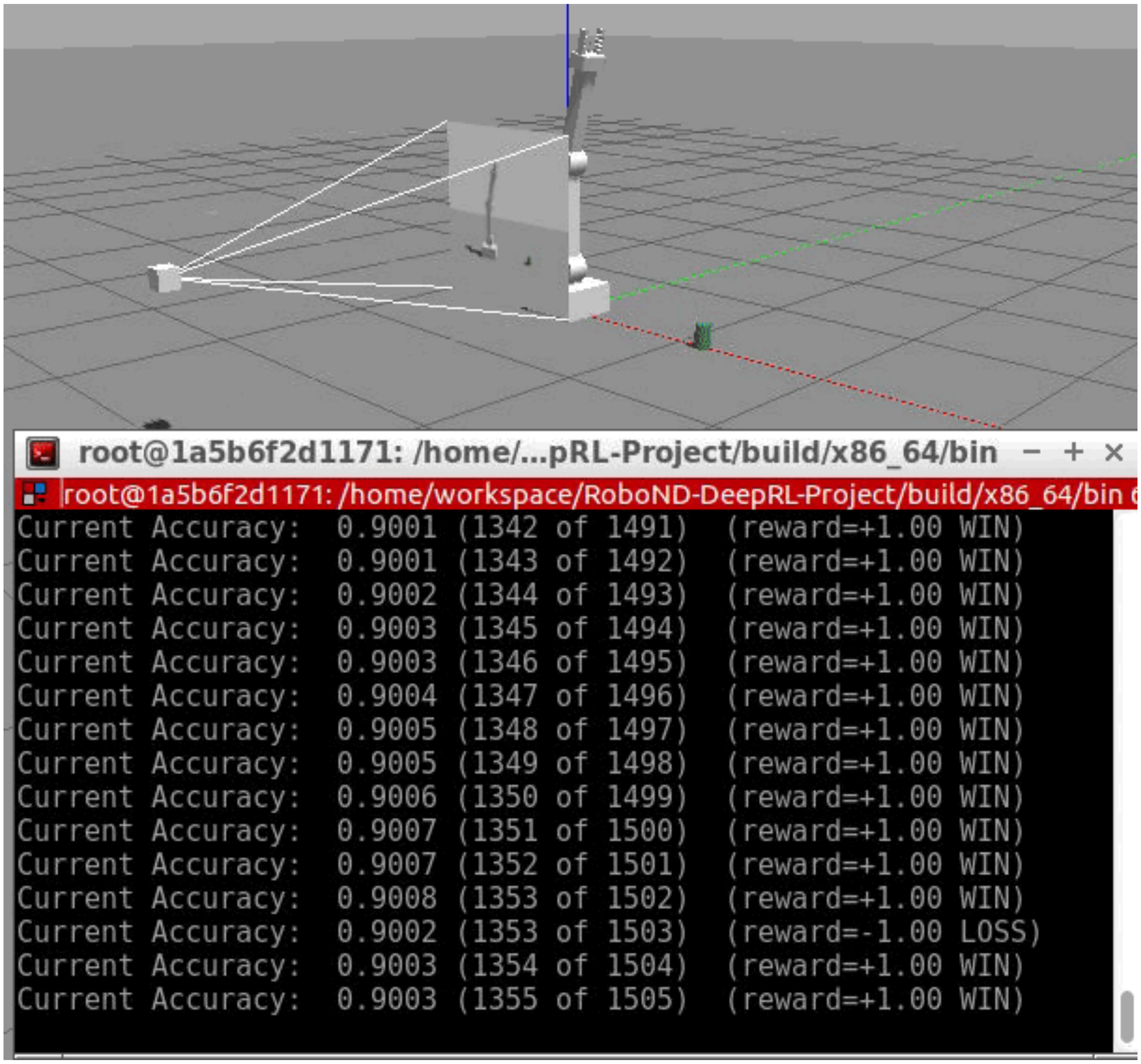
Results

Here is the screenshot of the first objective:



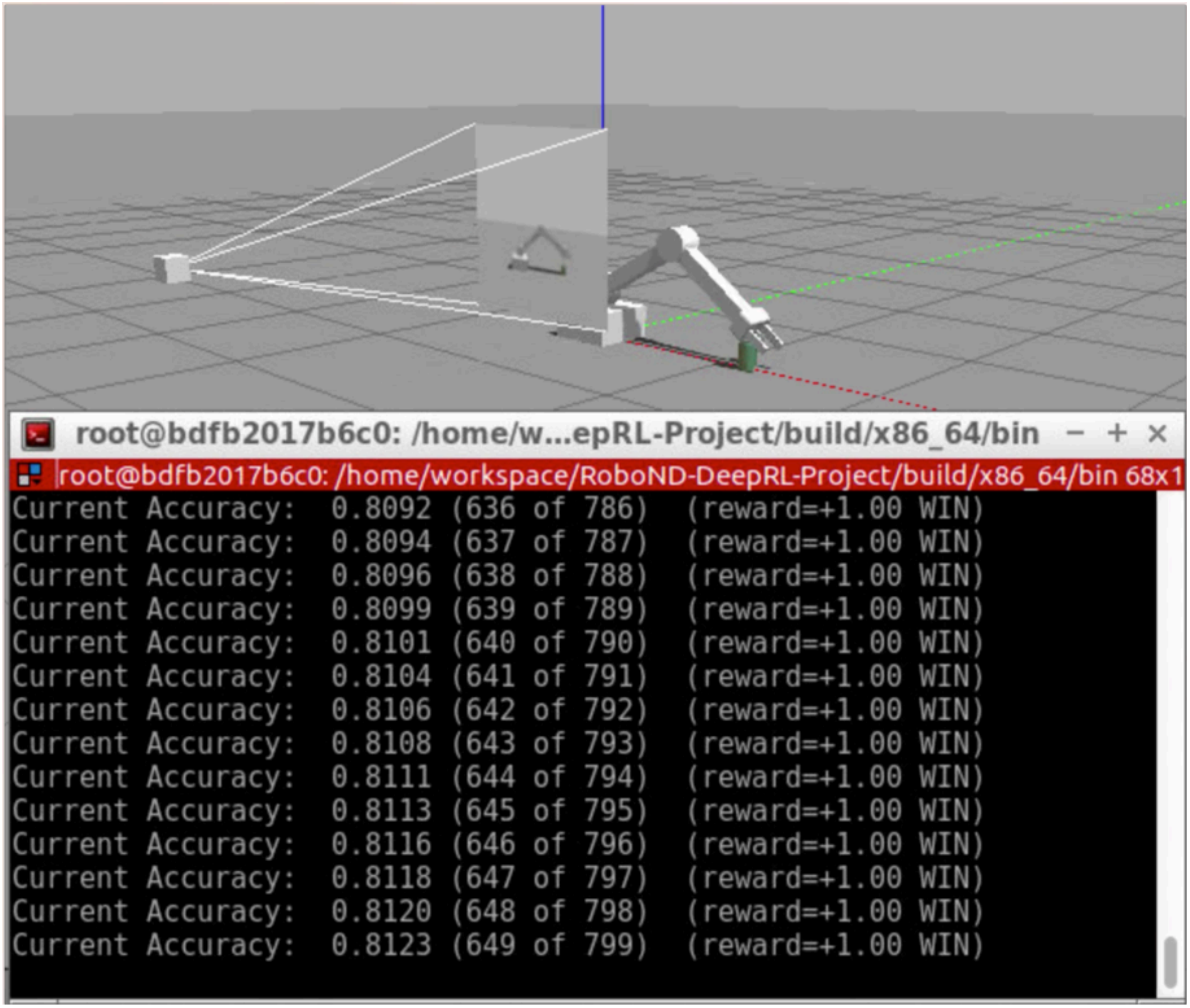
```
root@1a5b6f2d1171: /home/...pRL-Project/build/x86_64/bin
root@1a5b6f2d1171: /home/workspace/RoboND-DeepRL-Project/build/x86_64/bin
Current Accuracy:  0.9001 (1361 of 1512)  (reward=+1.00 WIN)
Current Accuracy:  0.9002 (1362 of 1513)  (reward=+1.00 WIN)
Current Accuracy:  0.9003 (1363 of 1514)  (reward=+1.00 WIN)
Current Accuracy:  0.9003 (1364 of 1515)  (reward=+1.00 WIN)
Current Accuracy:  0.9004 (1365 of 1516)  (reward=+1.00 WIN)
Current Accuracy:  0.9005 (1366 of 1517)  (reward=+1.00 WIN)
Current Accuracy:  0.9005 (1367 of 1518)  (reward=+1.00 WIN)
Current Accuracy:  0.9006 (1368 of 1519)  (reward=+1.00 WIN)
Current Accuracy:  0.9007 (1369 of 1520)  (reward=+1.00 WIN)
Current Accuracy:  0.9007 (1370 of 1521)  (reward=+1.00 WIN)
Current Accuracy:  0.9008 (1371 of 1522)  (reward=+1.00 WIN)
Current Accuracy:  0.9009 (1372 of 1523)  (reward=+1.00 WIN)
Current Accuracy:  0.9009 (1373 of 1524)  (reward=+1.00 WIN)
Current Accuracy:  0.9010 (1374 of 1525)  (reward=+1.00 WIN)
Current Accuracy:  0.9010 (1375 of 1526)  (reward=+1.00 WIN)
```

Along with a GIF:


root@1a5b6f2d1171: /home/...pRL-Project/build/x86_64/bin
root@1a5b6f2d1171: /home/workspace/RoboND-DeepRL-Project/build/x86_64/bin

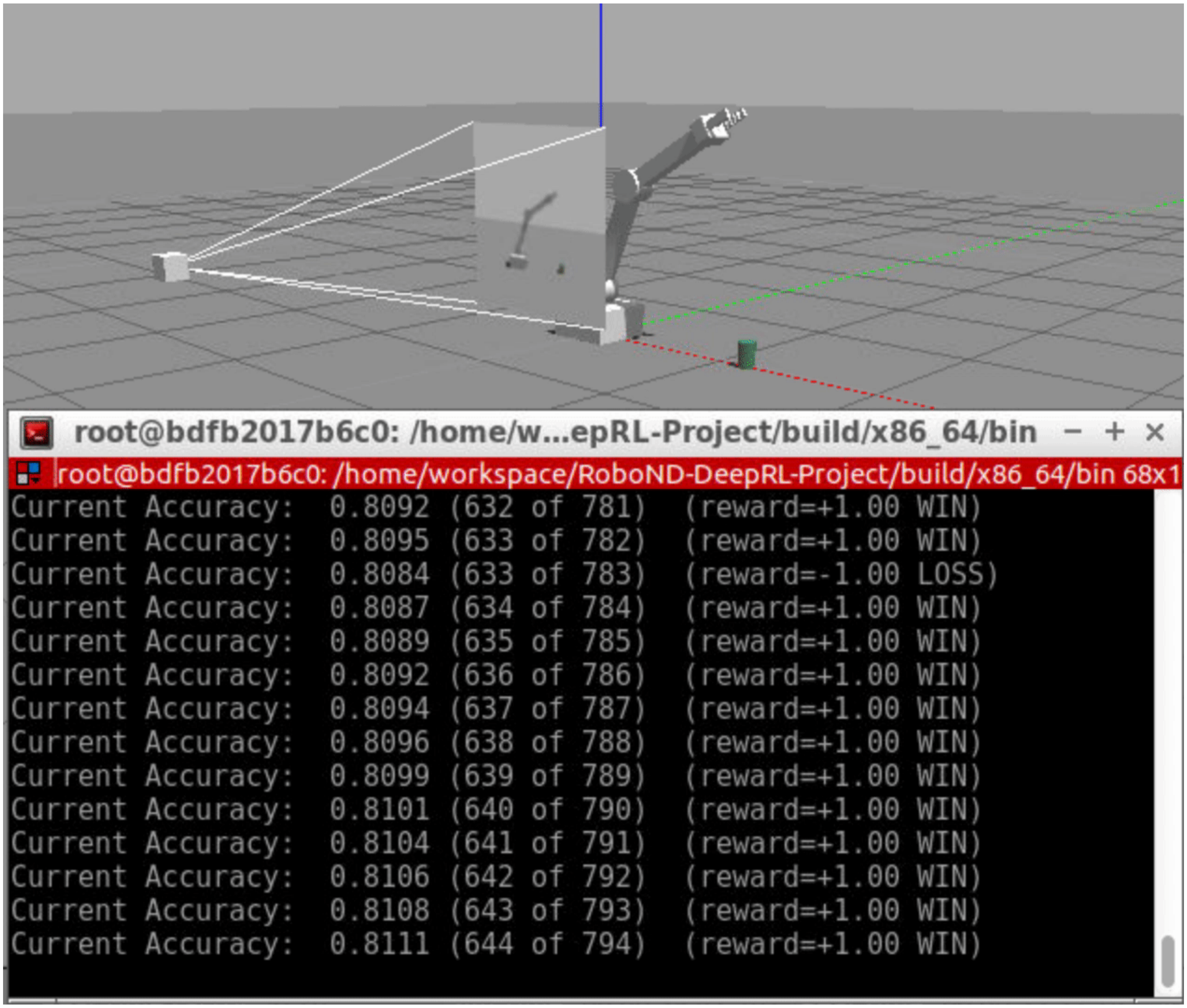
```
Current Accuracy:  0.9001 (1342 of 1491)  (reward=+1.00 WIN)
Current Accuracy:  0.9001 (1343 of 1492)  (reward=+1.00 WIN)
Current Accuracy:  0.9002 (1344 of 1493)  (reward=+1.00 WIN)
Current Accuracy:  0.9003 (1345 of 1494)  (reward=+1.00 WIN)
Current Accuracy:  0.9003 (1346 of 1495)  (reward=+1.00 WIN)
Current Accuracy:  0.9004 (1347 of 1496)  (reward=+1.00 WIN)
Current Accuracy:  0.9005 (1348 of 1497)  (reward=+1.00 WIN)
Current Accuracy:  0.9005 (1349 of 1498)  (reward=+1.00 WIN)
Current Accuracy:  0.9006 (1350 of 1499)  (reward=+1.00 WIN)
Current Accuracy:  0.9007 (1351 of 1500)  (reward=+1.00 WIN)
Current Accuracy:  0.9007 (1352 of 1501)  (reward=+1.00 WIN)
Current Accuracy:  0.9008 (1353 of 1502)  (reward=+1.00 WIN)
Current Accuracy:  0.9002 (1353 of 1503)  (reward=-1.00 LOSS)
Current Accuracy:  0.9003 (1354 of 1504)  (reward=+1.00 WIN)
Current Accuracy:  0.9003 (1355 of 1505)  (reward=+1.00 WIN)
```

And the result for the second objective is shown below:


root@bdfb2017b6c0: /home/w...epRL-Project/build/x86_64/bin
root@bdfb2017b6c0: /home/workspace/RoboND-DeepRL-Project/build/x86_64/bin 68x1

```
Current Accuracy:  0.8092 (636 of 786)  (reward=+1.00 WIN)
Current Accuracy:  0.8094 (637 of 787)  (reward=+1.00 WIN)
Current Accuracy:  0.8096 (638 of 788)  (reward=+1.00 WIN)
Current Accuracy:  0.8099 (639 of 789)  (reward=+1.00 WIN)
Current Accuracy:  0.8101 (640 of 790)  (reward=+1.00 WIN)
Current Accuracy:  0.8104 (641 of 791)  (reward=+1.00 WIN)
Current Accuracy:  0.8106 (642 of 792)  (reward=+1.00 WIN)
Current Accuracy:  0.8108 (643 of 793)  (reward=+1.00 WIN)
Current Accuracy:  0.8111 (644 of 794)  (reward=+1.00 WIN)
Current Accuracy:  0.8113 (645 of 795)  (reward=+1.00 WIN)
Current Accuracy:  0.8116 (646 of 796)  (reward=+1.00 WIN)
Current Accuracy:  0.8118 (647 of 797)  (reward=+1.00 WIN)
Current Accuracy:  0.8120 (648 of 798)  (reward=+1.00 WIN)
Current Accuracy:  0.8123 (649 of 799)  (reward=+1.00 WIN)
```

Also with a GIF to show it dynamically:


root@bdfb2017b6c0: /home/w...epRL-Project/build/x86_64/bin
root@bdfb2017b6c0: /home/workspace/RoboND-DeepRL-Project/build/x86_64/bin 68x1

```
Current Accuracy:  0.8092 (632 of 781)  (reward=+1.00 WIN)
Current Accuracy:  0.8095 (633 of 782)  (reward=+1.00 WIN)
Current Accuracy:  0.8084 (633 of 783)  (reward=-1.00 LOSS)
Current Accuracy:  0.8087 (634 of 784)  (reward=+1.00 WIN)
Current Accuracy:  0.8089 (635 of 785)  (reward=+1.00 WIN)
Current Accuracy:  0.8092 (636 of 786)  (reward=+1.00 WIN)
Current Accuracy:  0.8094 (637 of 787)  (reward=+1.00 WIN)
Current Accuracy:  0.8096 (638 of 788)  (reward=+1.00 WIN)
Current Accuracy:  0.8099 (639 of 789)  (reward=+1.00 WIN)
Current Accuracy:  0.8101 (640 of 790)  (reward=+1.00 WIN)
Current Accuracy:  0.8104 (641 of 791)  (reward=+1.00 WIN)
Current Accuracy:  0.8106 (642 of 792)  (reward=+1.00 WIN)
Current Accuracy:  0.8108 (643 of 793)  (reward=+1.00 WIN)
Current Accuracy:  0.8111 (644 of 794)  (reward=+1.00 WIN)
```

Generally speaking, the DQN agent is performing well. As seen from the screenshot, the second training was better than the first one, reaching 80% accuracy in about half the time.

Apart from several coding errors that are undoubtedly due to the unfamiliarity of the library, a few obstacles have severely slowed down the training. First of all, when checking the contact, I was over thinking the problem and checking all the contact messages. Instead, I should be checking only the ones containing the tube. This modification has dramatically speeded things up.

Secondly, when setting up the interim reward, I misunderstood the meaning of the formula and falsely assume that the reward and the smoothed average should be negatively related. The result was that the arm would learn, but with the wrong objective as to stay away as far as possible from the tube.

Improvements

The above is just a basic implementation to reach the goal, and there are numerous ways to improve it. Name a few:

1. Choose a different optimizer. RMSprop was the only optimizer that was used, and there might be other ones that are better suited for the task.
2. Increase REPLAY_MEMORY and BATCH_SIZE further.
3. Investigate and develop a better formula for the interim reward. Primarily, the alpha is a fixed number during each training. The training should converge even quicker if this alpha can be changed dynamically to reward a faster success than a slower one.
4. Perform the training on a local machine with lower learning rate and for a more extended period.