Deep RL Arm Manipulation

Abstract

The project is train a robot arm to reach goal based on a sample re-enforcement project from Nvidia by Deep Reinforcement Learning. After tuning rewards and hyper-parameters, accuracy achieved 98% in task 1 and 92% in task 2.

Background

Reinforcement Learning is to use rewards and goals to train agent to learn from experiment.

With Deep Reinforcement Learning, an agent is processing 2D images with Convolutional Neural Networks (CNN) with the ability to learn from vision with the end-to-end network (pixels to actions).

The Nvidia sample includes Deep Q-Learning (DQN), Long Short-Term Memory Units (LSTMs), and continuous A3G algorithms in PyTorch with Gazebo simulation in Linux.

There are two tasks to be performed for the project.

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.

The project has been checked in https://github.com/TianHuaBooks/DeepRL. Under the Udacity Workspace environment (/home/workspace), the project can be cloned from GitHub.

```
$ git clone https://github.com/TianHuaBooks/DeepRL
$ cd DeepRL
```

By creating a build directory and change to the build directory, the project can be built by command make.

```
$ mkdir build
$ cd build
$ make
$ cd x86_64/bin
```

Then the Gazebo simulation can be launched with gazebo-arm.sh. Due to the world file is in build/x86_64/bin, we need to execute the shell script (gazebo-arm.sh) in the directory.

```
$ ./gazebo-arm.sh
```

Approach

First, we need to subscribe camera image from the camera node and collision message from the collision node. ArmPlugin::onCameraMsg and ArmPlugin::onCollisionMsg routines were provided by the Nvidia sample

Then we need to create an Deep Q-Learning (DQN) agent with parameters.

- input of image dimension, 64x64
- input channel, 3
- number of action, Degree of Freedom (DOF) * 2 = 6
- optimizer, RMSprop (adaptive learning rate method proposed by Geoff Hinton)
- learning rate, 0.01
- allow random, true

Long Short-Term Memory Units (LSTMs)

- Replay memory of 10000
- Batch size 128
- Use_LSTM, true
- LSTM_SIZE, 256

Width and height has been set to the same as image dimension. Each image has three channels. Number of action is set to degree of freedom times two.

RMSprop optimization was used (default in dqnAgent) with learning rate 0.01 and batch size 128 after tries and errors. During test, RMSprop performed better than Adam optimization. It's discussed in detail with test results later in the result session.

LSTM size higher than 256 doesn't help performance or accuracy. For task 1, it can achieve 80% by lower LSTM memory and size. As we need to perform both task 1 and task 2, a set of parameters is used with an additional parameter TASK1 to tell which task to run.

In processing collision message, it goes through the contacts pointer array. Each contact object in the array has two collision contacts. We filter out ground plane contact messages.

If ether contact 1 or 2 is from collision item tube, we look into the other contact point. For task 1, as long as any part of the robot arm is contacted, we consider it good and grant a winning award. For task 2, it needs to be among gripper link, right gripper, left gripper, or the middle collision of the gripper base.

Two utility functions has been written. SameAsPattern compares string from contact pointer string with a contact string pattern. HasGripperHit checks wether there is a match of gripper collision by using the sameAsPattern function.

The wining award of right collision is summed up by two parted, REWARD_WIN * REWARD_MULT and (maxEpisodeLength - episodeFrames) * REWARD_FRAMES.

The base REWARD_WIN is defined to be one. REWARD_MULT is defined to be a factor of 100 to distinguish the winning award from interim rewards. The second part is to reward efficiency of goal achieving. MaxEpisodeLength is defined to 100 in the program and episodeFrames is the current frame index counted from zero. REWARD_FRAMES is a factor of 10. When the robot arm hits the goal in an earlier frame, it will receive a higher reward.

OnUpdate processing is another important routine to decide reward. First, it updates joint positions of the robot arm. If it uses more MaxEpisodeLength(100) frames without hitting the goal, a base

REWARD_LOSS (-1) will be returned as a loss. If gripper position is too low with a danger to hit the ground, it'll be rewarded with REWARD_LOSS * REWARD_GRND. REWARD_GRND is defined as 10. As hitting ground might damage the robot arm, it's given a larger loss penalty than timeout.

For interim rewards, distance between gripper and the target is calculated to factor it in. To smooth it, moving average with historical distances is taken into consideration. Distance Delta is subtracted from the prior distance delta to tell whether robot arm is moving closer to the target. The new average goal delta is calculated by the alpha factor times the prior average goal delta plus delta distance times one minus of alpha.

```
(avgGoalDelta * ALPHA) + distDelta * (1.0f - ALPHA);
```

Alpha is chosen to 0.5 to balance effects from historical value and the current value.

If delta distance is greater than zero, it means gripper is getting closer to that goal. The reward is calculated by REWARD_WIN * distDelta. The more distance it move closer to the goal, the higher reward it gets.
If delta distance is less than zero, a reward loss of REWARD_LOSS * distGoal is used to penalize moving away from the goal.

When delta distance is zero, a reward of zero is given to the system for transition.

Results

Fig. 1 shows test results of task 1 with 98% accuracy. Fig. 2 shows test results of task 2 with 92% accuracy.

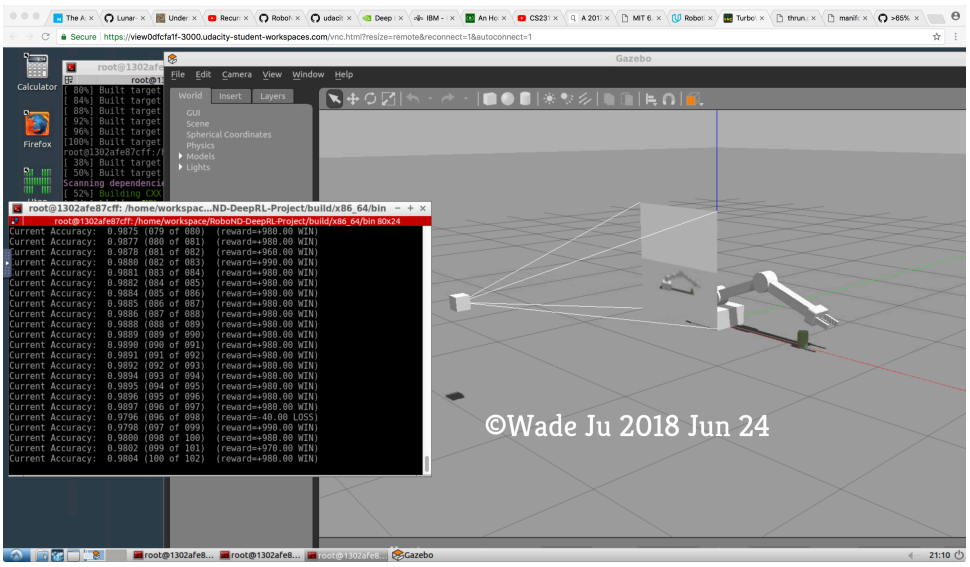

Fig. 1 Results of Task 1

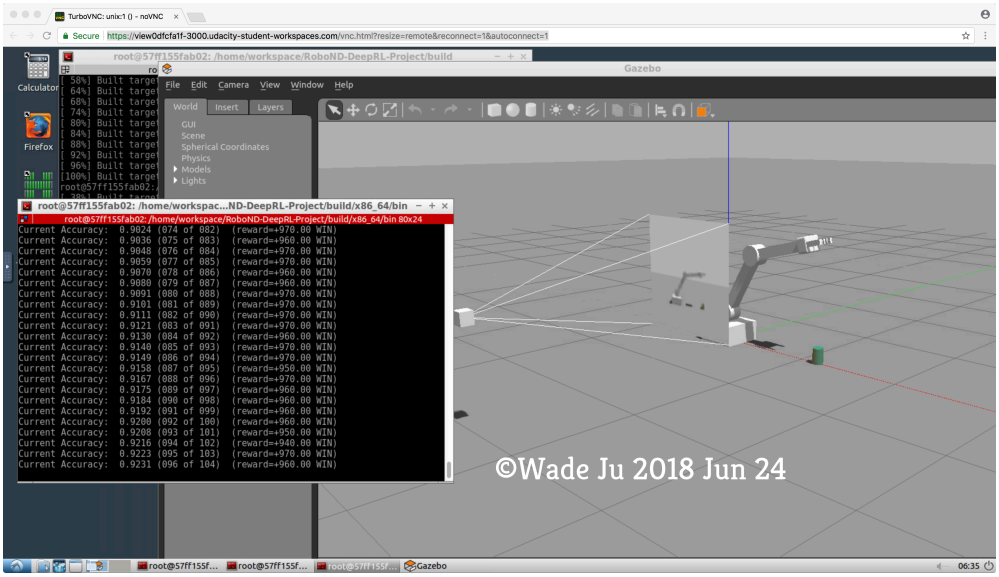

Fig. 2 Results of Task 2

There are two kinds of controls, position control and velocity control. After switching to velocity control, the accuracy goes down dramatically although testing finished quickly. Fig. 3 and Fig. 4 shows results for task 1 and task 4 with velocity control respectively.

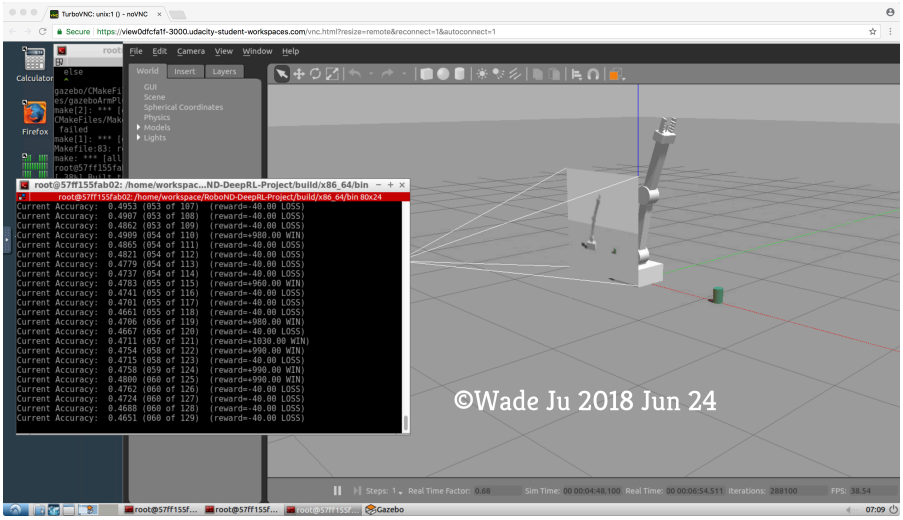

Fig. 3 Task 1 with Velocity Control

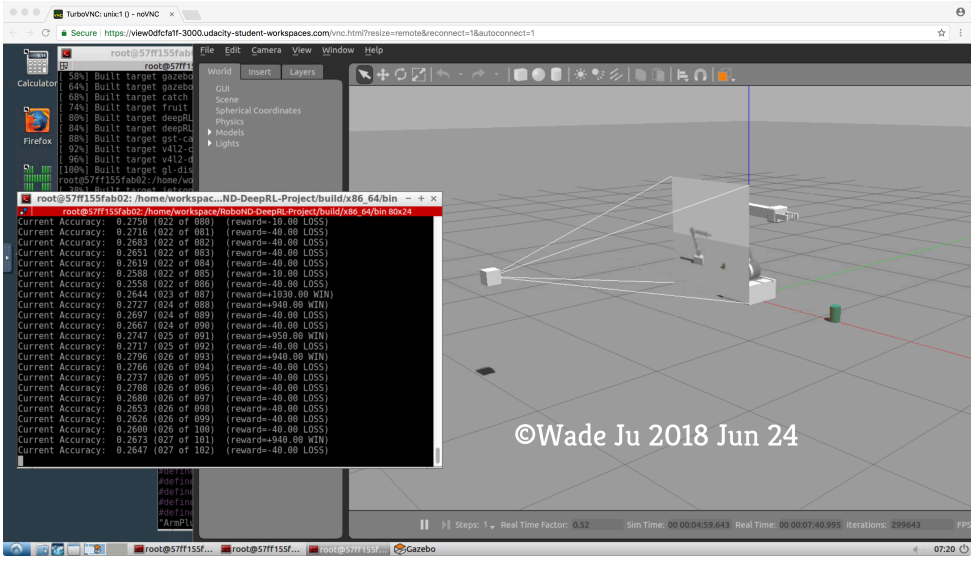

Fig. 4 Task 2 with Velocity Control

Table 1 shows results from three learning rates for task 2 with RMSprop optimization in 100 runs. Learning rate 0.01 performed best in accuracy.

Learning Rate	Accuracy
0.1	79.8%
0.01	92%
0.001	72%

Table 1 Learning Rate vs Accuracy for Task 2

Table 2 shows results with different batch sizes for task 2 with 100 runs. Batch size 128 performed best in both accuracy and time. For batch size 192 and 256, lots of runs failed in timeout that made it finished late besides low accuracy. From the results, it looks DQN not able to handle memory for large batch.

Batch	Accuracy	Time
64	82%	~ 7 minutes
128	92%	~ 6 minutes
192	32%	~12 min
256	0%	~ 39 minutes

Table 2 Batch Sizes for Task 2

Table 3 shows accuracy and time spent in Adam with learning rate of 0.001 and RMSprop with learning rate of 0.01 for task 2 in 100 runs. The test with Adam also ran into lots of time-out that contributed to low accuracy and longer time.

Optimization	**Accuracy**	Time
Adam	22%	~ 20 minutes
RMSprop	92%	~ 6 minutes

Table 3 Adam vs RMSprop for Task 2

With these configuration of hyper-parameters and reward policy, DQN agent performs well in simulator. One might observe the initial couple of runs takes long time to accomplish with difficulty in both task 1 and task 2.

However, once it hits goal with a large reward (> 800 points), it'll perform consistently all the way through 100 runs.

Hyper-parameters help the underneath CNN performing with stable results. The reward policy makes agent learning efficiently including components for hitting goal, efficiency with less frames (time), hitting ground, timeout, and interim rewards with alpha smooth in historical average distance.

Table 4 shows the breakout of reward policy. The reward of hitting goal has two parts, 100 plus time efficiency bonus. Time efficiency bonus is calculated by 10 times delta frames subtracting the current frame index from max frame number (100). It gives a big boost to have DQN agent trained with a reward that is two orders bigger than the base reward.

Reward	Number
Win base reward	1
Loss base reward	-1
Hitting Ground	-40
Timeout	-1
Hitting wrong part of robot	-10
Interim - closer	1 * (distance moved)
Interim - far away	-1 * (distance to goal)
Interim - same distance	0
Hitting goal	100 + 10 * (max_frame - frame index)

Table 4 Reward Policy

Future Improvement

The interim reward was calculated from distance between gripper and the goal. It might be good to use distance between current position and an optimal trajectory if available.

The calculation takes lots of computation power. For it to be executed in hardware, it would be good to optimize and improve on computation. To run and test it on hardware is the next thing to do.

Sometime, gripper gets a perfect position to have left and right grippers going through the target without hitting it. It would be good to adjust gripper to avoid timeout.

Another improvement is to start it from different positions and unlock the base for another degree of freedom or even adding more joins.

Reference

https://github.com/dusty-nv/jetson-reinforcement