Deep Reinforcement Learning for Robot Arm Manipulation

Deepak Trivedi

Abstract—Effective use of deep Q-learning is demonstrated using a robotic manipulator. The manipulator is trained using suitable rewards and punishments in order for its end effector (gripper) to come into contact with a cylindrical object. The benefit of this approach is that it does not require building a model of the manipulator or the environment. Q-learning can handle problems with stochastic transitions and rewards, without requiring adaptations.

Index Terms—Robot, Deep Reinforcement Learning, DQN

1 Introduction

Q-learning is a reinforcement learning technique that technique does not require a model of the environment. Another benefit of Q-learning is that it can also handle problems with stochastic transitions and rewards, without requiring adaptations [1]. Deep Q-learning combines with reinforcement learning a deep convolutional neural network. Layers of tiled convolutional filters mimic the effects of receptive fields.

The DQN technique uses experience replay, a biologically inspired mechanism that uses a random sample of prior actions instead of the most recent action to proceed. This removes correlations in the observation sequence and smoothing changes in the data distribution. Iterative update adjusts Q towards target values that are only periodically updated, further reducing correlations with the target. [1]

Nvidia provides an open source platform called jetson-reinforcement that could be integrated with Gazebo to implement DQN on robotic agents. Gazebo is an open source robotics simulator that allows realistic simulation of robots that could be controlled.

In this project, the jetson-reinforcement platform is used to implement DQN on a robotic manipulator. There are two tasks that the robot should be able to perform.

* Task 1: Any part of the robot arm should touch the cylindrical object with at least 90% accuracy for a minimum of 100 episodes. * Task 2: The gripper base of the robot arm should touch the cylinder with at least 80% accuracy for a minimum of 100 episodes. If any other part of the arm touches the cylinder, the robot has failed and the episode is over.

In both tasks, if the robotic arm touches the ground, the robot fails and the episode is over.

2 Background

Deep Reiforcmeent Learning was first described in a 2015 paper published in Nature [2]. The authors describe a Deep Reinforcmeent Learning system which combines Deep Neural Networks with Reinforcement Learning, and is able to master a diverse range of Atari 2600 games to superhuman level with only the raw pixels and score as inputs [3]. This is a powerful idea because of its generality. The algorithm makes it unnecessary to create separate algorithms to do a wide variety of tasks. This work represents the first demonstration of a general-purpose agent that is able to continually adapt its behavior without any human intervention, a major technical step forward in the quest for general AI [3].

3 DQN Agent Parameters

Successful convergence of the DQN agent depends on fine tuning of various parameters. For the purpose of this project, this notably includes the reward functions and the hyperparameters that governs the reinforcement learning and aspects of the deep neural network. In this section, the choice of these parameters will be explained.

3.1 Model hyperparameters

Table 1 describes the model hyperparameters, including the values chosen. The rationale for picking these values is also briefly noted.

3.2 Reward functions

Choosing appropriate reward function was a delicate task for this project since there was often a thin line dividing failure and success. Strict punishment for failure could discourage the robot from trying. A lot of trial and error was involved in coming up with functions that finally worked. Determining how effective the functions are was difficult at times because of the stochastic nature of the learning process. The final set of reward functions that worked are described in Table 2.

4 Results

With appropriate rewards, hyperparameters and training, the DQN agent was able to accomplish both tasks.

Figure 1 shows a screenshot of the simulation environment taken during training for Task 1. It can be seen that the DQN algorithm achieves an accuracy of more than 90% for more than 100 iterations. In fact, the accuracy is near 100% after about 50 iterations.

TABLE 1
Model hyperparameters for the DQN agent.

Parameter	Value for Task 1	Value for Task 2	Notes
VELOCITY_CONTROL	false	false	Displacement control was found to be more effective. Displacement control means the joint angles are controlled, while velocity control means joint velocities are controlled.
INPUT_CHANNELS	3	3	This refers to the number of channels in the input images. The three channels are red, green and blue
NUM_ACTIONS	6	6	This is twice the number of degrees of freedom of the manipulator.
ALLOW_RANDOM	true	true	This means that the DQN agent is allowed to take random actions. This is necessary so that the agent can explore the environment and stumble upon better rewards.
DEBUG_DQN	true	true	This flag allows details such as collisions and rewards to be output to the console for debugging.
GAMMA	0.9f	0.9f	
EPS_START	0.9f	0.9f	The probability of choosing a random action will start at EPS_START and will decay exponentially towards EPS_END. EPS_DECAY controls the rate of the decay.
EPS_END	0.05f	0.05f	See above. Larger numbers were tried too. Larger numbers helps more exploration but reduces final accuracy.
EPS_DECAY	200	200	See above.
INPUT_WIDTH	64	64	Width/Height of the image used as input to the convolutional neural network. Larger resolution slows down computation. Too coarse a resolution affects accuracy. 64 worked very well.
INPUT_HEIGHT	64	64	See above
OPTIMIZER	”RMSProp”	”Adam”	Optimizer to use for the neural network. The other option tried was RMSProp. It converged slower.
LEARNING_RATE	0.05f	0.05f	Learning rate for the above algorithm. Values from 0.01 to 0.1 were tried. Larger values were detrimental to getting a good final accuracy.
REPLAY_MEMORY	10000	20000	Memory size for experience replay, a random sample of prior actions instead of the most recent action to proceed.
BATCH_SIZE	8	256	Batch size for optimization. Smaller batch size affected convergence stability. Larger batch size made the process very slow. For initial tests, a much smaller size (4 to 16) was used and it was graduall increased when other parameters were tuned.
USE_LSTM	true	true	Long Short Term Memory. This was always used so that the Recurrent Neural Networks makes good use of the entire sequence of actions in order to learn what policies work.
LSTM_SIZE	256	256	Size of the LSTM
REWARD_WIN	100.0f	30000.0f	Parameter that determines the reward given to the agent if it achieves the goal.
REWARD_LOSS	-100.0f	-30000.0f	Parameter that determines the reward given to the agent if it fails to achieve the goal.

Figure 2 shows a screenshot of the simulation environment taken during training for Task 2. It can be seen that the DQN algorithm achieves an accuracy of more than 80% for more than 100 iterations. Just like in Task 1, accuracy keeps increasing gradually over time.

It was discovered that it is important to have a good start. Because of stochastic nature of the method, if there are no early successes, the DQN agent gets lost. In this case, it is appropriate to restart until a good start is achieved.

Videos for both the Tasks are available at [4] and [5].

5 Discussion / Future work

Several lessons were learned from this project that are worth mentioning. First, it was learned that developing appropriate reward functions could be in itself a delicate task, on the edge of being an art. It is difficult to immediately quantify the effect of various changes made in the parameters or rewards because of the stochastic nature of the algorithm. Therefore, deployment of this method may only be suitable where traditional methods such as physics-based modeling are prohibitively complicated. However, with more experience with the process it may become possible to come up with rewards and hyperparameters more easily.

Several challenging extensions of the project were suggested, such as those involving an additional degree of freedom. These would be interesting to try. With a few additional steps, it should be possible to deploy the solution on hardware. For example, a Jetson TX2 could be used to control motor drivers and interface with a camera. Given fewer time constraints, it would be interesting to deploy

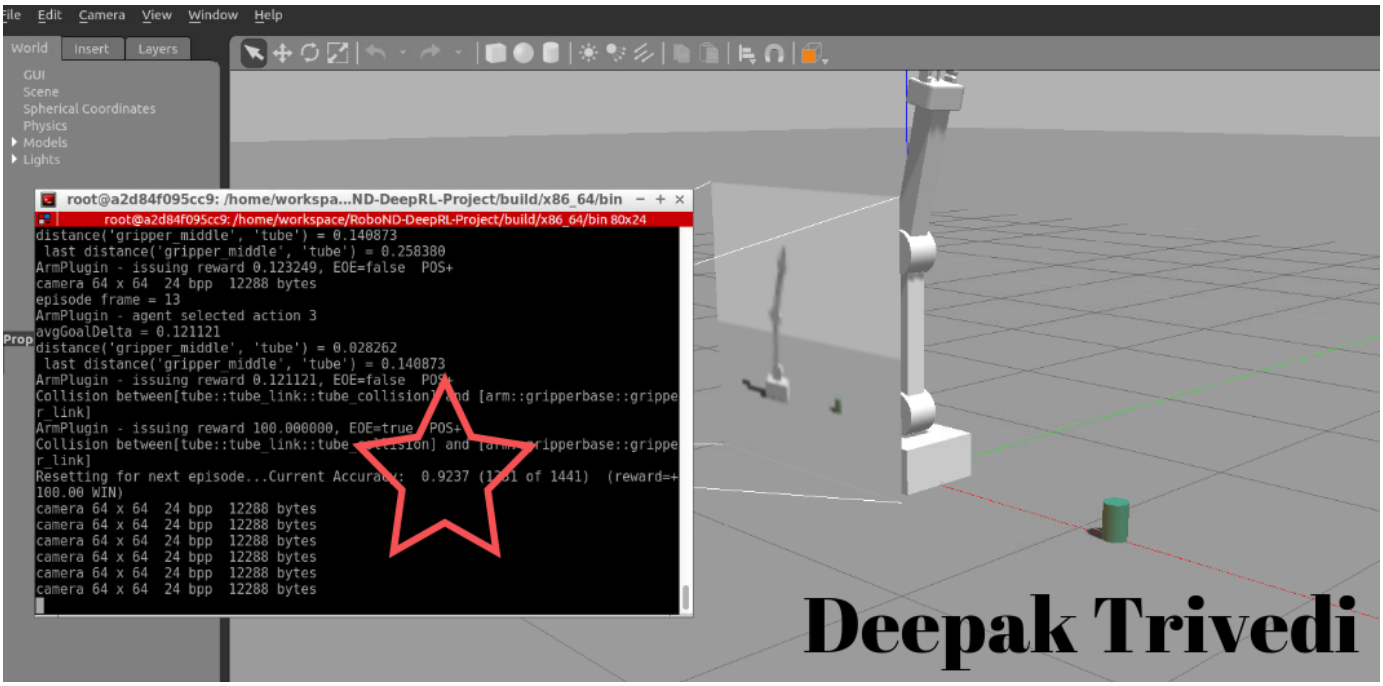

Fig. 1. A screenshot taken during the training of Task 1 showing an accuracy of more than 90% for more than 100 iterations

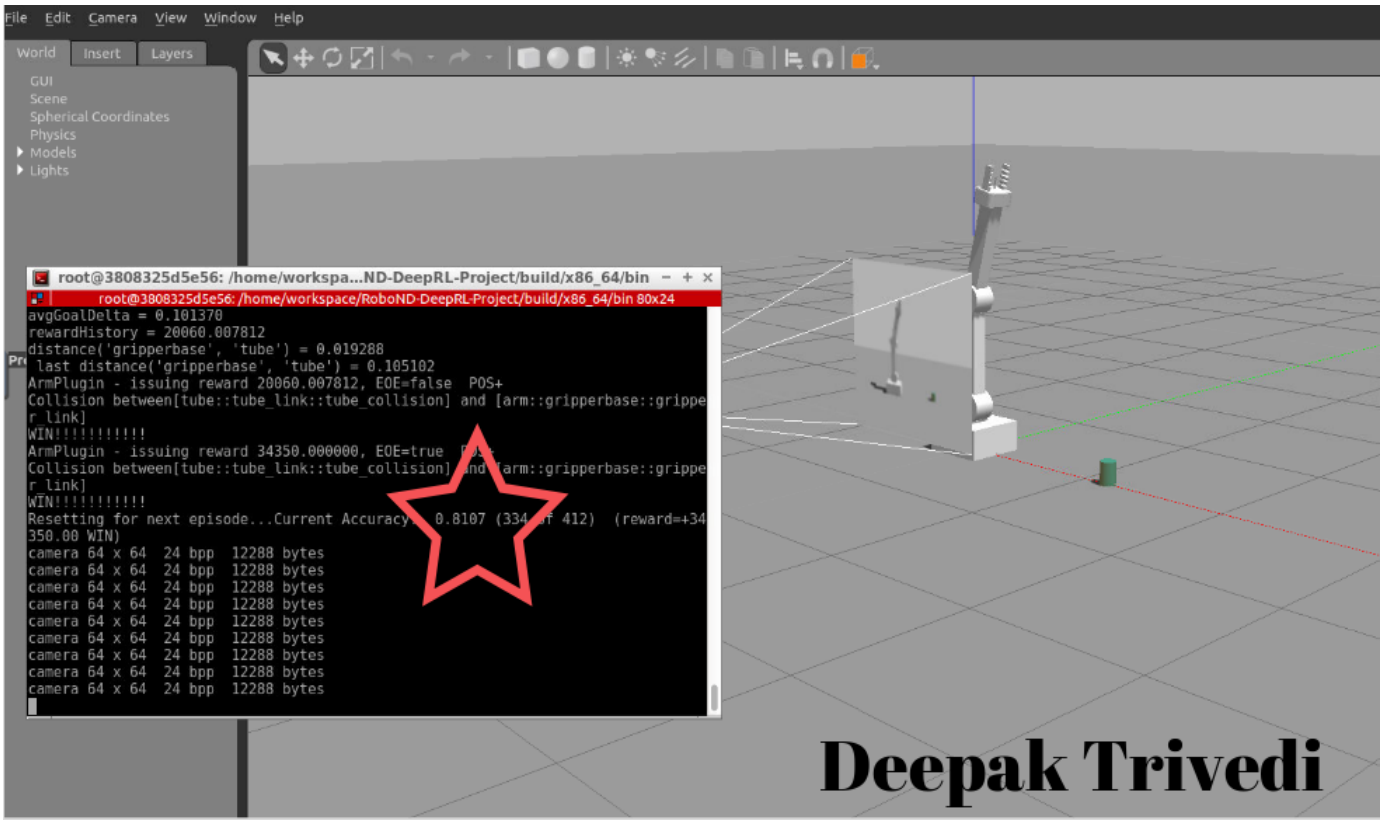

Fig. 2. A screenshot taken during the training of Task 2 showing an accuracy of more than 80% for more than 100 iterations

TABLE 2
Rewards for the DQN agent.

Condition	Task 1	Task 2	Notes
Arm touches ground	-100	-30000	Reward for failure.
Arm (excluding gripper base) touches cylinder	+100	-30000	This is success for Task 1 but failure for Task 2
Gripper base touches cylinder	100	+30000 + 50*number of runs remaining	Rewarding the robot for reaching the goal quickly.
Episode ends inconclusively	-100	-30000	Not getting anywhere is failure.
Interim rewards	avgGoalDelta	500*int(avgGoalDelta¿0.01) + int(avgGoalDelta¿0.01)* (200/(0.01+distGoal)- 150/(0.01+distArm) + 150/(0.01+distHorizGoal))	A combination of total and horizontal distance of gripper from the goal, penalizing stalled arm, and penalizing arm getting close to cylinder. Please see code for more details
Arm stalls	N/A	if (abs(avgGoalDelta)¡0.005) rewardHistory -=500;	Punish arm if it doesn't move much in any direction
Arm moves too far way	N/A	if (distGoal¿1.75) rewardHistory -=2000;	Discourage arm going in a totally wrong direction

this on a Jetson TX2.

REFERENCES

[1] "https://en.wikipedia.org/wiki/q-learning."
[2] "https://storage.googleapis.com/deepmind-media/dqn/dqnnaturepaper.pdf."
[3] "https://deepmind.com/research/dqn/."
[4] "https://www.youtube.com/watch?v=yjpzomyn2mu."
[5] "https://www.youtube.com/watch?v=svzgfhclgfy."