Deep Reinforcement Learning to Control a Robotic Arm

Rohan Paleja

***Abstract*—The process of deep reinforcement learning to control a robotic arm involves using trial and error and a designed reward system to iteratively learn the task at hand. An interface using a C++ API interacting with a robotic arm simulated in Gazebo acting as the DQN agent is used to tune, test, teach an agent to perform the task of touching the gripper of the arm to the object. The first simpler goal to have any part of the robotic arm touch the object took 812 iterations. The second task of training the gripper of the robotic arm to touch the object took 123 iterations.**

***Index Terms*—DQN, Robotics Nanodegree Program, Udacity, RTAB-Map, Reinforcement Learning.**

I. Introduction

A. Motivation for Study

As any roboticist will tell, designing robot controls for movement is hard, and can take months or even years of hard labor and research. As the goal of most robotic projects is to achieve some behavior, a higher level of abstraction can be used to directly work with the behavior rather than the specific kinematics or path planning. Deep reinforcement learning accomplishes behavior training through a trial and error based training. At its simplest level, reinforcement learning works by a repetitive process of the environment giving the agent an observation (can be interpreted as the agent's sensor taking a measurement), and the agent responding with some action. Based on this action, the environment gives the agent a reward that will help it determine its next task. While the concept of reinforcement learning is very simple, the method can be used to solve some very complex tasks, and thus this method is very preferable when it comes to designing robots to do something.

B. Problem Description

Deep reinforcement learning comes with several feats to overcome. For this project, the goal of the robotic arm is to touch the object with high accuracy. The quantitative goals of this project is for any part of the arm to touch the object with an accuracy of 90% and have only the gripper touch the object with an accuracy of 80%. To reach these goals, parameters of the deep Q network had to be tuned for each case. This included choosing learning rates and memory/reward parameters. This also included working with the C++ API and adding code to interface with the Gazebo physics engine. The last and most important step to achieve this goal involves creating and tuning a reward system to ensure that the task was accomplished.

II. Background

Reinforcement learning has been used successfully in tasks from self-driving cars to games. The specialty of this is that no rules of the task are given; only controls, the score, and some feedback. With a simple task of maximizing score, reinforcement learning simplifies complex problems into that of minimizing a cost function.

This cost function must be designed by the user to take into account the important categories for the task at hand. For example, Google trained a robot how to wake. For this, straying from the path and using excessive motor torques were penalized. On the other hand, duration of episode was negatively penalized. Minimizing this function led to developing the parameters for a controller to control the movement of the robot. In the end, a robot that could move through an environment well.

Several definitions should be stated before beginning this discussion. The learner/decision-maker is known as the agent. This is the robot or module that tries trial and error. Exploration is exploring potential hypothesis for how to choose actions. Exploitation is to use limited knowledge about what is already known to choose actions. The state is the observation of the environment at some time (discrete). There is also an associated action and reward for a given time; the definition of those terms are self-explanatory. There are two types of learning tasks, episodic and continuing. Episodic tasks have some defined end point; this is the case in this experiment as the episode ends when the robotic arm touches the object or fails. One sequence from start to finish in called an episode, and at the end of each episode, the reward is analyzed and the sequence is restarted with new weights. Tasks that go on forever are known as continuing tasks. Weights are updated simultaneously as rewards are analyzed and the agent interacts with the environment. Presentation of the episodic task will be shown below. The initial stages of the loop can be presented as follows:
At t=0: the agent receives S_0 and some action A_0 is chosen.
At t=1: reward R_1 and S_1 are received by the agent. Then, action A_1 is chosen. The remaining steps repeat that of t=1. This trial and error process can be very lengthy. Using simulation and a computer with high speed, some of these simulations can take hours or even days to learn complex tasks. Even after lengthy training, unwanted behavior may have been learned for some observation and the procedure still may fail (will be a very small percentage, but in real world applications, failure can cause large problems).

A. Goals and Rewards

The main goal in reinforcement learning is to maximize the cumulative reward. This should be kept in mind at all times, especially during the design of the reward function. The total reward of a function can easily be represented by the sum of the individual rewards at each time step.

$$G_t = R_1 + \gamma * R_2 + ...\gamma^{t-1} R_t + ... \quad (1)$$

where γ represents the discount and the reward function continues till the end of the episode. When γ is equal to 0, only the most immediate return is accounted for, and as γ approaches 1, the more future returns matter and will have effect on the weights inside the cost function. A common value for γ is .9.

The reward function is made up of several parameters that give meaning to the task. In the case of the robotic arm from this simulation, large negative rewards are given for hitting the ground or moving out of frame. Smaller negative rewards are also given for stand still and not reaching the object in time. A distance-based negative reward is also given for how far the arm is from the object. Positive rewards are given for completing the task properly. Thus, to ultimately maximize the reward, the robot should attempt to complete the task.

B. Dynamics, Policies, and Value Functions

Reinforcement Learning attempts to figure out the rules of the environment. Since this task is episodic, how well the agent is doing, or rewards it is collecting within the episode has no effect on how it will choose to respond to the agent. Given the current state and action, a robot can find the probability of the next state and reward by look at an action-reward map. In a real-world problem, the agent knows the state, action, and the discount factor, but does not know the future reward or one step dynamics.

Policies can be defined as how a robot chooses an action based on a state. The simplest type of policy is one that is deterministic or a direct mapping. Another type of policy is a stochastic policy that gives the probability of an action based on a state. Analyzing this and using the state-value function which corresponds to the expected return if agent starts in some state and follows the policy, a best action can be found. In general, if 2 state-value graphs are compared, the one with the higher state values has the better policy. This is due to the fact that the expected discounted return is higher for all states. While this comparison may not always be easily visible, an optimal policy and an optimal state value-function is guaranteed to exist. THe action-value function gives the value of taking some action in some state under some policy. This yields expected return for each action. Iteratively searching through each route of actions and choosing the one that yields the highest expected return allows for a choice of action.

C. Q-Learning

Q-Learning introduces a different type of reinforcement learning algorithm where the environment does not have to fully defined, state changes are assumed to be time based, and values are estimated from observations. The equation for the Q-learning algorithm is

$$Q(s_{t+1}, a_{t+1}) = (1-\alpha)*Q(s_t, a_t)+\alpha*(r_t+\gamma*maxQ(s_{t+1}, a) \quad (2)$$

where α refers to the learning rate, r refers the reward at some time t, a refers to an action at some time t, and Q refers to the estimate of future value. This equation recursively updates after each episode. At the beginning of learning, these q-values are set randomly. Each episode results in an update. Q-learning in it simplest form uses a table to keep track of q-values. These values determine the best action for each state; the best action will be determined when these q-values converge. When these values converge, the policy will be set in stone. During learning, after several iterations, an action of decently high value will be chosen. If the robot chooses to continue to explore what occurs when it chooses this action, it is called exploitation. If the number of iterations is still in the beginning stages, exploration is much better. Using an ϵ-greedy exploration, the amount of exploration in beginning stages will be high, and will reduce exponentially as time continues. This ensures all methods are tested. Once convergence is achieved, this can be removed and the best action can be trained upon.

An even more abstract approach than working with sensor observations from a robot is to use a camera to view the robot. This, then can become an image problem allowing utilization of the learning techniques used mainly in computer vision. Transforming from basic reinforcement learning to deep reinforcement learning allows approximation of the q-value function through a neural network. Using the new method of a camera to view the robot and that data to make observations, a neural network will again input pixels as the case of most computer vision problems, but now output actions for the robot to undertake. Even though this is slightly slower, distinguishing features of each image can be learned and then the state is determined based on the difference between the current and previous patch. This also allows to take into account a large amount of past trials which smooths out learning and increasing the chance of convergence. Some general input features are that the input is of a predefined pixel size. To speed up processing, this is converted into gray-scale. After this, 4 time steps are stacked together to produce sequential data. The general path of the neural network is a convolution followed by a batch norm followed by a convolution, followed by another set of batch norm and convolution, followed by another batch norm, and then a fully connected layer leading to the action vector. Between each of these is a RELU function. Definitions of each of these can be found by consulting literature. Convolutions in general extract properties of spatial and temporal data.

Experience Replay or memory playback is a special feature that allows for a higher chance of convergence. In the usual sense, states, actions, and rewards are used to learn with and then discarded. Instead of this, they are held in a replay buffer.

As these pairs are highly correlated in a sequential fashion, randomization is done so this correlation does not affect the parameters. Training can now use large sets of data to ensure a more robust function.

III. Simulations

Working with a 3D simulation in Gazebo, a C++ API must be used; This is because C++ has faster computation and this API provides an interface to Pytorch. Also, supplied is a rlAgent base class that can be used to create an agent. Each iteration, the environment gives the state to the NextAction() call, which returns the agent's action. Then, a reward function Nextreward() returns a reward to the user. Udacity graciously gives a lot of startup code, alongside two full examples to assist with generation of a reward function and the syntax in general. The environment consists of a robotic arm with gripper, a camera with a feed into the DQN, and a cylindrical object/prop. This learning process also utilizes LSTMs or Long Short Term Memory. This architecture keeps track of both long term and short term memory, where the short term is the prediction.

This template project is based on an open source project developed by Dustin Franklin. This project was also split into two sections, to achieve the two different goals. The first simpler goal is to have any part of the robotic arm touch the object with 90% accuracy with a minimum of 100 runs. The second task of training the gripper of the robotic arm to only touch the object, with at least 80% accuracy for a minimum of 100 runs, as stated previously. A depiction of the project environment is shown below.

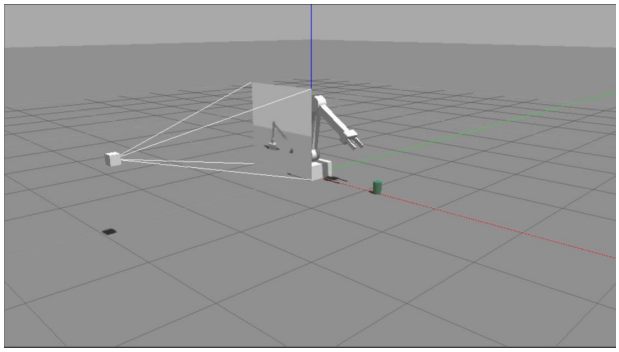

Fig. 1. Project Environment

A. Reward Functions

After finishing several tasks assigned by Udacity to have proper communication between sensors and creating the agent using the previously described class, the reward functions could be designed. This was all done in ArmPlugin.cpp. The revolute joints of the robotic arm were controlled using position control. This was chosen as it was simpler and was the default setting. For both tasks, the reward parameters were defined as
REWARD_WIN=0.10
REWARD_LOSS=-0.10
. The first reward that had to be determined was that of when the robot hit the ground. The corresponded to a very bad case, as if this were to happen in the real world, the robot would break. Thus, a large negative reward of 10*REWARD_LOSS was given. Two booleans should also be mentioned here, newReward and endEpisode. newReward is set to true if a new reward has been issued. endEpisode is set to true if the episode has ended. Here, both are set to true. Note that this reward was same for both tasks.

The next reward analyzed was that of the robot extending the episode length. This meant that the robot was either idle too long or oscillating in a region. This motion should not be punished to highly as the robot may have taken some proper actions to get to that region, but failed to complete its goal in time. Thus, a small negative reward of REWARD_LOSS was given. This episode length was set to 100 frames. Both booleans were again set to true here.

An interim reward was also given based on the distance between the gripper and the prop. A smoothed average of the delta of the distance was computed using

$$avgGoalDelta = (avgGoalDelta*\alpha)+(distDelta*(1.0f-\alpha)); \quad (3)$$

where α was set to .9. If the gripper was moving toward the object, a small positive reward of REWARD_WIN was issued. If not, a scaled negative reward of REWARD_LOSS*2*distGoal was issued, where distGoal is the distance between the gripper and the box. This interim reward was given for both tasks. For the first task, training iterations were very high, nearing about a 1000. Since the second task was more difficult, it could be expected the iterations would be even greater and take much longer. Thus, an additional interim reward was added to penalize arm stagnation. Thus, if the avgGoalDelta was less than .005, a reward of REWARD_LOSS would be issued. For iterim rewards, the newReward parameter was set to true.

For the first task, a parameter called collisionCheck was set to true if any part of the robotic arm collided with the prop. If this was true, a very big reward of REWARD_WIN*15 was issued. A very large reward like this ensured that the learning algorithm could easily tell how to achieve a high cumulative reward. For the second task, the same collisionCheck parameter was used. If this was true, then an if statement is used to see if the gripper is in contact with the object. If so, the task is accomplished and a large reward of REWARD_WIN*25 is issued. If not, then the arm is in contact and a negative reward of REWARD_LOSS*7.5 is issued. In both these cases, newReward and endEpisode are set to true. It is important to note that many of these reward parameters were tuned by trial and error. At this stage, the reward function had been developed to the point of expectation, and from a conceptual point a view, should teach the robot how to do the task. It is hard to generalize on if this method will result in the fastest method of training, as there is much randomness during training.

B. Hyper Parameters

The hyper parameters that could be altered were the dimensions of the image, the optimizer, the learning rate, the replay memory size, the batch size, and the LSTM size. The parameter for the image dimensions INPUT_WIDTH and INPUT_HEIGHT were both set to 64 to match the size of the input. Initially, these were set to the value of 512, which was too large and caused poor performance. The OPTIMIZER was chose as Adam as it performs better than its common alternative of RMSProp. Both were tested and provided very similar results during training. The learning rate parameter LEARNING_RATE was set to .1 for the first task and .01 for the second task. As one will be able to see after the results section, the first task was missing a penalty in the reward function for stagnation; thus, the learning rate was increased in an attempt to decrease the number of iterations required for convergence. After the reward function was improved, a smaller learning rate could be used to increase the chance of convergence while still successfully completing the task in a reasonable number of iterations. The REPLAY_MEMORY parameter was set to 10000 for the first task, 20000 for the second task, and the batch size to 512 for both. A higher amount of replay memory should generally improve the learning. Thus, this value was plainly doubled from its initial in the second task to see its result. LSTM was used in this experiment and thus, the parameter USE_LSTM was set to true. The LSTM size was set to 256 using trial and error.

IV. Results

For the first objective of having any part of the robot arm touch the object of interest, a simple goal reward structure was used. This simulation appeared very slow to learn and the poor start resulted in a large number of iterations. As one can imagine, if the first 40 trials are unsuccessful, at the least 400 more episodes are needed to hit the goal of 90%. Below is a picture depicting the terminal displaying the accuracy. A

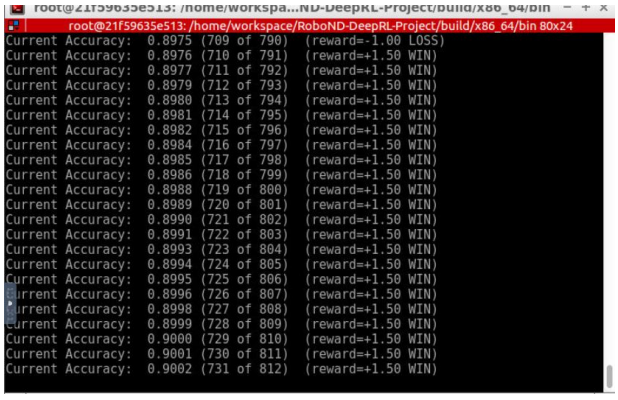

Fig. 2. Task 1 Results: Arm Collision with Prop 90% of Episodes

video of this stage can also be found on http://github.com/rohanpaleja27 that will depict the motion of the robot as this moment was approached. Here, once a winning motion was found and the ϵ from the ϵ-greedy approach had decayed, the robot arm would consistently touch the object about 98% of the time (looking at last 100 trials from end).

The second objective of only using the gripper contact to touch the prop was much more difficult, and if not for changing the reward function, would take many more iterations. 812 iterations took several hours; with the increased difficulty, this experimentation may have taken around eight of nine hours. Thus, after adding the penalty for stagnation into the reward function, the result displayed in figure 3 was achieved. As

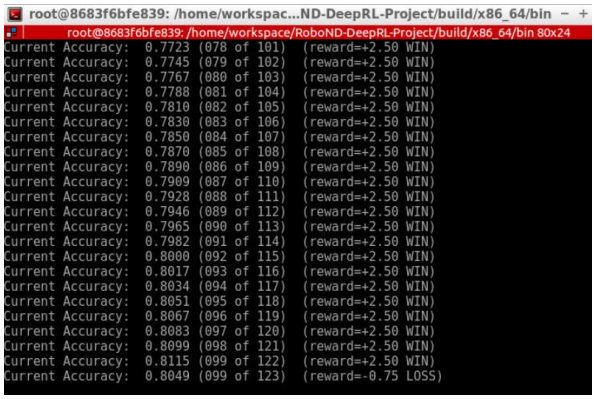

Fig. 3. Task 2 Results: Gripper Collision with Prop 80% of Episodes

these experiments were fairly lengthy, it was very hard to determine if a parameter had a significant effect on the learning accuracy or rate. Even while closely monitoring the simulation for many iterations, quick correlations between parameters and results could not be seen. The only trend that was noticeable that if the robot was lucky enough to start well, and achieve some success within 15 trials, it converged much earlier than simulations that did not.

V. Discussion

Using the Udacity workspace for simulations, long hours were spent training. A total hours of around 30 were spent training to achieve these results. To test each parameter, it seemed only fair to let at least 80 iterations be performed (this number was chosen based on the success range of most Udacity students). In comparison with other Udacity students, the results from the first simulation were much longer than the general consensus. The best explanation for this could be explained due a very poor start, and a mild reward system. After adding the penalty for stagnation, convergence was achieved in around 114 iterations (not shown).

Some observations from the experiment was that at times the robotic arm or gripper would register that it hit the ground when there was slight room left. This could have been fixed by tuning the size of collision boxes. For the first experiment, the robot found success by continuously moving down and allowing the middle of the second link touch the prop. In the second experiment, the robotic arm found success by grabbing the object from the top to ensure that it did not hit the ground and recieve a large negative penalty.

VI. Conclusion / Future work

Reinforcement learning proved to be very successful for a robotic arm attempting to pick up a prop. The generalization that reinforcement learning can be used to teach a robot repetitive tasks can easily be made, and with the help of Nvidia and the C++ API, the implementation only involves the development of a reward system.

Future work for this project is to test the platform on the Jetson TX2. After this, utilization of the CUDA architecture and GPU can be used. This should allow for faster testing and parameter tuning can be studied much more deeply. Adding a feature to the reward system to decrease the torque actuation and time to reach object would be of interest. Another study that is very interesting would be that of actually picking up the object and moving it.

The randomness factor and the initial set of iterations is a region of focus that is most interesting. Finding out what parameters would best ensure a better start to the learning, and ultimately cause earlier convergence would be an interesting study. Another interesting method would be to turn this task into a continuing task where the interim reward system would guide the arm to the prop in a few iterations.

References

[1] SUTTON, RICHARD S.. BARTO, ANDREW G. (2018). REINFORCEMENT LEARNING: An introduction. S.l.: MIT PRESS.

[2] Understanding LSTM Networks. (n.d.). Retrieved from http://colah.github.io/posts/2015-08-Understanding-LSTMs/