Reinforcement Learning for Continuous Control in PyDrake

Samir Wadhwania[1]

Abstract— **Reinforcement learning has been used to learn model-free solutions to various types of problems. Recent work has extended reinforcement learning solutions to continuous spaces, allowing for these methods to be utilized in robotic motion planning problems. TD3 and Soft Actor-Critic are two algorithms that have been widely used for investigating continuous control problems in RL. In this project, I attempt to apply both of these algorithms to the PyDrake Manipulation Station environment in simulation. While the true optimal solution does not emerge during learning, the final behavior significantly outperforms a random policy and shows promise for a successful application given more time.**

I. INTRODUCTION

Some of the most difficult tasks in the field of AI involve finding solutions to problems with high-dimensional, often partially observable, state spaces. Deep learning has leveraged advances in computational power with neural networks to learn policies for difficult problems. In Intelligent Robot Manipulation (6.881), this difficult problem manifests itself with a Manipulation Station that consists of a 7-DOF Kuka iiwa arm and a variety of objects to manipulate. In order to solve the problems presented to us in class, human intervention was necessary to some extent. For example, to open a door, separate trajectories are computed to approach the door handle, grab the handle, and open the door. I attempted to use reinforcement learning methods in order to learn a general policy that can perform a task without the need for direction.

II. LITERATURE REVIEW

A. Reinforcement Learning

The goal of reinforcement learning (RL) is to have an agent learn an optimal policy, π, within an environment E such that it maximizes a given reward function. In general, the reward function an agent attempts to maximize is a time-weighted cumulative reward and the optimal action-value function takes the form

$$Q^*(s,a) = \max_{\pi} \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$

where s represents the state and a represents the chosen action. Google DeepMind demonstrated how RL can be successful by utilizing general function-approximaters such (as neural networks) to train/represent their agent and training via the minimization of TD-Error [1].

Since then, a variety of different methods have emerged using neural networks in RL to speed up or stabilize the learning process, as well as raise the ceiling for optimal performance. Predating [1], [2] proposed the idea of policy gradient methods for RL. In recent years, policy gradient and TD-Error methods have been combined in what is known as actor-critic methods. Actor refers to a policy $\pi(s)$ that takes as input state, s and outputs an action a. The critic is often an action value function $Q(s,a)$ that approximates the value of taking action a in state s. Combining actor and critic allows for the actor to update with gradients with lower variance, often leading to better convergence properties and speeding up the learning process [3].

B. RL For Continuous Control

Q-Learning as used in [1] is difficult to directly apply to continuous action spaces due to the fact that it requires optimizing over the action space at each time step. [4] introduces an actor-critic method that can learn a *deterministic* behaviour in continuous action spaces - a deterministic policy gradient method (DPG). The work is extended as DDPG to stabilize performance and learn in an off-policy manner [5].

[6] combines the idea of energy-based models and maximum entropy policy search to create a stochastic policy that also operate in continuous spaces with Soft Q-Learning. Stochastic energy-based polices attempt to maximize exploration as an inherent part of their learning process (as opposed to being done by injecting noise during action selection). Furthermore, doing away with the notion that an optimal policy is always deterministic, Soft Q-Learning focuses on learning the true underlying Q-function and sampling instead. The Soft Actor-Critic algorithm utilizes the maximum entropy RL framework and creates an off-policy actor-critic method to stabilize and outperform Soft Q-Learning [7].

Separate from both of these efforts, [8] introduces a method to minimize the approximation error in critics by tracking two Q-functions at a time and using the minimum value in order to reduce variance during actor updates. Because of the easily generalizable solution, this approach can be used in most (if not all) actor-critic algorithms to improve performance. [8] does so with DDPG to introduce Twin Dueling Deep-Deterministic Policy Gradients (TD3), and [7] also explores a variant of SAC with this approach as well.

III. REINFORCEMENT LEARNING ENVIRONMENT

PyDrake is a package that allows for limited usage of the Drake C++ toolbox in Python. Drake itself is built for planning, control, and analysis of nonlinear dynamic systems [9]. In order to train a policy for the robot, the RL

[1]LIDS, Massachusetts Institute of Technology, Cambridge, USA samirw@mit.edu

environment must be built within the confines of PyDrake in order to make use of the simulation and inverse kinematics functionality. This implementation was informed by (and likewise informed) design choices for the actual RL problem as discussed below.

A. *Planning and Inverse Kinematics*

There are various different options for what the output of a learned policy ought to be. The choice directly affects the type of problem we are attempting to solve; I separate these into two categories: RL for planning, and RL for control.

RL for planning refers to the output of the policy making a decision about "what to do" or "where to go" next. The motion planning in this case would be handled by solving an inverse kinematics problem. One example would be to take the current state of the robot (and any other information necessary to complete the task) and output a 3D-coordinate of the desired end-effector (EE). This, however, leads to plans that vary in distance and time. It also leans heavily on the assumption that the inverse kinematics solution will converge to a true solution.

On the other hand, RL for control refers to the policy making decisions on the actual torques to apply to each motor. The position of the end-effector does not come in via an observation nor is it outputted - the policy would need to learn how the dynamics of the EE relate to the positions of each joint. This method, however, is easier to implement with fixed time-steps in the simulator as well as fixed plan lengths.

In order to make learning within the PyDrake *Simulator()* class as simple as possible, I took advantage of an existing plan type - JointSpacePlanRelative. The JointSpacePlanRelative takes as input a change for each joint of the robot (Δq) . This avoids the issue of solving an inverse kinematics problem at each time step (speeding up the learning process) and sets the learning problem up as an RL for control scenario. By limiting the policy's max δq, we can ensure that the given δq is actually reachable in a chosen constant duration.

B. *Function Calls*

Environments for reinforcement learning require, at the very least, the following three methods. I detail how each method was implemented in PyDrake below.

- **Step(action)** - the environment takes in a 7-vector (Δq) and steps forward the Simulator() one time-step with the given action.
- **Reset()** - a new starting configuration is picked and all contextual properties are reset.
- **Reward()** - a reward is calculated given the current state of the environment and returned to the policy.

1) Step: The environment expects an action that represents a relative configuration of the current joint positions (Δq). This relative configuration is passed into the JointSpacePlanRelative function from Lab 2 with a constant duration of 0.1 seconds. This plan can then be added to the Plan Scheduler leaf diagram with a desired gripper position. Because we are always stepping forward a constant time-step, we know what time the simulator needs to be at (the current time + 0.1). The simulator can then be stepped forward to that time.

Once the simulator has stepped forward, a new observation and reward can be calculated from the new state, and all of these values can be passed back to the RL algorithm.

2) Reset: Within the Simulator(), the *context* controls the properties of all entities that can have properties. In our cabinet/block/robot environment, this separates into: hinge angles of the two doors on the cabinet, an Isometry3() (containing position and rotation) of the block, and a value for each robot joint.

The properties of the doors and the block can be set directly. Because the starting position of the robot is much more complex, I pick a starting position of the end effector instead and solve the inverse kinematic problem to obtain the starting configuration for the robot. A new observation is returned to the RL algorithm after resetting.

Note: Some plans will lead to the Simulator failing to forward integrate (often when accelerations get too large or the robot is forced into a catastrophic position). In this situation, the normal methods used to reset the context will not work and a new simulator will need to be initialized.

3) Reward: The reward is a function of the current *context* and the chosen task. The reward is calculated and returned.

C. *Task Selection*

The reward function depends wholly on the task we are trying to solve. In choosing a task, it is important to take into consideration what the reward function would look like and how it would affect the learning process. The options I looked are are detailed below.

1) Opening a door: This task was inspired by Lab 2, where our goal was to use Trajectory Optimization and Coordinate Transforms to open the left door. If this were the chosen task, the reasoning behind the reward is straightforward - the agent receives more reward as the door is opened more. Therefore, a logical choice would be to tie the reward (make the reward proportional to) the current angle of the door.

While this can be considered a *dense* reward in that information is obtained at each time step and before the completion of the task, the robot does not begin near the door. Because of this, at the beginning of training, the reward is constant. A constant reward provides no information, so must continue to explore until there is a change in the reward obtained. In this case, it translates to the robot acting randomly until the correct door is slightly nudged and opened. Initial testing showed that this took an incredibly long amount of time and was not feasible for the time span of this project.

2) Placing a block in the cabinet: This task requires that the doors begin open. To make the problem even easier, the block begins in the grasp of the end effector. An end goal is defined in 3-space, and the reward is the negative of the squared Euclidean distance between the block and the end goal. The reward thus nudges the robot to move the block in its grasp to the end goal (and ideally drop the block).

This reward looks much better for training as meaningful information in conveyed from the beginning; however, contact dynamics make this problem surprisingly difficult. Even with a constant gripper setting, the block often falls due to the accelerations of moving around. A true solution would entail the agent learning to not drop the block. Similar to the previous task, however, dropping the block results in a lack of useful information (as the reward is constant until it is picked up again). Due to this, this task was also deemed in-feasible within time constraints.

3) Reaching a goal position: The final task is a de-scoped version of placing a block in the goal location. Since we have access to the location of the end effector, a similar reward can be calculated with the location of the end effector instead of the location of the box.

Note that this task does not aim to have the robot end up in the same configuration every episode - only the end effector's position is relevant to maximizing the obtained reward.

IV. THE ALGORITHM

Once an environment class was built with the Simulator() and the chosen reward function, a RL algorithm could be implemented to train either the TD3 or SAC algorithm.

A. *Soft Actor Critic*

The implementation of SAC was straightforward given the built environment (read: exactly similar to the original implementation from the paper). The code was written in PyTorch for simplicity.

In order to find usable parameters, multiple sessions were run with the same seed but varied parameters - the novel parameter in Soft Q-Learning and SAC (beyond parameters such as target update rate τ and learning update rate α) is the notion of temperature. This temperature parameter determines the relative importance of the entropy term in the algorithm against the reward and is usually low (0.01-0.2).

After an initial training session (5 runs in >1 day), there was no instance in which the learning seemed to stabilize or converge to any particular solution. Because training the SAC agent for one set of parameters took on the order of 18 hours (and the fact that I only have a limited number of AWS instances allowed at a time), I quickly decided to investigate the feasibility of TD3 before continuing to tune parameters.

B. *TD3*

The authors of TD3 [8] released their code for TD3 already written in PyTorch, making it even simpler to run with our PyDrake RL Environment. The first test showed some notion of stability, which led me to spend the rest of the time focusing on TD3.

As mentioned before, the output of the policy (δq) should be limited to ensure that the JointSpacePlanRelative does not attempt to move to a significantly different configuration in a short amount of time. This is done by adding an operation to the output of the last layer of the neural network:

$$x = maxval \times \tanh(x)$$

V. RESULTS

No configuration was found within the time-frame of the project that led to an optimal solution (reached within 0.1m of goal position as quickly as possible). However, training with TD3 did lead to markedly improved performance over testing with a random policy. Table I shows the final parameters used during training.

TABLE I
FINAL PARAMETERS

Parameter	Value	Description
max_val	0.2	max value in δq
timesteps	1e4	# timesteps to train
exploration Noise	0.1	standard dev. of gaussian noise
policy noise	0.2	noise added during critic update
batch size	100	batch size for training
γ	0.99	reward discount factor
τ	5e-3	target network update rate
α	1e-3	learning rate

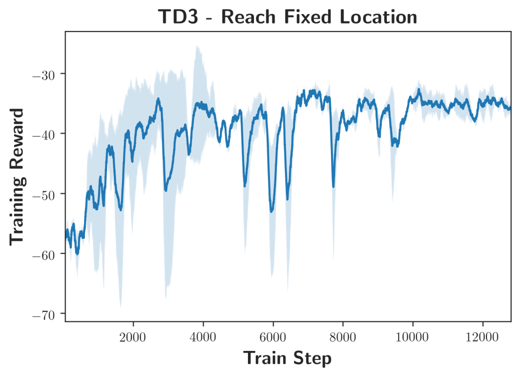

Fig. 1. Mean and std of reward during training for TD3 (various seeds)

Figure 1 shows the recorded episode reward during the training process. After the agent has been trained, a separate evaluation is done with zero noise. With no noise, the average cumulative reward is **-27.83** or an average distance of **0.37m** for each time-step. Videos[1] show that the final policy moves the end effector toward the goal position and stops a constant distance away (even before the end of the episode). It is my guess that this is a local minimum that results in the gradient at that state leading to no meaningful change to the policy. Adding more exploration is one method of avoiding this; Soft Q-Learning and SAC hope to avoid this altogether, but, as mentioned already, proved too time consuming to investigate fully for this project.

Note: Unfortunately, I lost the reward data while training SAC and do not have a plot for it's progress. It diverged often and randomly depending on the seed, leading to a plot without significant convergence and high standard deviation.

[1]Videos and code can be found at https://github.com/SamirW/6-881-Final-Project.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015, ISSN: 14764687. DOI: 10.1038/nature14236.

[2] R. Sutton, D. McAllester, S. P. Singh, and Y. Mansour, "Policy Gradient Methods for Reinforcement Learning with Function Approximation," *Advances in Neural Information Processing Systems 12*, 1999, ISSN: 0047-2875. DOI: 10.1.1.37.9714.

[3] V. R. Konda and J. N. Tsitsiklis, "OnActor-Critic Algorithms," *SIAM Journal on Control and Optimization*, 2003, ISSN: 0363-0129. DOI: 10.1137/S0363012901385691.

[4] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic Policy Gradient Algorithms," *Journal of Machine Learning Research*, 2014, ISSN: 1938-7228. DOI: 10.1017/jfm.2018.119.

[5] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," Sep. 2015.

[6] T. Haarnoja, H. Tang, P. Abbeel, and S. Levine, "Reinforcement Learning with Deep Energy-Based Policies," in *International Conference on Machine Learning (ICML)*, Feb. 2017.

[7] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," Jan. 2018. DOI: arXiv:1801.01290v2.

[8] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing Function Approximation Error in Actor-Critic Methods," Feb. 2018.

[9] (2018). Drake, [Online]. Available: https://drake.mit.edu/index.html (visited on 12/11/2018).