Reinforcement Learning for Automating Control on the daVinci Surgical Robot

Jingpei Lu and Soumyaraj Sreeman Bose

***Abstract*—Automation of robotic surgeons is the next milestone to be attained in medicine and, is already of prime importance to areas with extreme necessity but severe lack of medicare. With the motivation to solve this problem, we use a reinforcement learning (RL) approach to automate the arm of a da Vinci® surgical robot. This works solves the reach problem on the simulator of the da Vinci® robot arm. The simulator is enabled with dynamic properties with a goal to recreate realistic behaviour in the arm. A custom Gym environment is created for the same and is modified to control the arm in joint-space. The model is then trained using time-delayed methods, such as DDPG and TD3, to reach a randomly selected goal position. Training with TD3 with a dense reward-based model yields a better result than with DDPG, with the system reaching the goal in under thirty episodes.**

I. Introduction

The fact that artificial intelligence (AI) has penetrated into every aspect of life is redundant, to say the least. AI frameworks are continually used not just in analyzing daily phenomena but, in automating the simplest aspects of daily life. However, one field of work where the incorporation of AI is being practiced with restraint, primarily for the risks at hand, but will immensely profit from it is surgical robotics.

Richter et. al. [1] is one of the few efforts to employ aspects of AI with the motivation to automate it to perform standard surgical tasks. It realized a reinforcement learning (RL) framework to enable automation on the daVinci® surgical robot, building upon the computational structure from the daVinci Research Kit (dVRK) [2]. Known as the dVRL framework, it is currently used to train the Patient Side Manipulator (PSM) of the robot to pick an object or reach an end-effector position.

However, in case of the dVRL setup, the PSM arm in simulation lacks dynamic properties. Hence, in essence, it only approximates and does not recreate most responses it would have to interactions with bodies in the real-world. This property of the simulator is pivotal to checking collisions with objects (specifically, organs in the patient's body) and, learning complicated tasks such as guiding a tool to suture a wound.

This project aims to complete this aspect by incorporating dynamics into the simulator and, using a framework similar to the dVRL to solve a form of the reach problem. To generate an optimal policy for reaching a target position, this dynamically-enabled model is trained using deep reinforcement learning algorithm such as Deep Deterministic Policy Gradients (DDPG) [3] and Twin Delayed Deep Deterministic Policy Gradients (TD3) [4]. The result is a system using joint positions as inputs to successfully guide its end-effector to randomly generated goal positions, in the presence of other systems in its vicinity.

Jingpei Lu, and Soumyaraj Sreeman Bose are affiliated with the Department of Electrical and Computer Engineering and the Department of Mathematics, respectively, at University of California San Diego, La Jolla, CA 92093 USA. `{jil360, ssbose}@ucsd.edu`

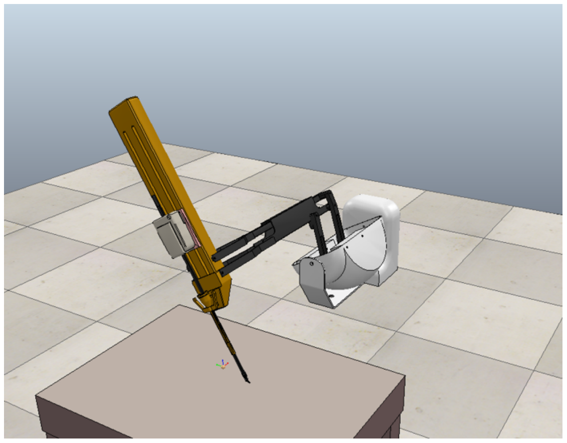
Fig. 1: A dynamically-enabled simulator of a dVRK PSM arm in the V-REP software environment

II. Background

A. The Training Environment for Reinforcement Learning

Reinforcement learning assumes that there is an agent that is situated in an environment. For each step, the agent takes an action and it receives an observation and a reward from the environment. Specifically, after the agent takes an action, following prevalent RL frameworks, it will receive a set of variables namely *observation*, *reward*, *done* and *info*, which are defined as (enclosed are the type of values they take)

- Observation (object): An environment-specific object representing the observation of the environment. For example, pixel data from a camera, joint positions and joint velocities of a robot, or the board state in a board game.
- Reward (float): Incentive received by the agent upon taking an action. The scale varies between environments, but the goal is always to increase the total reward.
- Done (boolean): This indicates whether it is time to reset the environment again. Most tasks are divided up into episodes (with a fixed number of steps per episode) and, done being True indicates the episode has terminated.
- Info (dict): Diagnostic information useful for debugging. This can be used in studies to gain more insight about the system (for example, it might contain the raw probabilities behind the environment's last state change).

An RL algorithm seeks to maximize some measure of the agent's total reward, as the agent interacts with the environment. In the RL literature, the environment is formalized as a partially observable Markov decision process (POMDP).

B. *Reinforcement Learning for Automating Control*

We consider the standard RL formalism consisting of an agent interacting with an environment, where we have a set of states S, actions A, the discount factor $\gamma \in [0, 1]$, a reward function $r : S \times A \rightarrow \mathbb{R}$, and policies $\pi : S \rightarrow A$. The agent's goal is to find a policy that maximizes its expected return $\mathbb{E}_{s_0}[R_0|s_0]$, where $R_t = \sum_{i=t}^{\infty} \gamma^{i-t} r_i$. One of the primary goals of the field of artificial intelligence is to solve complex tasks from unprocessed, high-dimensional, sensory input. One famous approach to solve this problem with reinforcement learning is the Deep Q Network (DQN) algorithm [5] that is capable of human level performance on many Atari video games using unprocessed pixels for input.

In DQN, we employ a neural network which approximates the optimal action-value function $Q : S \times A \rightarrow \mathbb{R}$. During training, for each episode, this setup generates the current approximation of the Q, and tuples for each transition (s_t, a_t, r_t, s_{t+1}), which are stored in the so-called replay buffer. The network is trained using samples in the replay buffer and by applying gradient descent to the loss function:

$$L = \mathbb{E}[(Q(s_t, a_t) - y_t)^2] \tag{1}$$

where $y_t = r_t + \gamma max_{a' \in A} Q(s_{t+1}, a')$.

However, while DQN solves problems with high-dimensional observation spaces, it can only handle discrete and low-dimensional action spaces. Lillicrap et. al. successfully adapted deep reinforcement learning methods such as DQN to continuous domains, thus developing the Deep Deterministic Policy Gradient (DDPG) algorithm [3]. DDPG uses two neural networks: one, an action-value function approximator (*critic*) $Q : S \times A \rightarrow \mathbb{R}$, and, the other, a target policy approximator (*actor*) $\pi : S \rightarrow A$. The transition tuples used for training are also sampled from the replay buffer. The critic network is trained with a similar loss function as (1) but the targets y_t are computed using actions generated by the actor network, i.e. $y_t = r_t + \gamma Q(s_{t+1}, \pi(s_{t+1}))$. Hence, the actor is trained using the loss function given by:

$$L_a = -\mathbb{E}_s[Q(s, \pi(s))] \tag{2}$$

Although DDPG is capable of providing excellent results, it has its drawbacks. Like many RL algorithms, training DDPG can be unstable and heavily reliant on finding the correct hyperparameters for the current task. This is caused by the algorithm continuously overestimating the Q values of the critic (value) network. These estimation errors build up over time and can lead to the agent falling into a local optima or experience *catastrophic forgetting*. Scott et al. addressed this issue by focusing on reducing the overestimation bias seen in previous algorithms by developing the Twin Delayed Deep Deterministic Policy Gradient (TD3) [4] algorithm. They improved upon DDPG by adding the following features: a pair of critic networks, delayed updates to the actor network and regularization of the action using noise (e.g. O-U noise).

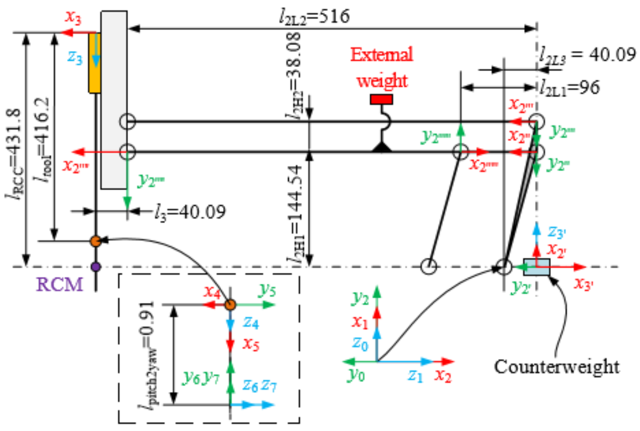

Fig. 2: Frame Definition of the daVinci® PSM

III. METHODOLOGY

A. *Dynamics for da Vinci PSM Simulator*

The Dynamic dVRL setup contains some noticeable upgrades and deviations with respect to the framework in [1]. The first is its lightweight synchronization of a Python interface with V-REP using the PyRep [6] module, instead of the V-REP remote API. The daVinci PSM scene in V-REP uses the same PSM model as in Fontanelli et. al. [7] (Fig. 1). However, unlike [1], the Dynamic dVRL simulator is enabled with dynamics data generated from the work of Wang et. al. [8].

Wang et. al. proposed a convex optimization-based method to identify dynamic parameters of the seven (with the gripper discretized as two for computational simplicity, one to control the yaw of the gripper and the other to adjust its jaw angle) joints associated with the daVinci®. A frame definition of the PSM realized for computing the dynamics is shown above (Fig. 2). [8] solves for these parameters by modelling the dynamics using the Euler-Lagrange equation:

$$\tau_i = \frac{d}{dt}\frac{\partial L}{\partial q_i} - \frac{\partial L}{\partial q_i} \tag{3}$$

Here, L represents an inertia tensor containing the moments of inertia about the frames associated with each of the joints. q_i explains the position of joint i and τ_i represents the torque on the motor due to the inertia in joint i. Note that, a few approximations have been made in the case of the Dynamic dVRL based on the results of [8]. Primarily, friction in the links and motor inertia have been neglected owing to their considerably low values (in the order of less than 10^{-3}).

Another deviation from [1] is the fact that the dynamic dVRK is controlled in joint configuration space, instead of end-effector space. This implies that joint positions and/or velocities (even torques) serve as inputs and are controlled to make the system reach a position. Operating the dVRK in joint space helps in realizing the contribution of and better controlling a joint to cause the collective motion of system.

This is especially helpful, when considering a scenario, where larger parts of the PSM are in collision with objects in the world outside the patient's body (a device or another individual), even though the gripper functions within its bounds.

Joints in the scene are enabled with PD control via the Spring-Damper mode in V-REP, which follows the relation:

$$F = Ke_i + C\frac{(e_i - e_{i-1})}{\Delta t} \tag{4}$$

The Spring-Damper mode enables V-REP's control mechanism to control the position (angle for revolute) that the joint attains by modifying force/torque on the joint's motor to regulate the error in position attained at time-step i or e_i. The mechanism applies proportional (P) control in the form of a spring constant (K) to check position error and, derivative (D) control by damping noise due to a net rate of change in error across two consecutive time-steps with a coefficient (C).

To verify dynamics, prior to training, the joints are driven to follow test trajectories given by [8], using functionality from [6] to set and get joint positions and velocities in a simulation. For this project, we also tune the PD control parameters (i.e. K and C) for the joints to check whether the system is stable upon enabling dynamics. To ensure that the tuning does not result in an overfit to the test trajectory, it is first performed on a simple response function such as a step or sine wave.

The verification yields the average error in tracking test trajectory for each joint, as tabulated in Table 1. The considerably low error values for this check help testify for the simulator to be used further by the Dynamic dVRL setup.

TABLE I: Average error in tracking test trajectory for each of the seven joints of the Dynamic dVRL simulator

Joint	Average Error
1	0.004345
2	0.001865
3	0.001005
4	0.006488
5	0.000680
6	0.006658
7	0.005178

B. The Customized Gym Environment

The Dynamic dVRL framework integrates the V-REP software environment (for robot simulations) with the framework of OpenAI Gym [9]. Figure 3 shows the block diagram of our framework. The problem at hand, as solved for a non-dynamic scenario by [1], involves a continuous state and action space, dictated by design configurations of the daVinci system. Since the aim is to generate an optimal policy, the simulator will learn from a reward model, depending on the objective of the problem being solved (if reach, then distance from goal position; if pick, then height above a base level).

This project is focused on solving the reach task, where the gripper is driven to reach a random goal position within the bounds of a table by passing target positions to the first five joints. Its position is, then, recorded for reward and convergence calculations. The goal and gripper are oriented about a base frame in the space of the simulator and, their positions are always computed with respect to this base frame.

To formulate this as an RL problem, we choose the state space to be the combination of joint positions and the goal position. The action space is composed of target positions for the five joints. We also incorporate two types of reward functions into the framework. We use the negative of the distance between the end-effector position and goal position as a dense reward, and 0/1 reward of successful reach as a sparse reward. Here are their symbolic representations:

- States: $s_t = [\boldsymbol{q}_t \quad p_{goal}]$, where $\boldsymbol{q}_t = (\theta_{1,t}, \theta_{2,t}, p_{3,t}, \theta_{4,t}, \theta_{5,t})$ are joint positions (for the first five joints) in radians and metres (for the 3rd and only prismatic joint) at time t and $p_{goal} = (x, y, z)$ is the goal position for that episode in metres.
- Actions: $a_t = (\hat{\theta_1}, \hat{\theta_2}, \hat{p_3}, \hat{\theta_4}, \hat{\theta_5})$ which are target positions for the joint to attain.
- Reward: We use two types of reward functions Continuous $r(s_t) = -||p_{actual} - p_{goal}||$ and, Discrete/Sparse reward

$$r(s_t) = \begin{cases} 1 & \text{if } ||p_{actual} - p_{goal}|| < \epsilon \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

  where p_{actual} is the end-effector position and ϵ is a threshold to determine whether the end-effector is close enough to the goal position.

Since we are setting the target joint positions to the simulator directly, to ensure stability of the simulation, we define the joint positions of the next state to be

$$q_{i,t+1} = \alpha \hat{q_i} + q_{i,t} \tag{6}$$

where $i = [1, 2, 3, 4, 5]$ indicates each joint and α is a scaling factor. We also clip the target joint positions to the range $[q_{i,min}, q_{i,max}]$ for each joint to make sure we do not exceed any limits imposed on the joint by the daVinci's actual design.

IV. Experiments

To learn the policy of reaching a goal, we trained the agent using DDPG and TD3 with dense reward. The threshold for successful reach is set to $\epsilon = 5$mm and the scaling factor is set to $\alpha = 0.1$. For training the model,

- the reward discount factor is set to $\gamma = 0.99$,
- the learning rate for both actor and critic networks is 1e-3.
- the parameters of the actor and critic networks and replay buffer are derived from the implementations in the original papers for DDPG [3] and TD3 [4]

The training is done for 10^6 steps with batch size equals to 100. For experiments, the agents is trained on a fixed goal position setting as well as a random goal position setting and the results are showed in Figures 4 and 5 respectively.

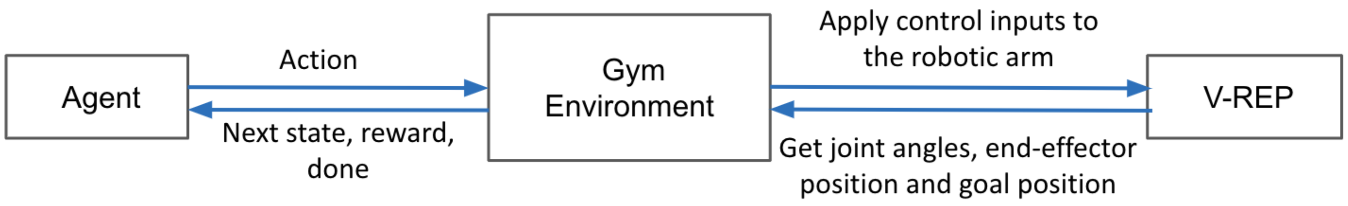
Agent
Action
Next state, reward, done
Gym Environment
Apply control inputs to the robotic arm
Get joint angles, end-effector position and goal position
V-REP

Fig. 3: We customized a gym-like reach environment for the daVinci® surgical robot by interfacing a dynamically-enabled simulator in a V-REP scene. An agent interacts with the environment with actions and the environment interfaces with the simulator by applying the control inputs corresponding to the actions. Then, the environment provides feedback to the agent by obtaining joint positions, the end-effector and goal positions from the simulator.

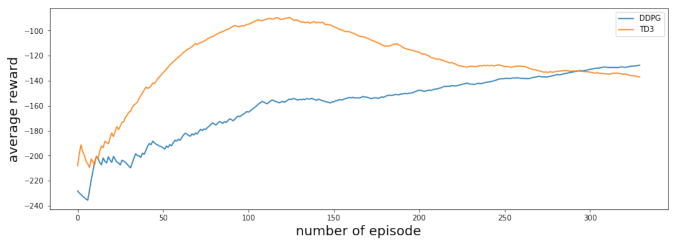

Fig. 4: The learning curves for the reach task with a fixed goal setting. The x-axis indicates the episode number and the y-axis indicates the moving average of the discounted reward over episodes. The orange line represents the learning curve for TD3 and the blue line represents DDPG. While DDPG converges slowly, TD3 converges quickly but it drops later.

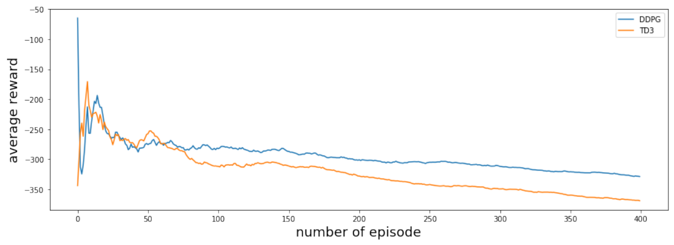

Fig. 5: The learning curves for the reach task with random goal setting. The x axis indicates the episode number and the y axis indicates the moving average of the discounted reward over episodes. The orange line represents the learning curve for TD3 and the blue line represents the DDPG. In this case, none of them converge. We will discuss this in Section V.

V. Discussion and Conclusion

While the dynamic model allows for successful training of algorithms such as DDPG and TD3, there are problems which arise in the system that call for attention. A major anomaly is the apparent reorientation of the entire model as it is reset over a considerable number of episodes. A plausible reason for this situation is that some of the state variables, though being sampled from a fixed range of values, may have eventually compounded in value after multiple episodes.

Another pitfall experienced by using this model is its inability to learn from a sparse reward setup. Unlike the non-dynamic case where the gripper is set to a particular position and reaches it unfailingly, joint space-control does not guarantee the gripper precisely reaching a target. However, it manages to sensitize the system to its surroundings and allows us to realize the extremes it can reach, thus helping develop a true representation by giving us the breadth of responses.

Our future investigations into solving this problem can take two particular routes. The first involves improving the dynamic model being used, since some failure cases in training may be attributed to the system hitting a limit in the dynamics. The second route can involve the identification and application of methods that enable learning from the experience of "failure in attaining goals", but for dense rewards. Alternatively, to allow training on a sparse reward-based model, the TD approach can be retained but modified to account for difference over a larger number of time-steps than three (as in TD3).

References

[1] F. Richter, R. K. Orosco, and M. C. Yip, "Open-sourced reinforcement learning environments for surgical robotics," 2019.
[2] P. Kazanzides, Z. Chen, A. Deguet, G. S. Fischer, R. H. Taylor, and S. P. DiMaio, "An open-source research kit for the da vinci® surgical system," in *2014 IEEE international conference on robotics and automation (ICRA)*, pp. 6434–6439, IEEE, 2014.
[3] 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," 2015.
[4] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International Conference on Machine Learning*, pp. 1582–1591, 2018.
[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.
[6] S. James, M. Freese, and A. J. Davison, "Pyrep: Bringing v-rep to deep robot learning," 2019.
[7] G. A. Fontanelli, M. Selvaggio, M. Ferro, F. Ficuciello, M. Vendittelli, and B. Siciliano, "A v-rep simulator for the da vinci research kit robotic platform," in *2018 7th IEEE International Conference on Biomedical Robotics and Biomechatronics (Biorob)*, pp. 1056–1061, Aug 2018.
[8] Y. Wang, R. Gondokaryono, A. Munawar, and G. S. Fischer, "A convex optimization-based dynamic model identification package for the da vinci research kit," *IEEE Robotics and Automation Letters*, vol. 4, pp. 3657–3664, Oct 2019.
[9] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.