Comparison of Control Methods: Learning Robotics Manipulation with Contact Dynamics

Keven Wang
Computer Science
Stanford University
Email: kvw@stanford.edu

Bruce Li
Mashgin
Email: yonglisc@gmail.com

***Abstract*—We compare the different control methods in learning a robotic manipulation task. The task is to push an object (a cube and sphere) from varying beginning position to a fixed goal position. Complex contact dynamics is involved. We used PPO as the learning algorithm trained from scratch with dense rewards. Comparison is performed on two dimensions: learning at joint level vs. end-effector level, as well as velocity control vs. position control. For end-effector learning, we use inverse jacobian to map from end-effector target velocity/position to joint velocity/position, and accounting for singularity, joint limits, and gimbal lock. Across the four methods proposed, joint velocity control demonstrated the fastest convergence on cube task across all control methods, and is the only successful method on sphere task. Video demonstration: https://www.youtube.com/watch?v=wh_qV58f95Y**

I. Introduction

Robotics manipulation is traditionally done using hand-crafted local controllers. A unique controller is built for each individual task. To repurpose the controller for a different task, a different set of parameters, such as gains, need to be tuned by hand. This tuning process requires expert knowledge, which makes this approach not as scalable as desired. On the other end of the spectrum, the learning approach commands at the joint level, while being more generalizable, disregards classical robotics principals such as forward and inverse kinematics.

Here we approach the robotic manipulation problem from a learning perspective, while attempting to leverage the well-studied principal of inverse kinematics. In particular, we compare two approaches to learning: commanding joint velocity, and commanding end-effector velocity. In the second case of end-effector velocity, we use inverse jacobian to compute the joint velocity needed to reach the desired positions. The task involves pushing an object task from start to goal, learned from scratch using reinforcement learning. The only human input is a dense reward function, which encodes the desired behavior and is easy to interpret. The neural network is trained end-to-end with object positions as input. The method is agnostic to the reinforcement learning algorithm used. Through experiments in simulation, we show that this method is generalizable to variation in start or goal position of the object.

The task is to push an object from a starting position to an end position on a plane. The robot used has six degrees of freedom rotational joints respectively. The pushing task, as compared to a pick-and-place task, invoves complex contact dynamics, since the object tends to rotate upon contact. The trajectory planning aspect is also present, where the robot needs to push the object from a starting position to a goal position.

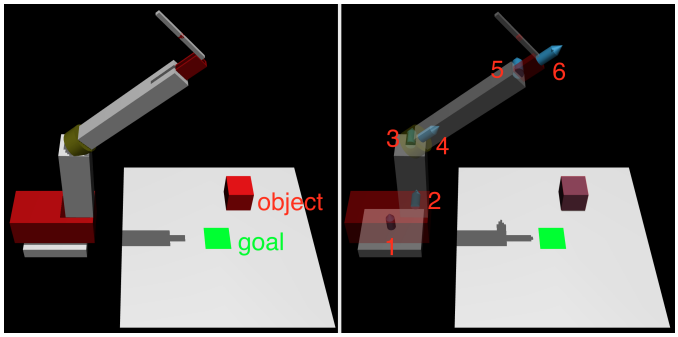

Fig. 1. Left: Robot Task. Right: Degree-of-Freedom.

The contributions made in this paper include:

- Demonstrating the data efficiency and asymptotic performance of learning at joint level vs. learning at end-effector level (with inverse jacobian), as well as the learning performance of position control vs. velocity control.
- Demonstrating the ability of the method to handle complex contact dynamics.

II. Related Work

Robotics manipulation is a well studied area with plenty of previous work. These work can be divided into two major camps: the classical trajectory optimization based approach, and the learning based approach.

A. Trajectory Planning

The trajectory planning based approach frame the task as a search problem. The task is defined as follows: given a configuration space, and an initial configuration point and a

goal configuration point, find the shortest path connecting the initial and goal configuration points. The configuration space has each dimension represented by a robot degree of freedom (joint angle), and points as obstacles. Popular algorithms such as Rapidly-exploring random tree LaValle [2], Probablistic Roadmap Kavraki et al. [1] uses a sampling mechanism to iteratively build the path, and optionally smoothes the path. While these solutions are probablistically complete, they are usually computationally expensive and are done offline. In order to perform closed-loop control, the trajectory planning needs to be re-computed frequently at each time step. This might not be feasible in practice depending on the dimensionality of the configuration space (robot degree of freedoms).

Furthermore, the trajectory planning approach assumes no contact between the robot and the environment. This does not handle pushing, where the end-effector and object experience complex contact dynamics.

B. Reinforcement Learning

On the other end of the spectrum, there is the learning approach which uses a reward function to reinforce a robot toward the desired behavior, such as in Lee et al. [3], Levine et al. [4]. There has been existing work that learn a visuomotor policy end-to-end. While previous work in this category produces a potentially more general policy with respect to different configurations, there are two main limitations with the current approaches.

- Firstly, the learning approach disregards classical robotics principals, such as forward kinematics and inverse kinematics. The learning approach commands joint velocity or position, and learns kinematics from scratch. This requires a large amount of data. One question this paper tries to answer is that, by leveraging existing analytical formulation to inverse-kinematics, can learning be more data-efficient?
- Secondly, the existing attempts deal with relatively simple dynamics moving from point A to point B, without involving complex contact dynamics. This paper addresses the this limitation by introducing pushing task, which involves complex contact dynamics between end-effectory and object.

C. Proximal Policy Optimization

Proximal Policy Optimization Schulman et al. [7] has achieved state-of-the-art results recently on continuous control tasks in mujoco environment. The PPO algorithm is similar to Trust Region Policy Optimization Schulman et al. [6] in the constrained policy update, but avoids the constrained optimization problem and is therefore easier to implement. The PPO algorithm computes the following loss. The key contribution of PPO is to bound the ratios between new and old policy action probability, to avoid a large update. In our experiment, we used $\epsilon = 0.2$. We chose entropy coefficient 0.001 to encourage exploration.

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \tag{1}$$

Where

$$r_t(\theta) = \frac{\pi_{\theta_{new}}(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} \tag{2}$$

The PPO paper also suggested an optional KL penalty, to further discourage a large policy update. The loss is defined as follows. In our experiment, we find a fixed β works better than the adaptive case, and added the KL penalty to our loss function. We chose $\beta = 0.01$ through hyper-parameter tuning.

$$L^{KLPEN}(\theta) = L^{CLIP} - \hat{\mathbb{E}}_t \left[\beta KL[\pi_{\theta_{old}}, \pi_{\theta_{new}}] \right] \tag{3}$$

D. Jacobian, Inverse Kinematics, Singularity

A jacobian maps from robot's joint velocity to end-effector velocity.

$$J = \frac{de}{d\theta} \tag{4}$$

$$= \begin{bmatrix} \frac{\partial e_x}{\partial \theta_1} & \frac{\partial e_x}{\partial \theta_2} & \dots & \frac{\partial e_x}{\partial \theta_M} \\ \frac{\partial e_y}{\partial \theta_1} & \frac{\partial e_y}{\partial \theta_2} & \dots & \frac{\partial e_y}{\partial \theta_M} \\ \frac{\partial e_z}{\partial \theta_1} & \frac{\partial e_z}{\partial \theta_2} & \dots & \frac{\partial e_z}{\partial \theta_M} \end{bmatrix} \tag{5}$$

In the equation above, e represents configuration (cartesian positions) of end-effector. θ represents joint angles. In this experiment, we want to control the end-effector velocity, based on which to compute the joint velocity. To achieve this, the inverse jacobian is needed:

$$d\theta = J^{-1} de \tag{6}$$

However, a jacobian is not necessarily a square matrix. The pseudo inverse is used:

$$J^{+} = (J^T J)^{-1} J^T \tag{7}$$

$$d\theta = J^{+} de \tag{8}$$

A robot reaches singularity, when two joints line up, so that the robot loses one degree of freedom. This cases the jacobian matrix to lose a rank, and becomes un-invertible. When a robot is near singularity, the jacobian value will become very large, and cause a small movement in the end-effector to result in large joint velocity. To alleviate the singularity problem, we use the damped least-squares Wampler [9] Nakamura and Hanafusa [5] method:

$$d\theta = J^T (JJ^T + \lambda^2 I)^{-1} de \tag{9}$$

Here we choose $\lambda = 0.01$. Even with the above formulation, the robot can still approach singularity, around which the joint velocity becomes large. In the experiments, we normalize the

joint velocities, such that the largest joint velocity element is one radians per second.

$$d\theta = [d\theta_1, d\theta_2, \cdots, d\theta_M] \tag{10}$$

$$d\theta_{normed} = [\frac{d\theta_1}{max(d\theta)}, \frac{d\theta_2}{max(d\theta)}, \cdots, \frac{d\theta_M}{max(d\theta)}] \tag{11}$$

III. Approach

A. Task Setup

The task involves pushing an object (cube: side 4cm, sphere: radius 2cm) from a starting position on a plane to a goal position. All objects are rigid, as is the limitation by the mujoco simulation environment. The episodes has 300 time steps, running at 50Hz. At each time step, the agent takes current state as input, and outputs an action to move the robot, collect the reward, and advance to the next state. The object is allowed six degrees of freedom with constant sliding friction coefficient of 1, and rotational friction coefficient of 0.1. The robot is a 6 degree of freedom robot arm. The end-effector is a thin stick of dimension (8cm x 0.5cm x 0.5cm). We chose this end-effector because the thin stick resembles a human fingertip. We hope that by following this biological inspiration, our robot is capable of versatile manipulation of the object.

B. Joint Control vs. End-effector Control

In previous learning approaches to robotics manipulation, the agent explores in joint space. As the task involves moving the end-effector, the robot essentially needs to learn inverse kinematics. An alternative formulation is to explore in end-effector space, and use inverse kinematics to compute joint velocity needed to achieve end-effector velocity. This way, the exploration is in a linear space and potentially easier to learn. The downside is more complex learning, and more computation per time step, and potential unstability when robot approaches singularity, joint limit, or gimbal lock.

- Joint control: the agent outputs joint velocity or position. The network output action is bounded to $[-1, 1]$, and denormalized to actual joint velocity/position ranges before executing on robot.
- End-effector control: the agent outputs end-effector velocity or position. The same bounding and denormalization is performed. The end-effector velocity is mapped to joint velocity using inverse jacobian with damped least-squares.

C. Velocity Control vs. Position Control

Recent learning approaches have used velocity control at joints. We experimented with both velocity and position control as follows. In the end-effector control, the control rules are slightly more complex than the joint control case.

In the case of joint control:

- Joint velocity control: the agent outputs target velocity (size: number of robot DOF) at each joint. We rely on mujoco's velocity and force controller (PD controller) to reach the target joint velocity.

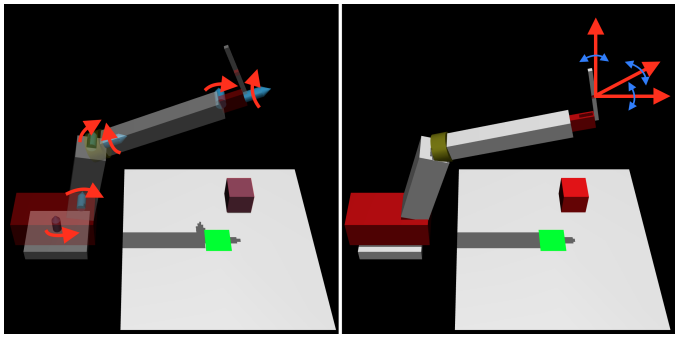

Fig. 2. Left: Joint Control. Right: End-effector Control.

- Joint position control: the agent outputs target position (size: number of robot DOF) at each joint. We rely on mujoco's position and velocity controller (PD controller) to reach the target joint position.

In the case of end-effector control:

- End-effector velocity control: the agent outputs target cartesian velocity (size of 3) and rotational velocity (size of 3) for the end-effector. It is mapped to joint velocity using inverse jacobian. We use mujoco's velocity and force controller (PD controller) to reach the target joint velocity.
- End-effector position control: the agent outputs target cartisian position (size of 3) and quaternion (size of 4) for the end-effector. At each time step, we use a proportional controller to find the translational velocity and quaternion derivative needed to reach the target position. We then map this to joint velocity using inverse Jacobian. Then we use mujoco's velocity and force controller (PD controller) to reach the target joint velocity. The reason for using quaternion is that, it avoids the gimbal lock issue introduced by using Euler angles.

For translational velocity:

$$v_p = \frac{p_{targ} - p}{dt} \tag{12}$$

$$\tag{13}$$

where p_{targ} is target position, and p is current position in cartesian coordinates.

For rotational velocity:

$$\omega = 2\dot{q}q^{-1} \tag{14}$$

$$= 2\dot{q}q^{-1} \tag{15}$$

$$= 2\frac{q_{targ}q^{-1}}{dt}q^{-1} \tag{16}$$

Where q_{targ} is the target orientation expressed in quaternion. q is the current quaternion. Both q_{targ}, q are unit quaternions.

D. Reward Function

This paper uses dense rewards to provide frequent feedback in the task. The reward is given per time step, and is the sum

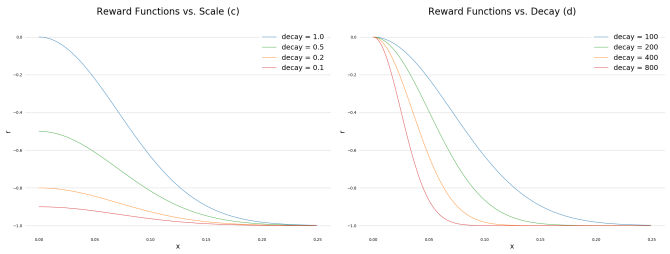

Fig. 3. Reward vs. Scale, Decay Parameters

of two parts:

- End-effector object reward: encourages end-effector to approach object. Reward is exponential to negative squared distance between end-effector and object. In order to push the object toward the goal, the end-effector must be in close proximity to the object.
- Object goal reward: encourages object to approach goal. Reward is exponential to negative squared distance between object and goal.

$$r_{dist} = c(e^{-d\|x_{targ}-x\|_2^2} - 1) \tag{17}$$

This reward formulation is intuitive. Each reward has two parameters and is easily tuned:

- Scale parameter c: controls the maximum reward per time step, if distance is zero. This can be set with the constraint that, the discounted sum of max rewards is within a reasonable range for the network to learn effectively. We chose this return range to be -30 to 30.
- Decay parameter d: controls the decay rate of reward, as distance increases. This can be set, such that at the reset (starting) position, the reward is sufficiently small as compared to the max reward. In our case, we chose 0.01.

In the case of end-effector control, a third reward is given by the mean-square of computed joint velocity from inverse jacobian. When a robot approaches singularity or joint limit, the computed joint velocity becomes large. Therefore we place a penalty on large joint velocity, before normalizing.

$$r_{singularity} = c(e^{-\mathbb{E}_t[\theta^2]} - 1) \tag{18}$$

Empirically, we found negative reward to be effective for neural network based learning. Theoretically, the learning is invariant to constant offset of reward as advantage is used. However, in practice subtracting rewards by a constant yields better learning. We hypothesize the reason being, in the beginning of the learning, the return is very negative, which results in larger gradient for the critic and more exploration. As the agent learns, the return gradually approaches zero, around which the critic loss becomes smaller and facilitates fine-tuning of a good policy by taking smaller gradient steps.

By giving frequent rewards at each time step, we hope to provide rapid feedback to the agent and learn the task faster. We experimented with sparse reward, with a $+1$ reward at episode end corresponding to successful task completion, and 0 otherwise. However, we had trouble learning with this sparse reward function, since the agent is trained from scratch and consumes no human demonstration.

E. Reinforcement Learning Algorithm

We used Proximal Policy Gradient as the reinforcement learning algorithm. We find the KL penalty crucial to the stability of our policy. Without the KL penalty, in our experiment, the agent would periodically take huge KL steps away from a good policy and completely destroyes its previously successful learning. We also experimented with adaptive KL penalty, and found it not effective.

F. Neural Network Architecture

The agent is made of a neural network, with two fully-connected layers of 64 neurons each. The network has the following input and output:

Input (called "Simple" features in experiment):

- Object center of mass (size: 3)
- Joint position normalized to $[-1, 1]$ (size: number of DOF)
- Sine (element-wise) of joint position (size: number of DOF)
- Cosine (element-wise) of joint position (size: number of DOF)

For end-effector control, we add the following input feature, in addition to the input above. Without the following features, the end-effector control does not learn. (called "Full" features in experiment)

- End-effector center of mass (size: 3)
- End-effector rotation (size: 3)
- Sine (element-wise) of end-effector position (size: 3)
- Cosine (element-wise) of end-effector position (size: 3)

We added sine and cosine, because these are used in rotation computation. By adding this non-linear transform as features, we hope the learning becomes easier. These features can be easily computed during robot execution time.

Output:

- target velocity or position of joint or end-effector. The neural network outputs a mean vector and a log standard deviation vector. The standard deviations are stand-alone variables directly learned via back-prop completely separate from the neural network. At training time, the output is chosen by randomly sampling from a normal distribution with zero mean and unit standard deviation, then scaling by standard deviation and offsetting by the mean. This sampling approach helps with exploration during training. For the exact output definition, see "Joint control vs. end-effector control" for more details.

We experimented with different number of hidden layers, and found two layers to be expressive enough to encode the pushing policy from varying start to fixed goal position. In our architecture, we have actor and critic network using separate

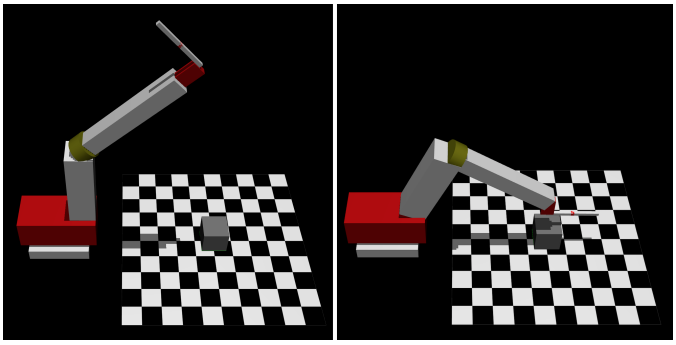

Fig. 4. Left: far configuration. Right: near configuration.

sets of weights. We experimented with shared hidden layer weights between the two networks, and found it does not improve performance.

IV. Experiments

All experiments are done in Mujoco simulation environment Todorov et al. [8] with 50 Hz control frequency.

A. Metric

We learned the task of pushing an object from start to goal position. The performance is evaluated on the euclidean distance from the final object position to the goal position. The lower distance corresponds to better performance. We compared the performance between position vs. velocity control, and between joint vs. end-effector control. We evaluated the performance on two initial end-effector positions. The reason for the near configuration is to avoid unnecessary exploration which might add variance to our experiment.

- Far: with all joints in neutral position (center of joint range).
- Near: with end-effector with a short distance directly above origin $(0, 0)$. The end-effector is closer in proximity to the object.

B. Generalization to Varying Start Configurations

We first validated our algorithm on a fixed-start, fixed-goal position. Here we fix start position and goal position of the cube. Our agent is able to learn the task successfully with low final distance. Then we proceeded to vary the start position of the cube, while fixing the goal position. At the beginning of each episode, we uniformly sample a radius r in range of $[0cm, 7.5cm]$, and an angle θ in range of $[0, 2\pi]$. In other words, we place the cube randomly in a circle, with uniform random probability of distance from origin, and angle from x-axis. We experimented with curriculum learning, where we gradually increase the circle size in 1cm increments (sampling the angle in range of $[0, 2\pi]$), and did not find it to be helpful to learning.

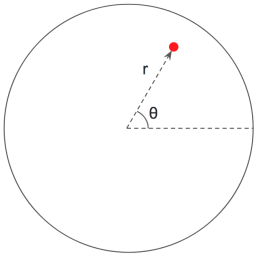

Fig. 5. Sampling the initial object position inside circle

C. Position vs. Velocity Control, Joint vs. End-effector Control

We evaluated performance of the fource control methods (joint velocity control, joint position control, end-effector vel control, end-effector position control). Joint velocity control learned the fastest. For end-effector control, the result is less clear. Velocity control performed better in near configuration, while position control performed better in far configuration.

D. Simple vs. Full Features

We also experimented with different input features for end-effector control. For the exact definition of the features, please refer to the secion on "Neural Network Architecture". We find the additional end-effector features only useful for position control. In velocity control with near configuration, the additional features didn't make a significant difference in performance. In velocity control with far configuration, the agent did not converge regardless of the input features.

E. Sphere Object

We used a sphere object to further test the ability to handle complex contact dynamics. A sphere has rotational inertia. Therefore more delicate control is required to make a sphere stop at the goal position. We performed learning with joint control using both velocity and position control. Velocity control is able to successfully learn the task with less than 1cm in final distance to goal, while position control is not able to learn the task.

V. Discussions

Across all experiments, joint velocity control demonstrates the fastest learning, with eventual performance on par with position control in some cases. This is likely enabled by velocity control's rapid feedback, allowing it to efficiently explore the surrounding. For joint control, it is much easier for the robot to command joint velocity than joint position. For position control, at each time step a target position is commanded. Therefore the robot might not have enough time to reach the desired position. Where for velocity control, the robot can rapidly change its target velocity at e ach time step.

End-effector control does not outperform joint control, despite the linear exploration in cartesian and rotational space. This is likely due to the constraints posed by singularity, joint limits, as well as gimbal lock for end-effector position control. Here are a few future directions to take:

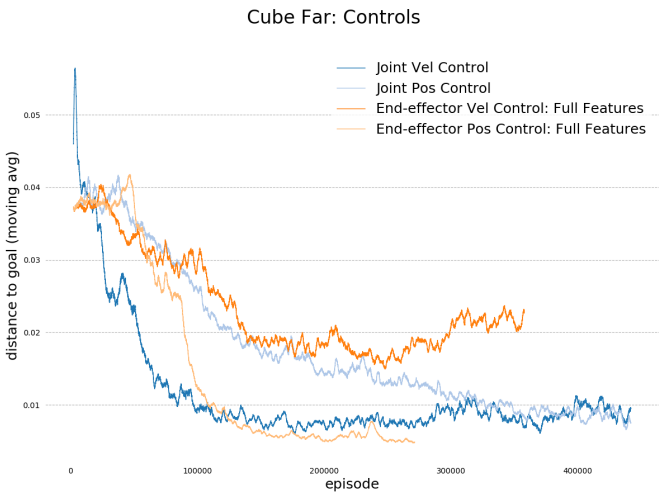

Fig. 6. Final distance to goal for far configuration

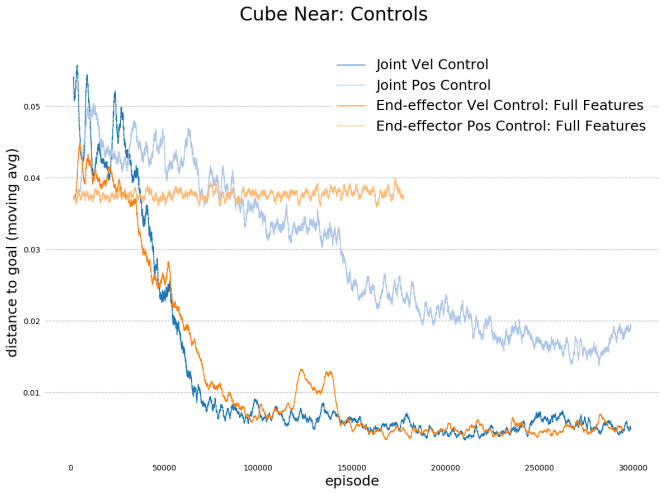

Fig. 7. Final distance to goal for near configuration

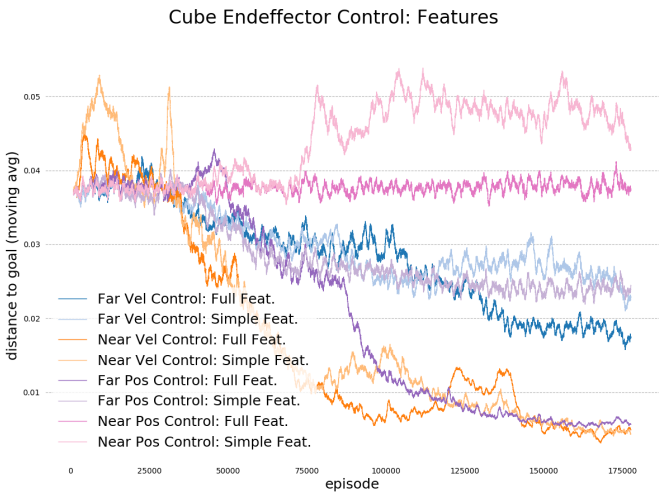

Fig. 8. Final distance to goal for end-effector control with different input features

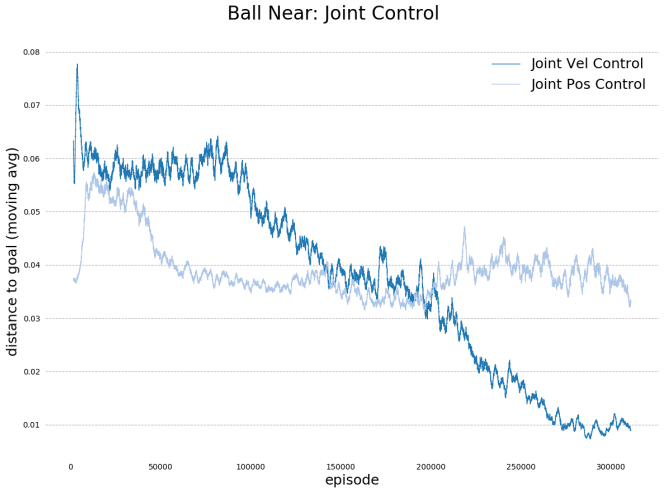

Fig. 9. Final distance to goal for pushing sphere using joint control

- Provide appropriate teaching signal with regard to singularity, joint limits, and gimbal lock constraints, such that the agent can receive consistent teaching signal, learns to operate within these constraints, and to explore effectively.
- Apply recurrent neural network to take into account temporal dependence between states for better policy.
- Use continus image frames as input, and train additional vision layer end-to-end for simulation to real transfer.

ACKNOWLEDGMENTS

Special thanks to Ajay Mandlekar, Michelle Lee, Julian Gao, and Jeannette Bohg for their help.

REFERENCES

[1] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4):566–580, 1996.

[2] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[3] Alex X Lee, Sergey Levine, and Pieter Abbeel. Learning visual servoing with deep features and fitted q-iteration. *arXiv preprint arXiv:1703.11000*, 2017.

[4] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

[5] Yoshihiko Nakamura and Hideo Hanafusa. Inverse kinematic solutions with singularity robustness for robot manipulator control. *ASME, Transactions, Journal of Dynamic Systems, Measurement, and Control*, 108:163–171, 1986.

[6] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference*

on Machine Learning (ICML-15), pages 1889–1897, 2015.

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[8] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012.

[9] Charles W Wampler. Manipulator inverse kinematic solutions based on vector formulations and damped least-squares methods. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):93–101, 1986.