CS230

GPU-Accelerated Deep Reinforcement Learning Simulation for Vision-Based Robotic Manipulation

Kihiro Bando
Department of Aeronautics & Astronautics
bandok@stanford.edu

Kingway Liang
Department of Mechanical Engineering
kingway@stanford.edu

Victor Zhang
Department of Mechanical Engineering
zhangvwk@stanford.edu

Abstract

Deep reinforcement learning for robotic manipulation relies on an important step of data collection, especially in continuous action space: whether a neural network is used to estimate the Q-function [4] or the policy [9], a large amount of transitions (s, a, r, s') must be collected so that the optimization step yields meaningful results. Using Proximal Policy Optimization (PPO) [12] on the GPU-based physics simulator FleX [6] for the robotic task of reaching an object yielded promising results in terms of time to convergence, both with and without visual information.

1 Introduction

Robotic manipulation methods that can be generalized to previously unseen objects is one of the largest open problem in the field of robotics. Because the use of traditional optimal control techniques has been found to be very difficult to implement for vision-based control, deep reinforcement learning (DRL) has been the focus of more recent research as a solution to this problem.

Given that the sim-to-real transition has shown promising results in robotics [3], it is of time-saving interest to collect data via simulation in a distributed fashion [2, 7] to accelerate the process. However, while distributed training is often done on a GPU, simulation is not: the goal of our project is thus to perform GPU-accelerated data collection for DRL of robotic manipulation tasks using the GPU-based physics simulator FleX as a cost-saving alternative to using hundreds of CPUs [4].

While robotic manipulation tasks can in principle be generalized to any robotic arm, this joint project with the Stanford Vision and Learning Lab (SVL) required us to focus on the Sawyer robot [11] designed by Rethink Robotics. Since the goal of the project was to demonstrate the higher efficiency obtained with GPU-based simulations, the work was split between training with data collected using the CPU-based physics simulator PyBullet and the code-base used in [10] and obtaining first DRL results for robotics using FleX. The following tasks were investigated, on the Sawyer robot:

- reaching a constant target (in FleX);
- reaching a randomized target (in FleX);
- verifying the performance of Deep Q-Learning for learning grasping [10] (in PyBullet).

For the first two tasks in FleX, depending on the chosen observation space (see Section 3), the input is either low-dimensional features or includes RGBD images taken from a camera located in front of the robot, as depicted in FIGURE. In the latter case, the image is encoded using a convolutional neural network (CNN) before being concatenated with the remaining observations. These are fed into two separate feed-forward networks for actor (policy) and critic (value). The output for the former is a vector of the actions means, and the estimated value function (scalar) for the latter. The reinforcement learning (RL) algorithm used is the PPO (see Section 4).

CS230: Deep Learning, Winter 2019, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

For the grasping task in PyBullet, the algorithm approximates the Q-function with a neural network by minimizing the Bellman error. The network consists of convolutional layers to process image observations and fully connected layers to process and integrate the action (see Section 4).

2 Related work

There exists a vast array of works that explore parallelizing the simulations and the training on CPUs, such as [7] and [2]. Scaling benefits have been observed by scaling up environment simulations on many hundreds of CPUs. [5] extended the approach by additionally performing policy training on the GPU. In their work, they propose using a GPU-accelerated reinforcement learning simulator to bring the benefits of the existing parallelism of the GPU to the simulations as well. The grasping task has been studied most notably in [10], wherein several learning algorithms are compared and simulation benchmarks are presented.

3 Modeling the problem

The reaching task is modelled as follows:

- the state space using low-dimensional information only $\mathcal{S}_{\text{low}} \subset \mathbb{R}^6$ is composed of the end-effector pose and the target pose relative to the end-effector;
- the state space with additional vision $\mathcal{S}_{\text{high}} \subset \mathbb{R}^{3+n\times n\times 4}$ is composed of the end-effector pose and the $n \times n \times 4$ RGBD image taken from a camera placed over the objects as shown on Figure 1 ($n = 84$ in our experiments);
- the action space $\mathcal{A} \subset \mathbb{R}^3$ is composed of the command translation velocities of the end-effector in each direction;
- the reward is defined as the l_1-distance between the end effector and the target pose.

A network is used to compute the mean and log-standard deviation (learned independently from the state) of a Gaussian distribution for each action's component. Instantaneous velocity of the end-effector is then sampled using this distribution and sent into the FleX environment.

For the grasping task, the observation is a $64 \times 64 \times 3$ RGB image, action space is 4 dimensional, commanding change in gripper pose (displacement in all three translations and the rotation about the z axis). Sparse reward is given for grasping episodes: 1 for successfully grasping at least 1 object and 0 otherwise.

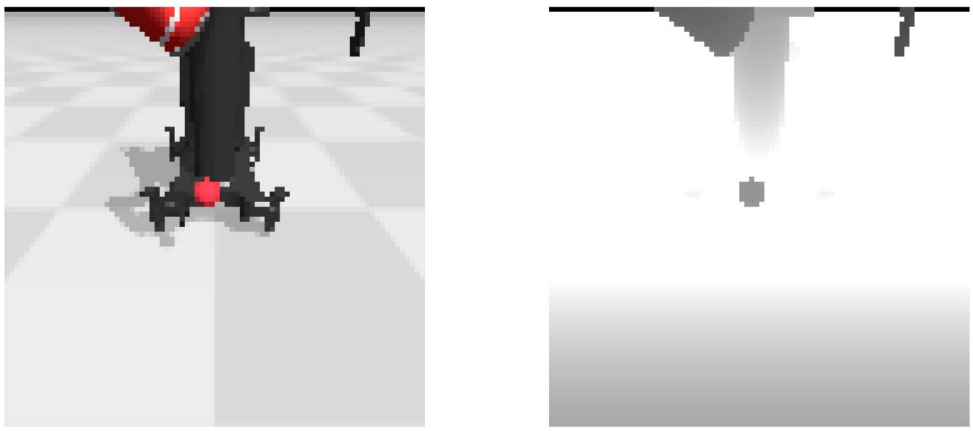

Figure 1: RGBD image as observation for the high-dimensional state space.

4 Methods and network architecture

The method used with the FleX environment is PPO [12], used in [5] for locomotion tasks. This algorithm is an on-policy algorithm similar to classical policy gradient that alternates between rollouts of data collection and optimization steps. In policy gradient, a policy network parameterized by θ is optimized by maximizing the following objective:

$$L^{\text{PG}}(\theta) = \mathbb{E}\Big[\log \pi_\theta(a_t|s_t)\hat{A}_t\Big].$$

Hyper-parameters	Value
Time-steps per step call	2
Episode length	500
Time-steps per batch (for 256 agents)	128
Number of optim. epochs	20
Mini-batch size	16
Discount factor γ	0.99
Clip param. ε	0.2
Target KL	0.02

Table 1: Hyperparameters

The use of the advantage function $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$ (for the notations, see [8]) is a common choice to reduce the variance of the expectation. The main idea of PPO is to mitigate updates in parameters that would substantially change the actual policy. The policy loss is replaced by

$$L(\theta) = \mathbb{E}\left[\min\Big(r_t(\theta),\ \text{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\Big)\hat{A}_t\right],$$

where $r_t(\theta) = \dfrac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ and the expectation is estimated over sampled trajectories. This essentially avoids large updates in θ in the direction of the surrogate objective gradient while maintaining the knowledge of directions that would make the objective worse. This is supplemented in an actor-critic fashion with a critic network that predicts the value function, which is then used to compute an estimator of the advantage function $\hat{A}_t$ in the former objective. The total loss is the combination of the policy loss and a squared-error loss for the critic network where the target is computed using temporal difference. Finally, the learning rate is adapted with respect to a target KL divergence between the new and old policy at each mini-batch optimization step.

After much experimenting, we found that the actor network yielding the best results for our tasks is comprised of two fully connected layers with 64 hidden units and a ReLU activation functions, followed by a fully connected layer with $\dim(\mathcal{A})$ hidden units. We pass a $\tanh$ activation function to have the output normalized between -1 and 1. The critic network's architecture is identical to the actor network, except for the last layer being a dense linear layer with a scalar output.

When vision is enabled, the CNN architecture used takes as input $84\times84\times4$ RGBD images, and performs the following operations: 2D convolution with 16 filters of kernel size 8 by 8 and a stride of 4, followed by a 2D convolution with 32 filters of kernel size 4 by 4 and a stride of 2, with ReLU activations. The outputs are flattened and passed into a fully connected layer of size 128, concatenated with low-dimensional features, and finally fed into the actor's and critic's feed-forward networks. Batch normalization is systematically applied.

The main hyperparameters are summarized in Table 1.

For the grasping task, the Bellman error is estimated by the equation below, where Q'_θ is a lagged version of the learned Q-function that is 50 gradients behind. This decorrelates the $\max$ operator from the learned Q-function. By minimizing this error for all states (in practice, only sampled states), the optimal Q-function can be learned, which induces the optimal policy.

$$\mathcal{E} = \frac{1}{2}\mathbb{E}_{s,a}\Big[\Big(Q_\theta(s,a) - (r(s,a) + \gamma Q_{\theta'}(s', \operatorname*{argmax}_{a'} Q_\theta(s',a')))^2\Big)\Big]$$

5 Results and discussion

All experiments were run on a single machine equipped with an Nvidia GeForce GTX 1080 GPU.

The results for the low-dimensional state inputs are presented first. As in [5], each rollout is made of a fixed number of collected transitions. This reinforcement learning problem is fairly easy which justifies the fairly simple architecture used. This can be compared with the network used in [9] of 3 layers with 256 units. The main metric of interest is the return averaged over all episodes simulated so far and the time needed to finish learning. The agent needs to learn to go towards the target and stay close to it. It was found to be important to set a sufficiently long episode length and not to reset the agent once it went sufficiently close to the target. Indeed, this helps the agent learn the second component of this task which is staying close to the target upon reaching it the first time. Finally, the l_1-norm reward gave much better results compared to the l_2-norm. This is related to the fact that the latter doesn't encourage the agent to continue moving towards the target at a close distance as much as the former.

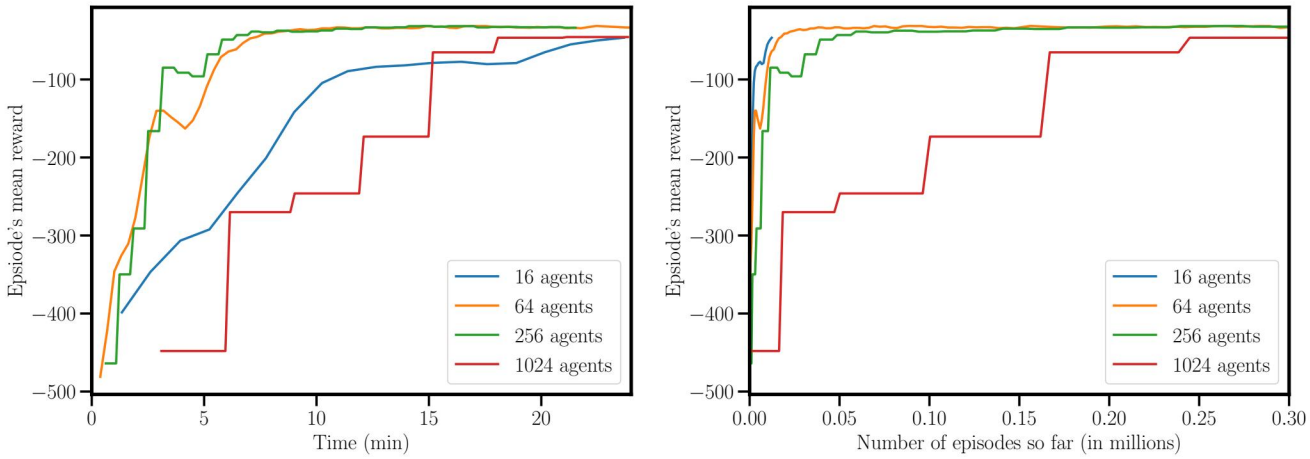

Figure 2: Scaling analysis for the randomized reaching task. The mean return is shown as a function of wall-clock time (left) and number of episodes collected so far (right).

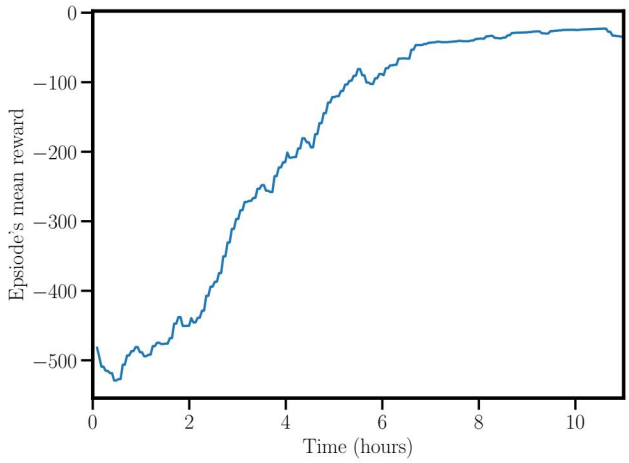

Figure 3: Preliminary results for vision-based learning. Mean return as a function of wall-clock time.

The advantage of the FleX environment is the ability to generate massively parallel simulations on a small number of hardware resources. The scaling analysis is presented in Figure 2. This shows that, at a given number of transitions per batch, there is an optimal balance between number of agents and transitions per agent (the product of the two being constant) that minimizes the wall-clock time to reach completion of the task. When too many agents are present, the optimizer sees only small portions of a full episode. When too few agents are present, each roll-out takes longer to simulate due to the large number of time-steps to complete. By taking advantage of the parallel architecture, it is much faster to simulate a greater number of episodes using more agents at a fixed compute time. The results can be compared with [9] which uses Deep Deterministic Policy Gradient (DDPG) to learn several robotic manipulation tasks. They require about 100k episodes to achieve completion but with a bigger network and larger input space.

Learning with vision inputs increases the number of parameters considerably, the size of the networks as well as the computational cost since it requires rendering. While the agent eventually manages to learn from this more complex observation space (as shown on Figure 3), the learning process revealed to be quite sensitive to the architecture. Often times, the mean output by the network reaches -1 or 1 in a few iterations and it becomes very difficult to sample opposite actions. Even increasing the initial standard deviation or decreasing the action frequency did not help. Moreover, batch normalization was necessary for stabilization and alternatives such as layer normalization were less effective. The sizes of the networks were taken from [1] as a baseline.

For grasping, off-policy episodes are collected with a scripted policy that randomly samples from the action space and overwrites the vertical velocity to -1 with 90% probability. The Q-function is then trained with these episodes to learn a grasping policy, which is evaluated throughout training by measuring the success rate of grasping. Early training shows that while the reward of the learned policy increases initially, it decreases with more training steps, as shown on Figure 4. The pattern was shown in several other configurations and is likely due to the number of training steps being disproportionately larger than the limited amount of episodes collected on a single machine, thus leading the network to overfit and not be able to generalize to unseen examples. After multiple trials, it is found the learned policy reaches optimal performance when the number of training steps is twice the number of episodes collected, as shown on Figure 5. However, even with the right training steps, the policy performance is significantly lower than the benchmarks presented in [10]. This is attributed to the simulation setting where originally the same objects were being spawned at each episode, thus resulting in a non-heterogeneous distribution of collected samples. After setting the environment to spawn random graspable objects at each episode, the policy learned from these episodes reaches a grasping success rate of 78%, comparable to the results presented in [10].

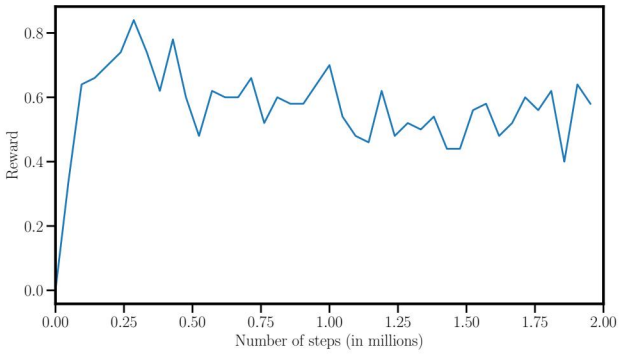

Figure 4: Reward with too many training steps

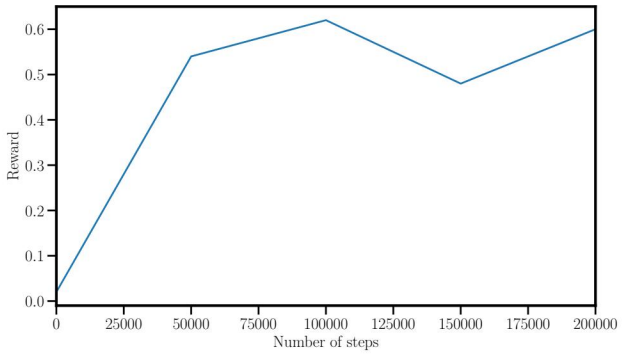

Figure 5: Reward with non-randomized objects.

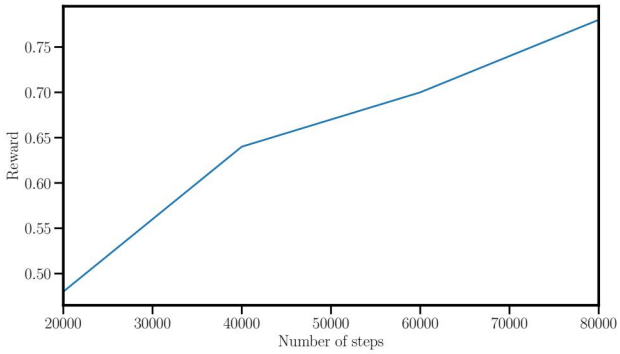

Figure 6: Reward with an appropriate number of iterations and object randomization.

6 Conclusion and future works

Our work has shown that DRL simulation for robotic manipulation using GPU is successful for reaching tasks and serves as a strong proof-of-concept for much more of what it has to offer as a better alternative to using hundreds of CPUs, in time and cost. Though only DQN was tested for grasping, PPO is shown to work well for reaching and generally seems like a more viable alternative for more complex tasks since the action space is continuous. Had we had more time, we would have tried more complex network architectures, such as updating the actor and critic networks through backpropagation through time with LSTM rollouts, as implemented in [1], so as to include more past information in the observation when using vision.

Had we had more resources, we would have trained our tasks on multiple GPUs, done a scaling analysis in terms of number of used GPUs, as well as compared the results with those obtained on a single-GPU. This would be a good benchmark in the domain of robotic manipulation – and more specifically, the reaching task.

7 Contributions

Kihiro and Victor contributed to the customization of the Sawyer robot environment in FleX, the implementation of the networks used in PPO for the reaching task, and the writing of this report.

Kingway contributed to the customization of the Sawyer robot environment in PyBullet, the implementation of a scripted policy used in DQN for the grasping task, and the writing of this report.

References

[1] Linxi Fan, Yuke Zhu, Jiren Zhu, Zihua Liu, Orien Zeng, Anchit Gupta, Joan Creus-Costa, Silvio Savarese, and Li Fei-Fei. Surreal: Open-source reinforcement learning framework and robot manipulation benchmark. In *Conference on Robot Learning*, 2018.

[2] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. In *International Conference on Learning Representations*, 2018.

[3] Stephen James, Paul Wohlhart, Mrinal Kalakrishnan, Dmitry Kalashnikov, Alex Irpan, Julian Ibarz, Sergey Levine, Raia Hadsell, and Konstantinos Bousmalis. Sim-to-Real via Sim-to-Sim: Data-efficient Robotic Grasping via Randomized-to-Canonical Adaptation Networks. *arXiv e-prints*, page arXiv:1812.07252, Dec 2018.

[4] Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, and Sergey Levine. Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation. *CoRR*, abs/1806.10293, 2018.

[5] Jacky Liang, Viktor Makoviychuk, Ankur Handa, Nuttapong Chentanez, Miles Macklin, and Dieter Fox. Gpu-accelerated robotic simulation for distributed reinforcement learning, 2018.

[6] Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Unified particle physics for real-time applications. *ACM Transactions on Graphics (TOG)*, 33(4):104, 2014.

[7] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively Parallel Methods for Deep Reinforcement Learning. *arXiv e-prints*, page arXiv:1507.04296, Jul 2015.

[8] OpenAI. Openai: Spinning up with deep rl. `https://spinningup.openai.com/en/latest/spinningup/rl_intro.html#advantage-functions`.

[9] Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, Vikash Kumar, and Wojciech Zaremba. Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research. *arXiv e-prints*, page arXiv:1802.09464, Feb 2018.

[10] Deirdre Quillen, Eric Jang, Ofir Nachum, Chelsea Finn, Julian Ibarz, and Sergey Levine. Deep reinforcement learning for vision-based robotic grasping: A simulated comparative evaluation of off-policy methods. *IEEE International Conference on Robotics and Automation*, 2018.

[11] Rethink Robotics. Sawyer collaborative robots for industrial automation. `http://www.rethinkrobotics.com/sawyer/`.

[12] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.