A Journey to Cool Robotic Cognition

K. Park

September, 2024

Contents

1 Introduction **2**

2 Foundation of Reinforcement Learning **3**
2.1 Bellman Equations . . . 3
2.1.1 Bellman Equation in MRP . . . 3
2.1.2 Bellman Equation in MDP . . . 4
2.1.3 Bellman Optimality Equation . . . 5
2.2 Dynamic Programming . . . 6
2.3 Monte Carlo Method . . . 7
2.4 Temporal Difference . . . 8
2.5 Multi-step Bootstrapping . . . 8
2.6 Policy Gradient Method . . . 9

3 Modern Reinforcement Learning **10**
3.1 Proximal Policy Optimization . . . 10

4 Implementations **13**
4.1 PPO: cheetah-v5 . . . 13

5 Robotic Applications **14**
5.1 Kinova Arm with PPO . . . 14
5.1.1 Grasping Task . . . 14
5.1.2 Lessons . . . 21

Chapter 1

Introduction

Aims of this journey are first, to preview ideas of robot learning including reinforcement learning (RL) second, to implement well-known algorithms and the last, to propose & to extend ideas to multi-modal system (perceiving states using RGB-D, lidar, radar, haptic whatever and moving actuators). I believe RL would inspire some ideas to develop artificial psychological entity, so called robotic mind. No one knows what mind is, but so what? It is not necessary to be similar as the organism's one. This journey has a couple of stages to address the goal in my life(?).

- Having solid background of robot learning.
- Understanding idea of modern RL algorithms such as PPO, SAC or diffusion policy model.
- Implement and test the ideas in simulation environment.
- Robot implementation with ROS2.
- R&D cool RL algorithm including 'local' perception (e.g. in robot's view).
- Evaluate what I did, and contemplate the missing points.

The stages above should cycle several times to achieve better understanding, as well as to devise new paradigms (maybe, 3+1 decomposition to solve Einstein's field equation is applicable but it needs 'scientific' interpretation).

Chapter 2

Foundation of Reinforcement Learning

Followings are key references of this chapter.

- Reinforcement Learning, an introduction, written by R. S. Sutton & A. G. Barto
- Reinforcement Learning Lecture Slides, written by O. Wallscheid, Paderborn Univ.
- Reinforcement Learning Lecture Slides, written by D. Silver, UCL

2.1 Bellman Equations

Note that it is also called as Bellman Expected Equation, except the optimal version.

2.1.1 Bellman Equation in MRP

Let V be the value function with respect to Markov reward process (MRP). Then,

$$\begin{aligned} V(x_k) &= \mathbf{E}\left[G_k | X_k = x_k\right] \\ &= \mathbf{E}\left[R_{k+1} + \gamma G_{k+1} | X_k = x_k\right] \\ &= \mathbf{E}\left[R_{k+1} | X_k = x_k\right] + \gamma \sum_g g\mathbf{p}(g|x_k) \\ &= \mathbf{E}\left[R_{k+1} | X_k = x_k\right] + \gamma \sum_g \sum_r \sum_{x_{k+1}} g\mathbf{p}(g, r, x_{k+1}|x_k) \end{aligned} \tag{2.1}$$

Employing multiplication rule such that

$$\mathbf{p}(x, y|z) = \frac{\mathbf{p}(x, y, z)}{\mathbf{p}(z)} \tag{2.2}$$

Thus, the second term in Equation 2.1 turns to

$$\begin{aligned}
\gamma \sum_g g\mathbf{p}(g|x_k) &= \gamma \sum_g \sum_r \sum_{x_{k+1}} g\mathbf{p}(g, r, x_{k+1}|x_k) \\
&= \gamma \sum_g \sum_r \sum_{x_{k+1}} g\mathbf{p}(g|x_{k+1}, r, x_k)\mathbf{p}(r, x_{k+1}|x_k) \\
&= \gamma \sum_g \sum_r \sum_{x_{k+1}} g\mathbf{p}(g|x_{k+1})\mathbf{p}(r, x_{k+1}|x_k) \\
&= \gamma \sum_g \sum_{x_{k+1}} g\mathbf{p}(g|x_{k+1})\mathbf{p}(x_{k+1}|x_k) \\
&= \gamma \sum_{x_{k+1}} \mathbf{E}\left[G_{k+1}|X_{k+1} = x_{k+1}\right]\mathbf{p}(x_{k+1}|x_k) \\
&= \gamma \sum_{x_{k+1}} V(x_{k+1})\mathbf{p}(x_{k+1}|x_k)
\end{aligned} \tag{2.3}$$

Remember marginalization,

$$\sum_r \mathbf{p}(r, x_{k+1}|x_k) = \mathbf{p}(x_{k+1}|x_k) \tag{2.4}$$

Keep in mind Markov property, memory-less. It means that $\mathbf{p}(x|y, z, l) = \mathbf{p}(x|y)$ if z, l are determined in the previous stage. Therefore,

$$\begin{aligned}
V(x_k) &= \mathbf{E}\left[G_k|X_k = x_k\right] \\
&= \mathbf{E}\left[R_{k+1}|X_k = x_k\right] + \gamma \sum_{x_{k+1}} \mathbf{p}(x_{k+1}|x_k)V(x_{k+1}) \\
&= \mathcal{R}_x + \gamma \sum_{x_{k+1}} \mathbf{p}(x_{k+1}|x_k)V(x_{k+1})
\end{aligned} \tag{2.5}$$

2.1.2 Bellman Equation in MDP

Now, the action u performs given state x: decision-making. From the definition of the value function,

$$\begin{aligned}
V_\pi(x_k) &= \mathbf{E}\left[G_k|X_k = x_k\right] \\
&= \sum_{u_k} \sum_g g\mathbf{p}(g|u_k, x_k)\mathbf{p}(u_k|x_k) \\
&= \sum_{u_k} Q_\pi(u_k, x_k)\pi(u_k|x_k)
\end{aligned} \tag{2.6}$$

using marginalization. Now action-state value Q is,

$$\begin{aligned} Q_\pi(x_k, u_k) &= \mathbf{E}\left[G_k | X_k = x_k, U_k = u_k\right] \\ &= \mathbf{E}\left[R_{k+1} | X_k = x_k, U_k = u_k\right] + \gamma \sum_g \sum_{x_{k+1}} g\mathbf{p}(g, x_{k+1} | x_k, u_k) \\ &= \mathcal{R}_x^u + \gamma \sum_g \sum_{x_{k+1}} g\mathbf{p}(g | x_{k+1}, x_k, u_k)\mathbf{p}(x_{k+1} | x_k, u_k) \\ &= \mathcal{R}_x^u + \gamma \sum_g \sum_{x_{k+1}} g\mathbf{p}(g | x_{k+1})\mathbf{p}(x_{k+1} | x_k, u_k) \\ &= \mathcal{R}_x^u + \gamma \sum_{x_{k+1}} \mathbf{E}\left[G_{k+1} | X_{k+1} = x_{k+1}\right] \mathbf{p}(x_{k+1} | x_k, u_k) \\ &= \mathcal{R}_x^u + \gamma \sum_{x_{k+1}} \mathbf{p}(x_{k+1} | x_k, u_k) V_\pi(x_{k+1}) \end{aligned} \tag{2.7}$$

Please keep in mind "memory-less"!! By employing Equation 2.7 and Equation 2.6, Bellman equation can be described in terms of Q or V alone. For example, substitute Equation 2.6 to Equation 2.7, then

$$Q_\pi(x_k, u_k) = \mathcal{R}_x^{u_k} + \gamma \sum_{u_{k+1}} \sum_{x_{k+1}} \mathcal{P}^u_{x_k x_{k+1}} Q_\pi(x_{k+1}, u_{k+1})\pi(u_{k+1} | x_{k+1}) \tag{2.8}$$

2.1.3 Bellman Optimality Equation

A theorem tell us that following is satisfied for any finite MDP.

- There exists an optimal policy that is always better or equal to all other policies.
- All optimal policies achieve same optimal state value.
- All optimal policies achieve same optimal state-action value.

From the definition of the value function,

$$\begin{aligned} \max_\pi V_\pi(x_k) &= \max_\pi \mathbf{E}_\pi\left[G_k | X_k = x_k\right] \\ &= \max_\pi \sum_{u_k} Q_\pi(x_k, u_k)\pi(u_k | x_k) \end{aligned} \tag{2.9}$$

Let's assume that the policy is deterministic. Then, the optimal policy would determine the particular actions that should be performed based on the specific states. At step k, every optimal actions has same Q values which are maximum, with same probability. Thus,

$$\begin{aligned} \max_\pi V_\pi(x_k) &= \max_\pi \sum_{u_k} Q_\pi(x_k, u_k)\pi(u_k | x_k) \\ &= \sum_{u_k} Q_{\pi^*}(x_k, u_k)\pi^*(u_k | x_k) \\ &= Q_{\pi^*}(x_k, u_k) \\ &= \max_{u_k} Q_\pi(x_k, u_k) \end{aligned} \tag{2.10}$$

Recall Equation 2.7, the optimality equation for state value becomes

$$\begin{aligned}\max_{\pi} V_{\pi}(x_k) &= V_{\pi^*}(x_k) \\ &= \max_{u_k} Q_{\pi}(x_k, u_k) \\ &= \max_{u_k} \left(\mathcal{R}_x^u + \gamma \sum_{x_{k+1}} \mathbf{p}(x_{k+1}|x_k, u_k) V_{\pi}(x_{k+1}) \right) \\ &= \max_{u_k} \mathcal{R}_x^u + \gamma \sum_{x_{k+1}} \mathbf{p}(x_{k+1}|x_k, u_k) V_{\pi^*}(x_{k+1})\end{aligned} \tag{2.11}$$

For state-action value,

$$\begin{aligned}Q_{\pi^*}(x_k, u_k) &= \max_{\pi} \left(\mathcal{R}_x^u + \gamma \sum_{x_{k+1}} \mathbf{p}(x_{k+1}|x_k, u_k) V_{\pi}(x_{k+1}) \right) \\ &= \mathcal{R}_x^u + \gamma \sum_{x_{k+1}} \mathbf{p}(x_{k+1}|x_k, u_k) Q_{\pi^*}(x_{k+1}, u_{k+1})\end{aligned} \tag{2.12}$$

Unlike Equation 2.11, the first term in Equation 2.12 isn't maximized. What the fuck? Why? Note that this is optimality equation for state=action, which indicates the action u_k at current step k is already determined. In case Equation 2.11, the action u_k should be determined after the state is given.

2.2 Dynamic Programming

Main concept of dynamic programming (DP) is to separate a complex problem into subproblems. Let's look at a couple of computation techniques beforehand. First of all, Richardson iteration is one way to solve Bellman equation in matrix form. It has $\mathcal{O}(kn^2)$ complexity, where k is the number of iterations, so it is faster than inverse matrix (complexity of $\mathcal{O}(n^3)$) if the matrix has bigger size and is sparse. The method finds all optimal state value simultaneously utilizing same residual vector (synchronous full-backup). On the other hand, in-place method using Equation 2.11, it updates the values sequentially. For example, state value of $V(x_3)$ is updated from the terminal state x_4 and it repeats until it reaches out the initial state.
For given dynamics $\mathbf{p}(x_{k+1}|x_k, u_k)$, essence of DP is a two-fold, policy evaluation, and policy improvement. Repeat this process until the iteration algorithm reaches out the convergence. Followings are key formulations. For the policy evaluation,

$$\begin{aligned}V_{\pi}(x_k) &= \mathbf{E}\left[G_k|X_k = x_k\right] \\ &= \sum_{u_k} \pi(u_k|x_k) \left(\mathcal{R}_x^u + \gamma \sum_{x_{k+1}} \mathbf{p}(x_{k+1}|x_k, u_k) V_{\pi}(x_{k+1}) \right)\end{aligned} \tag{2.13}$$

For the greedy policy improvement,

$$\begin{aligned}\pi'_j(x_k) &= \text{argmax}_u Q_j(x_k, u) \\ &= \text{argmax}_u \left(\mathcal{R}_x^u + \gamma \sum_{x_{k+1}} \mathbf{p}(x_{k+1}|x_k, u_k) V_\pi(x_{k+1}) \right)\end{aligned} \tag{2.14}$$

In control problem, there are two main branches: policy iteration and value iteration. Those have a subtle difference of strategy to find optimal solution. The **policy iteration** needs evaluation and improvement, which are repeated steps to reach out the optimum. The evaluation is conducted by Bellman (expectation) equation. In the improvement stage, a policy picks specific actions that maximizes the value. On the other hand, **value iteration** merges the two steps in a one step. It employs Bellman optimality equation to maximize the value function.

2.3 Monte Carlo Method

Monte Carlo (MC) approach replaces known dynamics to sampling, it collect data (states, actions, rewards, etc) from a series of experiments to estimate state values and state-action values. Let the initial policy generates a sequence such that $\pi_0 : x_0, u_0, r_1, x_1, u_1, r_2, ..., x_T, u_T, r_{T+1}$ at the $j-$th episode. Take the sequence reversed in order of $T, T-1, ..., 0$, and estimate return G. For the policy evaluation,

$$\begin{aligned} g &= \gamma g + r \\ V_j(x_k) &= V_{j-1}(x_k) + \frac{1}{J}(g_j + V_{j-1}(x_k)) \end{aligned} \tag{2.15}$$

where $V_j(x_k)$ represents sample mean, and J represents the number of episodes. Same principle is applied to $Q_j(x_k, u_k)$, as well. For the greedy policy improvement,

$$\begin{aligned} g &= \gamma g + r \\ Q_j(x_k, u_k) &= Q_{j-1}(x_k, u_k) + \frac{1}{J}(g_j + Q_{j-1}(x_k, u_k)) \\ \pi'_j(x_k) &= \text{argmax}_u Q_j(x_k, u) \end{aligned} \tag{2.16}$$

If the agent improves the policy that is used to generate the data, it is called on-policy learning. Whereas if it develops completely different policy from the one that is used for the data collection, it is called off-policy. **Exploring starts** is on-policy method that employs random starting states. $\epsilon-$**greedy on-policy** is that it gives small chance to explore new states and actions. If there are N possible actions, then the agent has $\frac{\epsilon}{N}$ probability to choose random actions that are different from the greedy search. With Equation 2.16,

$$\begin{aligned} \tilde{u} &= \text{argmax}_u Q(x_k, u) \\ \pi'(u|x_k) &= \tilde{u}, \ (\mathbf{p} = 1 - \epsilon + \epsilon/N) \\ \pi'(u|x_k) &= u \neq \tilde{u}, \ (\mathbf{p} = \epsilon/N) \end{aligned} \tag{2.17}$$

2.4 Temporal Difference

Temporal difference (TD) is hybrid version of DP and MC. DP has analytic formulation for the value functions, it takes care of every possible states at once (breath-first). On the other hand, MC looks data collected from episodes that continue until the agent reaches out terminal step (depth-first). TD takes the middle of DP and MC. Recalling Equation 2.15, MC demands end of an episode due to return g. Whereas TD, especially for one-step TD method, requires one-step further.

$$V(x_k) = V(x_k) + \alpha(r_{k+1} + \gamma V(x_{k+1}) - V(x_k)) \tag{2.18}$$

where α is a forgetting factor. Please note that it is bootstrapping because value of x_k is estimated from the another estimated value, $V(x_{k+1})$. Wait a second... How to estimate $V(x_{k+1})$, then? We can employ function approximates such as neural network or other machine-learning approaches. For the control problem, there are two basic ways: state-action-reward-state-action (SARSA) learning and Q-learning. SARSA literally needs five elements to estimate state-action value at k, $Q(x_k, u_k)$ and is on-policy method.

$$Q(x_k, u_k) = Q(x_k, u_k) + \alpha(R_{k+1} + \gamma Q(x_{k+1}, u_{k+1}) - Q(x_k, u_k)) \tag{2.19}$$

On the other hand, Q-learning doesn't need action of the next state and it directly finds optimal Q value while the current value updates. Note that it is off-policy.

$$Q(x_k, u_k) = Q(x_k, u_k) + \alpha(R_{k+1} + \gamma \max_u Q(x_{k+1}, u) - Q(x_k, u_k)) \tag{2.20}$$

A significant difference is contribution of action to estimate Q value. In case of on-policy, action that is determined by policy is directly used for Q value update. On the other hand, off-policy employs a particular action that can maximize the Q value of the next step. If the policy is greedy, both methods are identical. If the policy is $\epsilon-$greedy, then on-policy method sometimes takes random actions u_{k+1} to update Q values. However, off-policy always chooses specific actions that maximize the $Q(x_{k+1}, u_{k+1})$.

2.5 Multi-step Bootstrapping

An idea of bootstrapping is "an estimation based on estimation". TD method, especially one-step TD, is the simplest bootstrapping because $Q(x_k, u_k)$ is estimated from the $Q(x_{k+1}, u_{k+1})$ which is also estimated. Multi-step TD method is generalization of one-step TD, it looks forward n-step further to update the values. Therefore, agent should wait until the n-step forward looking is done. In the context of learning, even though it allows far-seeing, it leads to higher variance. λ return can resolve such a problem by weighting power of λ $(0 < \lambda < 1)$ on a series of n-step TD. Backward view is equivalent to forward view but new weight called eligibility trace $z(x_k)$ is necessary. It controls importance of states visited in the past: eligibility is decaying over iteration but once the agent visit same state, then it is increased.

$$z(x_k) \leftarrow \gamma\lambda z(x_{k+1}) + \delta_{xx'} \tag{2.21}$$

where $\delta_{xx'}$ is Kronecker-delta. The Q value concerning eligibility is

$$Q(x_k, u_k) = Q(x_k, u_k) + \alpha(R_{k+1} + \gamma Q(x_{k+1}, u_{k+1}) - Q(x_k, u_k))z(x_k) \tag{2.22}$$

Eligibility trace can be extended to continuous environment. In this case, value is modeled by functions such as neural network or linear regression. Then, the importance of specific states is described by gradient along with trained parameter vectors θ, a directional derivative.

$$z(x_k) \leftarrow \gamma\lambda z(x_{k+1}) + \nabla_\theta Q(x_k, u_k; \theta) \tag{2.23}$$

2.6 Policy Gradient Method

So far, we deal with DP, MC and TD with a few variations. There are three major categories of RL approaches: model-based (DP), value-based (MC, RD), and policy-based that is dealt in this section. The essence of policy gradient is functional approximation, it models policy $\pi(u|x;\theta)$ with parameter θ. The objective function to be maximized is $J(\theta) = \mathbf{E}(R(\tau))$, expectation of cumulative reward R along with state-action trajectory τ. By policy gradient theorem, no matter the process is continuous or not, the gradient of the objective is described as

$$\nabla_\theta J(\theta) = \mathbf{E}\left[Q_\pi(x, u)\nabla \ln \pi(u|x;\theta)\right] \tag{2.24}$$

Note that policy gradient approach has many equivalent form according to value formulation, $Q_\pi(x, u)$. The state-action value can be replaced to other functions such as advantage function, state value, weighting eligibility trace, and so on. Though value estimation contributes to find cool parameters, policy doesn't take it into account, which is vital difference from the previous approaches. If the value function is approximated by parameters as the policy, it is called actor(policy)-critic(value estimation) method. It seems similar to GAN algorithm in deep neural network.

Chapter 3

Modern Reinforcement Learning

I would like to begin with PPO! First of all, understanding idea of TRPO is essential. Here, unlike the previous chapters, reward is replaced to cost to follow up the authors' notation. Let the game begin. Followings are key references of the next section.

- Trusted Region Policy Optimization, written by J. Schulman et. al.
- Proximal Policy Optimization Algorithm, written by J. Schulman et. al.
- Approximately Optimal Approximate Reinforcement Learning, witten by S. Kakade & J. Langford

I would cover the diffusion policy after investigating PPO. Diffusion policy is inspired by generative model that is theoretically devised from non-equilibrium thermodynamics. It might be interesting. Please check followings.

- Deep unsupervised learning using nonequilibrium thermodynamics, J. Sohl-Dickstein, et. al.
- Diffusion policies as an expressive policy class for offline reinforcement learning, Z. Wanhg, et. al.

3.1 Proximal Policy Optimization

TRPO starts from conservative policy iteration proposed by Kakade & Langford. Basically, it is policy gradient method so that it searches optimized behaviors. The algorithm introduces a method to guarantee monotonic policy improvement by adding followings: surrogate function representing cumulative cost and improvement constraint. Let' start from the identity proved by S. Kakade & J. Langford.

$$\eta(\tilde{\pi}) = \eta(\pi) + \sum_{s} \rho_{\tilde{\pi}}(s) \sum_{a} \tilde{\pi}(a|s) A_{\pi}(s,a) \tag{3.1}$$

This equation holds when $\pi \rightarrow \tilde{\pi}$. The symbol $\eta(\cdot)$ indicates expected discounted cost. $\rho_{\pi}(s)$ is discounted visitation frequency,

$$\rho_{\pi}(s) = P(s_0 = s) + \gamma P(s_1 = s) + ... \tag{3.2}$$

In the second term, if $\sum_a \tilde{\pi}(a|s)A_\pi(s,a) \leq 0$, the Equation 3.1 guarantees the improvement. The authors mentioned that it would be difficult reaching out optimization due to complexity of $\rho_{\tilde{\pi}}(s)$. Instead, they introduced local approximation ignoring changes of the visitation frequency.

$$L_\pi(\tilde{\pi}) = \eta(\pi) + \sum_s \rho_\pi(s) \sum_a \tilde{\pi}(a|s) A_\pi(s,a) \tag{3.3}$$

Question 1. *What is the benefit of Equation 3.3? It just keeps old distribution, right? What is the problem of complexity of new discounted distribution exactly?*

Leave behind the fucking question, Equation 3.3 is the first order approximation of Equation 3.1. Assume that the policy π' such that

$$\pi' = \arg\min_{\pi'} L_\pi(\pi') \tag{3.4}$$

is taken based on the old policy π then the new policy π_{new} is defined as following, maybe, because of the first order approximation.

$$\pi_{\text{new}}(a|s) = (1-\alpha)\pi(a|s) + \alpha\pi'(a|s) \tag{3.5}$$

If α is fucking small, $\alpha << 1$, then

$$\eta(\pi_{\text{new}}) \leq L_\pi(\pi_{\text{new}}) + \frac{2\epsilon\gamma}{(1-\gamma)^2}\alpha^2 \tag{3.6}$$

Question 2. *Be honest, formulation seems tricky, α^2 should be vanished. See the original paper of TRPO.*

Authors proved that replacing the second term in Equation 3.6 to KL divergence also hold the inequality.

$$\eta(\pi_{\text{new}}) \leq L_\pi(\tilde{\pi}) + CD_{KL}^{\max}(\pi, \tilde{\pi}) \tag{3.7}$$

where $C = \frac{2\epsilon\gamma}{(1-\gamma)^2}$ and $D_{KL}^{\max}(\pi,\tilde{\pi}) = \max_s D_{KL}(\pi,\tilde{\pi})$. The right-hand side is the surrogate function representing dynamics of the policy. Thus, to guarantee improvement of the objective η, need to satisfy following.

$$\min_\pi \left[L_\pi(\pi_{\text{new}}) + CD_{KL}^{\max}(\pi,\tilde{\pi})\right] \tag{3.8}$$

Again, only very small value of step size of policy allows monotonic improvement. Trusted region can provide larger step size, by constraining acceptable changes.

$$\begin{aligned} &\min_\pi L_\pi(\pi_{\text{new}}) \\ &\text{subject to } D_{KL}^{\max}(\pi,\tilde{\pi}) \leq \delta \end{aligned} \tag{3.9}$$

I'd like to remark that TRPO needs MC rollouts to construct objective function as well as constraint. The objective function is represented by sampling as

$$\begin{aligned} &\min_\theta \mathbf{E}\left[\frac{\pi(a|s)}{\pi_{\text{new}}(a|s)}A\right] \\ &\text{subject to } \mathbf{E}\left[D_{KL}(\pi,\pi_{\text{new}})\right] \leq \delta \end{aligned} \tag{3.10}$$

TRPO is the second-order method because KL divergence is approximated by Hessian matrix. PPO is the first-order method, it employs clipped gradient for 'safe' improvement as TRPO and CPI did. Let ratio be $r_t(\theta) = \frac{\pi(a|s)}{\pi_{\text{old}}(a|s)}$. Then,

$$L(\theta) = \mathbf{E}\left[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)\right] \tag{3.11}$$

Note that it is on-policy method, it explores new actions through the sampling from the normal distribution $\pi(a|s)$.

Chapter 4

Implementations

I implement several modern approaches of RL. Capability of each algorithms is examined in simulation environment offered by gymnasium, OpenAI. All tests run once due to computation time. Followings are key references for the implementations.

- https://github.com/upb-lea/reinforcement_learning_course_materials
- https://spinningup.openai.com/en/latest/

4.1 PPO: cheetah-v5

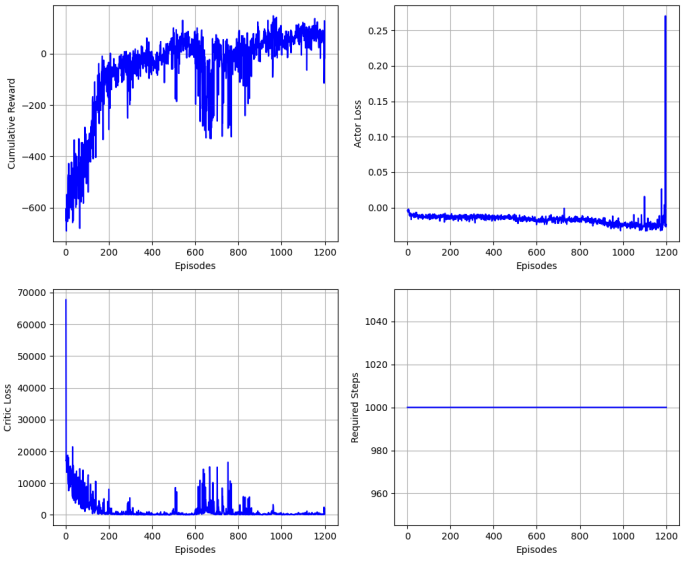


Figure 4.1: PPO in cheetah environment. Cumulative reward is increasing over episodes.

Chapter 5

Robotic Applications

I'm planning to conduct simple experiments to get some ideas for multi-modal robot learning. The baseline is encoder-decoder structure: perceiving states using sensors (camera, lidar, etc) and decoding the disparate information to move arms, for example. I need to think about

- state representation observed from the sensors
- interpretation: how to decode the state representation?
- if the idea is failed then what is the missing point?

For the application, I prefer to use ROS2 rather ROS which would not be maintained after Noetic (it ends middle of 2025). Different robotic platforms require specific distribution of ROS2 fitting to their design, so the primary job is building a work station according to the robot's configuration. All environments are managed by Apptainer.

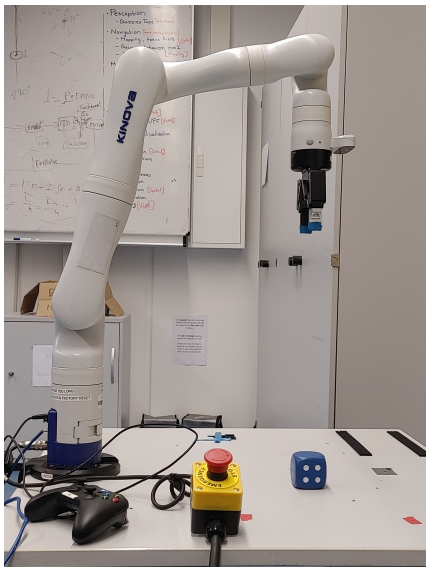
5.1 Kinova Arm with PPO

A model Gen3 with a RGB-D camera and a gripper will perform some tasks. All functionalities of the robot is controlled by kortex driver. The system should comprise, at least, ROS2 Humble, Gazebo Fortress, and Ubuntu 22.04 to install the driver.

5.1.1 Grasping Task

As far as I know, typically, tasks of the robots are reduced to independent functions and the robots achieve objectives based on state machine. What I want to do is, a robot arm learns how to grab unknown object via PPO after perceiving the states using RGB-D camera, status of the joints, or other supplements. While the agent learns the grasping unknown objects, it should notice "object instance" in its sight. Main difference might be actor network, the structure comprises variational encoder. Figure 5.1 stands for "raw" state (RGB), which is an input of variational encoder. The network would return seven-dimensional vectors meaning end-effector's next waypoint $(w, x, y, z, \theta_x, \theta_y, \theta_z)$ where w is width of gripper and the rest is Cartesian pose. The initial position of the gripper is fully open as well as the target is always in the sight. Thus, the optimized solution is that the arm directly goes

(a) States represented by RGB images



(b) Initial pose of Kinova arm

Figure 5.1: Kinova arm learns a grasping skill starting from the initial position. States are purely described by RGB images. As there's no supplementary algorithms to automate the learning process, reward signals are assigned manually after each step.

close to the target and grasps it at one or two waypoint(s). In the experiment, all episodes would be truncated by five waypoints. I also want to check if the robot is able to recognize the target while it learns how to grasp. The Kinova seems to have no feedback to indicate grasping objects, I would specify reward signal after the end-effector moves to waypoints. A goal is that **Kinova arm achieves grasping-a-dice skill.** While it learns the skill, I also want to see **how actor network understand raw state** before it infers the next action, via class activation map. Following is a list of reward signals.

- dice off-sight: -100
- dice off-sight, grasp air: -120
- dice on-sight: -10
- dice on-sight, grasp air: -30
- dice on-sight, grasp the target: 300

Experiment Setup

In the experiments, a couple of variables are considered to bridge RL algorithms to cognitive capabilities. Major goals are

- Stretching and grasping a target object
- Recognizing a target object

- Tracking a target object
- The skill should be achieved within 100 steps

At the initial pose, the arm need to notice the target and how far the arm is from it. And the arm stretches out while the camera keeps looking it. The robot would benefit if the target is on-sight of the camera but penalized if it is off-sight. Can the robot understand this rule of game at several attempts? Well, let's see!

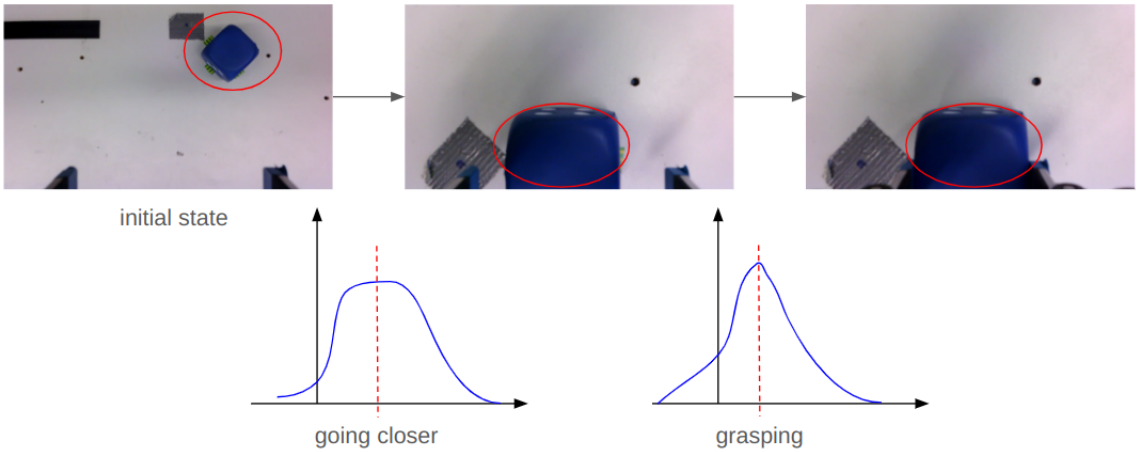


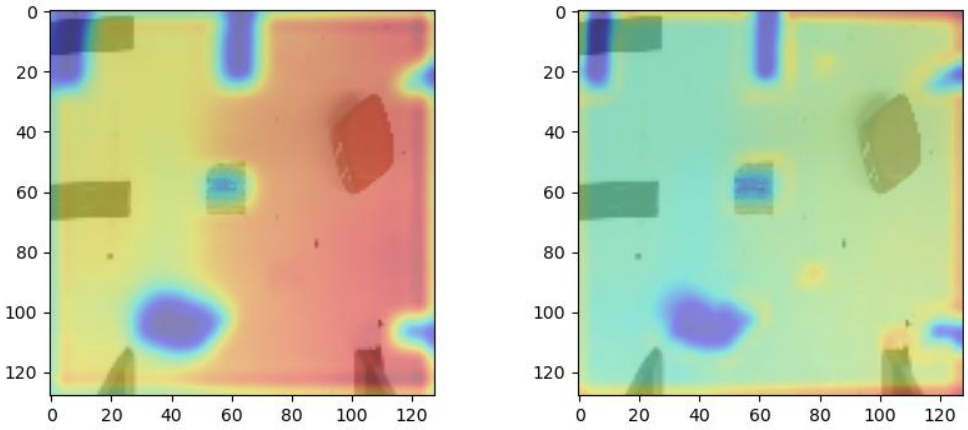
Figure 5.2: Desired behavior of PPO agent. Ovals highlight highly weighted area in activation map.

Scenario A: first trial

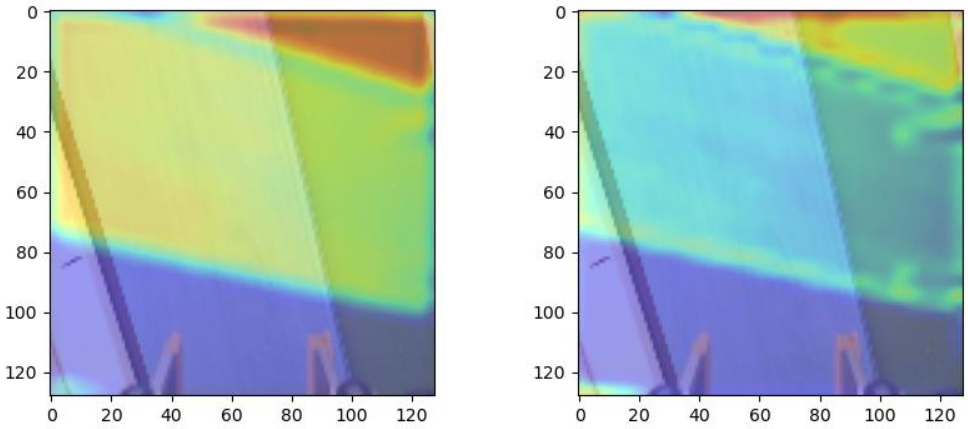
The scenario A assumes that the position of the target is always same. I examine two extreme cases: case I (-cI), without disturbance, and case II (-cII), 100% chance of disturbance. Here, disturbance means that I manually move the arm at my desired position. As you might expect, continuous space hardly allows the mission success through exploration. Due to a matter of time, I evaluate performance of a single trial of each case rather than mean value of multiple trials. Neural networks for function approximation of actor and critic are simple residual network. The structure of backbones are same, they are finalized by global average pooling (GAP). Only head components have difference varying dimension of the output. The input frame images are resized as (128, 128). All networks get batch size as step numbers in each episode. The maximum step is 5 so the roll-out buffer contain one to five data points to train the models right before the new episode starts. As Figure 5.3 shows, the results are way far away from what I expected. The agent neither recognizes, tracks nor grasps the target. At the initial pose, the agent moves the arm toward a door rather than the target. Once it perceive the next state (a door, now target is off-sight), it has no idea how to perform next action. It stays there watching the stupid door for the rest of steps. There's no crucial difference between A-cI and A-cII: I thought the network might be biased in A-cII because I perform same (or similar) behaviors every episode. However, the trained mean values let the agent watch the door again! I also try relatively large learning rate $(0.01 - 0.05)$ to

draw noticeable behavior changes. But the agent is still stuck after initial pose in test mode. Followings are summary of what I thought during the experiments.

- The networks seem to need more data or the networks are invalid to train using small amount of data.
- Sparse reward signal?
- Does policy improvement guarantee object recognition? This is what I want to check. I need to read the papers precisely. Those seem somethings separated each other.
- What is cognitive capabilities? If the agent achieves a particular skill then it is cognitive? Maybe I need neuroscience at this moment...?



(a) Activation maps of actor (left) and critic (right) at initial state.



(b) Activation maps of the next state.

Figure 5.3: The activation maps of Scene A-cI are captured at the last episode (20 episodes $\times$ 5 steps). The agent keeps staying foolish. Disturbances don't improve cognitive capability.

Scenario A: second trial

I observed that both actor and critic networks are not trained well because positive actor loss indicates decline of new policy. Furthermore, critic loss bring too large amount of loss. This is because of sparsity of the reward signal, the networks struggles to map continuous space to five-fold categories. Now I assign new reward signal as sum of cost and incentive such that

$$c = -\left(\sum_{i=1}^{3} x_i + \sum_{j=1}^{3} \theta_j/180 + f\right) \tag{5.1}$$

where f represents gripper's normalized angle. An incentive i is manually given according to actions performed. At least, the agent notices that the target should be on-sight. However, it has no idea to go closer and to grasp it. The agent just try to keep the object in its sight and moving slightly. Activation maps of every episode are similar to the first trial, which means it stays in particular coordinates (statistically biased) rather than recognizes the object. Let me summarize the observation.

- The agent seems to have no idea except watching the target. No going closer, no grasping the target: it falls into local pit
- Still, it could not recognize (or specify) the target. Networks, reward signals or even amount of data might lack to inform context of the state. From now on, I would not take into account an amount of data because it is one of main challenges of this project.
- So far, disturbance is not helping

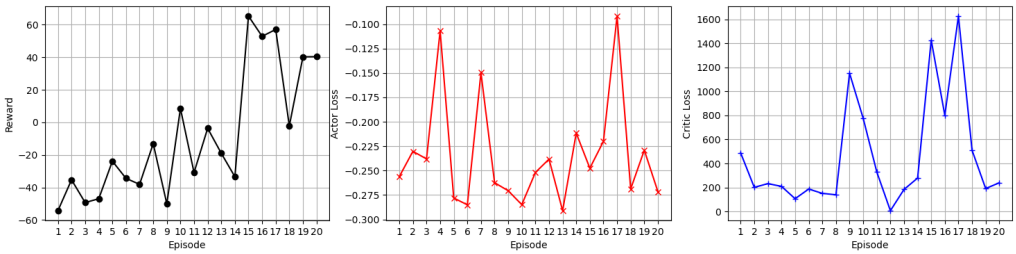


Figure 5.4: From the left, plots indicate evolution of cumulative reward, actor loss and critic loss over iterations, respectively. The experiment needs more data but it clearly shows that the agent starts to understand the desired behavior from the second trial.

I will try one more trial for the scenario A employing a multi-head actor network. One for the action (next coordinate), and the other is for (simple) recognition. It means that target identification is explicitly contribute to policy improvement.

Scenario A: third trial

In the third trial, the actor network has two heads where the second head returns context in binary manner: object-ness and grippability. In loss, context penalizes policy improvement.

$$\mathcal{L} = \mathcal{L}_{\text{policy}} + \alpha * \mathbf{E}\left[(l_{\text{pred}} - l_{\text{truth}})^2\right] \tag{5.2}$$

where context is two-element vector. I choose weight α as 0.1. Neither behavior nor explanability is improved: still, too broad area is regarding as an 'important instance'. Furthermore, the agent has no idea what the next job is after capturing (make on-sight) the target. How can I let it know going close and grasping a fucking dice? Hmmm... it is time to reconsider about the theoretical background. Note that

- images (128×128) serve as states
- a few images (five per each episode) are used to train the model
- actions are next coordinates to move (end-effector control)
- the agent could not specify the target instance alone
- the agent falls into local pit: once it captures the target, it is roaming around rather than going closer and grasping the target
- reward signal is normalized sum of the coordinates but the agent could get incentives if it performs as I want.

Review PPO

PPO optimizes a clipped surrogate objective function which is the first order approximation of Equation 3.1. Once the states are given, an actor network returns means and standard deviations according to the action space dimensionality.

$$\mathcal{G}\left(s;\theta\right) = \mu, \sigma^2 \tag{5.3}$$

Each parameters is given to one-dim'l normal distribution to sample new actions,

$$\pi\left(a|s\right) = \mathcal{N}\left(a|\mu,\sigma\right) \tag{5.4}$$

Thus, the network needs significant updates that decode the current state to a particular action which is likely beneficial. And the benefits are determined by generalized advantage estimation such that ($\lambda = 0$)

$$A_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{5.5}$$

where value function is approximated by a critic network. Please remember that summation over actions in the second term of Equation 3.3 is equivalent to $\mathbf{E}\left[r_t A_t\right]$ where r_t is a distribution ratio. In algorithm implementation, advantages are normalized. Back to my experiments, each episode comprises five steps in maximum and the agent always falls into local maxima: just keep watching the target without any movement. Even though I intentionally repeat same (similar) behaviors a few episodes (by 100 % disturbance), the agent could not learn a skill well. I would point out

- lack of data points at each episode: five images are too less
- limited degree of improvement: clipping does not allow quantum jump

- avoid local pit: need to explore outside of the trained distribution
- maybe more dense reward signal: linear sum of coordinate degenerates the actions (e.g. 1, 1, 1 and 3, 0, 0 get same reward)
- varying initial state: so far, it was always same
- one more fc layer in a head

Each problem is correlated each other. Next experiment, every episode has 16 steps (x 20 episodes). Entropy loss will be added to encourage the exploration.

Advanced Experiment

Before I conduct experiments, additional/improved functionalities should be embodied. The following is a list to do:

- more steps in each episode: ~~req. mini-batch due to sucking memory~~ - in very first experiment, the networks failed to learn due to shitty reward signal, not memory shortage
- more dense reward signal: check heatmap of three-coordinate
- varying initial state: req. a bounded region looking the target
- adding entropy loss to actor network
- disturbance is always is bigger than 0: the agent has no idea to go closer or grasp in a few hundreds steps

First I design new reward signal using linear sum of asymmetric functions, $c : \mathbf{R}^7 \to \mathbf{R}$ in range of $-1 \leq x \leq 1$.

$$c = x + \theta_x/180 + \sigma(y) + \sigma(\theta_y/180) + z^3 + (\theta_z/180)^3 + f + \beta \tag{5.6}$$

where β is a buffer parameter to ensure one-sided sign. To achieve surrogate objective, the agent would seek the shortest path for grasping skill. Because of origin's neighborhood, it is not completely one-to-one function with respect to actions but this is even denser than the old signal. Incentives will be manually assigned after I observe the agent's behavior. I define the signal density d_s as

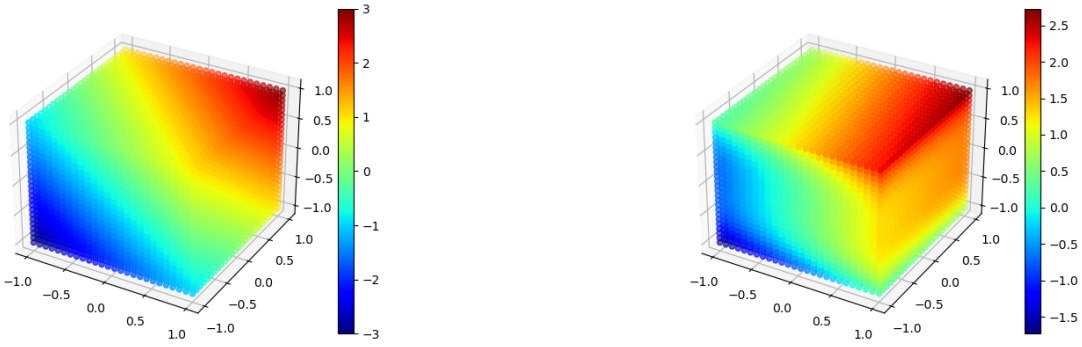
$$d_s = \frac{N_{\text{unique}}}{N_{\text{value}}} \tag{5.7}$$

Figure 5.5 show that new signal has higher density than the old. I add entropy so that the agent has more chance to explore action space rather than roaming around prematured locality.

$$\mathcal{H} = -\sum_a \pi(a|s) \log\left(\pi(a|s)\right) \tag{5.8}$$

The final form of the loss function for actor network is

$$\mathcal{L} = \mathcal{L}_{\text{policy}} + \beta\mathcal{H} \tag{5.9}$$



(a) Old reward signal, $d_s = 0.01$

(b) New reward signal, $\beta = 0, d_s = 0.99$

Figure 5.5: Reward (charge) signals in three-coordinate. The new one shows 99 times denser than the old one. In seven-dim'l action space, $\beta = -10$ is selected to make all values negative.

where β is chosen as 1. I set disturbance as 0.15 meaning that there's 15% of chance to show temporal demonstration from tutor's side. Nevertheless, the result could not deliver significant changes. Still, the critic network poorly performs regression of value with respect to the given states. Though the actor loss implies improvement of policy, path of gradient is misleading due to under/over-estimated states. Disturbance cannot handle this malfunction even in extreme case such as disturbance ratio is 1.

5.1.2 Lessons

From the grasping task, I learn practical bounds of RL (well, I used PPO only for this exp but all RLs have a problem in common: data req.) especially in robotic application. As far as I know (and discussion), RL would work better if it performs as fine-tuning after learning from demonstration (LfD). Why? At least, demonstration lets the agent know what the goal is. If the agent is fully trained from the scratch using RL, it hardly notice what the most valuable behaviors are. In other word, waiting until it shows the desired behaviors through exploration is infeasible and inefficient. I could not examine interrelationship between cognitive behavior and RL mechanism but the agent might be able to specify the target if it is trained in more dynamic environment (or data augmentation?). Ok then, what is next?

LfD, SINDy, or Manifold: Building Constrained System

Let's assume that I got states & actions dataset from LfD. How can the knowledge achieved from the LfD be transferred to RL agent?. I have a family of trajectories, starting from the initial states to the end states indicating grasping a target. I regard each trajectory as a unique solution of differentiable system, which is determined by a specific initial or boundary condition. Then how to figure out "differentiable system"? I employ sparse identification of dynamics (SINDy, see Discovering governing equations from data: Sparse identification of nonlinear dynamical systems, Steven L. Brunton et. al) framework to APPROXIMATE the dynamic system. It can displace clipped gradient which performs as constraints. I can find

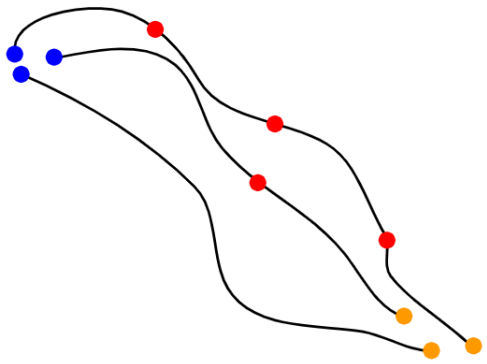


Figure 5.6: Example sequences of behavior trajectories from LfD. From this data, SINDy estimates differential equation constraining the behavior. It means that we have infinite number of bundles of trajectories and pick one of them according to initial condition. Actor network would learn policy that minimize the violation, distance from the estimated curve and explored actions.

several approaches that constrain the searching space using geometry or geometric primitives such as manifolds. Apart from the theory, still, I am very skeptical of employing RL training agents from scratch. For feature extraction, at least, the agent should exploit pre-knowledge rather than learning it while the agent is under the training.

Transfer Learning from Simulation

We can exploit the knowledge from simulation through transfer learning. In this virtual environment, it allows for a large amount of iterations with less time cost. The significant defect is the domain gap between the real and virtual world, the graphical assets cause poor cognitive capabilities for genuine objects. There are a few efforts to deal with such problems employing fine-tuning, domain randomization or learning representations.