A Journey to Fucking Cool Robotic Cognition

Angry K. Park

September, 2024

Contents

Chapter 1

Read Me

Aims of this journey are first, to preview ideas of reinforcement learning (RL) second, to implement well-known algorithm e.g. proximity policy optimization (PPO) 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 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 RL.
- Understanding idea of modern RL algorithms such as PPO, SAC or diffusion policy model.
- Implementation and test in openAI gym environment using mujoco.
- 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). This is the first run!

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

It's time to deal 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 this chapter.

- 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 recommend reading in order of Kakade & Langford → J. Schulman et. al. TRPO → J. Schulman et. al. PPO. Please read another approaches such as REINFORCE or SAC, at least papers suggested to read in openai's spinning up.

3.1 Trust Region 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

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

3.2 Proximity Policy Optimization

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 (and I'm lazy engineer). Followings are key references for the implementations.

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

4.1 Proximal Policy Optimization

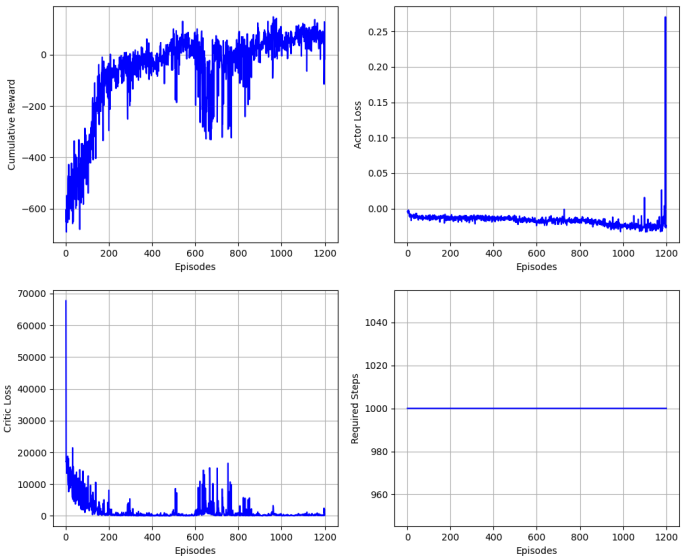

Figure 4.1: PPO in cheetah environment. Cumulative reward is increasing over episodes.

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

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-D), which is an input of variational encoder. The network would return seven-dimensional vectors meaning end-effector's next waypoint $(w, x, y, z, \theta_x, \theta_y, \theta_z)$. 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 close to the target and grasps it at a single waypoint. In the experiment,

Figure 5.1: Frame image from Kinova arm. A dice is a target to grasp.

all episodes would be truncated by ten waypoints. I also want to check if the robot is able to recognize the target while it learns how to grasp.