Deep Reinforcement Learning Notes (UBC)

Dongda Li
dli160@syr.edu

Contents

1 Background

This note is the class note of UBC Deep Reinforcement Learning, namely CS294-112 or CS285. The lecturer is Sergey Levine. The lecture videos can be found on YouTube. I wrote two notes on reinforcement learning before, one is basic RL, the other is the David Silver class note.

Different from the previous courses, this course includes a deeper theoretical view, more recent methods, and some advanced topics, especially in model-based RL and meta-learning. It is more suitable for those who are interested in robotic control and a deeper understanding of reinforcement learning.

This class is a little bit hard to study, so make sure you follow it closely.

2 Imitation Learning

In imitation learning, the objective is to train a policy that mimics the behavior of an expert. However, one of the key challenges is the *distribution drift* problem, where the distribution of states encountered during training differs from the distribution encountered when the learned policy is deployed. In this section, we discuss the main problem of distribution drift, describe a solution using DAgger (Dataset Aggregation), and elaborate on additional challenges and a formulation of a reward function for imitation learning.

2.1 The Main Problem: Distribution Drift

A central challenge in imitation learning is that the training dataset is often collected from an expert or human demonstrator, i.e.,

$$p_{\text{data}}(o_t),$$

which is different from the distribution of observations encountered under the learned policy,

$$p_{\pi_\theta}(o_t).$$

When the learned policy π_θ is executed, it may visit states that are rarely (or never) seen in the expert demonstrations. This mismatch can cause the policy to perform poorly when deployed, as it has not learned how to act in these unfamiliar states.

2.2 DAgger: Dataset Aggregation

One effective method to address the distribution drift problem is the DAgger algorithm (Dataset Aggregation). The main idea behind DAgger is to gradually collect training data from the distribution induced by the current policy, $p_{\pi_\theta}(o_t)$, rather than relying solely on the original expert dataset $p_{\text{data}}(o_t)$.

2.2.1 DAgger Procedure

The goal is to collect training data that better reflects the distribution of observations under π_θ while still having expert labels for the correct actions. The procedure is as follows:

1. **Initial Training:** Train an initial policy $\pi_\theta(a_t \mid o_t)$ using the expert data
$$\mathcal{D} = \{o_1, a_1, \ldots, o_N, a_N\}.$$
2. **Data Collection:** Run the current policy $\pi_\theta(a_t \mid o_t)$ in the environment to collect a dataset of observations:
$$\mathcal{D}_\pi = \{o_1, \ldots, o_M\}.$$
3. **Expert Labeling:** Have an expert label the observations in $\mathcal{D}_\pi$ by providing the corresponding actions.
4. **Dataset Aggregation:** Aggregate the new labeled data with the existing dataset:
$$\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_\pi.$$

By iteratively repeating these steps, the training dataset gradually becomes more representative of the state distribution encountered by the policy, thereby mitigating the distribution drift problem.

2.3 Challenges in Imitation Learning

Even with techniques like DAgger, there are additional challenges that can prevent the model from perfectly mimicking the expert:

2.3.1 Non-Markovian Behavior

- **Issue:** The expert's behavior may depend on the history of observations rather than solely on the current observation.
- **Solution:** Incorporate historical information (e.g., using recurrent neural networks or stacking multiple observations) to better capture the context.

2.3.2 Multimodal Behavior

- **Issue:** In many cases, there may be multiple valid actions for a given observation (multimodality), which complicates learning.
- **Discrete Actions:**
  - A Softmax output is typically used to represent a probability distribution over actions, naturally handling multimodality.
- **Continuous Actions:** Special care is needed for continuous actions:
  - **Mixture of Gaussians:** The model can output parameters for a mixture of Gaussian distributions, allowing for multiple modes.
  - **Latent Variable Models:** Injecting noise into the network input or using latent variables can help capture multimodal behavior.
  - **Autoregressive Discretization:** Discretizing the continuous action space in an autoregressive manner can also be an effective strategy.

2.3.3 Other Practical Issues

- **Limited Expert Data:** Human-labeled data is often finite, which can limit the diversity of training examples.
- **Expert Performance:** In some domains, human experts may not provide optimal demonstrations, potentially limiting the performance ceiling of imitation learning.

2.4 Reward Function for Imitation Learning

An alternative perspective on imitation learning is to view it as a reinforcement learning problem where the reward function is designed to encourage the policy to match the expert's behavior. One commonly used reward formulation is:

$$r(s, a) = \log p(a = \pi^*(s) \mid s),$$

where $\pi^*(s)$ represents the expert's action at state s. This reward function assigns higher rewards to actions that are more likely under the expert's policy, thereby guiding the learning process toward expert-like behavior.

In summary, imitation learning seeks to replicate expert behavior by addressing the distribution drift between training data and the policy-induced state distribution. Methods like DAgger help in collecting a more representative dataset, while various modeling strategies are employed to handle non-Markovian and multimodal behaviors. Finally, reward formulations based on expert likelihood can further guide the policy toward expert performance.

3 MDP and RL Introduction

In this section, we introduce the fundamental concepts of Markov Decision Processes (MDPs) and Reinforcement Learning (RL). We begin with the goal of RL, followed by definitions of the value function V and the action-value function Q. We then describe the various types of RL algorithms, discuss the trade-offs inherent in these methods, and state common assumptions made in RL.

3.1 The Goal of Reinforcement Learning

The objective in reinforcement learning is to learn a policy $\pi_\theta(a|s)$ that maximizes the expected cumulative reward over a sequence (or trajectory) of states and actions. A trajectory τ is defined as the sequence

$$\tau = \{s_1, a_1, s_2, a_2, \ldots, s_T, a_T\}.$$

The probability distribution over trajectories induced by the policy and the environment dynamics is given by

$$p_\theta(\tau) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t \mid s_t)\, p(s_{t+1} \mid s_t, a_t). \tag{1}$$

The goal is to find the optimal policy parameters θ^* that maximize the expected cumulative reward:

$$\theta^* = \arg\max_\theta \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} r(s_t, a_t) \right]. \tag{2}$$

Here, $r(s_t, a_t)$ is the immediate reward received at time step t, and the expectation is taken over the distribution of trajectories $p_\theta(\tau)$.

3.2 Value Functions: Q and V

To evaluate the quality of a policy, two fundamental functions are defined:

State-Value Function V^π: The state-value function represents the expected cumulative reward starting from state s_t and following the policy π thereafter:

$$V^\pi(s_t) = \mathbb{E}_\pi \left[\sum_{t'=t}^{T} r(s_{t'}, a_{t'}) \,\middle|\, s_t \right]. \tag{3}$$

Action-Value Function Q^π: The action-value function represents the expected cumulative reward starting from state s_t, taking action a_t, and then following the policy π:

$$Q^\pi(s_t, a_t) = r(s_t, a_t) + \mathbb{E}_\pi \left[V^\pi(s_{t+1}) \,\middle|\, s_t, a_t \right]. \tag{4}$$

Sometimes, the state-value function is also expressed in terms of Q^π as:

$$V^\pi(s_t) = \mathbb{E}_{a_t \sim \pi(a_t|s_t)} \left[Q^\pi(s_t, a_t) \right]. \tag{5}$$

3.3 Types of RL Algorithms

RL algorithms can be broadly classified into several categories, each with its own advantages and challenges:

- **Policy Gradient Methods:** These methods directly optimize the policy $\pi_\theta(a|s)$ by estimating gradients of the expected reward with respect to the policy parameters.
- **Value-Based Methods:** These methods learn the value function $Q^\pi(s, a)$ (or $V^\pi(s)$) and derive a policy by acting greedily with respect to the value estimates. Examples include Q-learning and Deep Q-Networks (DQN).
- **Actor-Critic Methods:** These methods combine policy gradient (actor) and value-based (critic) approaches. The critic estimates the value function, which is then used to update the actor.

- **Model-Based RL:** These methods build a model of the environment dynamics. Model-based methods can be subdivided into:
  - **Planning:** Using optimal control or discrete planning techniques.
  - **Policy Improvement:** Using the learned model to simulate experience and improve the policy.
  - **Other Aspects:** Incorporating ideas from dynamic programming or leveraging simulated experience to boost sample efficiency.

3.3.1 Trade-offs

Different RL algorithms often involve trade-offs in several key areas:

- **Sample Efficiency:**
  - *Off-policy* methods can reuse past experience to update the policy without generating new samples from the current policy.
  - *On-policy* methods require new data every time the policy is updated, which can be less sample efficient.
- **Stability:** Ensuring stable learning is a major challenge in RL. Some methods (e.g., policy gradients) can be sensitive to hyperparameters and the scale of rewards.
- **Ease of Use:** While supervised learning often benefits from straightforward gradient descent, RL methods frequently need additional techniques (e.g., target networks, replay buffers, and careful exploration strategies) to converge reliably.

3.3.2 Assumptions in RL

RL algorithms are typically designed under various assumptions about the environment and the problem setup:

- **Stochastic vs. Deterministic:** The environment dynamics may be stochastic or deterministic, which affects the design of the policy and value function estimators.
- **Continuous vs. Discrete:** The action and state spaces can be continuous or discrete, influencing the choice of algorithm and function approximators.
- **Episodic vs. Infinite Horizon:** Problems may be episodic (with a clear terminal state) or formulated over an infinite horizon, which affects the use of discount factors and convergence criteria.

3.3.3 Sample Efficiency in RL

Sample efficiency is crucial for many RL applications:

- **Off-policy Methods:** These methods allow the algorithm to learn from previously collected data, improving sample efficiency by reusing experiences.
- **On-policy Methods:** These require collecting new samples after every policy update, which can be less efficient in terms of data usage.

3.3.4 Stability and Ease of Use

Unlike supervised learning, which almost always relies on gradient descent for optimization, RL methods may not strictly use gradient descent due to issues such as:

- High variance in gradient estimates.
- The need for exploration, which can lead to unstable behavior.
- The complexity of the environment dynamics.

These factors contribute to challenges in achieving convergence and reliable performance in RL.

In summary, reinforcement learning aims to optimize a policy to maximize expected cumulative reward by interacting with an environment modeled as an MDP. The field includes a diverse set of algorithms, each with different assumptions, trade-offs in sample efficiency, and stability. Understanding these fundamental aspects is crucial for selecting and designing RL methods for a given application.

4 Policy Gradient

In this section, we derive the policy gradient formulation from first principles. We begin with the objective function and then show how to differentiate it using the log-derivative trick. We also discuss practical issues such as variance reduction via causality and baselines, and finally outline extensions to off-policy learning using importance sampling.

4.1 Objective Function

The goal in policy gradient methods is to find the optimal policy parameters θ^* that maximize the expected cumulative reward. A trajectory τ is defined as a sequence of states and actions:

$$\tau = \{s_1, a_1, s_2, a_2, \ldots, s_T, a_T\}.$$

The probability of a trajectory under the policy π_θ and the dynamics of the environment is given by

$$p_\theta(\tau) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t \mid s_t)\, p(s_{t+1} \mid s_t, a_t). \tag{6}$$

The overall objective is to maximize the expected total reward:

$$\theta^* = \arg\max_\theta \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} r(s_t, a_t) \right], \tag{7}$$

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} r(s_t, a_t) \right] \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} r(s_{i,t}, a_{i,t}). \tag{8}$$

Here, $r(s_t, a_t)$ is the immediate reward, and the expectation is approximated using N sampled trajectories.

4.2 Policy Differentiation

To optimize $J(\theta)$ with respect to the policy parameters θ, we differentiate the objective using the log-derivative trick.

4.2.1 Log Derivative Trick

Consider the identity:

$$\pi_\theta(\tau)\, \Delta \log \pi_\theta(\tau) = \pi_\theta(\tau)\, \frac{\Delta \pi_\theta(\tau)}{\pi_\theta(\tau)} = \Delta \pi_\theta(\tau). \tag{9}$$

Since the trajectory probability can be written as:

$$\begin{aligned} \pi_\theta(\tau) &= \pi_\theta(s_1, a_1, \ldots, s_T, a_T) \\ &= p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t \mid s_t) p(s_{t+1} \mid s_t, a_t), \end{aligned} \tag{10}$$

its logarithm becomes:

$$\log \pi_\theta(\tau) = \log p(s_1) + \sum_{t=1}^{T} \log \pi_\theta(a_t \mid s_t) + \sum_{t=1}^{T} \log p(s_{t+1} \mid s_t, a_t). \tag{11}$$

Noting that the environment dynamics $p(s_{t+1}|s_t, a_t)$ and the initial state distribution $p(s_1)$ do not depend on θ, differentiating with respect to θ yields:

$$\Delta_\theta \log \pi_\theta(\tau) = \sum_{t=1}^{T} \Delta_\theta \log \pi_\theta(a_t \mid s_t). \tag{12}$$

4.2.2 Differentiating the Objective Function

The objective function can be written as an integral over trajectories:

$$\theta^* = \arg\max_\theta \int \pi_\theta(\tau)\, r(\tau)\, d\tau, \tag{13}$$

$$r(\tau) = \sum_{t=1}^{T} r(s_t, a_t). \tag{14}$$

Differentiating $J(\theta)$ with respect to θ gives:

$$\begin{aligned}
\Delta_\theta J(\theta) &= \int \Delta_\theta \pi_\theta(\tau)\, r(\tau)\, d\tau \\
&= \int \pi_\theta(\tau)\, \Delta_\theta \log \pi_\theta(\tau)\, r(\tau)\, d\tau \\
&= \mathbb{E}_{\tau\sim\pi_\theta}\left[\Delta_\theta \log \pi_\theta(\tau)\, r(\tau)\right] \\
&= \mathbb{E}_{\tau\sim\pi_\theta}\left[\left(\sum_{t=1}^{T} \Delta_\theta \log \pi_\theta(a_t \mid s_t)\right)\left(\sum_{t=1}^{T} r(s_t, a_t)\right)\right].
\end{aligned} \tag{15}$$

4.2.3 Evaluating the Policy Gradient

In practice, the policy gradient is estimated by sampling N trajectories:

$$\Delta_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T} \Delta_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t})\right)\left(\sum_{t=1}^{T} r(s_{i,t}, a_{i,t})\right). \tag{16}$$

The policy parameters are then updated using gradient ascent:

$$\theta \leftarrow \theta + \alpha\, \Delta_\theta J(\theta), \tag{17}$$

where α is the learning rate.

4.3 REINFORCE Algorithm

The REINFORCE algorithm is a Monte Carlo policy gradient method that directly implements the above gradient estimation procedure:

1. **Sample Trajectories:** Generate N episodes τ^i by running the policy π_θ in the environment.
2. **Compute Gradient Estimate:** Estimate the gradient:

$$\Delta_\theta J(\theta) \approx \sum_{i=1}^{N}\left(\sum_{t=1}^{T} \Delta_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t})\right)\left(\sum_{t=1}^{T} r(s_{i,t}, a_{i,t})\right).$$

3. **Update Parameters:** Update θ using the gradient estimate:

$$\theta \leftarrow \theta + \alpha\, \Delta_\theta J(\theta).$$

4.3.1 Policy Gradient Estimator and Variance Reduction

A compact form of the policy gradient estimator is:

$$\Delta_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N} \Delta_\theta \log \pi_\theta(\tau^i)\, r(\tau^i). \tag{18}$$

However, this estimator can have high variance. Two common techniques are used to reduce variance:

Causality: Since an action at time t' cannot affect rewards obtained at times $t < t'$, we can refine the estimator as:

$$\Delta_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \Delta_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t}) \left(\sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'}) \right)$$

$$= \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \Delta_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t}) \, \hat{Q}_{i,t}, \tag{19}$$

where $\hat{Q}_{i,t}$ is an estimate of the cumulative reward (or return) starting from time t.

Baseline: A baseline b can be subtracted from the return to reduce variance without introducing bias:

$$\Delta_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \Delta_\theta \log \pi_\theta(\tau^i) \, [r(\tau^i) - b]. \tag{20}$$

A simple choice is to use the average return over the batch:

$$b = \frac{1}{N} \sum_{i=1}^{N} r(\tau^i). \tag{21}$$

Proof that baseline does not introduce bias:

$$\mathbb{E}\left[\Delta_\theta \log \pi_\theta(\tau)\, b\right] = \int \pi_\theta(\tau)\, \Delta_\theta \log \pi_\theta(\tau)\, b \, d\tau$$

$$= b \, \Delta_\theta \int \pi_\theta(\tau) \, d\tau$$

$$= b \, \Delta_\theta 1$$

$$= 0. \tag{22}$$

An optimal baseline that minimizes variance is given by

$$b = \frac{\mathbb{E}\left[g(\tau)^2 e(\tau)\right]}{\mathbb{E}\left[g(\tau)^2\right]},$$

but in practice, the sample mean is often used due to its simplicity.

Note: Policy gradient methods are *on-policy* algorithms, meaning that they require samples generated by the current policy.

4.4 Off-Policy Learning & Importance Sampling

In some settings, it is desirable to learn from data generated by a behavior policy $\bar{\pi}(\tau)$ that is different from the target policy $\pi_\theta(\tau)$. The objective can be rewritten using importance sampling:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[r(\tau) \right]$$

$$= \mathbb{E}_{\tau \sim \bar{\pi}(\tau)} \left[\frac{\pi_\theta(\tau)}{\bar{\pi}(\tau)} \, r(\tau) \right]. \tag{23}$$

Since

$$\pi_\theta(\tau) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) \, p(s_{t+1}|s_t, a_t), \tag{24}$$

$$\frac{\pi_\theta(\tau)}{\bar{\pi}(\tau)} = \frac{\prod_{t=1}^{T} \pi_\theta(a_t|s_t)}{\prod_{t=1}^{T} \bar{\pi}(a_t|s_t)}, \tag{25}$$

we can express the off-policy gradient as:

$$\Delta_{\theta'} J(\theta') = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\frac{\pi_{\theta'}(\tau)}{\pi_\theta(\tau)} \Delta_{\theta'} \log \pi_{\theta'}(\tau)\, r(\tau) \right]. \tag{26}$$

This formulation requires careful handling of the state distribution ratios. In practice, a common workaround is to treat the ratio $\frac{p_{\theta'}(s_t)}{p_\theta(s_t)}$ as approximately constant or bounded, leading to methods such as TPRO.

For practical implementation, one can define a "pseudo-loss" that is equivalent to weighted maximum likelihood:

$$\bar{J}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \log \pi_\theta(a_{i,t}|s_{i,t})\, \hat{Q}_{i,t}, \tag{27}$$

which can then be differentiated automatically.

4.5 Policy Gradient in Practice

Practical implementation of policy gradient methods comes with several challenges:

- **High Variance:** The gradient estimates are typically noisy compared to those in supervised learning. This motivates the use of large batches, variance reduction techniques (such as causality and baselines), and careful tuning of hyperparameters.
- **Learning Rate Sensitivity:** Choosing an appropriate learning rate is critical. Adaptive methods such as ADAM are commonly used, though policy gradient-specific adjustments may also be beneficial.

In summary, the policy gradient method leverages the log derivative trick to obtain an unbiased estimator of the gradient of the expected return. Variance reduction strategies, such as using causality and baselines, are crucial for practical success. Extensions to off-policy learning via importance sampling enable the reuse of data generated by different policies, albeit with additional considerations.

5 Actor-Critic Method

The Actor-Critic method is a popular approach in reinforcement learning that combines ideas from policy gradient (actor) methods with value function (critic) estimation. The actor is responsible for learning the policy $\pi_\theta(a \mid s)$, while the critic estimates the value function $V^\pi(s)$ (or action-value function $Q^\pi(s,a)$). This combination helps to reduce the high variance associated with pure policy gradient methods while still maintaining the flexibility of learning a stochastic policy.

5.1 Basics and Recap of Policy Gradient

Recall that the basic policy gradient estimator is given by:

$$\Delta_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\Delta_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t})\right)\left(\sum_{t=1}^{T} r(s_{i,t}, a_{i,t})\right) \tag{28}$$

$$\approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\Delta_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t})\left(\sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'})\right) \tag{29}$$

$$= \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\Delta_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t})\,\hat{Q}_{i,t}, \tag{30}$$

where $\hat{Q}_{i,t}$ is an unbiased sample return computed from trajectories but typically suffers from high variance.

One way to reduce the variance is to use the expected cumulative reward from time t onwards:

$$\hat{Q}_{i,t} \approx \sum_{t'=t}^{T} E_{\pi_\theta}\big[r(s_{t'}, a_{t'}) \mid s_t, a_t\big]. \tag{31}$$

We then define the value function as:

$$\hat{Q}_{i,t} = \sum_{t'=t}^{T} E_{\pi_\theta}\big[r(s_{t'}, a_{t'}) \mid s_t, a_t\big], \tag{32}$$

$$V(s_t) = E_{a_t\sim\pi(\cdot|s_t)}\Big[Q(s_t, a_t)\Big]. \tag{33}$$

Substituting the value function into the gradient estimate leads to:

$$\Delta_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\Delta_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t})\Big(Q(s_{i,t}, a_{i,t}) - V(s_{i,t})\Big). \tag{34}$$

Advantage Function

The advantage function is defined as:

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t), \tag{35}$$

so that the policy gradient becomes:

$$\Delta_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\Delta_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t})\,A^\pi(s_{i,t}, a_{i,t}). \tag{36}$$

A more accurate (or lower-variance) estimate of $A^\pi(s_t, a_t)$ leads to more efficient learning.

Value Function Fitting

The action-value function satisfies the Bellman equation:

$$Q^\pi(s_t, a_t) = r(s_t, a_t) + E_{s_{t+1}\sim p(\cdot|s_t,a_t)}\Big[V^\pi(s_{t+1})\Big]. \tag{37}$$

A common approximation is to use a one-step bootstrapped estimate:

$$Q^\pi(s_t, a_t) \approx r(s_t, a_t) + V^\pi(s_{t+1}). \tag{38}$$

Thus, the advantage function can be approximated as:

$$A^\pi(s_t, a_t) \approx r(s_t, a_t) + V^\pi(s_{t+1}) - V^\pi(s_t). \tag{39}$$

With this approximation, only the value function $V^\pi(s)$ needs to be estimated accurately.

Policy Evaluation

The state-value function under policy π is given by:

$$V^\pi(s_t) = \sum_{t'=t}^{T} E_{\pi_\theta}\big[r(s_{t'}, a_{t'}) \mid s_t\big], \tag{40}$$

$$J(\theta) = E_{s_1 \sim p(s_1)}\Big[V^\pi(s_1)\Big]. \tag{41}$$

In Monte Carlo policy evaluation, one might approximate:

$$V^\pi(s_t) \approx \sum_{t'=t}^{T} r(s_{t'}, a_{t'}), \tag{42}$$

or, if possible, average over multiple rollouts:

$$V^\pi(s_t) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t'=t}^{T} r(s_{t'}, a_{t'}). \tag{43}$$

With function approximation, we use supervised regression to fit the value function:

$$\mathcal{L} = \frac{1}{2} \sum_i \left\| \hat{V}_\phi^\pi(s_i) - y_i \right\|^2, \tag{44}$$

where the target y_i can be set either as the Monte Carlo return,

$$y_{i,t} = \sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'}),$$

or using a bootstrapped target:

$$y_{i,t} \approx r(s_{i,t}, a_{i,t}) + \hat{V}_\phi^\pi(s_{i,t+1}).$$

5.2 Actor-Critic Algorithm

The actor-critic algorithm combines the policy gradient (actor) with value function estimation (critic). A typical batch actor-critic algorithm proceeds as follows:

1. **Data Collection:** Sample a batch of state-action pairs $\{(s_i, a_i)\}$ by running the current policy $\pi_\theta(a \mid s)$.
2. **Critic Update:** Fit the value function $\hat{V}_\phi^\pi(s)$ using the sampled data. For example, use a target computed either via Monte Carlo returns or using a bootstrapped one-step target:

   $$y_i \approx r(s_i, a_i) + \gamma \hat{V}_\phi^\pi(s_i').$$

   The loss for the value function is:

   $$\mathcal{L} = \frac{1}{2} \sum_i \left\| \hat{V}_\phi^\pi(s_i) - y_i \right\|^2.$$

3. **Advantage Estimation:** Compute the advantage estimate for each sample:

   $$\hat{A}^\pi(s_i, a_i) = r(s_i, a_i) + \gamma \hat{V}_\phi^\pi(s_i') - \hat{V}_\phi^\pi(s_i).$$

4. **Actor Update:** Compute the policy gradient:

$$\Delta_\theta J(\theta) \approx \sum_i \Delta_\theta \log \pi_\theta(a_i \mid s_i)\, \hat{A}^\pi(s_i, a_i),$$

and update the policy parameters:

$$\theta \leftarrow \theta + \alpha\, \Delta_\theta J(\theta).$$

For online (step-by-step) actor-critic, the update is applied at each time step:

1. Take action $a \sim \pi_\theta(a \mid s)$ and observe the transition (s, a, s', r).
2. Update the critic using the target $r + \gamma\, \hat{V}_\phi^\pi(s')$.
3. Compute the advantage:

$$\hat{A}^\pi(s, a) = r + \gamma\, \hat{V}_\phi^\pi(s') - \hat{V}_\phi^\pi(s).$$

4. Update the actor using:

$$\Delta_\theta J(\theta) \approx \Delta_\theta \log \pi_\theta(a \mid s)\, \hat{A}^\pi(s, a),$$

and set $\theta \leftarrow \theta + \alpha\, \Delta_\theta J(\theta)$.

5.3 Discount Factors

When the episode length T is infinite, the value function can diverge. To address this, a discount factor $\gamma \in [0, 1]$ is introduced to prioritize immediate rewards:

$$V^\pi(s_{i,t}) \approx r(s_{i,t}, a_{i,t}) + \gamma\, \hat{V}_\phi^\pi(s_{i,t+1}). \tag{45}$$

The discounted policy gradient becomes:

$$\Delta_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \Delta_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t}) \left(\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{i,t'}, a_{i,t'}) \right). \tag{46}$$

5.4 Architecture Design

Network Architecture:

- **Separate Networks:** The value network (critic) and policy network (actor) can be implemented as separate networks, which has been observed to be more stable and sample efficient.
- **Shared Features:** Alternatively, some architectures share lower-level features between the actor and critic, while using separate output layers.

Batch updates generally yield better performance in actor-critic algorithms.

5.5 Trade-off and Balance

There is a trade-off between the unbiased nature of pure policy gradient methods (which have high variance) and the lower-variance but slightly biased actor-critic methods. For instance:

Policy Gradient:

$$\Delta_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \Delta_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t}) \left(\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{i,t'}, a_{i,t'}) - b \right). \tag{47}$$

Actor-Critic:

$$\Delta_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \Delta_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t}) \left(r(s_i, a_i) + \hat{V}_\phi^\pi(s_i') - \hat{V}_\phi^\pi(s_i) \right). \tag{48}$$

Using the critic as a state-dependent baseline typically leads to lower variance:

$$\Delta_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \Delta_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t}) \left(\sum_{t'=t}^{\infty} \gamma^{t'-t} r(s_{i,t'}, a_{i,t'}) - \hat{V}_\phi^\pi(s_{i,t}) \right). \tag{49}$$

5.6 Eligibility Traces & n-Step Returns

To further reduce variance and better trade off bias and variance, multi-step returns and eligibility traces can be used.

n-Step Returns:

$$\hat{A}_n^\pi(s_t, a_t) = \sum_{t'=t}^{t+n} \gamma^{t'-t} r(s_{t'}, a_{t'}) + \gamma^n \hat{V}_\phi^\pi(s_{t+n}) - \hat{V}_\phi^\pi(s_t). \tag{50}$$

Using $n > 1$ often yields better empirical performance.

Generalized Advantage Estimation (GAE): GAE combines multiple-step returns by weighting them with a factor λ (typically $\lambda = 0.95$). The advantage is computed as:

$$\hat{A}_{GAE}^\pi(s_t, a_t) = \sum_{t'=t}^{\infty} (\gamma\lambda)^{t'-t} \delta_{t'}, \tag{51}$$

$$\delta_{t'} = r(s_{t'}, a_{t'}) + \gamma \hat{V}_\phi^\pi(s_{t'+1}) - \hat{V}_\phi^\pi(s_{t'}). \tag{52}$$

This approach provides a flexible trade-off between bias and variance by effectively combining information over multiple time scales.

In summary, the Actor-Critic method leverages the strengths of both policy gradient and value function estimation. By using the critic as a baseline (or even a more refined estimator such as with GAE), actor-critic methods achieve lower-variance gradient estimates, leading to more stable and efficient learning.

6 Value Based Methods

Value based methods estimate a value function, such as the action-value function $Q(s, a)$ or the state-value function $V(s)$, and use these estimates to select actions. In many cases, the best action at state s_t is given by:

$$\arg\max_{a_t} A^\pi(s_t, a_t),$$

which is equivalent to

$$\arg\max_{a_t} Q^\pi(s_t, a_t).$$

A greedy policy induced by these estimates is then:

$$\pi'(a_t \mid s_t) = \begin{cases} 1, & \text{if } a_t = \arg\max_{a_t} A^\pi(s_t, a_t) \\ 0, & \text{otherwise.} \end{cases}$$

This deterministic policy is at least as good as any stochastic policy $a_t \sim \pi(a_t \mid s_t)$ when following π.

6.1 Policy Iteration

A common framework for value based methods is **policy iteration**, which alternates between two steps:

1. **Policy Evaluation:** Estimate the value function $A^\pi(s, a)$ (or $Q^\pi(s, a)$ and $V^\pi(s)$) under the current policy π.
2. **Policy Improvement:** Update the policy by acting greedily with respect to the estimated value function:

   $$\pi'(a_t \mid s_t) = \begin{cases} 1, & \text{if } a_t = \arg\max_{a_t} A^\pi(s_t, a_t) \\ 0, & \text{otherwise.} \end{cases}$$

   Then, set $\pi \leftarrow \pi'$ and repeat.

6.2 Dynamic Programming

When the dynamics $p(s' \mid s, a)$ are known and the state and action spaces are discrete (and small), dynamic programming methods can be applied. The bootstrapped update for the value function is given by:

$$V^\pi(s) \leftarrow E_{a\sim\pi(a|s)}\Big[r(s, a) + \gamma\, E_{s'\sim p(s'|s,a)}\big[V^\pi(s')\big]\Big].$$

For a deterministic policy $\pi(s) = a$, this simplifies to:

$$V^\pi(s) \leftarrow r(s, \pi(s)) + \gamma\, E_{s'\sim p(s'|s,\pi(s))}\big[V^\pi(s')\big].$$

Recall that:

$$Q^\pi(s, a) = r(s, a) + \gamma\, E\big[V^\pi(s')\big],$$

and therefore,

$$\arg\max_{a_t} A^\pi(s_t, a_t) = \arg\max_{a_t} Q^\pi(s_t, a_t).$$

The overall policy iteration process becomes:

1. Compute or update $Q^\pi(s, a) \leftarrow r(s, a) + \gamma\, E\big[V^\pi(s')\big]$.
2. Update the state-value function: $V(s) \leftarrow \max_a Q(s, a)$.

6.3 Function Approximators

In many practical problems, the state and/or action spaces are too large or continuous for tabular methods. In these cases, function approximators (such as neural networks) are used. For example, one can minimize the loss:

$$\mathcal{L} = \frac{1}{2}\sum_i \Big\| V_\phi(s_i) - \max_a Q(s_i, a) \Big\|^2,$$

to train a value function approximator $V_\phi(s)$.

6.3.1 Fitted Value Iteration

The fitted value iteration algorithm iteratively updates the value function using a batch of data:

1. For each sample s_i, compute the target:
$$y_i \leftarrow \max_{a_i} \Big(r(s_i \mid a_i) + \gamma \, E\big[V_\phi(s_i')\big] \Big).$$
2. Update the parameters by minimizing the squared error:
$$\phi \leftarrow \arg\min_{\phi} \frac{1}{2} \sum_i \Big\| V_\phi(s_i) - y_i \Big\|^2.$$

When the dynamics are unknown, one may instead focus on learning the Q-function.

6.3.2 Fitted Q-Iteration

Fitted Q-Iteration is a batch method for learning the Q-function. The algorithm proceeds as follows:

1. **Data Collection:** Collect a dataset $\{(s_i, a_i, s_i', r_i)\}$ using a given policy.
2. **Target Computation:** For each sample, compute the target:
$$y_i \leftarrow r(s_i, a_i) + \gamma \max_{a_i'} Q_\phi(s_i', a_i').$$
3. **Function Approximation:** Update the parameters by minimizing:
$$\phi \leftarrow \arg\min_{\phi} \frac{1}{2} \sum_i \Big\| Q_\phi(s_i, a_i) - y_i \Big\|^2.$$

Repeat the target computation and parameter update for k iterations before collecting new data.

6.4 Exploration Strategies

Since value based methods rely on accurate estimation of $Q(s, a)$, effective exploration is crucial. Common strategies include:

1. **Epsilon-Greedy:** With probability $1 - \epsilon$, choose the action with the highest estimated Q-value, and with probability ϵ, choose an action uniformly at random:
$$\pi(a_t \mid s_t) = \begin{cases} 1 - \epsilon, & \text{if } a_t = \arg\max Q_\phi(s_t, a_t), \\ \frac{\epsilon}{|\mathcal{A}| - 1}, & \text{otherwise.} \end{cases}$$
2. **Boltzmann Exploration:** Choose actions probabilistically according to the softmax of Q-values:
$$\pi(a_t \mid s_t) \propto \exp\Big(Q_\phi(s_t, a_t) \Big).$$

6.5 Value Function Learning Theory

In the tabular case, the standard Bellman updates converge. The basic process is:

1. Update $Q(s, a)$ via:
$$Q(s, a) \leftarrow r(s, a) + \gamma \, E\big[V(s')\big].$$
2. Update $V(s)$ as:
$$V(s) \leftarrow \max_a Q(s, a).$$

However, when using function approximation or in non-tabular cases, convergence is not guaranteed. Actor-critic methods that use bootstrapped estimates of V or Q inherit these challenges.

7 Practical Q-learning

In practical applications, online Q-learning suffers from several issues:

- It is not a true gradient descent method, as it does not compute the gradient of the target Q-value in y.
- The samples are often assumed to be independent and identically distributed (i.i.d.), which is not true for sequential data.

7.1 Replay Buffer

A replay buffer $\mathcal{B}$ stores past experiences so that each sample can be used multiple times. The procedure is:

1. **Data Collection:** Collect dataset $\{(s_i, a_i, s_i', r_i)\}$ using some behavior policy, and add it to $\mathcal{B}$.
2. **Batch Update:**
   (a) Sample a mini-batch (s_i, a_i, s_i', r_i) from $\mathcal{B}$.
   (b) Update the Q-network parameters ϕ using:

$$\phi \leftarrow \phi - \alpha \sum_i \frac{\partial Q_\phi(s_i, a_i)}{\partial \phi} \cdot \frac{1}{2}\Big(Q_\phi(s_i, a_i) - \Big[r(s_i, a_i) + \gamma \max_{a_i'} Q_\phi(s_i', a_i')\Big]\Big)^2.$$

   Repeat these updates k times.

7.2 Target Network

The target network helps to stabilize learning by decoupling the target y from the rapidly changing Q-network:

1. Maintain a target network with parameters ϕ' that is periodically updated from ϕ, e.g., $\phi' \leftarrow \phi$.
2. Use the target network to compute the target:

$$y_i = r(s_i, a_i) + \gamma \max_{a_i'} Q_{\phi'}(s_i', a_i').$$

3. Perform several mini-batch updates with the current replay buffer.

An alternative is to use **Polyak averaging** (soft updates):

$$\phi' \leftarrow \tau\phi' + (1-\tau)\phi, \quad \text{with } \tau \text{ close to 1 (e.g., } \tau = 0.999).$$

7.3 Double Q-learning

One common problem with Q-learning is overestimation of the Q-values due to the maximum operator. The target in standard Q-learning is:

$$y = r + \gamma\, Q_{\phi'}\Big(s', \arg\max_{a'} Q_{\phi'}(s', a')\Big).$$

This uses the same network to select and evaluate the action, which can amplify estimation noise.

Double Q-learning addresses this by decoupling the action selection and evaluation:

$$y = r + \gamma\, Q_{\phi'}\Big(s', \arg\max_{a'} Q_{\phi}(s', a')\Big).$$

Here, the current network Q_ϕ is used to choose the action while the target network $Q_{\phi'}$ evaluates it. Alternatively, one can maintain two separate networks Q_{ϕ_A} and Q_{ϕ_B}:

$$Q_{\phi_A} \leftarrow r + \gamma\, Q_{\phi_B}\Big(s', \arg\max_{a'} Q_{\phi_A}(s', a')\Big),$$

$$Q_{\phi_B} \leftarrow r + \gamma\, Q_{\phi_A}\Big(s', \arg\max_{a'} Q_{\phi_B}(s', a')\Big).$$

7.4 Multi-step Returns

Instead of using one-step returns, multi-step returns can provide a better trade-off between bias and variance. A typical n-step return is:

$$y_{j,t} = \sum_{t'=t}^{t+N-1} r_{j,t'} + \gamma^N \max_a Q_{\phi'}\Big(s_{j,t+N}, a\Big).$$

Strictly speaking, multi-step returns are correct for on-policy data. When using off-policy data, one might:

1. Ignore the discrepancy if N is small.
2. Dynamically choose N such that the data remains mostly on-policy.
3. Use importance sampling corrections (see, e.g., "Safe and Efficient Off-Policy Reinforcement Learning" by Munos et al. (2016)).

7.5 Q-learning with Continuous Actions

In continuous action spaces, performing the $\arg\max$ over actions is nontrivial. Several approaches are used:

1. **Optimization Methods:**
   - Use gradient-based optimization (e.g., SGD) in the inner loop, though it may be slow.
   - For low-dimensional action spaces, sample a discrete set of actions:
   $$\max_a Q(s,a) \approx \max\{Q(s,a_1), \ldots, Q(s,a_N)\}.$$
   - More advanced methods include the cross-entropy method (CEM) or CMA-ES.
2. **Structured Function Classes:** Use a quadratic form for Q-values:
   $$Q_\phi(s,a) = -\frac{1}{2}(a - \mu_\phi(s))^T P_\phi(s)(a - \mu_\phi(s)) + V_\phi(s).$$
   In this case,
   $$\arg\max_a Q_\phi(s,a) = \mu_\phi(s) \quad \text{and} \quad \max_a Q(s,a) = V_\phi(s).$$
   This method, known as Normalized Advantage Functions (NAF), limits representational power but simplifies optimization.
3. **Actor-Critic for Continuous Actions (DDPG):** Learn an approximate maximizer by training a separate policy network $\mu_\theta(s)$ such that:
   $$\mu_\theta(s) \approx \arg\max_a Q_\phi(s,a).$$
   The network $\mu_\theta(s)$ is updated by solving:
   $$\theta \leftarrow \arg\max_\theta Q_\phi\Big(s, \mu_\theta(s)\Big),$$
   with gradients given by:
   $$\frac{\partial Q_\phi}{\partial \theta} = \frac{\partial \mu_\theta(s)}{\partial \theta}\frac{\partial Q_\phi}{\partial a}.$$
   This approach is the basis of the Deep Deterministic Policy Gradient (DDPG) algorithm.

7.6 Tips for Q-learning

Practical tips for successful Q-learning include:

- **Gradient Clipping or Huber Loss:** The Bellman error can produce large gradients. Clip gradients or use the Huber loss:
$$L(x) = \begin{cases} \frac{x^2}{2}, & \text{if } |x| \le \delta, \\ \delta|x| - \frac{\delta^2}{2}, & \text{otherwise.} \end{cases}$$

- **Double Q-learning:** Incorporate double Q-learning to reduce overestimation bias.
- **Multi-step Returns:** Use n-step returns to balance bias and variance.
- **Exploration and Learning Rate Scheduling:** Gradually reduce exploration (e.g., decrease ϵ) and adjust learning rates over time. Optimizers like Adam can help.
- **Multiple Random Seeds:** Due to high variability, run experiments with multiple seeds.

8 Advanced Policy Gradients

This note explains advanced topics in policy gradient methods. We first review the basic REINFORCE algorithm and the policy gradient theorem. Next, we derive a surrogate objective using importance sampling to enable off-policy evaluation. We then explain how to bound the objective to ensure that policy updates yield performance improvements and finally discuss constrained optimization methods that lead to practical algorithms such as TRPO and PPO.

8.1 Recap: Policy Gradient and the REINFORCE Algorithm

8.1.1 Policy Gradient Objective

The goal in reinforcement learning is to maximize the expected return:

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t) \right],$$

where τ denotes a trajectory and π_θ is the policy parameterized by θ.

The *policy gradient theorem* states that:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) Q^{\pi_\theta}(s_t, a_t) \right].$$

In practice, it is common to use the advantage function

$$A^{\pi_\theta}(s_t, a_t) = Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)$$

to reduce the variance of the gradient estimator.

8.1.2 REINFORCE Algorithm

The REINFORCE algorithm is a simple Monte Carlo method that implements policy gradients:

1. **Sampling:** Run the current policy π_θ to sample trajectories $\{\tau^i\}$.
2. **Gradient Estimation:** Estimate the policy gradient:

$$\Delta_\theta J(\theta) \approx \sum_i \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i) \left(\sum_{t'=t}^{T} r(s_{t'}, a_{t'}) \right) \right).$$

3. **Parameter Update:** Update the policy parameters:

$$\theta \leftarrow \theta + \alpha \, \Delta_\theta J(\theta).$$

This method works because policy gradient methods can be seen as a type of policy iteration where we improve the policy based on its own performance estimates.

8.1.3 Performance Improvement via the Advantage Function

A key result is that the difference in performance between a new policy $\pi_{\theta'}$ and the current policy π_θ can be expressed as:

$$J(\theta') - J(\theta) = \mathbb{E}_{\tau \sim p_{\theta'}(\tau)} \left[\sum_{t=0}^{\infty} \gamma^t A^{\pi_\theta}(s_t, a_t) \right]. \tag{53}$$

The derivation (omitted here for brevity) relies on telescoping sums and the definition of the value function. The intuition is that if actions taken by the new policy have positive advantages under the old policy, then the new policy will yield a higher return.

8.2 Deriving the Surrogate Objective via Importance Sampling

8.2.1 Need for Off-Policy Estimation

In practice, it is desirable to use data collected from the current policy π_θ to evaluate the effect of a potential update $\pi_{\theta'}$. However, Equation (53) is expressed as an expectation over trajectories generated by $\pi_{\theta'}$. To address this, we use importance sampling.

8.2.2 Importance Sampling Rewriting

For any function $f(s, a)$, we can rewrite the expectation under $\pi_{\theta'}$ in terms of π_θ:

$$\mathbb{E}_{a\sim\pi_{\theta'}(\cdot|s)}[f(s,a)] = \mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\left[\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)}\, f(s,a)\right].$$

Applying this to the performance difference, we have:

$$\begin{aligned}\mathbb{E}_{\tau\sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty}\gamma^t A^{\pi_\theta}(s_t,a_t)\right] &= \sum_{t=0}^{\infty}\mathbb{E}_{s_t\sim p_{\theta'}(s_t)}\left[\mathbb{E}_{a_t\sim\pi_{\theta'}(a_t|s_t)}\left[\gamma^t A^{\pi_\theta}(s_t,a_t)\right]\right] \\ &= \sum_{t=0}^{\infty}\mathbb{E}_{s_t\sim p_{\theta'}(s_t)}\left[\mathbb{E}_{a_t\sim\pi_\theta(a_t|s_t)}\left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\,\gamma^t A^{\pi_\theta}(s_t,a_t)\right]\right].\end{aligned} \tag{54}$$

The ratio

$$r(s,a) = \frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)}$$

is called the *importance sampling ratio*.

8.2.3 Approximating the State Distribution

The expression in Equation (54) still samples states from $p_{\theta'}(s_t)$ (the state distribution under the new policy). However, if $\pi_{\theta'}$ is close to π_θ, we approximate:

$$p_{\theta'}(s_t) \approx p_\theta(s_t).$$

This approximation allows us to form a *surrogate objective* that can be computed using data from the current policy:

$$L(\theta') = \sum_{t=0}^{\infty}\mathbb{E}_{s_t\sim p_\theta(s_t)}\left[\mathbb{E}_{a_t\sim\pi_\theta(a_t|s_t)}\left[r(s_t,a_t)\,\gamma^t A^{\pi_\theta}(s_t,a_t)\right]\right].$$

Maximizing $L(\theta')$ approximately maximizes the performance improvement $J(\theta') - J(\theta)$ (up to a constant factor).

8.3 Bounding the Objective and Enforcing a Trust Region

8.3.1 Motivation for Bounding

The above derivation relies on the assumption that $\pi_{\theta'}$ is close to π_θ. To formalize this, we assume that for all s_t:

$$|\pi_{\theta'}(a_t \mid s_t) - \pi_\theta(a_t \mid s_t)| \le \epsilon.$$

Under this assumption, one can show that the difference in state distributions satisfies:

$$|p_{\theta'}(s_t) - p_\theta(s_t)| \le 2\epsilon\, t.$$

Thus, a small deviation in the policy leads to a controlled deviation in the state visitation distribution. More precisely, for any function $f(s_t)$:

$$\mathbb{E}_{p_{\theta'}}[f(s_t)] \ge \mathbb{E}_{p_\theta}[f(s_t)] - 2\epsilon\, t \max_{s_t} f(s_t). \tag{55}$$

8.3.2 Bounding with KL Divergence

It is often more convenient to bound the change in the policy using the Kullback-Leibler (KL) divergence:

$$D_{KL}\Big(\pi_{\theta'}(a_t \mid s_t) \,\|\, \pi_\theta(a_t \mid s_t)\Big) = \mathbb{E}_{a_t \sim \pi_{\theta'}(a_t|s_t)} \left[\log \frac{\pi_{\theta'}(a_t \mid s_t)}{\pi_\theta(a_t \mid s_t)}\right].$$

A useful inequality shows that:

$$|\pi_{\theta'}(a_t \mid s_t) - \pi_\theta(a_t \mid s_t)| \leq \sqrt{\frac{1}{2} D_{KL}\Big(\pi_{\theta'}(a_t \mid s_t) \,\|\, \pi_\theta(a_t \mid s_t)\Big)}.$$

Thus, rather than enforcing a hard bound ϵ on the policy difference, we can impose a KL divergence constraint:

$$D_{KL}\Big(\pi_{\theta'}(a_t \mid s_t) \,\|\, \pi_\theta(a_t \mid s_t)\Big) \leq \epsilon.$$

This constraint is more natural when working with probability distributions.

8.3.3 The Final Constrained Surrogate Objective

Combining the importance sampling approximation and the KL bound, the surrogate optimization becomes:

$$\theta' \leftarrow \arg\max_{\theta'} L(\theta') \quad \text{subject to} \quad D_{KL}\Big(\pi_{\theta'}(a_t \mid s_t) \,\|\, \pi_\theta(a_t \mid s_t)\Big) \leq \epsilon,$$

where

$$L(\theta') = \sum_{t=0}^{\infty} \mathbb{E}_{s_t \sim p_\theta(s_t)} \left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} \left[\frac{\pi_{\theta'}(a_t \mid s_t)}{\pi_\theta(a_t \mid s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t)\right]\right].$$

For sufficiently small ϵ, the surrogate objective guarantees that $J(\theta') > J(\theta)$.

8.4 Solving the Constrained Optimization Problem

There are several methods to solve the above constrained problem. We describe two approaches: dual gradient descent and the natural gradient method.

8.4.1 Dual Gradient Descent

To enforce the KL constraint, we form the Lagrangian:

$$\mathcal{L}(\theta', \lambda) = L(\theta') - \lambda\Big(D_{KL}\Big(\pi_{\theta'}(a_t \mid s_t) \,\|\, \pi_\theta(a_t \mid s_t)\Big) - \epsilon\Big).$$

The optimization then proceeds in two steps:

1. **Maximization Step:** For a fixed λ, maximize $\mathcal{L}(\theta', \lambda)$ with respect to θ'.
2. **Dual Update:** Update the multiplier λ by

$$\lambda \leftarrow \lambda + \alpha\Big(D_{KL}\big(\pi_{\theta'}(a_t \mid s_t) \,\|\, \pi_\theta(a_t \mid s_t)\big) - \epsilon\Big).$$

This method ensures that the KL constraint is gradually enforced during optimization.

8.4.2 Natural Gradient and Second-Order Approximation

An alternative is to use a first-order Taylor expansion of the surrogate objective around θ. Define

$$\bar{A}(\theta') = \sum_t \mathbb{E}_{s_t \sim p_\theta(s_t)} \left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} \left[\frac{\pi_{\theta'}(a_t \mid s_t)}{\pi_\theta(a_t \mid s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t)\right]\right], \tag{56}$$

$$\bar{A}(\theta) = \sum_t \mathbb{E}_{s_t \sim p_\theta(s_t)} \left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} \left[\gamma^t A^{\pi_\theta}(s_t, a_t)\right]\right]. \tag{57}$$

A first-order expansion leads to:

$$\theta' \leftarrow \arg\max_{\theta'} \nabla_\theta \bar{A}(\theta)^T(\theta' - \theta)$$

subject to

$$D_{KL}\Big(\pi_{\theta'} \,\|\, \pi_\theta\Big) \leq \epsilon.$$

To solve the constrained problem, a second-order Taylor expansion of the KL divergence is used:

$$D_{KL}(\pi_{\theta'} \,\|\, \pi_\theta) \approx \frac{1}{2}(\theta' - \theta)^T \mathbf{F}(\theta' - \theta),$$

where $\mathbf{F}$ is the Fisher information matrix:

$$\mathbf{F} = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a \mid s)\, \nabla_\theta \log \pi_\theta(a \mid s)^T\right].$$

This leads to the natural gradient update:

$$\theta' = \theta + \alpha\, \mathbf{F}^{-1} \nabla_\theta J(\theta),$$

with the step size chosen as

$$\alpha = \sqrt{\frac{2\epsilon}{\nabla_\theta J(\theta)^T \mathbf{F}\, \nabla_\theta J(\theta)}}.$$

This update ensures that the KL constraint is satisfied and is a key idea behind Trust Region Policy Optimization (TRPO).

8.5 Practical Methods and Summary

- **Natural Policy Gradient:**
$$\theta' = \theta + \alpha\, \mathbf{F}^{-1} \nabla_\theta J(\theta).$$
This method stabilizes training by taking into account the geometry of the policy space.
- **Trust Region Policy Optimization (TRPO):** TRPO uses the above natural gradient idea together with a line search to enforce the KL constraint:
$$D_{KL}\Big(\pi_{\theta'} \,\|\, \pi_\theta\Big) \leq \epsilon.$$
- **Proximal Policy Optimization (PPO):** Instead of using a hard KL constraint, PPO uses a clipped surrogate objective that limits the change in the policy ratio $r(s, a)$, yielding a similar effect in a simpler implementation.

8.6 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a policy gradient method designed to achieve reliable and stable improvements in policy updates while avoiding the complexities of second-order optimization techniques. It introduces a clipped surrogate objective to restrict large updates and maintain the new policy within a "proximal" region of the old policy.

8.6.1 Clipped Surrogate Objective

The key idea in PPO is to modify the standard policy gradient objective by incorporating an importance sampling ratio and then clipping this ratio to limit policy updates. Define the probability ratio between the new policy π_θ and the old policy $\pi_{\theta_{\text{old}}}$ at time step t as

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}.$$

The unclipped surrogate objective (or likelihood ratio objective) is given by

$$L^{\text{PG}}(\theta) = \mathbb{E}_t\Big[r_t(\theta)\, \hat{A}_t\Big],$$

where $\hat{A}_t$ is an estimator of the advantage function at time t.

To prevent excessively large policy updates, PPO employs a clipping mechanism:

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

with ϵ as a hyperparameter (typically around 0.1 or 0.2). The clipping function $\text{clip}(r_t(\theta),\, 1-\epsilon,\, 1+\epsilon)$ truncates the ratio so that it remains within the interval $[1-\epsilon,\, 1+\epsilon]$.

Intuition:

- When $\hat{A}_t > 0$ (indicating that action a_t is better than average), increasing $r_t(\theta)$ above $1 + \epsilon$ will not yield further improvement in the objective, as the term is capped.
- When $\hat{A}_t < 0$ (indicating that action a_t is worse than average), reducing $r_t(\theta)$ below $1 - \epsilon$ is similarly capped.

Thus, the clipping mechanism ensures that the objective is not overly sensitive to large deviations in the policy update.

8.6.2 Additional Objective Terms

In practice, PPO is often implemented in an actor-critic framework, where the following additional terms are included:

- **Value Function Loss:** To fit the value function $V_\theta(s)$, a mean-squared error loss is employed:
$$L^{\text{VF}}(\theta) = \mathbb{E}_t\Big[\big(V_\theta(s_t) - V_t^{\text{target}}\big)^2\Big],$$
where V_t^{target} is the target return, often computed using methods such as Generalized Advantage Estimation (GAE).
- **Entropy Bonus:** An entropy term is added to encourage exploration by preventing premature convergence:
$$L^{\text{S}}(\theta) = \mathbb{E}_t\Big[\mathcal{H}\big(\pi_\theta(\cdot \mid s_t)\big)\Big],$$
where $\mathcal{H}(\pi_\theta(\cdot \mid s_t))$ denotes the entropy of the policy at state s_t.

8.6.3 Final PPO Objective

The overall PPO objective is a weighted combination of the clipped surrogate objective, the value function loss, and the entropy bonus:

$$L^{\text{PPO}}(\theta) = \mathbb{E}_t\Big[L^{\text{CLIP}}(\theta) - c_1\, L^{\text{VF}}(\theta) + c_2\, L^{\text{S}}(\theta)\Big],$$

where c_1 and c_2 are hyperparameters that balance the contributions of the value loss and the entropy bonus, respectively.

8.6.4 Optimization Procedure

The optimization process in PPO typically follows these steps:

1. **Collect Data:** Use the current policy $\pi_{\theta_{\text{old}}}$ to generate a batch of trajectories.
2. **Compute Advantages:** Estimate the advantages $\hat{A}_t$ for each time step using techniques such as GAE.
3. **Optimize:** Perform several epochs of stochastic gradient ascent on the PPO objective $L^{\text{PPO}}(\theta)$ using mini-batches drawn from the collected data.
4. **Update Policy:** After the optimization, update the old policy parameters:
$$\theta_{\text{old}} \leftarrow \theta,$$
and repeat the process.

8.6.5 Summary

PPO achieves stable and reliable policy updates by:

- Utilizing an importance sampling ratio to relate the new policy to the old one.
- Clipping the ratio to prevent overly large updates.

- Incorporating additional terms for value function learning and exploration.

The final PPO objective is given by:

$$L^{\text{PPO}}(\theta) = \mathbb{E}_t \Bigg[\min\Big(r_t(\theta)\, \hat{A}_t,\ \text{clip}\big(r_t(\theta),\, 1-\epsilon,\, 1+\epsilon\big)\, \hat{A}_t \Big) \tag{58}$$

$$- c_1 \Big(V_\theta(s_t) - V_t^{\text{target}} \Big)^2 \tag{59}$$

$$+ c_2\, \mathcal{H}\Big(\pi_\theta(\cdot \mid s_t) \Big) \Bigg]. \tag{60}$$

This objective is optimized with respect to θ using first-order gradient methods, resulting in a robust and effective policy optimization algorithm.

In summary: We started with the basic policy gradient objective and derived a surrogate objective using importance sampling, which allows us to approximate the performance improvement when updating the policy. By bounding the difference between the new and old policies (using either a simple ϵ-bound or the KL divergence), we can guarantee that our updates improve the policy. Practical methods such as TRPO and PPO implement these ideas to achieve stable learning in high-dimensional and complex environments.

9 Optimal Control and Planning

In reinforcement learning (RL), the objective is to maximize the expected cumulative reward over trajectories:

$$p_\theta(s_1, a_1, \ldots, s_T, a_T) = p_\theta(\tau) = p(s_1) \prod_{t=1}^{T} \pi(a_t \mid s_t) p(s_{t+1} \mid s_t, a_t)$$

$$\theta^* = \arg\max_\theta E_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} r(s_t, a_t) \right].$$

In model-free RL, we do not assume knowledge of the dynamics $p(s_{t+1} \mid s_t, a_t)$. However, in many practical scenarios we either know the dynamics or can learn an approximate model. Knowing the dynamics allows us to use model-based approaches, which include methods from optimal control, trajectory optimization, and planning.

9.1 Model-based Reinforcement Learning

Model-based RL typically involves the following steps:

1. **Learn or use known dynamics:** Obtain $p(s_{t+1} \mid s_t, a_t)$ either from prior knowledge or by learning it from data.
2. **Decision Making:** With dynamics in hand, choose actions by leveraging planning or optimal control methods. This can be done by:
   (a) Solving an optimal control problem or performing trajectory optimization.
   (b) Planning over a horizon using methods like Monte Carlo Tree Search (MCTS).
3. **Policy Learning:** One may also imitate the behavior of an optimal controller to learn a policy.

9.2 The Control/Planning Objective

When dynamics are known (or estimated), the planning or optimal control problem is typically formulated as:

$$\min_{a_1, \ldots, a_T} \sum_{t=1}^{T} c(s_t, a_t) \quad \text{s.t. } s_t = f(s_{t-1}, a_{t-1}),$$

or equivalently, maximizing the cumulative reward:

$$a_1, \ldots, a_T = \arg\max_{a_1, \ldots, a_T} \sum_{t=1}^{T} r(s_t, a_t) \quad \text{s.t. } s_t = f(s_{t-1}, a_{t-1}).$$

Here, $c(s_t, a_t)$ is a cost function (often defined as the negative reward) and $f(s_{t-1}, a_{t-1})$ denotes the deterministic dynamics.

Deterministic Case

In the deterministic case, the optimal action sequence is given by:

$$a_1, \ldots, a_T = \arg\max_{a_1, \ldots, a_T} \sum_{t=1}^{T} r(s_t, a_t) \quad \text{s.t. } s_t = f(s_{t-1}, a_{t-1}).$$

This formulation is common in traditional optimal control and trajectory optimization.

Stochastic Open-Loop Case

In an open-loop formulation, the agent chooses an entire sequence of actions without receiving intermediate feedback. The dynamics are stochastic:

$$p_\theta(s_1, \ldots, s_T \mid a_1, \ldots, a_T) = p(s_1) \prod_{t=1}^{T} p(s_{t+1} \mid s_t, a_t).$$

The optimization objective is then:

$$a_1, \ldots, a_T = \arg\max_{a_1,\ldots,a_T} E\left[\sum_{t=1}^{T} r(s_t, a_t) \,\middle|\, a_1, \ldots, a_T \right].$$

Open-loop methods choose the entire action sequence in one shot, as opposed to closed-loop methods which update actions based on observed states.

Stochastic Closed-Loop Case

For closed-loop control (i.e., feedback control), the agent's policy is defined over state-action pairs:

$$p_\theta(s_1, a_1, \ldots, s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi(a_t \mid s_t) p(s_{t+1} \mid s_t, a_t),$$

and the optimal policy is obtained by solving:

$$\pi^* = \arg\max_{\pi} E_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} r(s_t, a_t) \right].$$

This is the classical formulation in RL, where decisions are made at each time step using state feedback.

9.3 Stochastic Optimization for Control and Planning

Optimal control and planning can also be viewed as a stochastic optimization problem:

$$a_1, \ldots, a_T = \arg\max_{a_1,\ldots,a_T} J(a_1, \ldots, a_T).$$

Let $A = (a_1, \ldots, a_T)$; then the problem reduces to:

$$A = \arg\max_{A} J(A).$$

Cross-Entropy Method (CEM)

The Cross-Entropy Method (CEM) is a popular stochastic optimization technique for trajectory optimization:

1. **Sampling:** Sample $A_1, \ldots, A_n$ from an initial distribution $p(A)$.
2. **Evaluation:** Compute the objective values $J(A_1), \ldots, J(A_n)$.
3. **Selection:** Select M elite samples $A_{i_1}, \ldots, A_{i_M}$ with the highest values, where $M < n$.
4. **Refitting:** Refit the distribution $p(A)$ (e.g., update the mean and variance) using the elite samples.
5. **Iteration:** Repeat until convergence.

Monte Carlo Tree Search (MCTS)

MCTS is a planning algorithm that constructs a search tree by exploring promising actions:

1. **Tree Expansion:** Starting at the root s_1, use `TreePolicy` to traverse the tree until a leaf s_l is reached.

2. **Rollout:** Use `DefaultPolicy` to simulate a trajectory from s_l and obtain an estimate of the value.
3. **Backpropagation:** Update the value Q and visit count N along the path from s_1 to s_l.
4. **Action Selection:** After several iterations, choose the best action from the root s_1 and repeat the process.

Each node in the tree stores an estimate of $Q(s)$ and the number of visits $N(s)$. A common tree policy, such as UCT (Upper Confidence Bound for Trees), selects the next node using:

$$\text{Score}(s_t) = \frac{Q(s_t)}{N(s_t)} + 2C\sqrt{\frac{2\ln N(s_{t-1})}{N(s_t)}},$$

where C is a tunable constant.

9.4 Optimal Control

When the dynamics are known, optimal control theory provides tools for solving the control problem. Consider the deterministic case:

$$\min_{u_1,\dots,u_T} \sum_{t=1}^{T} c(s_t, u_t) \quad \text{s.t. } x_t = f(x_{t-1}, u_{t-1}).$$

This expands as:

$$\min_{u_1,\dots,u_T} \Big[c(x_1, u_1) + c\big(f(x_1, u_1), u_2\big) + \cdots + c\big(f(f(\dots), u_T\big)\Big].$$

Shooting Methods vs. Collocation

- **Shooting Methods:** These methods (such as CEM) optimize over the sequence of control inputs $u_1, \dots, u_T$ by forward simulating the system.
- **Collocation Methods:** These methods simultaneously optimize over states and control inputs while enforcing the dynamics constraints $x_t = f(x_{t-1}, u_{t-1})$.

Linear Quadratic Regulator (LQR)

For linear dynamics and quadratic cost, the problem becomes tractable via the LQR framework. Suppose the dynamics are:

$$f(x_t, u_t) = F_t \begin{bmatrix} x_t \\ u_t \end{bmatrix} + f_t,$$

and the cost is quadratic:

$$c(x_t, u_t) = \frac{1}{2} \begin{bmatrix} x_t \\ u_t \end{bmatrix}^T C_t \begin{bmatrix} x_t \\ u_t \end{bmatrix} + \begin{bmatrix} x_t \\ u_t \end{bmatrix}^T c_t.$$

At the terminal time T, we define:

$$C_T = \begin{bmatrix} C_{x_T,x_T} & C_{x_T,u_T} \\ C_{u_T,x_T} & C_{u_T,u_T} \end{bmatrix}, \quad c_T = \begin{bmatrix} c_{x_T} \\ c_{u_T} \end{bmatrix}.$$

The base case is solved by minimizing the quadratic cost with respect to u_T:

$$Q(x_T, u_T) = \text{const} + \frac{1}{2} \begin{bmatrix} x_T \\ u_T \end{bmatrix}^T C_T \begin{bmatrix} x_T \\ u_T \end{bmatrix} + \begin{bmatrix} x_T \\ u_T \end{bmatrix}^T c_T, \tag{61}$$

$$\Delta_{u_T} Q(x_T, u_T) = C_{u_T,x_T} x_T + C_{u_T,u_T} u_T + c_{u_T} = 0, \tag{62}$$

$$u_T = -C_{u_T,u_T}^{-1}\Big(C_{u_T,x_T} x_T + c_{u_T}\Big). \tag{63}$$

We can write this as:

$$u_T = K_T x_T + k_T, \quad \text{with} \quad K_T = -C_{u_T,u_T}^{-1} C_{u_T,x_T}, \quad k_T = -C_{u_T,u_T}^{-1} c_{u_T}.$$

Substituting u_T into the cost, we express the terminal cost as a quadratic function of x_T:

$$V(x_T) = \text{const} + \frac{1}{2} x_T^T V_T x_T + x_T^T v_T.$$

Then, the backward recursion is performed from $t = T$ to 1:

1. At time t, the Q-function is given by:

$$Q(x_t, u_t) = \text{const} + \frac{1}{2} \begin{bmatrix} x_t \\ u_t \end{bmatrix}^T Q_t \begin{bmatrix} x_t \\ u_t \end{bmatrix} + \begin{bmatrix} x_t \\ u_t \end{bmatrix}^T q_t,$$

   where

$$Q_t = C_t + F_t^T V_{t+1} F_t, \quad q_t = c_t + F_t^T V_{t+1} f_t + F_t^T v_{t+1}.$$

2. The optimal control law is obtained by:

$$u_t = \arg\min_{u_t} Q(x_t, u_t) = K_t x_t + k_t, \quad K_t = -Q_{u_t,u_t}^{-1} Q_{u_t,x_t}, \quad k_t = -Q_{u_t,u_t}^{-1} q_{u_t}.$$

3. The value function is updated by:

$$V_t = Q_{x_t,x_t} + Q_{x_t,u_t} K_t + K_t^T Q_{u_t,x_t} + K_t^T Q_{u_t,u_t} K_t, \quad v_t = q_{x_t} + Q_{x_t,u_t} k_t + K_t^T q_{u_t}.$$

Finally, a forward recursion computes the state trajectory:

$$u_t = K_t x_t + k_t, \tag{64}$$

$$x_{t+1} = f(x_t, u_t). \tag{65}$$

Stochastic Dynamics

If the dynamics are stochastic, for example when

$$p(x_{t+1} \mid x_t, u_t) = \mathcal{N}\Big(F_t \begin{bmatrix} x_t \\ u_t \end{bmatrix} + f_t,\, \Sigma_t\Big),$$

and if the probability distribution is Gaussian with linear mean and fixed covariance, similar methods can be applied using the symmetry of the Gaussian distribution.

Nonlinear Systems: DDP and Iterative LQR

For nonlinear dynamics, one common approach is to use a Taylor expansion around a nominal trajectory. Let $\hat{x}_t, \hat{u}_t$ be a nominal trajectory. Then:

$$f(x_t, u_t) \approx f(\hat{x}_t, \hat{u}_t) + \frac{\partial f}{\partial x, u}\Big|_{(\hat{x}_t,\hat{u}_t)} \begin{bmatrix} x_t - \hat{x}_t \\ u_t - \hat{u}_t \end{bmatrix}, \tag{66}$$

$$c(x_t, u_t) \approx c(\hat{x}_t, \hat{u}_t) + \frac{\partial c}{\partial x, u}\Big|_{(\hat{x}_t,\hat{u}_t)} \begin{bmatrix} x_t - \hat{x}_t \\ u_t - \hat{u}_t \end{bmatrix} + \frac{1}{2} \begin{bmatrix} x_t - \hat{x}_t \\ u_t - \hat{u}_t \end{bmatrix}^T \frac{\partial^2 c}{\partial x, u^2}\Big|_{(\hat{x}_t,\hat{u}_t)} \begin{bmatrix} x_t - \hat{x}_t \\ u_t - \hat{u}_t \end{bmatrix}. \tag{67}$$

Defining the perturbations $\delta x_t = x_t - \hat{x}_t$ and $\delta u_t = u_t - \hat{u}_t$, one obtains a linear-quadratic approximation that can be solved by iterative LQR (iLQR) or Differential Dynamic Programming (DDP). These methods are essentially Newton's method applied to the trajectory optimization problem.

For more details on trajectory optimization, see:

1. **Differential Dynamic Programming** (1970).
2. **Synthesis and Stabilization of Complex Behaviors through Online Trajectory Optimization** (2012) — a practical guide for implementing nonlinear iterative LQR.
3. **Learning Neural Network Policies with Guided Policy Search under Unknown Dynamics** (2014) — which presents a probabilistic formulation and a trust region alternative to deterministic line search.

10 Model-Based Reinforcement Learning (Learning the Model)

Model-based reinforcement learning (MBRL) aims to learn a model of the environment's dynamics and, optionally, the reward function, so that planning or control techniques from optimal control can be applied. In this section, we describe several versions of MBRL, discuss challenges such as overfitting and distribution mismatch, and explain strategies to incorporate model uncertainty and deal with high-dimensional observations.

10.1 Basic Approach

Why learn the model?

If we knew $f(s_t, a_t) = s_{t+1}$ (or $p(s_{t+1} \mid s_t, a_t)$ in the stochastic case), we could directly apply optimal control and planning techniques from our previous courses.

A simple version of model-based RL (version 0.5) follows these steps:

1. **Data Collection:** Run a base policy $\pi_0(a_t \mid s_t)$ (e.g., a random policy) to collect a dataset
$$\mathcal{D} = \{(s, a, s')_i\}.$$
2. **Model Learning:** Learn a dynamics model $f(s, a)$ by minimizing the prediction error:
$$\min_f \sum_i \|f(s_i, a_i) - s'_i\|^2.$$
3. **Planning:** Use the learned model $f(s, a)$ to plan a sequence of actions.

Does it work?

- This approach mirrors the concept of *system identification* in classical robotics.
- The quality of the base policy is important because it determines the data distribution.
- It is particularly effective when we can incorporate physics knowledge into the model design (thus reducing the number of parameters to learn).
- A major limitation is that the learned model fits the data collected under the base policy; when the policy is updated, a **distribution mismatch problem** may occur.

10.2 Over-Fitting and Distribution Mismatch

10.2.1 Distribution Mismatch Problem

Can we do better?

The goal is to make the state distribution under the base policy match that of the final (improved) policy, i.e., $p_{\pi_0}(s_t) = p_{\pi_f}(s_t)$. One strategy (version 1.0) is to iteratively augment the dataset:

1. Run the base policy $\pi_0(a_t \mid s_t)$ to collect an initial dataset $\mathcal{D} = \{(s, a, s')_i\}$.
2. Learn the dynamics model $f(s, a)$ to minimize
$$\sum_i \|f(s_i, a_i) - s'_i\|^2.$$
3. Plan using $f(s, a)$ to choose actions.
4. Execute these actions, collect the resulting transitions $\{(s, a, s')_j\}$, and add them to $\mathcal{D}$.
5. Repeat steps 2–4.

However, model errors may accumulate, leading to poor performance if not handled carefully.

10.2.2 Model Predictive Control (MPC)

A further refinement (version 1.5) uses MPC to mitigate model error accumulation:

1. Collect an initial dataset $\mathcal{D}$ using $\pi_0(a_t \mid s_t)$.
2. Learn the dynamics model $f(s, a)$ from $\mathcal{D}$.
3. Plan through $f(s, a)$ to generate a sequence of candidate actions.
4. Execute only the **first** planned action and observe the resulting state s'.
5. Append the new transition (s, a, s') to $\mathcal{D}$ and re-plan every N steps.

MPC re-plans frequently, thereby reducing the impact of model errors.

10.3 Incorporating Model Uncertainty

Can we improve performance by considering model uncertainty?

When a model is learned from limited data, its predictions may be unreliable in regions not well covered by the data. We can improve robustness by estimating the model's uncertainty.

How to get uncertainty?

1. **Output Entropy:** One might use the entropy of the model's output distribution, though this is generally not effective.
2. **Bayesian Methods:** Compute

$$\int p(s_{t+1} \mid s_t, a_t, \theta)\, p(\theta \mid \mathcal{D})\, d\theta,$$

   which requires a distribution $p(\theta \mid \mathcal{D})$ over model parameters. Bayesian neural networks (BNN) can be employed for this purpose.
3. **Bootstrap Ensembles:** Train multiple models (an ensemble) and use the diversity among their predictions as a proxy for uncertainty:

$$p(\theta \mid \mathcal{D}) \approx \frac{1}{N} \sum_{i=1}^{N} \delta(\theta - \theta_i),$$

   leading to the predictive distribution approximation:

$$\int p(s_{t+1} \mid s_t, a_t, \theta)\, p(\theta \mid \mathcal{D})\, d\theta \approx \frac{1}{N} \sum_{i=1}^{N} p(s_{t+1} \mid s_t, a_t, \theta_i).$$

Training Considerations: To obtain diverse models, one can use re-sampling with replacement (bootstrapping) to generate multiple training datasets. However, in practice, random initialization and SGD often provide sufficient diversity without explicit re-sampling.

For evaluating a candidate action sequence $a_1, \ldots, a_H$:

1. Sample a model parameter θ from the approximate posterior.
2. For each time step t, sample s_{t+1} from $p(s_{t+1} \mid s_t, a_t, \theta)$.
3. Compute the total reward $R = \sum_{t=1}^{H} r(s_t, a_t)$.
4. Repeat the process to average the reward over multiple samples.

10.4 Model-Based RL with Images (POMDP)

High-dimensional observations, such as images, pose additional challenges due to redundancy and partial observability. One solution is to learn a latent space representation.

10.4.1 Model-Based RL with Latent Space Models

Challenges with Complex Observations:

- High dimensionality.
- Redundancy in the observations.
- Partial observability.

Approach:

- Learn an encoder $g_\psi(o_t)$ that maps raw observations o_t to a latent state s_t. For simplicity, we assume a deterministic encoder:

$$q_\psi(s_t \mid o_t) = \delta(s_t = g_\psi(o_t)) \quad \Longrightarrow \quad s_t = g_\psi(o_t).$$

- Learn a dynamics model $p_\phi(s_{t+1} \mid s_t, a_t)$ in the latent space, as well as a decoder $p_\phi(o_t \mid s_t)$ to reconstruct observations.
- Optionally, learn a reward model $p_\phi(r_t \mid s_t)$.

The training objective can combine prediction and reconstruction losses:

$$\max_{\phi,\psi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} E\Big[\log p_\phi\big(g_\psi(o_{t+1,i}) \mid g_\psi(o_{t,i}), a_{t,i}\big) + \log p_\phi(o_{t,i} \mid g_\psi(o_{t,i}))\Big],$$

or include a reward term:

$$\max_{\phi,\psi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} E\Big[\log p_\phi\big(g_\psi(o_{t+1,i}) \mid g_\psi(o_{t,i}), a_{t,i}\big) \tag{68}$$

$$+ \log p_\phi(o_{t,i} \mid g_\psi(o_{t,i})) \tag{69}$$

$$+ \log p_\phi(r_{t,i} \mid g_\psi(o_{t,i}))\Big]. \tag{70}$$

Overall Procedure:

1. **Data Collection:** Run a base policy $\pi_0(o_t \mid a_t)$ (e.g., a random policy) to collect data $\mathcal{D} = \{(o, a, o')_i\}$.
2. **Model Learning:** Learn the encoder $g_\psi(o_t)$, latent dynamics $p_\phi(s_{t+1} \mid s_t, a_t)$, decoder $p_\phi(o_t \mid s_t)$, and optionally the reward model $p_\phi(r_t \mid s_t)$.
3. **Planning:** Plan actions by simulating rollouts in the latent space.
4. **Execution (MPC):** Execute the **first** action from the planned sequence, observe the new observation o', and update $\mathcal{D}$.
5. **Iteration:** Repeat planning and execution, and periodically re-learn the model.

10.4.2 Learning Directly in Observation Space

An alternative is to directly learn a predictive model in observation space:

$$p(o_{t+1} \mid o_t, a_t),$$

and plan using image prediction. However, this approach typically requires much larger models and more data.

11 Model-Based RL and Policy Learning

In some applications, rather than relying solely on planning with a learned model, one may wish to learn an explicit policy that generalizes well and allows for closed-loop control without re-planning at every time step.

11.1 Basic Idea

Why learn a policy?

- Avoid the computational cost of re-planning at every time step.
- Achieve better generalization across states.
- Enable efficient closed-loop control.

The main idea is to use the learned dynamics model to generate training data (or even gradients) for a policy $\pi_\theta(a_t \mid s_t)$.

11.2 Model-Based RL Version 2.0: Policy Learning

A typical procedure is:

1. **Data Collection:** Run a base policy $\pi_0(a_t \mid s_t)$ (e.g., a random policy) to collect
$$\mathcal{D} = \{(s, a, s')_i\}.$$
2. **Model Learning:** Learn a dynamics model $f(s, a)$ by minimizing
$$\sum_i \|f(s_i, a_i) - s_i'\|^2.$$
3. **Policy Update:** Back-propagate through the model $f(s, a)$ into the policy parameters to optimize $\pi_\theta(a_t \mid s_t)$.
4. **Data Augmentation:** Execute the updated policy $\pi_\theta(a_t \mid s_t)$ in the environment, collect new transitions (s, a, s'), and add them to $\mathcal{D}$. Repeat steps 2–4.

Challenges:

- The process can be highly sensitive to parameters, similar to shooting methods.
- Unlike LQR-like methods, there is no convenient second-order update because the policy parameters affect the entire trajectory.
- Back-propagation through long sequences (akin to training deep RNNs with BPTT) can lead to vanishing or exploding gradients.

11.3 Guided Policy Search (GPS)

GPS combines trajectory optimization with policy learning by treating optimal control as an expert that guides the policy. Consider the constrained trajectory optimization problem:

$$\min_{u_1,\ldots,u_T,\, x_1,\ldots,x_T} \sum_{t=1}^{T} c(s_t, u_t) \quad \text{s.t.} \quad x_t = f(x_{t-1}, u_{t-1}), \quad u_t = \pi_\theta(x_t).$$

This can be solved by dual gradient descent, where the Lagrangian is constructed as:

$$\bar{\mathcal{L}}(\tau, \theta, \lambda) = c(\tau) + \sum_{t=1}^{T} \lambda_t\big(\pi_\theta(x_t) - u_t\big) + \sum_{t=1}^{T} \rho_t\big(\pi_\theta(x_t) - u_t\big)^2.$$

The overall process involves:

1. **Trajectory Optimization:** Optimize the trajectory τ (using methods like iLQR) with respect to a surrogate cost.

2. **Policy Update:** Train the policy π_θ in a supervised manner to match the optimized actions.
3. **Dual Updates:** Update the dual variables λ (and possibly ρ) to enforce the constraint $u_t = \pi_\theta(x_t)$.

GPS can be interpreted as:

- A constrained trajectory optimization method.
- Imitation learning from an adaptive optimal control expert.

11.3.1 Imitation Optimal Control with DAgger and PLATO

An alternative is to use a DAgger-like process:

1. From the current state, run a planning algorithm (e.g., MCTS) to generate a trajectory.
2. Add the state-action pair (s_t, a_t) from the planner to a dataset $\mathcal{D}$.
3. Execute the action from the current policy $\pi(a_t \mid s_t)$ and collect new data.
4. Repeat and update the policy with supervised learning.

PLATO is a variant that uses a stochastic behavior policy (e.g., a Gaussian policy)

$$\hat{\pi}(u_t \mid x_t) = \mathcal{N}(K_t x_t + k_t, \Sigma_{u_t}),$$

and optimizes a planning objective with a KL constraint:

$$\hat{\pi}(u_t \mid x_t) = \arg\min_{\hat{\pi}} \sum_{t'=t}^{T} E_{\hat{\pi}}\Big[c(x_{t'}, u_{t'})\Big] + \lambda\, D_{KL}\Big(\hat{\pi}(u_t \mid x_t) \,\|\, \pi_\theta(u_t \mid o_t)\Big).$$

This adaptive expert mitigates the issues in standard DAgger.

11.4 Model-Free Optimization with a Model

Even with a learned model, one can apply model-free RL methods (e.g., policy gradients) by treating the model as a simulator. In some cases, using the model gradients may work better than planning gradients.

11.4.1 Dyna

Dyna is an integrated approach that combines model-free learning with a learned model:

1. Given a state s, select an action a using an exploration policy.
2. Execute a, observe s' and reward r, and store the transition (s, a, s', r) in a buffer.
3. Update the model $\hat{p}(s' \mid s, a)$ (and optionally, $\hat{r}(s, a)$) using the collected data.
4. Perform Q-learning updates using both real transitions and simulated transitions from the model.

Dyna-style methods require only short rollouts from the model to avoid error accumulation.

11.5 Algorithm Summary and Trade-offs

11.5.1 Methods

- **Learn Model and Plan (Without Policy):**
  - Iteratively collect data to reduce distribution mismatch.
  - Use MPC to re-plan at each step, mitigating model error.
- **Learning a Policy:**
  - Back-propagate through the learned model to update the policy (e.g., PILCO).
  - Use guided policy search (GPS) to combine trajectory optimization with policy learning.
  - Apply imitation learning techniques such as DAgger or PLATO.
  - Use model-free methods on simulated data (e.g., Dyna).

11.5.2 Limitations of Model-Based RL

- **Requirement for a Model:** A model may not be available, and learning an accurate model can be as challenging as learning a policy.
- **Time and Data Demands:** Expressive models (e.g., deep networks) require significant data and computational resources.
- **Additional Assumptions:** Assumptions such as linearizability, smoothness, or the ability to reset the environment are often necessary.

11.5.3 Intuitive Understanding

- **Why use model-based RL?** Model-free RL relies on extensive exploration over large state spaces. MBRL leverages a learned model to plan promising trajectories, thereby reducing the need for random exploration.
- **Why not use optimal control alone?** Direct application of optimal control (e.g., MPC) is often limited by model inaccuracies and the difficulty in handling high-dimensional or complex observations. Learning a policy from planned trajectories (or using guided policy search) can yield better closed-loop performance.
- **Available Methods:** Options include planning without a policy, learning a policy via guided search, and using model-based simulators to enhance model-free learning.

11.6 Choosing the Right Algorithm

The appropriate algorithm depends on the trade-off between sample efficiency and computational efficiency. Consider the following:

- **Gradient-free Methods:** (e.g., NES, CMA-ES) often require many samples but less computational effort per sample.
- **Online Methods:** (e.g., A3C) update the policy at each step.
- **Policy Gradient Methods:** (e.g., TRPO) can offer stable updates but may be computationally intensive.
- **Replay-Buffer Value Estimation:** (e.g., Q-learning, DDPG, NAF, SAC) balance sample efficiency and stability.
- **Model-Based Deep RL:** (e.g., PETS, guided policy search) offer high sample efficiency at the cost of increased computation.
- **Model-Based "Shallow" RL:** (e.g., PILCO) can be extremely sample efficient in low-dimensional tasks.

The choice depends on factors such as the task complexity, available computational resources, and the need for sample efficiency.

In summary, model-based reinforcement learning leverages a learned or known dynamics model to reduce exploration requirements by shifting the problem to planning. By iteratively updating the model and addressing issues such as distribution mismatch and model uncertainty, MBRL can provide significant advantages in sample efficiency. Moreover, integrating model-based techniques with policy learning (via guided policy search, DAgger, or Dyna) can yield robust and generalizable controllers.

12 Variational Inference and Generative Models

Variational inference provides a framework for learning probabilistic models with latent variables. Generative models such as Variational Autoencoders (VAEs) leverage latent variable models to capture complex data distributions. In this section, we first review probabilistic models with latent variables, then derive the variational lower bound, and finally explain how to optimize these models using the re-parameterization trick and amortized inference.

12.1 Probabilistic Models and Latent Variable Models

Latent variable models assume that the observed data x is generated from some unobserved (latent) variable z. In general, the marginal likelihood of x is given by:

$$p(x) = \sum_z p(x \mid z)p(z)$$

or, in the continuous case,

$$p(x) = \int p(x \mid z)p(z)\, dz.$$

For conditional models, where one wishes to predict y given x, the model is:

$$p(y \mid x) = \sum_z p(y \mid x, z)p(z).$$

A common approach is to assume a simple prior for z, such as a Gaussian:

$$p(z) = \mathcal{N}(0, I),$$

and to model the conditional likelihood with a neural network:

$$p(x \mid z) = \mathcal{N}(\mu_{\text{nn}}(z), \sigma_{\text{nn}}(z)).$$

Thus, by feeding a random Gaussian vector z into the neural network, we can model a wide range of complex distributions over x.

In reinforcement learning (RL), latent variable models are useful for representing multi-modal policies or behaviors, where the latent variables capture different modes of the expert's behavior.

12.2 Training Latent Variable Models via Maximum Likelihood

Suppose we wish to model the data distribution with a model $p_\theta(x)$ given a dataset $\mathcal{D} = \{x_1, x_2, \ldots, x_N\}$. In a latent variable model, the marginal likelihood is

$$p_\theta(x) = \int p_\theta(x \mid z)p(z)\, dz.$$

A maximum likelihood approach would optimize

$$\theta^* = \arg\max_\theta \frac{1}{N} \sum_{i=1}^N \log p_\theta(x_i) = \arg\max_\theta \frac{1}{N} \sum_{i=1}^N \log \left(\int p_\theta(x_i \mid z)p(z)\, dz \right).$$

An alternative is to maximize the expected joint log-likelihood under a posterior over the latent variable:

$$\theta^* = \arg\max_\theta \frac{1}{N} \sum_{i=1}^N E_{z \sim p(z \mid x_i)} \log p_\theta(x_i, z).$$

Since the true posterior $p(z \mid x_i)$ is generally intractable, we instead introduce a variational distribution $q_i(z)$ to approximate it.

12.3 The Variational Approximation

For a given observation x_i, the model defines

$$p_\theta(x_i) = \int p_\theta(x_i \mid z) p(z)\, dz.$$

We can write:

$$\begin{aligned}
\log p_\theta(x_i) &= \log \int p_\theta(x_i \mid z) p(z)\, dz \\
&= \log \int p_\theta(x_i \mid z) p(z) \frac{q_i(z)}{q_i(z)}\, dz \\
&= \log E_{z\sim q_i(z)} \left[\frac{p_\theta(x_i \mid z) p(z)}{q_i(z)} \right] \\
&\geq E_{z\sim q_i(z)} \left[\log \frac{p_\theta(x_i \mid z) p(z)}{q_i(z)} \right] \quad \text{(Jensen's inequality)} \\
&= E_{z\sim q_i(z)} \left[\log p_\theta(x_i \mid z) + \log p(z) \right] - E_{z\sim q_i(z)} \left[\log q_i(z) \right] \\
&= E_{z\sim q_i(z)} \left[\log p_\theta(x_i \mid z) + \log p(z) \right] + \mathcal{H}(q_i),
\end{aligned}$$

where the entropy of q_i is defined as

$$\mathcal{H}(q_i) = -E_{z\sim q_i(z)} \left[\log q_i(z) \right].$$

The Kullback-Leibler (KL) divergence between $q_i(z)$ and the true posterior $p(z \mid x_i)$ is

$$\begin{aligned}
D_{KL}(q_i(z) \,\|\, p(z \mid x_i)) &= E_{z\sim q_i(z)} \left[\log \frac{q_i(z)}{p(z \mid x_i)} \right] \\
&= E_{z\sim q_i(z)} \left[\log \frac{q_i(z) p_\theta(x_i)}{p_\theta(x_i, z)} \right] \\
&= -E_{z\sim q_i(z)} \left[\log p_\theta(x_i \mid z) + \log p(z) \right] + E_{z\sim q_i(z)} \left[\log q_i(z) \right] + \log p_\theta(x_i) \\
&= -\mathcal{L}_i(p, q_i) + \log p_\theta(x_i),
\end{aligned}$$

where we define the variational lower bound (ELBO) for x_i as

$$\mathcal{L}_i(p, q_i) = E_{z\sim q_i(z)} \left[\log p_\theta(x_i \mid z) + \log p(z) \right] + \mathcal{H}(q_i).$$

Thus, we obtain:

$$\log p_\theta(x_i) = D_{KL}(q_i(z) \,\|\, p(z \mid x_i)) + \mathcal{L}_i(p, q_i),$$

which implies

$$\log p_\theta(x_i) \geq \mathcal{L}_i(p, q_i).$$

Maximizing $\mathcal{L}_i(p, q_i)$ (and thereby minimizing the KL divergence) yields a better approximation of the true posterior $p(z \mid x_i)$.

Our training objective becomes:

$$\theta^* = \arg\max_\theta \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_i(p, q_i),$$

with

$$\mathcal{L}_i(p, q_i) = E_{z\sim q_i(z)} \left[\log p_\theta(x_i \mid z) + \log p(z) \right] + \mathcal{H}(q_i).$$

12.4 Amortized Variational Inference

Learning a separate $q_i(z)$ for each data point x_i can be inefficient. Instead, we use a shared inference network (encoder) $q_\phi(z \mid x)$ that approximates the posterior for all data points. For example, we can choose:

$$q_\phi(z \mid x) = \mathcal{N}\big(\mu_\phi(x), \sigma_\phi(x)\big).$$

The variational lower bound for each x_i becomes:

$$\mathcal{L}(x_i) = E_{z\sim q_\phi(z|x_i)}\left[\log p_\theta(x_i \mid z) + \log p(z)\right] + \mathcal{H}\big(q_\phi(z \mid x_i)\big).$$

We then optimize the parameters θ (of the generative model) and ϕ (of the inference network) jointly by maximizing:

$$\theta^*, \phi^* = \arg\max_{\theta,\phi} \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(x_i).$$

12.5 The Re-parameterization Trick

A key challenge in optimizing the ELBO with respect to ϕ is that the expectation is taken over $q_\phi(z \mid x)$, whose parameters depend on ϕ. A naive estimator based on the REINFORCE trick has high variance. Instead, the re-parameterization trick expresses the latent variable z as a deterministic function of ϕ and a noise variable ϵ:

$$z = \mu_\phi(x) + \epsilon\, \sigma_\phi(x), \quad \epsilon \sim \mathcal{N}(0, I).$$

Then, the expectation becomes:

$$E_{z\sim q_\phi(z|x)}\Big[\cdot\Big] = E_{\epsilon\sim\mathcal{N}(0,I)}\Big[\cdot\,\Big(\mu_\phi(x) + \epsilon\,\sigma_\phi(x)\Big)\Big].$$

The gradient with respect to ϕ can then be computed by back-propagation through the deterministic function, yielding a low-variance estimator:

$$\Delta_\phi \mathcal{L}(x) \approx \frac{1}{M}\sum_{j=1}^{M} \nabla_\phi \log p_\theta\Big(x,\, \mu_\phi(x) + \epsilon_j\, \sigma_\phi(x)\Big),$$

where $\epsilon_1, \ldots, \epsilon_M$ are samples from $\mathcal{N}(0, I)$.

12.6 Variational Autoencoder (VAE)

The variational autoencoder (VAE) is a concrete implementation of amortized variational inference in which the generative model $p_\theta(x \mid z)$ (the decoder) and the inference network $q_\phi(z \mid x)$ (the encoder) are jointly trained by maximizing the variational lower bound:

$$\mathcal{L}(x) = E_{z\sim q_\phi(z|x)}\left[\log p_\theta(x \mid z)\right] - D_{KL}\Big(q_\phi(z \mid x)\,\|\,p(z)\Big).$$

This objective encourages the encoder to produce latent representations that both allow accurate reconstruction of the input and remain close to the prior $p(z)$. The re-parameterization trick makes it possible to optimize this objective end-to-end via back-propagation.

12.7 Conditional Models and Extensions

For conditional generative models, where the goal is to model $p(y \mid x)$, the latent variable model takes the form:

$$p(y \mid x) = \int p_\theta(y \mid x, z) p(z \mid x)\, dz.$$

In the variational framework, we introduce an approximate posterior $q_\phi(z \mid x, y)$ and optimize the corresponding lower bound:

$$\mathcal{L}(x, y) = E_{z\sim q_\phi(z|x,y)}\Big[\log p_\theta(y \mid x, z) + \log p(z \mid x)\Big] + \mathcal{H}\big(q_\phi(z \mid x, y)\big).$$

Variational inference has broad applications, including:

- Modeling complex multi-modal distributions.
- Enhancing exploration in reinforcement learning.
- Capturing multi-modal policies in RL using conditional latent variable models.

Intuition: Variational inference enables us to represent complex data distributions by introducing latent variables that capture the underlying factors of variation. In the VAE framework, the encoder compresses the data into a latent representation, and the decoder reconstructs the data from this representation. The variational lower bound ensures that the learned latent space is both informative for reconstruction and regularized to follow a known prior (typically a standard Gaussian). The re-parameterization trick provides a means to efficiently back-propagate gradients through stochastic sampling operations, allowing end-to-end training of these generative models.

This approach is fundamental to modern generative modeling and is widely used in applications ranging from image synthesis to reinforcement learning, where latent variable models can capture multi-modal and complex behaviors.

13 Re-framing Control as an Inference Problem

In this section, we reinterpret the classical control and decision-making problem as one of probabilistic inference. Instead of directly optimizing a policy to maximize cumulative reward, we cast control as inferring the most probable trajectory under a distribution that favors high rewards. This perspective unifies planning, optimal control, and reinforcement learning under a common probabilistic framework.

13.1 From Policy to Objective via Inference

The key idea is to reinterpret decision making as an inference problem. In many real-world scenarios, human behavior is stochastic and not perfectly optimal—but overall it achieves good performance. By introducing a probabilistic model of optimality, we can capture this behavior.

13.2 A Probabilistic Graphical Model of Decision Making

We introduce an *optimality variable* $\mathcal{O}_t$ at each time step, which is a Boolean variable indicating whether the action taken at time t is "optimal" or desirable. We define this variable through the likelihood:

$$p(\mathcal{O}_t \mid s_t, a_t) = \exp\big(r(s_t, a_t)\big),$$

so that higher rewards correspond to higher probabilities of optimality. Under this formulation, the joint probability of a trajectory $\tau = (s_1, a_1, \ldots, s_T, a_T)$ and all optimality variables is:

$$p(\tau, \mathcal{O}_{1:T}) = p(\tau) \prod_{t=1}^{T} \exp\big(r(s_t, a_t)\big),$$

where $p(\tau) = p(s_1) \prod_{t=1}^{T} \pi(a_t \mid s_t) p(s_{t+1} \mid s_t, a_t)$ is the usual trajectory probability. Thus, the posterior over trajectories conditioned on optimality is given by

$$p(\tau \mid \mathcal{O}_{1:T}) = \frac{p(\tau, \mathcal{O}_{1:T})}{p(\mathcal{O}_{1:T})} \propto p(\tau) \exp\Big(\sum_{t=1}^{T} r(s_t, a_t)\Big).$$

This result shows that maximizing the cumulative reward is equivalent to finding the most probable trajectory given that all actions are "optimal."

13.3 Inference via Backward Messages

To perform inference in this graphical model, we define backward messages that capture the probability of future optimality given the current state and action. Let

$$\beta_t(s_t, a_t) = p(\mathcal{O}_{t:T} \mid s_t, a_t),$$

which can be computed recursively. Assuming that the optimality variables factorize over time, we have:

$$\beta_t(s_t, a_t) = p(\mathcal{O}_t \mid s_t, a_t)\; E_{s_{t+1} \sim p(s_{t+1} \mid s_t, a_t)}\big[\beta_{t+1}(s_{t+1})\big],$$

and by marginalizing over actions:

$$\beta_t(s_t) = \int \beta_t(s_t, a_t)\, da_t.$$

It is useful to work in the log-domain. Define

$$V_t(s_t) = \log \beta_t(s_t) \quad \text{and} \quad Q_t(s_t, a_t) = \log \beta_t(s_t, a_t).$$

Then the relationship between state and action values is:

$$V_t(s_t) = \log \int \exp\big(Q_t(s_t, a_t)\big)\, da_t.$$

In the limit of large Q_t values, $V_t(s_t)$ approximates $\max_{a_t} Q_t(s_t, a_t)$. The recursion for the Q-function is then:

$$Q_t(s_t, a_t) = r(s_t, a_t) + \log E_{s_{t+1} \sim p(s_{t+1} \mid s_t, a_t)}\Big[\exp\big(V_{t+1}(s_{t+1})\big)\Big].$$

For deterministic transitions, this simplifies to the familiar recursion:

$$Q_t(s_t, a_t) = r(s_t, a_t) + V_{t+1}(s_{t+1}).$$

13.4 Policy Computation via Inference

The optimal policy can be derived by conditioning on the optimality variables. We want to compute:

$$\pi(a_t \mid s_t) = p(a_t \mid s_t, \mathcal{O}_{1:T}).$$

Using Bayes' rule and the definition of backward messages, we have

$$\begin{aligned} p(a_t \mid s_t, \mathcal{O}_{1:T}) &= \frac{p(\mathcal{O}_{1:T} \mid s_t, a_t)\, p(a_t \mid s_t)}{p(\mathcal{O}_{1:T} \mid s_t)} \\ &= \frac{\beta_t(s_t, a_t)}{\beta_t(s_t)}\, p(a_t \mid s_t). \end{aligned}$$

If we assume a uniform (or uninformative) prior $p(a_t \mid s_t)$, this simplifies to:

$$\pi(a_t \mid s_t) = \frac{\beta_t(s_t, a_t)}{\beta_t(s_t)}.$$

Expressing this in the log-domain using our value functions, we obtain:

$$\pi(a_t \mid s_t) = \exp\Big(Q_t(s_t, a_t) - V_t(s_t)\Big).$$

More generally, introducing a temperature parameter α yields:

$$\pi(a_t \mid s_t) = \exp\Big(\frac{1}{\alpha} Q_t(s_t, a_t) - \frac{1}{\alpha} V_t(s_t)\Big) = \exp\Big(\frac{1}{\alpha} A_t(s_t, a_t)\Big),$$

where $A_t(s_t, a_t) = Q_t(s_t, a_t) - V_t(s_t)$ is analogous to an advantage function. As α approaches zero, the policy becomes nearly deterministic (favoring the action with the maximum Q_t); for larger α, the policy is more stochastic.

13.5 Forward Messages and Their Role

In addition to backward messages, forward messages propagate information from the past. Define the forward message as

$$\alpha_t(s_t) = p(s_t \mid \mathcal{O}_{1:t-1}),$$

which can be written as

$$\alpha_t(s_t) = \int p(s_t, s_{t-1}, a_{t-1} \mid \mathcal{O}_{1:t-1})\, ds_{t-1}\, da_{t-1}.$$

Expanding this using the dynamics and the policy (conditioned on past optimality), one can show that the smoothed state distribution given all optimality is proportional to the product of forward and backward messages:

$$p(s_t \mid \mathcal{O}_{1:T}) \propto \alpha_t(s_t)\, \beta_t(s_t).$$

This decomposition is useful for algorithms that perform inference over entire trajectories.

13.6 The Optimism Problem in Inference-Based Control

A challenge arises when conditioning on optimality: the backward recursion becomes

$$\beta_t(s_t, a_t) = p(\mathcal{O}_t \mid s_t, a_t)\; E_{s_{t+1} \sim p(s_{t+1} \mid s_t, a_t)}\big[\beta_{t+1}(s_{t+1})\big],$$

and defining

$$V_t(s_t) = \log \beta_t(s_t), \quad Q_t(s_t, a_t) = \log \beta_t(s_t, a_t),$$

we obtain

$$Q_t(s_t, a_t) = r(s_t, a_t) + \log E\Big[\exp\big(V_{t+1}(s_{t+1})\big)\Big].$$

Because we condition on trajectories with high cumulative reward, the inferred transition probabilities become biased toward optimistic outcomes. In other words, $p(s_{t+1} \mid s_t, a_t, \mathcal{O}_{1:T})$ is not equal to the true dynamics $p(s_{t+1} \mid s_t, a_t)$. To address this *optimism problem*, one can employ variational inference techniques to find a distribution $q(s_{1:T}, a_{1:T})$ that is close to $p(s_{1:T}, a_{1:T} \mid \mathcal{O}_{1:T})$ while respecting the true dynamics.

13.7 Control via Variational Inference

We can cast control as a variational inference problem. Suppose we define a variational distribution over trajectories as

$$q(s_{1:T}, a_{1:T}) = p(s_1) \prod_{t=1}^{T} p(s_{t+1} \mid s_t, a_t)\, q(a_t \mid s_t),$$

where $q(a_t \mid s_t)$ is a variational policy. By applying standard variational inference, we obtain a lower bound on $\log p(\mathcal{O}_{1:T})$:

$$\begin{aligned}\log p(\mathcal{O}_{1:T}) &\geq E_{(s_{1:T}, a_{1:T}) \sim q}\left[\sum_{t=1}^{T} \log p(\mathcal{O}_t \mid s_t, a_t) - \log q(a_t \mid s_t)\right] \\ &= E_{(s_{1:T}, a_{1:T}) \sim q}\left[\sum_{t=1}^{T} \big(r(s_t, a_t) + \mathcal{H}\big(q(a_t \mid s_t)\big)\big)\right].\end{aligned}$$

Maximizing this lower bound corresponds to maximizing the expected reward while also encouraging high-entropy (exploratory) policies.

A soft Bellman backup then follows:

$$Q_t(s_t, a_t) = r(s_t, a_t) + E_{s_{t+1} \sim p(s_{t+1} \mid s_t, a_t)}\Big[V_{t+1}(s_{t+1})\Big],$$

with

$$V_t(s_t) = \alpha \log \int \exp\Big(\frac{1}{\alpha} Q_t(s_t, a_t)\Big) da_t,$$

where the temperature α controls the trade-off between reward maximization and entropy.

13.8 Policy Gradient with Soft Optimality

Under the variational formulation, the optimal policy that maximizes the variational lower bound is given by

$$\pi(a_t \mid s_t) = \frac{\beta_t(s_t, a_t)}{\beta_t(s_t)} = \exp\Big(Q_t(s_t, a_t) - V_t(s_t)\Big).$$

Introducing a temperature parameter α, we can write:

$$\pi(a_t \mid s_t) = \exp\Big(\frac{1}{\alpha} Q_t(s_t, a_t) - \frac{1}{\alpha} V_t(s_t)\Big) = \exp\Big(\frac{1}{\alpha} A_t(s_t, a_t)\Big).$$

This form naturally connects to soft policy gradient and soft Q-learning methods, where policies are proportional to the exponentiated Q-values, leading to smoother and more robust learning.

13.9 Benefits of the Inference Formulation

Re-framing control as an inference problem offers several advantages:

- **Exploration via Entropy:** Including an entropy term in the objective encourages exploration and prevents premature convergence to suboptimal deterministic policies.
- **Robustness:** The soft Bellman backup integrates over all actions, leading to smoother value estimates and inherent tie-breaking.
- **Human Behavior Modeling:** The probabilistic formulation naturally captures the stochastic and sometimes suboptimal behavior observed in human decision making.
- **Connection to Soft Q-Learning:** As the temperature parameter α decreases, the formulation approaches the hard optimal control case, while higher α values yield more exploratory, stochastic policies.

In summary, by re-framing control as an inference problem, we derive a probabilistic framework in which the task of control becomes equivalent to inferring a trajectory (or policy) that is most likely under a distribution weighted by rewards. This leads to soft value functions, policy representations that blend reward and entropy, and robust algorithms that are closely connected to soft Q-learning and policy gradient methods.

14 Inverse Reinforcement Learning

Inverse Reinforcement Learning (IRL) seeks to infer the underlying reward function from expert demonstrations. Unlike standard imitation learning—which focuses solely on copying actions—IRL tries to recover the expert's intent or objective. This approach is particularly useful when the reward function is complicated, ambiguous, or unknown.

14.1 Why Should We Worry About Learning Rewards?

There are two perspectives on imitation:

- **Standard Imitation Learning:**
  - Simply copy the action performed by the expert.
  - No reasoning is done regarding the outcomes or long-term consequences.
- **Human Imitation Learning:**
  - Aim to infer the *intent* of the expert.
  - Different actions may be taken as long as the outcome is similar.

From the reinforcement learning perspective, the reward function itself can be complex and unclear. Instead of imitating actions directly, it may be more beneficial to infer the underlying reward structure and then learn an optimal policy accordingly.

14.2 Inverse Reinforcement Learning Formulation

In IRL, we are given:

- A state space $\mathcal{S}$ and an action space $\mathcal{A}$.
- (Optionally) a transition model $p(s' \mid s, a)$.
- A set of demonstration trajectories $\{\tau_i\}$ sampled from the expert policy $\pi^*(\tau)$.

Our goal is to learn a reward function $r_\psi(s, a)$ such that, when used in a reinforcement learning algorithm, the resulting policy $\pi^*(a \mid s)$ reproduces the expert's behavior.

14.3 Learning the Optimality Variable

We introduce an optimality variable $\mathcal{O}_t$ that encodes whether an action at time t is "optimal." Define its likelihood as:

$$p(\mathcal{O}_t \mid s_t, a_t, \psi) = \exp\big(r_\psi(s_t, a_t)\big).$$

Then the probability of a trajectory conditioned on all optimality variables is

$$p(\tau \mid \mathcal{O}_{1:T}, \psi) \propto p(\tau) \exp\left(\sum_{t=1}^{T} r_\psi(s_t, a_t)\right).$$

This formulation means that trajectories yielding higher cumulative rewards are exponentially more likely to be considered optimal.

14.4 The IRL Partition Function and Likelihood Objective

Given a set of expert trajectories $\{\tau_i\}_{i=1}^N$, we can define the maximum likelihood objective as

$$\max_\psi \frac{1}{N} \sum_{i=1}^{N} \log p(\tau_i \mid \mathcal{O}_{1:T}, \psi) = \max_\psi \frac{1}{N} \sum_{i=1}^{N} \Big(r_\psi(\tau_i) - \log Z\Big),$$

where

$$Z = \int p(\tau) \exp\big(r_\psi(\tau)\big)\, d\tau$$

is the partition function that normalizes the probabilities. The gradient with respect to ψ is

$$\Delta_\psi \mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \Delta_\psi r_\psi(\tau_i) - \frac{1}{Z} \int p(\tau) \exp\big(r_\psi(\tau)\big) \Delta_\psi r_\psi(\tau)\, d\tau.$$

This can be rewritten as

$$\Delta_\psi \mathcal{L} = E_{\tau \sim \pi^*(\tau)} \big[\Delta_\psi r_\psi(\tau)\big] - E_{\tau \sim p(\tau|\mathcal{O}_{1:T},\psi)} \big[\Delta_\psi r_\psi(\tau)\big].$$

Thus, the update drives the learned reward to make expert trajectories more likely while penalizing trajectories generated by the current policy.

14.5 Estimating the Expectation

Let the visitation frequency of state-action pairs under the optimality-conditioned distribution be defined as

$$\mu_t(s_t, a_t) \propto \beta(s_t, a_t)\, \alpha(s_t),$$

where β and α are backward and forward messages respectively. Then, the expectation over trajectories can be decomposed as

$$E_{\tau \sim p(\tau|\mathcal{O}_{1:T},\psi)} \big[\Delta_\psi r_\psi(\tau)\big] = \sum_{t=1}^{T} E_{(s_t,a_t) \sim p(s_t,a_t|\mathcal{O}_{1:T},\psi)} \big[\Delta_\psi r_\psi(s_t, a_t)\big] = \sum_{t=1}^{T} \mu_t^\top \Delta_\psi \mathbf{r}_\psi.$$

14.6 The MaxEnt IRL Algorithm

The maximum entropy (MaxEnt) IRL framework seeks to infer the reward function that best explains the expert's behavior while favoring high-entropy policies. The overall algorithm proceeds as follows:

1. **Backward Pass:** Given the current reward parameters ψ, compute the backward message $\beta(s_t, a_t)$ for all state-action pairs.
2. **Forward Pass:** Compute the forward message $\alpha(s_t)$ and the visitation frequencies $\mu_t(s_t, a_t) \propto \beta(s_t, a_t)\alpha(s_t)$.
3. **Gradient Evaluation:** Evaluate the gradient

$$\Delta_\psi \mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \Delta_\psi r_\psi(s_{i,t}, a_{i,t}) - \sum_{t=1}^{T} \int \mu_t(s_t, a_t)\, \Delta_\psi r_\psi(s_t, a_t)\, ds_t\, da_t.$$

4. **Update:** Adjust ψ by a small step:

$$\psi \leftarrow \psi + \eta\, \Delta_\psi \mathcal{L}.$$

When the reward is linear in features, $r_\psi(s, a) = \psi^\top f(s, a)$, it can be shown that maximizing the likelihood is equivalent to matching feature expectations between the expert and the learned policy while maximizing policy entropy (see Ziebart et al., 2008).

14.7 Efficient Sample-Based Updates

Direct evaluation of the partition function and visitation frequencies can be intractable for large or continuous state-action spaces. Practical approaches include:

Guided Cost Learning: Approximate the gradient using samples:

$$\Delta_\psi \mathcal{L} \approx \frac{1}{N} \sum_{i=1}^{N} \Delta_\psi r_\psi(\tau_i) - \frac{1}{M} \sum_{j=1}^{M} \Delta_\psi r_\psi(\tau_j),$$

where the τ_j are generated from a soft optimal policy induced by the current reward.

Importance Sampling: Use importance weights w_j to reweight samples from the current policy:

$$\Delta_\psi \mathcal{L} \approx \frac{1}{N} \sum_{i=1}^{N} \Delta_\psi r_\psi(\tau_i) - \frac{1}{\sum_j w_j} \sum_{j=1}^{M} w_j \, \Delta_\psi r_\psi(\tau_j),$$

with

$$w_j = \frac{p(\tau) \exp\big(r_\psi(\tau_j)\big)}{\pi(\tau_j)} = \frac{\exp\Big(\sum_t r_\psi(s_t, a_t)\Big)}{\prod_t \pi(a_t \mid s_t)}.$$

IRL as a Generative Adversarial Network (GAN): Alternatively, one can view IRL through the lens of GANs. Define a discriminator $D_\psi(\tau)$ that distinguishes expert trajectories from those generated by the policy:

$$D_\psi(\tau) = \frac{p(\tau) \frac{1}{Z} \exp\big(r_\psi(\tau)\big)}{p(\tau) \frac{1}{Z} \exp\big(r_\psi(\tau)\big) + \pi(\tau)}.$$

The discriminator is trained to maximize:

$$\psi \leftarrow \arg\max_\psi E_{\tau \sim p^*(\tau)} \big[\log D_\psi(\tau)\big] + E_{\tau \sim \pi(\tau)} \big[\log(1 - D_\psi(\tau))\big],$$

and the policy is updated to minimize:

$$\Delta_\theta \mathcal{L} \approx \frac{1}{M} \sum_{j=1}^{M} \Delta_\theta \log \pi_\theta(\tau_j) \, \log D_\psi(\tau_j).$$

This adversarial formulation forces the learned policy to imitate the expert by making the generated trajectories indistinguishable from the expert's.

14.8 Summary and Intuitive Remarks

Inverse reinforcement learning seeks to recover a reward function $r_\psi(s, a)$ that explains expert behavior. In the maximum entropy framework, the likelihood of a trajectory is weighted by the exponential of its cumulative reward:

$$p(\tau \mid \mathcal{O}_{1:T}, \psi) \propto p(\tau) \exp\Big(\sum_t r_\psi(s_t, a_t)\Big).$$

The gradient update for the reward parameters involves the difference between the gradients computed on expert demonstrations and those computed on trajectories generated by the current (soft) policy. To make these updates tractable, sample-based methods, importance sampling, and even adversarial formulations (similar to GANs) are employed.

Intuitive Points:

- IRL is underspecified; many reward functions can explain the same behavior. Maximum entropy IRL biases the solution toward reward functions that lead to high-entropy (i.e., diverse) policies.
- Instead of merely imitating actions, IRL infers the underlying objectives that generate those actions.
- Efficient sample-based updates are critical for handling large or continuous state-action spaces.

15 Transfer and Multi-task Learning

Transfer learning in reinforcement learning (RL) involves using prior knowledge or experience acquired in one set of tasks (source domain) to improve the learning speed or performance on a new task (target domain). In RL, a task is typically defined by its Markov Decision Process (MDP).

Shot Types:

- **0-shot:** Directly applying a policy trained in the source domain to the target domain without any additional adaptation.
- **1-shot:** The target task is attempted once, relying on minimal adaptation from the transferred knowledge.
- **Few-shot:** The agent is allowed a small number of attempts in the target domain to fine-tune or adapt the transferred policy.

Approaches to Transfer and Multi-task Learning:

1. **Forward Transfer:** Train on a single source task and then transfer to a new target task.
   (a) *Direct Transfer:* Simply apply the learned policy to the target task and hope that it performs well.
   (b) *Fine-tuning:* Use the pre-trained policy as an initialization and further train it on the new task.
   (c) *Domain Randomization:* Train on a range of randomized source tasks to develop a more robust policy that generalizes better to new environments.
2. **Multi-task Transfer:** Train simultaneously on a variety of tasks, so that the learned policy (or representation) can generalize to new tasks.
   (a) *Highly Randomized Source Domains:* Expose the agent to a diverse set of tasks during training.
   (b) *Model-Based RL:* Use models that capture the underlying structure common to multiple tasks.
   (c) *Model Distillation:* Distill knowledge from several task-specific policies into a single, more general policy.
   (d) *Contextual Policies:* Learn policies that condition on a task-specific context or embedding.
   (e) *Modular Policy Networks:* Design policies with interchangeable modules that can be recombined for new tasks.
3. **Multi-task Meta-learning:** Learn to learn by training on many tasks such that the agent quickly adapts to a new task with only a small amount of data.
   (a) *RNN-based Meta-learning:* Use recurrent neural networks to capture task structure and adapt online.
   (b) *Gradient-based Meta-learning:* Methods such as MAML (Model-Agnostic Meta-Learning) that explicitly optimize for rapid adaptation.

This section provides a high-level overview of the various approaches and trade-offs in transfer and multi-task learning. For more detailed information and specific algorithms, refer to the lecture slides and recommended papers.

16 Distributed Reinforcement Learning

Distributed reinforcement learning scales up RL algorithms by leveraging parallel computation and multiple agents (actors) interacting with the environment simultaneously. This approach is crucial for improving sample efficiency and accelerating training, especially in environments where obtaining data is costly.

Historical Timeline and Key Methods:

- **2013/2015:** *DQN* utilizes a replay buffer to decorrelate training samples.
- **2015:** *GORILA* (General Reinforcement Learning Architecture) introduces distributed training for value-based methods.
- **2016:** *A3C* (Asynchronous Advantage Actor-Critic) employs a single learner with multiple actors running in parallel; the actors compute gradients which are sent asynchronously to the central learner.
- **2018:** *IMPALA* (Importance Weighted Actor-Learner Architecture) extends A3C by using multiple learners and actors, and employs importance sampling (V-trace) to correct for delays between policy updates.
- **2018:** *Ape-X* and *R2D2* reintroduce the replay buffer concept into distributed RL, leading to significant performance improvements.
- **2019:** *R2D3* further advances the distributed framework.
- **RLlib:** Provides abstractions and a framework for implementing distributed reinforcement learning algorithms, as presented in ICML 2018.

Distributed RL systems share several common design elements:

- **Actors:** Multiple agents interact with the environment in parallel, generating a diverse set of experiences.
- **Learners:** One or more central learners update the policy or value function parameters using the aggregated experiences.
- **Replay Buffers:** In some architectures, experiences are stored in a centralized buffer and sampled for training to improve sample efficiency.
- **Importance Sampling:** Techniques such as V-trace in IMPALA correct for the lag between the actors' policies and the learner's updated policy.

In summary, distributed reinforcement learning enables the scaling of RL algorithms by leveraging parallelism. The choice of architecture (e.g., A3C, IMPALA, Ape-X) depends on the specific requirements for sample efficiency, computational resources, and the nature of the environment.

17 Exploration

Exploration is a critical aspect of reinforcement learning (RL) that determines how effectively an agent discovers new, potentially rewarding states and actions. In this section, we discuss various exploration strategies, starting with the classical multi-armed bandit setting and then extending these ideas to deep RL.

17.1 Exploration in Bandits

In the multi-armed bandit setting, the goal is to minimize regret over a time horizon T. The regret is defined as

$$\text{Reg}(T) = T\,E[r(a^*)] - \sum_{t=1}^{T} r(a_t),$$

where a^* is the optimal arm and $r(a_t)$ is the reward received at time t.

17.1.1 Optimistic Exploration

A popular exploration strategy is *optimistic exploration*, where each arm is assigned an optimistic estimate of its reward. For example, if $\hat{\mu}_a$ is the empirical mean reward for arm a and σ_a its uncertainty, one might select

$$a = \arg\max_a \big(\hat{\mu}_a + C\,\sigma_a\big).$$

A common instance is

$$a = \arg\max_a \left(\hat{\mu}_a + \sqrt{\frac{2\ln T}{N(a)}}\right),$$

which yields a regret of $O(\log T)$.

17.1.2 Probability Matching / Posterior Sampling

Another effective exploration strategy is *posterior sampling* (or Thompson Sampling). Assume the reward for each arm a_i is generated according to a parameterized distribution $p_{\theta_i}(r_i)$. This approach can be summarized as follows:

1. Sample $\theta_1, \ldots, \theta_n$ from the current posterior $\hat{p}(\theta_1, \ldots, \theta_n)$.
2. Pretend that the sampled parameters represent the true model.
3. Choose the arm that is optimal under the sampled model.
4. Update the posterior with new observations and repeat.

17.1.3 Information Gain

An alternative perspective is to choose actions that maximize information gain. Let $\mathcal{H}(\hat{p}(z))$ denote the current entropy of our belief over some variable z, and $\mathcal{H}(\hat{p}(z) \mid y)$ the entropy after observing y. The information gain is defined as

$$IG(z, y) = E_y\Big[\mathcal{H}(\hat{p}(z)) - \mathcal{H}(\hat{p}(z) \mid y)\Big].$$

Conditioned on an action a, it is written as

$$IG(z, y \mid a) = E_y\Big[\mathcal{H}(\hat{p}(z)) - \mathcal{H}(\hat{p}(z) \mid y) \mid a\Big].$$

In the bandit setting, if we set $y = r_a$ and $z = \theta_a$, and let

$$g(a) = IG(\theta_a, r_a \mid a),$$

and define the expected regret for arm a as

$$\Delta(a) = E\big[r(a^*) - r(a)\big],$$

one might select the arm

$$a = \arg\min_a \frac{\Delta(a)^2}{g(a)},$$

balancing the potential loss against the information gained.

17.2 Exploration in Deep Reinforcement Learning

17.2.1 Optimistic Exploration in RL

In deep RL, optimistic exploration strategies extend the ideas from bandits. One common approach is to add an exploration bonus to the reward based on state or state-action counts:

$$r^+(s,a) = r(s,a) + \mathcal{B}(N(s)),$$

where $N(s)$ (or $N(s,a)$) is the visit count and $\mathcal{B}(N(s))$ is a bonus function. Because exact counts are infeasible in large state spaces, similarity measures or density models (e.g., $p_\theta(s)$) are used to estimate pseudo-counts.

17.2.2 Exploring with Pseudo-Counts

Pseudo-count methods work as follows:

1. Fit a density model $p_\theta(s)$ to all states observed so far ($\mathcal{D}$).
2. When a new state s_i is encountered, update the density model to $p_{\theta'}(s)$ based on $\mathcal{D} \cup \{s_i\}$.
3. Estimate an effective count:
$$\hat{N}(s_i) = \hat{n}\, p_\theta(s_i),$$
where
$$\hat{n} = \frac{1 - p_{\theta'}(s_i)}{p_{\theta'}(s_i) - p_\theta(s_i)}\, p_\theta(s_i).$$
4. Define the exploration bonus as a function of $\hat{N}(s)$, for example:
$$\mathcal{B}(\hat{N}(s)) = \sqrt{\frac{2\ln T}{\hat{N}(s)}}.$$

17.2.3 Common Bonus Functions

Several bonus functions are commonly used:

- **UCB:** $\mathcal{B}(N(s)) = \sqrt{\frac{2\ln T}{N(s)}}$.
- **MBIE-EB:** $\mathcal{B}(N(s)) = \sqrt{\frac{1}{N(s)}}$.
- **BEB:** $\mathcal{B}(N(s)) = \frac{1}{N(s)}$.

17.2.4 Models for Estimating $p_\theta(s)$

Various methods exist for modeling the state density:

- **CTS Models:** As proposed by Bellemare et al., a context tree switching (CTS) model conditions each pixel on its top-left neighborhood.
- **Hashing:** Compress s into a k-bit code using a function $\phi(s)$, and count occurrences $N(\phi(s))$.
- **Exemplar Models:** Train a classifier to distinguish a new state from past states; a state is novel if the classifier performs well. The density can be estimated as:
$$p_\theta(s) = \frac{1 - D_s(s)}{D_s(s)},$$
where $D_s(s)$ is the classifier's output.
- **Prediction Errors:** Use a target function $f^*(s,a)$ and define a bonus based on the prediction error:
$$\xi(s,a) = \|\hat{f}_\theta(s,a) - f^*(s,a)\|^2.$$
A common choice is $f^*(s,a) = s'$ or to use a randomly initialized network (Random Network Distillation).

17.2.5 Posterior Sampling in Deep RL

Posterior sampling, also known as Thompson Sampling, can be approximated in deep RL using bootstrap methods:

1. Given a dataset $\mathcal{D}$, generate N bootstrap datasets by sampling with replacement.
2. Train a separate model f_{θ_i} on each bootstrap dataset.
3. At decision time, select one model f_{θ_i} at random and act according to its prediction.

To reduce computational overhead, one can share a common base network with multiple output heads.

17.2.6 Reasoning About Information Gain

An alternative to count-based bonuses is to measure the information gain from observing a new state. Define the information gain (IG) as

$$IG(z, y) = E_y\Big[\mathcal{H}(\hat{p}(z)) - \mathcal{H}(\hat{p}(z) \mid y)\Big],$$

and conditionally,

$$IG(z, y \mid a) = E_y\Big[\mathcal{H}(\hat{p}(z)) - \mathcal{H}(\hat{p}(z) \mid y) \mid a\Big].$$

For bandits, setting $y = r_a$ and $z = \theta_a$ and defining

$$g(a) = IG(\theta_a, r_a \mid a),$$

we can choose the arm that minimizes

$$\frac{\Delta(a)^2}{g(a)},$$

where $\Delta(a) = E[r(a^*) - r(a)]$ is the expected regret of arm a.

17.3 Exploration in Deep RL: Bonus-Driven Methods

17.3.1 Optimistic Exploration via Count-Based Bonuses

In deep RL, optimistic exploration is implemented by augmenting the reward function:

$$r^+(s, a) = r(s, a) + \mathcal{B}(N(s)),$$

where $N(s)$ is replaced by pseudo-counts or density estimates when explicit counts are not available.

17.3.2 Exploring with Pseudo-Counts

Using a density model $p_\theta(s)$ and its updated version $p_{\theta'}(s)$ after observing a new state, an effective count can be estimated as:

$$\hat{N}(s) = \hat{n}\, p_\theta(s), \quad \text{with} \quad \hat{n} = \frac{1 - p_{\theta'}(s)}{p_{\theta'}(s) - p_\theta(s)}\, p_\theta(s).$$

The exploration bonus is then set as a function of $\hat{N}(s)$.

17.4 Imitation Learning vs. Reinforcement Learning for Exploration

Imitation Learning:

- Requires expert demonstrations.
- Suffers from distributional shift if the learned policy deviates from the expert.
- Relies on stable supervised learning.

Reinforcement Learning:

- Requires a well-defined reward function.
- Must tackle the exploration–exploitation dilemma.
- Can, in principle, achieve higher performance by discovering novel strategies.

Combining Both: When both demonstrations and reward signals are available, one can combine them—using inverse RL, for example—to leverage the advantages of both approaches.

17.4.1 Pre-training and Fine-tuning

A simple method to combine imitation and RL is:

1. **Pre-train:** Initialize the policy π_θ by behavior cloning using demonstration data:
$$\max_\theta \sum_i \log \pi_\theta(a_i \mid s_i).$$
2. **Fine-tune:** Use an RL algorithm to further improve π_θ on the target task.

This approach is effective but may suffer from distribution shift if the initial performance is poor.

17.4.2 Off-Policy RL with Demonstrations

Off-policy RL methods can incorporate demonstration data into the replay buffer. For example, one may modify the policy gradient as follows:

$$\Delta J(\theta) = \sum_{\tau \in \mathcal{D}} \left[\sum_{t=1}^{T} \Delta_\theta \log \pi_\theta(a_t \mid s_t) \left(\prod_{t'=1}^{t} \frac{\pi_\theta(a_{t'} \mid s_{t'})}{q(a_{t'} \mid s_{t'})} \right) \left(\sum_{t'=t}^{T} r(s_{t'}, a_{t'}) \right) \right],$$

where $\mathcal{D}$ includes both demonstration and self-generated data. Challenges include determining the appropriate weighting for demonstration data and handling multiple data distributions.

17.4.3 Imitation as an Auxiliary Loss

Alternatively, one can add an imitation loss to the RL objective:

$$\text{Hybrid Objective: } E_{\pi_\theta}[r(s,a)] + \lambda \sum_{(s,a) \in \mathcal{D}_{demo}} \log \pi_\theta(a \mid s).$$

This auxiliary loss encourages the policy to mimic expert behavior while still optimizing for reward.

17.5 Summary

Exploration in RL is essential for discovering rewarding behaviors, especially in environments with sparse or deceptive rewards. Classical strategies include optimistic exploration (e.g., UCB), posterior sampling (Thompson Sampling), and information gain. In deep RL, count-based exploration is extended via pseudo-counts and density models. Balancing exploration and exploitation remains a central challenge, and combining imitation learning with RL can help bootstrap exploration when demonstrations are available.

18 Meta Reinforcement Learning

This part introduced many meta learning methods at a high level. You may find and read the papers later.

19 Information Theory, Challenges, Open Problems

19.1 Information Theory

Entropy: The entropy of a distribution $p(x)$ is defined as

$$\mathcal{H}(p(x)) = -E_{x\sim p(x)}[\log p(x)].$$

Mutual Information: The mutual information between random variables X and Y is given by

$$\begin{aligned}\mathcal{I}(X;Y) &= D_{KL}\big(p(x,y) \,\|\, p(x)p(y)\big) \\ &= E_{(x,y)\sim p(x,y)}\left[\log \frac{p(x,y)}{p(x)p(y)}\right] \\ &= \mathcal{H}(p(y)) - \mathcal{H}(p(y \mid x)).\end{aligned}$$

If $\pi(s)$ denotes the state *marginal* distribution of a policy π, then $\mathcal{H}(\pi(s))$ is the entropy of the state distribution under the policy.

Empowerment: Empowerment is a measure of the influence an agent has over its future states. It is defined as the mutual information between actions and subsequent states:

$$\mathcal{I}(s_{t+1}; a_t) = \mathcal{H}(s_{t+1}) - \mathcal{H}(s_{t+1} \mid a_t).$$

19.2 Learning without a Reward Function by Reaching Goals

One approach to learning in the absence of an explicit reward function is to provide goal states and train the agent to reach them. For example, a variational autoencoder (VAE) can be used to generate diverse goal states. A possible procedure is:

1. **Propose a goal:** Sample a latent variable $z_g \sim p(z)$ and generate a goal state $x_g \sim p_\theta(x_g \mid z_g)$.
2. **Attempt to reach the goal:** Use a policy $\pi(a \mid x, x_g)$ conditioned on the current state x and the goal x_g to produce a final state $\bar{x}$.
3. **Update the policy:** Use the collected data $(x, x_g, \bar{x})$ to update the policy.
4. **Update the generative model:** Refine the goal generator $p_\theta(x_g \mid z_g)$ and the encoder $q_\phi(z_g \mid x_g)$ using the observed outcomes.

But how to diversify goals? One approach is to modify the maximum likelihood estimation (MLE) objective:

- **Standard MLE:** Optimize
$$(\theta, \phi) \leftarrow \arg\max_{\theta,\phi} E\big[\log p(\bar{x})\big].$$
- **Weighted MLE:** Optimize
$$(\theta, \phi) \leftarrow \arg\max_{\theta,\phi} E\Big[w(\bar{x}) \log p(\bar{x})\Big],$$
where the weight is defined as
$$w(\bar{x}) = p_\theta(\bar{x})^\alpha.$$

A key result is that for $\alpha \in [-1, 0)$, the entropy $\mathcal{H}(p_\theta(x))$ increases, effectively maximizing the entropy over the goal distribution $\mathcal{H}(p(G))$.

Link to RL: A policy $\pi(s \mid S, G)$ is trained to reach a goal G such that as the policy improves, the final state S becomes closer to G. Equivalently, the conditional distribution $p(G \mid S)$ becomes more deterministic. Thus, the overall objective can be expressed as:

$$\max \mathcal{H}(p(G)) - \mathcal{H}(p(G \mid S)) = \max \mathcal{I}(S; G),$$

which maximizes the mutual information between states and goals.

19.3 Learning Diverse Skills

To learn a diverse set of skills, consider policies of the form $\pi(a \mid s, z)$, where z is a task or skill identifier. The intuition is that different values of z should lead the policy to explore different regions of the state space. A diversity-promoting reward function is given by:

$$r(s, z) = \log p(z \mid s).$$

Maximizing this reward encourages the policy to visit states that are uniquely associated with each task index z. This is equivalent to maximizing the mutual information between the latent variable z and the state s:

$$\mathcal{I}(z; s) = \mathcal{H}(z) - \mathcal{H}(z \mid s).$$

A uniform prior on z (maximizing $\mathcal{H}(z)$) combined with a minimization of $\mathcal{H}(z \mid s)$ results in diverse skill acquisition. This approach is explored in the paper *Diversity is All You Need* (ICLR 2019).

19.4 Unsupervised Reinforcement Learning for Meta-Learning

Unsupervised RL can be employed to propose tasks for meta-learning:

1. First, use unsupervised meta-RL to generate a set of diverse tasks.
2. Then, apply meta-learning algorithms on these tasks to enable fast adaptation to new tasks.

19.5 Challenges in Deep Reinforcement Learning

Core Algorithmic Challenges:

- **Stability:** Many RL algorithms suffer from unstable training dynamics.
- **Efficiency:** Sample complexity is a major issue, particularly in real-world applications.
- **Generalization:** RL agents often struggle to generalize beyond the specific tasks or environments on which they were trained.

Assumptions:

- The formulation of the problem (e.g., state space, action space, MDP structure).
- The availability and quality of supervision (e.g., demonstrations, language, or human feedback).

19.5.1 Stability and Hyper-parameter Tuning

- Algorithms such as Q-learning and policy gradients require careful tuning of numerous hyper-parameters (learning rates, target network update frequency, replay buffer size, etc.).
- Model-based RL faces additional challenges in back-propagating through time and avoiding exploitation of model errors.

19.5.2 Sample Complexity

- Many RL algorithms require large amounts of data, making them impractical in real-world settings.
- This motivates research into more sample-efficient methods, such as model-based approaches and meta-learning.

19.5.3 Scaling & Generalization

- While many studies focus on performance in controlled environments, generalization to new and varied tasks remains an open problem.
- RL often requires re-collecting data when the environment changes.

19.5.4 Single Task vs. Multi-task Learning

- Training on multiple tasks can improve generalization.
- Techniques include policy distillation, actor-mimic, and meta-learning methods like MAML.
- Unsupervised or weakly supervised approaches (e.g., using stochastic neural networks or energy-based models) may facilitate learning of diverse behaviors.

Supervision Signals:

- Tasks can be defined via demonstrations, language instructions, or human preferences.
- Unsupervised or self-supervised objectives (e.g., predicting future states) also provide valuable signals.

19.6 Rethinking Reinforcement Learning from the Perspective of Generalization (Chelsea Finn)

19.6.1 Meta-Learning

Meta-learning, or "learning to learn," aims to train agents that can quickly adapt to new tasks:

- Gradient-based meta-learning methods (e.g., MAML) optimize for fast adaptation.
- RNN-based meta-learning leverages recurrent networks to capture task-specific information.
- Recent work has focused on off-policy meta-reinforcement learning via probabilistic context variables.

These approaches typically require that the meta-training task distribution matches the meta-testing task distribution to achieve good generalization.

Key Considerations:

- **Algorithms:** More general algorithms may be needed than those based solely on demonstration or trial-and-error.
- **Task Representation:** How tasks are described (e.g., using language or goal specifications) is critical.
- **Data:** Datasets such as RoboNet provide valuable resources for training agents across diverse tasks.