Deep Reinforcement Learning

CS 285, University of California, Berkeley

Harry Zhang

December 2019

Contents

Preface

This is a compilation of notes from UC Berkeley's CS 285 (formerly CS 294-112) taught by Professor Sergey Levine. The notes are based on Fall 2018, Fall 2019, and Fall 2020 offerings of the course. The author is currently a EECS student at UC Berkeley, and the readers are welcome to contact the author via harryhzhang@berkeley.edu.

The notes assume the reader have some familiarity with key concepts such as machine learning, neural networks, Markov Decision Process (MDP), and optimal control.

Rest in Peace, Andy.

Chapter 1: Introduction

Here we review some of the terminologies that frequently appear in the field of Reinforcement Learning.

1.1 Important Concepts

Imagine a **discrete-time** system (environment), meaning that the system is discretized into time steps. At time step t, we define the state of the system to be s_t. The state of a system/agent could be some intrinsic data of it. For example, for a car, the state of a car at a given time step could be the car's angular velocity, acceleration, and mass. We also define an agent's action a_t, which is equivalent to the notion of input in control theory. A **policy** is a function that takes in a state and outputs an action, determining what the agent should do given current time step's state.

A policy function could be deterministic such that $a_t = \pi_\theta(s_t)$. The policy function could also be a distribution, which we can define as $\pi_\theta(a_t|s_t)$. Note that in many cases, a state is not **fully observable**, so we might only partially observe the state of the agent via an observation o_t. In this case, the policy function should condition on the observation o_t.

We should also define a **transition function** of the environment, which is also called the **model** of the system. In the most general case, the transition should be stochastic, meaning that a state could evolve into a number of other states potentially. Therefore, this function should be a distribution, defined as $p(s_{t+1}|s_t, a_t)$. Note that this distribution is conditioned on both the state and the action at time step t. If you are familiar with control theory, you would probably notice the resemblance of the transition distribution with a discrete-time system's dynamics function. A sequence of states and actions become a **trajectory**, which we call τ.

Meanwhile, we also define a **reward function** of the environment. For example, in Pacman, the player gains one point after eating a dot, and loses 100 points after being eaten by a ghost. Formally, the reward function should be a function of both state and action, so we denote the reward function as $r(s, a)$. The rewards could be a hand-engineered function, or it could be implicit that the agent actually needs to learn the rewards first, as seen in the case of Inverse RL (IRL).

In many reinforcement learning problems, we assume the states transitions are **Markovian**, meaning that the state at time step t is only responsible for contributing to the state at time step $t + 1$. Fig. 1.1 an illustration of a Markov chain, and the direction of the arrow means causality.

In control theory, such as the LQR optimization problem, we aim to minimize the cost function of the system. What would be an equivalent notion in reinforcement learning? Recall we defined a reward function, $r(s, a)$, so naturally we want to collect as much reward

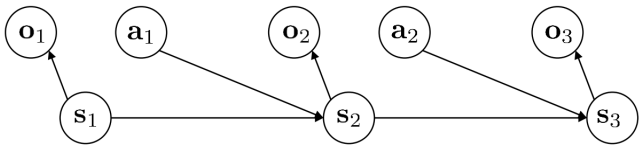

Figure 1.1: A simple Markov chain.

as possible in the environment. Without loss of generality, we assume that the environment is stochastic. Define a trajectory distribution $p_\theta(\tau)$ according to Bayes' Rule:

$$p(\tau) := p(s_1, a_1, ..., s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

where T is the length of episode horizon. Therefore, given this trajectory distribution, we can calculate the expected value of the total reward function induced by following this trajectory as $\mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t)\right]$. Therefore, to optimize this objective, we want to find a parameter θ, such that θ maximizes the above expectation:

$$\theta = \arg\max_\theta \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t)\right]$$

1.2 Value Function and Q Function

To facilitate our calculation of the above expectation, and to simplify the notations, we introduce two important types of functions: Q function and value function. In most cases these two functions are not given to us in closed form, and one needs to approximate and improve the functions using some deep neural net, hence the notion of "deep" in deep reinforcement learning.

1.2.1 Q Function

Q function is a function of both state and action, so it is denoted as $Q(s_t, a_t)$ and it quantitatively measures the quality of taking action a_t at state s_t. Mathematically, it is the expected sum of reward from the current time step given state s_t and action a_t. This is very similar to the "cost-to-go" function in control theory, especially Model Predictive Control. We define $Q(s, a)$ as:

$$Q^\pi(s_t, a_t) = \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}[r(s'_t, a'_t)|s_t, a_t]$$

1.2.2 Value Function

Unlike the Q function, value function is only a function of state, so intuitively it quantitatively measures the value of being in state s_t. Mathematically, it is defined as $V^\pi(s_t) = \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}[r(s'_t, a'_t)|s_t]$. Again by Bayes' rule we can obtain the relation between Value function and Q function: $V^\pi(s_t) = \mathbb{E}_{a_t \sim \pi(a_t|s_t)}[Q^\pi(s_t, a_t)]$.

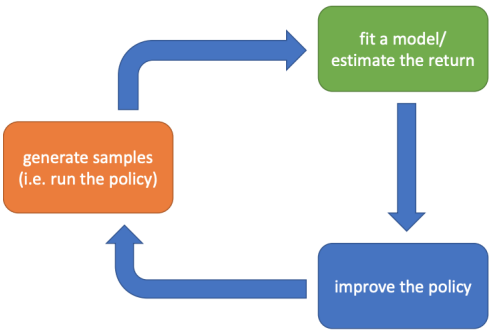

Figure 1.2: Three steps of reinforcement learning.

Furthermore, if we sum the value function over all possible initial states, we essentially recovered the objective of reinforcement learning: $\mathbb{E}_{s_1 \sim p(s_1)}[V^{\pi}(s_1)]$, where $p(s_1)$ is a known distribution of all possible initial states.

1.3 Reinforcement Learning Anatomy

In RL, we usually have three parts in the whole pipeline. We need to keep generating data for the agent to learn, and use the data to generate samples in order to fit and regress onto a model. Then with this model, we estimate the reward and based on the reward we update our policy to maximize the reward, and we go back to step 1. Therefore, our primal concern is to efficiently run the three parts so that the agent can learn optimally with less data and computation. Here is an illustration of the three steps in Fig. 1.2.

Chapter 2: Imitation Learning

Imitation learning is also called behavioral cloning. The basic idea of imitation learning is "train to fit the expert behavior". In other words, given a demonstration, we want to make the agent follow the demonstration as closely as possible, to best imitate the demonstration's behaviors.

2.1 Distribution Mismatch

However, a big problem of such type of approach is that it does not generalize well, if at all. For example, one can imagine that the agent makes a small mistake and ends up being in a slightly different state from what it has seen (trained) before, but since the state is novel, the agent does not know how to act, thus behaving randomly, diverging from the learned trajectory. An illustration of such mismatches is shown in Fig. 2.1.

2.2 Dataset Aggregation

The aggregation of mistakes that the agent makes often makes imitation learning not feasible. But imitation learning does work in some cases. Intuitively, if the agent could somehow learn from the mistakes, and we keep appending data to the agent's dataset so that the agent is exposed to a variety of states, then the expected trajectory would get closer to the training trajectory. If the actions applied to those states are correct, then eventually, the agent can, ideally, converge to an optimal trajectory. This is essentially the idea behind an imitation learning algorithm called Dataset Aggregation (DAgger) [1]. DAgger essentially makes the training trajectory close to the expected trajectory by constantly appending test data to training data. In step 3 of Algorithm 1, what we are doing is basically discarding the actions from running trained policy $\pi_\theta(a_t|o_t)$. Instead, we ask a human in the loop to label what they

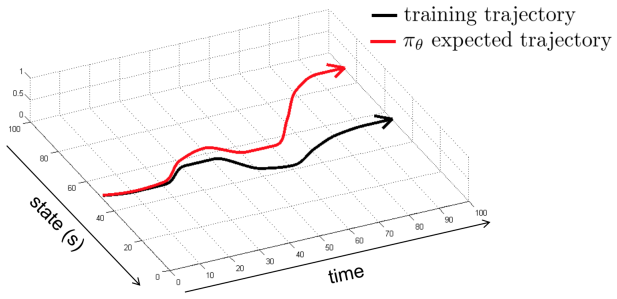

Figure 2.1: Mistakes aggregate in behavior cloning.

Algorithm 1 Dataset Aggregation (DAgger)

Require: Human data $\mathcal{D} = \{o_1, a_1, ..., o_N, a_N\}$
1: **while** true **do**
2: Train $\pi_\theta(a_t|o_t)$ from human data $\mathcal{D} = \{o_1, a_1, ..., o_N, a_N\}$.
3: Run $\pi_\theta(a_t|o_t)$ to get dataset $\mathcal{D}_\pi = \{o_1, ..., _M\}$
4: Ask human to label $\mathcal{D}_\pi$ with actions a_t
5: Aggregate $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_\pi$
6: **return** optimal imitation-learned trajectory as τ^{return}

would have done based on the observations in D_π. Take an autonomous car for example, the training data would be images labeled with steering commands, and we let the car collect more data, which are only images. Then we give those images to a human expert, and let the human determine, based on each image, what action (steer left, right, or go straight) that the human would have applied if he observed such an image.

It can be proven that DAgger resolves the distribution "drift" issue. However, one problem with DAgger is that human might be error-prone, so the human labelled data might be flawed to use. Furthermore, more subtly, human, in most cases, does not make decisions based on a Markovian process. Therefore, the current time step's action might be dependent on a state/observation some number of time steps ago.

2.3 When Does Imitation Learning Fail?

In general, there are some cases where one may fail to fit the expert data, which lead to undesirable outcomes of imitation learning.

2.3.1 Non-Markovian Behaviors

First, the data is Non-Markovian, as mentioned above. Actually Non-Markovian process is more natural and intuitive for human in that human learns from their past mistakes. Why is this wrong? Essentially, we are fitting the wrong distribution. Since we are fitting a policy distribution based on a Markovian process, our goal is fitting $\pi(a_t|o_t)$. However, if the expert data is not Markovian, then we are trying to fit $\pi(a_t|o_t)$ from another distribution $\pi(a_t|o+1, ..., o_t)$. One solution is to use a lot of previous memory frames, and concatenate them as one huge frame, effectively augmenting the state space. However, this solution might require too many weights in the neural net encoder, significantly increasing the computational complexity. Another solution is that one can fit the expert data using an RNN with shared weights in the convolutional encoder, and one possible implementation is shown in Fig. 2.2. Usually, an LSTM cell works well.

The underlying reason why having a full history makes imitation learning difficult is that history data tends to exacerbate the causality misclassification, which is also called **causal confusion**. Having more history might make the agent learn the wrong direction of causality, and in many cases, the wrong direction is actually easier to learn. This can be illustrated in an autonomous car example. Consider a car that has a brake light on the dashboard that lights up every time the brake pedal is pressed. When the brake pedal is pressed due to the red light/obstacle in front of the vehicle, it is probably easier for the agent

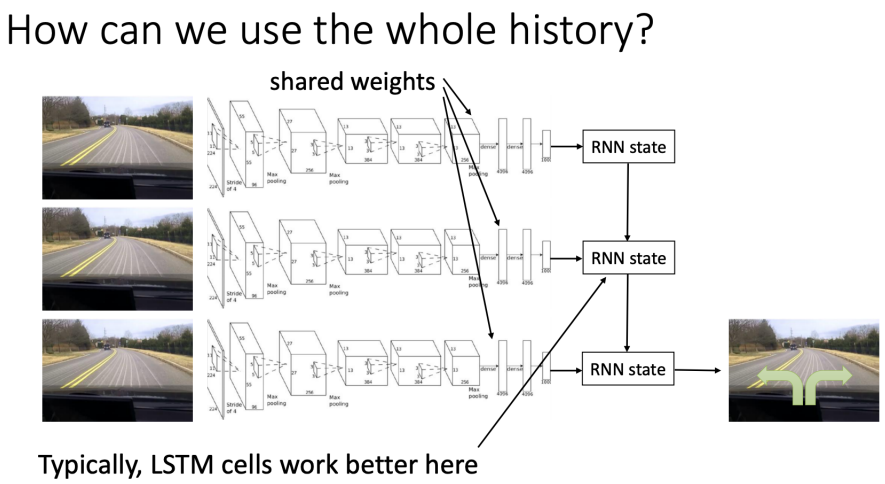

Figure 2.2: Using RNN to address non-Markovian expert data.

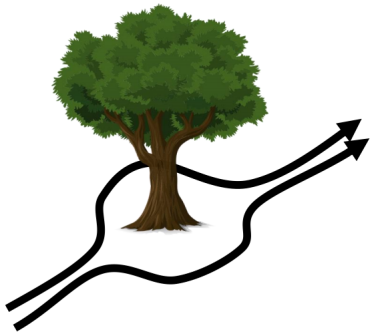

Figure 2.3: Multimodal behaviors.

to associate the brake action with the brake light rather than with the red light/obstacle in front of the car. The causal confusion issue can be alleviated with the use of DAgger because the human annotator is able to provide the correct causal relation. For more information, please refer to this paper [2].

2.3.2 Multimodal Behaviors

Another scenario where fitting expert might fail is that the expert has **multi-modal** behaviors. An example of this is that when you are controlling a drone to dodge a tree ahead, you either steer left or steer right. However, if you choose the wrong parametric form of the distribution (e.g. a simple Gaussian) of the actions, the distribution might average out left and right and choose to go straight, as shown in figure 2.3. Some methods to mitigate this issue include: first, one can use a mixture of different Gaussian distributions, instead of just one. Second, construct a latent space variables model, which we will talk more about in variational inference. Third, we can use autogregressive discretization. Specifically, a mixture of Gaussians means that the policy distribution should be a weighted sum of different Gaussians with different means and variances.

2.4 Theoretical Analysis of Imitation Learning's Error

First we define two different reward functions for imitation learning. To make our analysis easier, we assume the policy function is deterministic. First, we can define the reward function as $r(s,a) = \log p(a = \pi^*(s)|s)$. This function measures the log likelihood of the action equal to expert policy's action. Another (simpler) choice can just be a counter to count the number of mistakes. Specifically,

$$c(s,a) = \begin{cases} 0 & \text{if } a = \pi^*(s) \\ 1 & \text{o.w.} \end{cases}$$

To analyze this, let's introduce a lower bound on the probability of making mistakes: $\pi_\theta(a \neq \pi^*(s)|s) \leq \epsilon$ for all $s \sim p_{train}(s)$, where p_{train} is training data distribution. The fit distribution of states $p_\theta(s)$ is consisted of two parts: the first part is the probability of no mistakes made, and the second part is the probability of making some mistakes. Using Bayes' rule, we can calculate $p_\theta(s)$ as follows:

$$p_\theta(s_t) = (1-\epsilon)^t p_{train}(s_t) + (1-(1-\epsilon)^t) p_{mistake}(s_t)$$

so to measure the divergence of p_θ from p_{train}, we take the difference of the two distributions (naive, total variation divergence):

$$\begin{aligned} |p_\theta(s_t) - p_{train}(s_t)| &= (1-(1-\epsilon)^t)|p_{mistake} - p_{train}| \leq 2(1-(1-\epsilon)^t) \\ &\leq 2\epsilon t \end{aligned}$$

where we used the identity that $(1-\epsilon)^t \geq 1 - \epsilon t$ for $\epsilon \in [0,1]$. Thus, we can calculate the expected number of mistakes the agent makes using this scheme by:

$$\begin{aligned} \sum_t \mathbb{E}_{p_\theta(s_t)}[c_t] = \sum_t \sum_{s_t} p_\theta(s_t) c_t(s_t) &\leq \sum_t \sum_{s_t} p_{train}(s_t) c_t(s_t) + |p_\theta(s_t) - p_{train}(s_t)| c_{max} \\ &\leq \sum_t \epsilon + 2\epsilon t \\ &\in O(\epsilon T^2) \end{aligned}$$

Also note that with DAgger $p_{train}(s) \to p_\theta(s)$. So we no longer have the second item inside the summation for DAgger. Thus for DAgger, the expected value should be in $O(\epsilon T)$.

As we see, when we have longer horizon length, the errors are going to be aggregated, thus making more mistakes, and this is one of the most fundamental disadvantages of imitation learning, as discussed in [1].

2.5 Summary

Overall, what are some disadvantages of imitation learning? We have a human factor to provide data in the entire loop, which is potentially finite, and to generate a good policy, one need to learn from a lot of data. Moreover, human cannot provide all kinds of data. Specifically, a human may have trouble with providing data such as the joint angle/torque of a robotic arm. Therefore, we wish that machines can learn automatically, from unlimited data.

Chapter 3: Policy Gradient Methods

Recall the objective of Reinforcement Learning:

$$\theta = \arg\max_{\theta} \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

This is actually framed as an optimization problem. Therefore, we can use a variety of optimization techniques, such as gradient descent, to optimize this objective. To be more concrete, let us define a function $J(\theta)$:

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

By definition, $r(\tau)$ is the sum of reward incurred in this trajectory, so it can be equivalently defined as $\sum_{t=1}^{T} r(s_t, a_t)$, and by definition of expectation, we can more conveniently express the J function as an integral of the product of policy and reward:

$$J(\theta) = \int \pi_\theta r(\tau) d\tau$$

. With this integral, we can easily take the gradient to perform gradient descent/ascent. A convenient expression of the gradient of $J(\theta)$ is shown below.

3.1 Policy Gradient Theorem

In this section, we will derive the mathematical expression of the policy gradient theorem.

Recall a convenient identity:

$$\pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) = \pi_\theta(\tau) \frac{\nabla_\theta \pi_\theta(\tau)}{\pi_\theta(\tau)} = \nabla_\theta \pi_\theta(\tau)$$

Using this identity, we can take the gradient of $J(\theta)$ in a cleaner fashion:

$$\begin{aligned} \nabla_\theta J(\theta) &= \int \nabla_\theta \pi_\theta r(\tau) d\tau \\ &= \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) r(\tau) d\tau \\ &= \mathbb{E}_{\tau \sim \pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) r(\tau)] \end{aligned}$$

Now, we want to get rid of the huge $\log \pi_\theta(\tau)$ from our equation. Recall that a trajectory τ is a list of states and actions, so $\pi_\theta(s_1, a_1, ..., s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$

Algorithm 2 REINFORCE Algorithm

Require: Base policy $\pi_\theta(a_t|s_t)$, sample trajectories τ^i

1: **while** true **do**
2: Sample $\{\tau^i\}$ from $\pi_\theta(a_t|s_t)$ (run it on a robot).
3: $\nabla_\theta J(\theta) \simeq \frac{1}{N}\sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_t r(s_{i,t}, a_{i,t})\right)$
4: Improve policy by $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$
5: **return** optimal trajectory from gradient ascent as τ^{return}

by Bayes' rule. Then we take the log on both sides, and we end up getting $\log \pi_\theta(\tau) = \log p(s_1) + \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t)$. Plugging into our original gradient:

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\nabla_\theta\left(\log p(s_1) + \sum_{t=1}^{T}\log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t)\right) r(\tau)\right] \\ &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\right)\left(\sum_{t=1}^{T} r(s_t, a_t)\right)\right]\end{aligned} \tag{3.1}$$

Note that in the above calculation, we cancel out $\log p(s_1)$ and $\log p(s_{t+1}|s_t, a_t)$ because we are taking the gradient with respect to θ, but those two expressions do not depend on θ. The first item in the final expectation is similar to maximum likelihood.

3.2 Evaluating the Policy Gradient

In our derivation, we mathematically derived an expression for policy gradient, which involves calculating an expectation. However, in most cases we cannot easily obtain this expectation easily because it is highly possible that it involves a huge, intractable integral. Therefore, what are we going to do if the expectation (integral) is hard to evaluate? The answer is to approximate, and more specifically, we use Monte Carlo approximation. The idea is to take N samples, and average them out:

$$\nabla_\theta J(\theta) \simeq \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_{t=1}^{T} r(s_{i,t}, a_{i,t})\right)$$

where the subscripts i, t means time step t in the i-th rollout. With the above gradient, we can do gradient descent (ascent) on the parameter θ by:

$$\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$$

Now we are ready to propose a vanilla policy gradient algorithm by direct gradient ascent on the Monte Carlo-approximated policy gradient parameters, the REINFORCE Algorithm, as shown in Algorithm 2.

3.3 Example: Gaussian Policy

Now let us work out a simple case of running Algorithm 2 on a simple Gaussian policy. A Gaussian policy means that the policy function is a Gaussian distribution. Specifically,

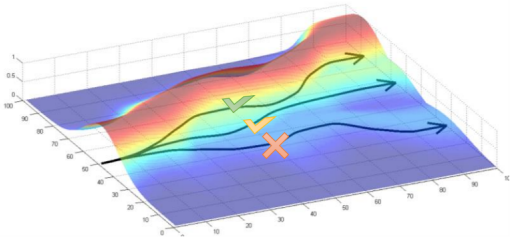

Figure 3.1: More rewarding trajectories are more probable.

$\pi_\theta(a_t|s_t) = \mathcal{N}\left(f_{\text{neural net}}(s_t); \Sigma\right)$. One advantage of using a Gaussian policy is that it is easy to obtain a closed-form expression for the Gaussian derivative. We simply write out the quadratic discriminant function in a multivariate Gaussian distribution:

$$\log \pi_\theta(a_t|s_t) = -\frac{1}{2}\left(f(s_t) - a_t\right)^T \Sigma^{-1} \left(f(s_t) - a_t\right) + C = -\frac{1}{2}\|f(s_t) - a_t\|_\Sigma^2 + C$$

Taking the derivative, we have:

$$\nabla_\theta \log \pi_\theta(a_t|s_t) = -\frac{1}{2}\Sigma^{-1}(f(s_t) - a_t)\frac{df}{d\theta}$$

And we use Gradient ascent as discussed above.

3.4 Intuition behind Policy Gradient: What are We Actually Doing?

Recall we mentioned that the first term inside the expectation is similar to maximum likelihood. Maximum Likelihood can be written as maximizing the log likelihood of an event. Let us compare the two side by side. Recall the expression of the policy gradient is:

$$\nabla_\theta J(\theta) \simeq \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(\tau_i) r(\tau_i)$$

And the Maximum Likelihood is defined as:

$$\nabla_\theta J_{ML}(\theta) \simeq \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(\tau_i)$$

As we discussed before, the first term in the policy gradient is exactly the same as the definition of maximum likelihood!

So what are we doing here when taking this gradient? Intuitively, we are assigning more weight to more rewarding trajectories by making trajectories with higher rewards more probable. Equivalently, higher-reward trajectories are likely to have more probability to be chosen. This intuition is crucial to the policy gradient methods and is illustrated in Fig. 3.1.

3.5 Partial Observability

Can we use the policy gradient on a Partially Observed Markov Decision Process (POMDP)? The short answer is Yes. Why? Recall (yet again) the policy gradient expression:

$$\nabla_\theta J(\theta) \simeq \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_{t=1}^{T} r(s_{i,t}, a_{i,t})\right)$$

In this expression, we do not even have the transition function in it. Long story short, the Markovian property is not actually used! So we can use policy gradient on a POMDP without any modification except instead of s_t, we use o_t.

Note that we do not care about what the state actually is. Any Non-Markovian proces can be made Markovian by setting the state as the whole history.

3.6 Disadvantages of the Policy Gradient

Recall the intuition behind the policy gradient update: we make trajectories with more reward more probable. Let's consider the following scenario: say two trajectories have similar positive rewards, while another trajectory has a low, negative reward. Then policy gradient is going to assign zero to low probability to the third trajectory, and high probabilities to the other two. Now imagine we add a large constant number to our reward function, and apparently it does not change the relative relation between different trajectories' rewards because we are only adding in a constant. Now the negative reward becomes positive, and policy gradient is likely to spread out the likelihood for the three trajectories since all three rewards are positive now. This is bad because our reward distribution does not change at all, but after adding in a constant, the distribution of policy gradient changed substantially.

This problem shows the high variance in our naive policy gradient algorithm. Therefore, we need to come up with some methods to reduce the variance introduced in the policy gradients.

3.7 Reducing Policy Gradients Variance using Baselines

3.7.1 Causality

One simple fix for high variance is to use the fact of causality: policy at time t' cannot affect reward at t if $t < t'$. This simple, commonsensical idea allows us to discard some operands in the summation:

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

and we define the second item in the summation as the "reward-to-go". Notice that in the reward-to-go term, we start the summation from time t instead of 1, by causality. The idea is that we are multiplying the likelihood by smaller numbers due to the reduction of the summation term, so we can reduce the variance to some extent.

3.7.2 Baselines

Another common approach is to use baselines. By baselines, we mean that instead of making all high-reward trajectories more likely, we only make trajectories **better than average** more likely. So naturally, we define a baseline b as the average reward:

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

Incorporating the baseline b into our original policy gradient expression:

$$\nabla_\theta J(\theta) \simeq \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(\tau)\left[r(\tau) - b\right]$$

But, are we allowed to that? Yes, in fact, we can show that the expectation is the same with baseline b. To show this, we can express the expectation of baseline as:

$$\begin{aligned}
\mathbb{E}_{\pi_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(\tau) b\right] &= \int \pi_\theta(\tau) \nabla \log \pi_\theta(\tau) b \, d\tau \\
&= \int \nabla_\theta \pi_\theta(\tau) b \, d\tau \\
&= b \nabla_\theta \int \pi_\theta(\tau) \, d\tau \\
&= b \nabla_\theta 1 \\
&= 0
\end{aligned}$$

Therefore, by subtracting a baseline, our policy gradient is still unbiased in expectation!

3.7.3 Analyzing the Variance with Baselines

Let us explicitly write down the variance of the policy gradient. Recall the definition of variance:

$$\mathrm{Var}[x] = \mathbb{E}[x^2] - \mathbb{E}[x]^2$$

And the policy gradient with baselines is written as:

$$\nabla_\theta J(\theta) \simeq \mathbb{E}_{\tau \sim \pi_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(\tau)\left(r(\tau) - b\right)\right]$$

Therefore, the variance of the policy gradient can be written as follows:

$$\mathrm{Var} = \mathbb{E}_{\tau \sim \pi_\theta(\tau)}\left[\left(\nabla_\theta \log \pi_\theta(\tau)\left(r(\tau) - b\right)\right)^2\right] - \mathbb{E}_{\tau \sim \pi_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(\tau)\left(r(\tau) - b\right)\right]^2$$

Note that in the second squared expectation term of variance, it can be equivalently written as $\mathbb{E}_{\tau \sim \pi_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(\tau) r(\tau)\right]^2$ since baselines are unbiased in expectation.

Now we have an expression of variance with respect to baseline b, we can calculate the optimal b that minimizes the variance by setting the gradient of variance to 0:

$$\begin{aligned}\frac{d\text{Var}}{db} &= \frac{d}{db}\mathbb{E}\left[g(\tau)^2(r(\tau)-b)^2\right] \\ &= \frac{d}{db}\mathbb{E}\left[g(\tau)^2 r(\tau)^2\right] - 2\mathbb{E}\left[g(\tau)^2 r(\tau) b\right] + b^2\mathbb{E}\left[g(\tau)^2\right] \\ &= -2\mathbb{E}\left[g(\tau)^2 r(\tau)\right] + 2b\mathbb{E}\left[g(\tau)^2\right] \\ &= 0\end{aligned}$$

Solving the equation, we will have

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

where b^{opt} is the optimal baseline value for reducing the variance.

In practice, we just use the average reward for baseline.

3.8 On-Policy vs. Off-Policy

We now introduce two concepts in RL: on-policy and off-policy. On-policy means that we learn only from using the current policy π_θ, and off-policy means we learn also from other policies. Apparently, policy gradient is an on-policy method because $\nabla_\theta J(\theta) \simeq \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(\tau)\left(r(\tau)-b\right)\right]$, and the expectation is taken under the current, known trajectory of interest. Therefore, every time we have a new policy, we need to use new samples. Since we are changing θ, π_θ also changes overtime in policy gradient. One can imagine that this is extremely inefficient in neural networks, because in a neural network, θ only changes a little and the overhead for changing the policy is large.

One solution is to use off-policy learning.

3.8.1 Off-policy Learning and Importance Sampling

We first introduce an important technique called importance sampling. Given a distribution $p(x)$, how do we calculate the expectation from samples from another distribution $q(x)$? This is the idea of importance sampling, by using an importance ratio, we can calculate the expectation from another distribution, thus enabling to learn off-policy.

In importance sampling:

$$\begin{aligned}\mathbb{E}_{x\sim p(x)}\left[f(x)\right] &= \int p(x)f(x)\,dx \\ &= \int \frac{q(x)}{q(x)}p(x)f(x)\,dx \\ &= \mathbb{E}_{x\sim q(x)}\left[\frac{p(x)}{q(x)}f(x)\right]\end{aligned}$$

Then we can plug it into the off-policy policy gradient. Say we have a trained policy $\pi_\theta(\tau)$, and we have samples from another policy $\bar{\pi}(\tau)$, we can use the samples from $\bar{\pi}(\tau)$ to

calculate $J(\theta)$ function using importance sampling:

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[r(\tau)\right] \\ &= \mathbb{E}_{\tau\sim\bar{\pi}(\tau)}\left[\frac{\pi_\theta(\tau)}{\bar{\pi}(\tau)}r(\tau)\right] \end{aligned}$$

Now we want to look closely at the importance ratio. Recall that $\pi_\theta(\tau) = p(s_1)\prod_{t=1}^{T}\pi_\theta(a_t|s_t)p(s_{t+1}|s_t,a_t)$. Then we can simplify the ratio in the following way:

$$\begin{aligned} \frac{\pi_\theta(\tau)}{\bar{\pi}(\tau)} &= \frac{p(s_1)\prod_{t=1}^{T}\pi_\theta(a_t|s_t)p(s_{t+1}|s_t,a_t)}{p(s_1)\prod_{t=1}^{T}\bar{\pi}(a_t|s_t)p(s_{t+1}|s_t,a_t)} \\ &= \frac{\prod_{t=1}^{T}\pi_\theta(a_t|s_t)}{\prod_{t=1}^{T}\bar{\pi}(a_t|s_t)} \end{aligned}$$

3.8.2 Deriving Policy Gradient with Importance Sampling

It turns out that we can recover the original policy gradient theorem using off-policy learning using importance sampling. Recall the objective of RL as defined in the first chapter:

$$\theta^* = \arg\max_\theta J(\theta)$$

and we defined $J(\theta)$ as $J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[r(\tau)\right]$. Now if we want to estimate J with some new parameter θ', we can use importance sampling as discussed above:

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

then we take the gradient as:

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

Now if we estimate it locally, by setting $\theta = \theta'$, then we will cancel out the importance ratio, ending up with $\mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\nabla_{\theta'}\log\pi_{\theta'}(\tau)r(\tau)\right]$.

3.9 First Order Approximation for Importance Sampling

Now we focus on the cases where we do not use local approximation, when $\theta \neq \theta'$.

$$\begin{aligned} J(\theta') &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[r(\tau)\right] \\ \nabla_{\theta'}J(\theta') &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\frac{\pi_{\theta'}(\tau)}{\pi_\theta(\tau)}\nabla_{\theta'}\log\pi_{\theta'}(\tau)r(\tau)\right] \\ &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\left(\frac{\prod_{t=1}^{T}\pi_{\theta'}(a_t|s_t)}{\prod_{t=1}^{T}\pi_\theta(a_t|s_t)}\right)\left(\sum_{t=1}^{T}\nabla_{\theta'}\log\pi_{\theta'}(a_t|s_t)\right)\left(\sum_{t=1}^{T}r(s_t,a_t)\right)\right] \end{aligned}$$

Now there is a problem in the equation. Note that the ratio of the two products can be very small or very big if T is big, thus increasing the variance. To alleviate the issue, one can make use of causality as we discussed before:

$$\begin{aligned}\nabla_{\theta'} J(\theta') &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\left(\frac{\prod_{t=1}^{T}\pi_{\theta'}(a_t|s_t)}{\prod_{t=1}^{T}\pi_\theta(a_t|s_t)}\right)\left(\sum_{t=1}^{T}\nabla_{\theta'}\log\pi_{\theta'}(a_t|s_t)\right)\left(\sum_{t=1}^{T} r(s_t,a_t)\right)\right] \\ &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\sum_{t=1}^{T}\nabla_{\theta'}\log\pi_{\theta'}(a_t|s_t)\left(\prod_{t'=1}^{t}\frac{\pi_{\theta'}(a_{t'}|s_{t'})}{\pi_\theta(a_{t'}|s_{t'})}\right)\left(\sum_{t'=t}^{T} r(s_{t'},a_{t'})\left(\prod_{t''=t}^{t'}\frac{\pi_{\theta'}(a_{t''}|s_{t''})}{\pi_\theta(a_{t''}|s_{t''})}\right)\right)\right]\end{aligned}$$

Here we used the fact of causality that future actions don't affect the current weight. Also note that the last ratio of products can be deleted, and we essentially get the policy iteration algorithm, which we will discuss in later chapters.

So when we delete the last weight, we end up having

$$\nabla_{\theta'} J(\theta') = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\sum_{t=1}^{T}\nabla_{\theta'}\log\pi_{\theta'}(a_t|s_t)\left(\prod_{t'=1}^{t}\frac{\pi_{\theta'}(a_{t'}|s_{t'})}{\pi_\theta(a_{t'}|s_{t'})}\right)\left(\sum_{t'=t}^{T} r(s_{t'},a_{t'})\right)\right]$$

The product of ratio is again exponential in T, so we may have high variance.

Recall on-policy policy gradient:

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

Similarly in off-policy policy gradient:

$$\begin{aligned}\nabla_{\theta'} J(\theta') &= \frac{1}{N}\sum_{i=1}^{T}\sum_{t=1}^{T}\frac{\pi_{\theta'}(s_{i,t},a_{i,t})}{\pi_\theta(s_{i,t},a_{i,t})}\nabla_{\theta'}\log\pi_{\theta'}(a_{i,t}|s_{i,t})\hat{Q}_{i,t} \\ &= \frac{1}{N}\sum_{i=1}^{T}\sum_{t=1}^{T}\frac{\pi_{\theta'}(s_{i,t})}{\pi_\theta(s_{i,t})}\frac{\pi_{\theta'}(s_{i,t}|a_{i,t})}{\pi_\theta(s_{i,t}|a_{i,t})}\nabla_{\theta'}\log\pi_{\theta'}(a_{i,t}|s_{i,t})\hat{Q}_{i,t}\end{aligned}$$

In later chapters, we can see that we can pretty much ignore the first states priors ratio.

3.9.1 Advanced Policy Gradients

Recall the policy gradients update rule:

$$\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$$

In many cases, some parameters have more impact on the outcome than others. Therefore, intuitively, we would like to set higher learning rate for parameters with less impact and lower learning rate for parameters with more impact. To do this, we leverage covariant/natural policy gradient. Let us look at at the constrained view of iterative gradient descent:

$$\begin{aligned}\theta' \leftarrow \arg\max_{\theta'}(\theta'-\theta)^T\nabla_\theta J(\theta) \\ \text{s.t. } ||\theta'-\theta||^2 \le \epsilon\end{aligned} \tag{3.2}$$

where this ϵ controls how far we should go. But this ϵ is defined in the parameters' space, which means that we do not have much control over individual parameters. To resolve this, we would like to **rescale** this constraint so that we can constrain the step size in terms of the policy space, thus giving us more control on individual parameters. For example, we can use:

$$\begin{aligned} \theta' \leftarrow \arg\max_{\theta'} & (\theta' - \theta)^T \nabla_\theta J(\theta) \\ \text{s.t. } & D(\pi_{\theta'} - \pi_\theta) \leq \epsilon \end{aligned} \tag{3.3}$$

where $D(\cdot, \cdot)$ is a parameterization-independent divergence measure, which usually is the KL-divergence: $D_{KL}(\pi_{\theta'} - \pi_\theta) = \mathbb{E}_{\pi_{\theta'}}[\log \pi_\theta - \log \pi_{\theta'}]$.

We can also estimate the KL divergence locally using second-order Taylor expansion by:

$$D_{KL}(\pi_{\theta'} - \pi_\theta) \approx (\theta' - \theta)^T F (\theta' - \theta)$$

where F is the Fisher-information matrix defined as:

$$F = \mathbb{E}_{\pi_{\theta'}}[\nabla_\theta \log \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s)^T]$$

Thus, with F, the rescaled constraint optimization problem can be equivalently rewritten as:

$$\begin{aligned} \theta' \leftarrow \arg\max_{\theta'} & (\theta' - \theta)^T \nabla_\theta J(\theta) \\ \text{s.t. } & ||\theta' - \theta||_F^2 \leq \epsilon \end{aligned} \tag{3.4}$$

Using Lagrangian, one could solve this optimization problem iteeratively as follows:

$$\theta \leftarrow \theta + \alpha F^{-1} \nabla_\theta J(\theta)$$

This is the basic idea behind TRPO (PPO).

Chapter 4: Actor-Critic Algorithms

Recall from last chapter, we derived the policy gradient theorem:

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

where we defined the summed reward as the "reward-to-go" function $\hat{Q}_{i,t}$, and it represents the estimate of expected reward if we take action $a_{i,t}$ in state $s_{i,t}$. We have shown that this estimate has very high variance, and we shall see how we can improve policy gradients from using better estimation of the reward-to-go function.

4.1 Reward-to-Go

Let us take a closer look at the reward-to-go. To improve the estimation, one way is to get closer to the precise value of the reward-to-go. We can define the reward-to-go using expectation:

$$Q(s_t, a_t) = \sum_{t'=t}^{T}\mathbb{E}_{p(\theta)}\left[r(s_{t'}, a_{t'})|s_t, a_t\right]$$

this is the **true, expected** value of the reward-to-go.

Therefore, one could imagine using this true expected value, combined with our original Monte Carlo approximation to yield a better estimate:

$$\nabla_\theta J(\theta) \simeq \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})Q(s_{i,t}, a_{i,t})$$

.

4.2 Using Baselines

As we have seen in last chapter, one can reduce the high variance of the policy gradient using baselines. We have also seen that it is possible to calculate the optimal baseline value to yield the minimum variance, although people often use the average reward for sake of simplicity.

Motivated by this, let us recall the definition of the value function (defined in the introduction section):

$$V(s_t) = \mathbb{E}_{a_t\sim\pi_\theta(a_t|s_t)}\left[Q(s_t, a_t)\right]$$

By definition, the value function is the average of Q-function value.

Similarly, we can use the **average** reward-to-go as a baseline to reduce the variance. Specifically, we could use the value function $V(s_t)$ as the baseline, thus improving the estimate of the gradient in the following way:

$$\nabla_\theta J(\theta) \simeq \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \left(Q(s_{i,t}, a_{i,t}) - V(s_{i,t})\right)$$

and the value function we used is a better approximation of the baseline $b_t = \frac{1}{N} \sum_i Q(s_{i,t}, a_{i,t})$.

What have we done here? What is the intuition behind subtracting the value function from the Q-function? Essentially, we are quantifying how much an action $a_{i,t}$ is better than the average actions. In some sense, it measures the **advantage** of applying an action over the average action. Therefore, to formalize our intuition, let us define the advantage as follows:

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$$

which quantitatively measures how much better action a_t is.

Putting it all together, now a better baseline-backed policy gradient estimate using Monte Carlo estimate can be written as:

$$\nabla_\theta J(\theta) \simeq \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) A^\pi(s_{i,t}, a_{i,t})$$

.

4.3 Value Function Fitting

The better the estimate of the advantage function, the lower the variance, and we can have better policy gradient. Let us massage the definition of the Q-function a little in order to find some interesting mathematical relations between Q and V:

$$\begin{aligned}
Q^\pi(s_t, a_t) &= \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta} \left[r(s_{t'}, a_{t'})|s_t, a_t\right] \\
&= r(s_t, a_t) + \sum_{t'=t+1}^{T} \mathbb{E}_{\pi_\theta} \left[r(s_{t'}, a_{t'})|s_t, a_t\right] \\
&= r(s_t, a_t) + V^\pi(s_{t+1}) \\
&= r(s_t, a_t) + \mathbb{E}_{s_{t+1} \sim p(s_{t+1}|s_t, a_t)} \left[V^\pi(s_{t+1})\right]
\end{aligned}$$

The last expectation of the value function is used because we do not know what the next state actually is. Note that we can be a little crude with respect to that expectation in such a way that we just use the full value function $V^\pi(\cdot)$ on one single sample of the next state, and use the value as the expectation, ignoring the fact that there are multiple other next states. With this estimate, we can plug into the advantage function:

$$A^\pi(s_t, a_t) \simeq r(s_t, a_t) + V^\pi(s_{t+1}) - V^\pi(s_t)$$

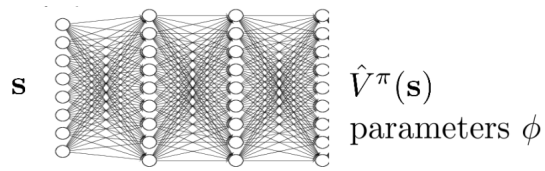

Figure 4.1: Fitting the value function

Therefore, it is almost enough to just approximate the value function, which solely depends on state, to generate approximations of other functions. To achieve this, we can use a neural network to fit our value function $V(s)$, and use the fit value function to approximate our policy gradient, as illustrated in Fig. 4.1

4.4 Policy Evaluation

Here in this section, we discuss the process and purpose of fitting the value function.

4.4.1 Why Do We Evaluate a Policy

Policy evaluation is a process that given a fixed policy π, we figure out how good it is by fitting the value function $V^\pi(\cdot)$ by using this expectation:

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

Having the value function allows us to figure out how good the policy is because the reinforcement learning objective can be equivalently written as $J(\theta) = \mathbb{E}_{s_1 \sim p(s_1)}\left[V^\pi(s_1)\right]$, where we take the expectation of the value function value of the initial state over all possible initial states.

4.4.2 How to Evaluate a Policy

To evaluate a policy, we can use an approach similar to the policy gradient - Monte Carlo approximation. Specifically, we can estimate the value function by summing up the reward collected from time step t:

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

and if we are able to reset the simulator, we could indeed ameliorate this estimate by taking multiple samples (N) as follows:

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

In practice, we can just use the single sample approximation.

Here is a question, if our original objective is to use V^π to reduce the variance, but we end up using a single sample estimation to estimate V^π, does it actually help? The

answer is yes, because we are using a neural net to fit the Monte Carlo targets from a variety of different states, so even though we do single sample estimate, the value function does generalize when we visit similar states.

4.4.3 Monte Carlo Evaluation with Function Approximation

To fit our value function, we could use a supervised learning approach. Essentially, we can use our single sample estimation of the value function as our function value, and fit a function that maps the states to the value function values. Therefore, our training data will be $\left\{(s_{i,t}, \sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'}))\right\}$, and we denote the function value labels as $y_{i,t}$, and we define a typical supervised regression loss function which we try to minimize as $\mathcal{L}(\phi) = \frac{1}{2}\sum_i ||\hat{V}_\phi^\pi(s_i) - y_i||^2$.

4.4.4 Improving the Estimate Using Bootstrap

In fact, we can improve our training process because the original applied target $y_{i,t}$ is not perfect. We could use a technique called **bootstrapping**. Recall the definition of our ideal target in the supervised regression:

$$\begin{aligned} y_{i,t} &= \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}\left[r(s_{t'}, a_{t'})|s_{i,t}\right] \\ &\simeq r(s_{i,t}, a_{i,t}) + \sum_{t'=t+1}^{T} \left[r(s_{t'}, a_{t'})|s_{i,t+1}\right] \\ &\simeq r(s_{i,t}, a_{i,t}) + V^\pi(s_{i,t+1}) \end{aligned}$$

, compared with our Monte Carlo targets: $y_{i,t} = \sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'})$.

Bootstrapping means applying our previous estimation on our current estimation. In our ideal targets, the last estimation is accurate if we knew the actual V^π. But if the actual value function is not known, we can just apply bootstrapping by using the current fit estimate $\hat{V}_\phi^\pi$ to estimate the next state's value: $\hat{V}_\phi^\pi(s_{i,t+1})$. Such an estimate is biased, but it has low variance.

Consequently, our training data using bootstrapping becomes:

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

. Such bootstrapped targets work well with highly stochastic environments.

4.5 Batch Actor-Critic Algorithm

Now we are ready to devise our first actor-critic algorithm. The reason why we call it actor-critic is that we use a critic (value function) to decrease the high variance of the actor (Q-function/policy). The full algorithm is shown in Alg. 3 and we call it a batch algorithm because it is not online. We shall see the online version later. In Algorithm 3, the way how we fit $\hat{V}_\phi$ is by minimizing the supervised regression norm $\mathcal{L}(\phi) = \frac{1}{2}\sum_i ||\hat{V}_\phi^\pi(s_i) - y_i||^2$.

Algorithm 3 Batch Actor-Critic Algorithm

Require: Base policy $\pi_\theta(a_t|s_t)$

1: **while** true **do**
2: Sample $\{s_i, a_i\}$ from $\pi_\theta(a|s)$ (run it on a robot)
3: Fit $\hat{V}_\phi(s)$ to sampled reward sums
4: Evaluate $\hat{A}^\pi(s_i, a_i) = r(s_i, a_i) + \hat{V}_\phi(s'_i) - \hat{V}_\phi(s_i)$
5: $\nabla_\theta J(\theta) \simeq \sum_i \nabla_\theta \log \pi_\theta(a_i|s_i)\hat{A}^\pi(s_i, a_i)$
6: Improve policy by $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$
7: **return** optimal policy from gradient ascent as π^{return}

4.6 Aside: Discount Factors

Imagine if we had an infinite horizon environment ($T \to \infty$), then our estimated value function $\hat{V}^\pi_\phi(s)$ can get infinitely large in many cases. Therefore, one possible way to address this issue is to say that it is better to get rewards sooner than later. Therefore, instead of labeling our values as $y_{i,t} \simeq r(s_{i,t}, a_{i,t}) + \hat{V}^\pi_\phi(s_{i,t+1})$, we can shrink the value function value as we progress to the next time step. To achieve this, we introduce a hyperparameter called a **discount factor**, denoted as γ, where $\gamma \in [0, 1]$:

$$y_{i,t} \simeq r(s_{i,t}, a_{i,t}) + \gamma \cdot \hat{V}^\pi_\phi(s_{i,t+1})$$

in most cases, $\gamma = 0.99$ works well.

Let us apply the discount factor to policy gradients. Basically, we have two options to impose this discount factor. The first option is :

$$\nabla_\theta J(\theta) \simeq \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_{t'=t}^{T}\gamma^{t'-t}r(s_{i,t'}, a_{i,t'})\right)$$

and the second option is:

$$\begin{aligned}
\nabla_\theta J(\theta) &\simeq \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_{t=1}^{T}\gamma^{t-1}r(s_{i,t}, a_{i,t})\right) \\
&\simeq \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_{t'=t}^{T}\gamma^{t'-1}r(s_{i,t'}, a_{i,t'})\right) \text{ (causality)} \\
&\simeq \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\gamma^{t-1}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_{t'=t}^{T}\gamma^{t'-t}r(s_{i,t'}, a_{i,t'})\right)
\end{aligned}$$

Intuitively, the second option assigns less weight to later step's gradient, so it essentially means that later steps matter less in our discount.

In practice, we can show that option 1 gives us better variance, so it is actually what we use. The full derivation can be found in this paper [3]. Now in our actor-critic algorithm, after we impose the discount factor, we have the following gradient:

$$\nabla_\theta J(\theta) \simeq \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\right)\left(r(s_{i,t}, a_{i,t}) + \gamma\hat{V}^\pi_\phi(s_{i,t+1}) - \hat{V}^\pi_\phi(s_{i,t})\right)$$

Algorithm 4 Batch Actor-Critic Algorithm with Discount Factor

Require: Base policy $\pi_\theta(a_t|s_t)$, hyperparameter γ

1: **while** true **do**
2: Sample $\{s_i, a_i\}$ from $\pi_\theta(a|s)$ (run it on a robot)
3: Fit $\hat{V}_\phi(s)$ to sampled reward sums
4: Evaluate $\hat{A}^\pi(s_i, a_i) = r(s_i, a_i) + \gamma\hat{V}_\phi(s_i') - \hat{V}_\phi(s_i)$
5: $\nabla_\theta J(\theta) \simeq \sum_i \nabla_\theta \log \pi_\theta(a_i|s_i)\hat{A}^\pi(s_i, a_i)$
6: Improve policy by $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$
7: **return** optimal policy from gradient ascent as π^{return}

Algorithm 5 Online Actor-Critic Algorithm

Require: Base policy $\pi_\theta(a_t|s_t)$, hyperparameter γ

1: **while** true **do**
2: Take action $a \sim \pi_\theta(a|s)$, get (s, a, s', r)
3: Update $\hat{V}_\phi^\pi$ using target $r + \gamma\hat{V}_\phi^\pi(s')$
4: Evaluate $\hat{A}^\pi(s, a) = r(s, a) + \gamma\hat{V}_\phi^\pi(s') - \hat{V}_\phi^\pi(s)$
5: $\nabla_\theta J(\theta) \simeq \sum_i \nabla_\theta \log \pi_\theta(a|s)\hat{A}^\pi(s, a)$
6: Improve policy by $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$
7: **return** optimal policy from gradient ascent as π^{return}

Now we can incorporate the discount factor with our actor-critic algorithm in Algorithm 4.

4.7 Online Actor-Critic Algorithm

Now that we have seen actor-critic algorithms with a batch of samples, we can further improve the performance by making it fully online. Namely, we are taking the gradient step based on the current sample so that we are not storing any large number of samples, which is more efficient. In the online version of actor-critic, we essentially use two neural nets: one for the policy, the other one for the value function. This is simple and stable, but as the states dimension becomes higher, we are not giving any shared features between the actor and the critic. Therefore, we can also make the network shared between the policy and the value function. For example, in image-based observations scenarios, we could share the conv layers' weights for the two networks and only differ the two in the final fully connected layers.

In each step, we can only take one sample and gradually improve our value function using that sample. Here is the sketch of the online version of actor-critic algorithm in Algorithm 5. Note that in steps 3-5, we are only taking a gradient step from one sample. In reality, this works best if we use a batch of samples instead of just one, and one can use parallel workers (simulations) either synchronously or asynchronously to achieve it, as illustrated in Fig 4.2.

One caveat about the asynchronous version is that as the parameter server gets updated, the data collection policy might not be updated, which means the newly collected data might not come from the latest policy, thus making the current acting policy slightly outdated. Such problem is less of an issue in practice because the policy only gets updated by a tiny bit

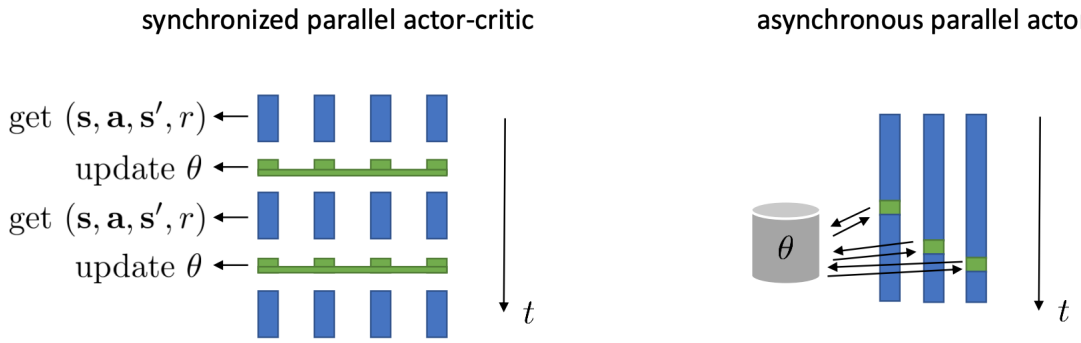

Figure 4.2: Parallel simulations for online actor-critic

every time.

4.8 Critics as State-Dependent Baselines

Now let us further discuss the connection between a baseline and a critic. Recall in the Monte Carlo version of policy gradient, the gradient is defined as:

$$\nabla_\theta J(\theta) \simeq \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \right) \left(\sum_{t'=1}^{T} r(s_{i,t'}, a_{i,t'}) - b \right)$$

and in actor-critic algorithm, we estimate the gradient by estimating the advantage function:

$$\nabla_\theta J(\theta) \simeq \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \right) \left(r(s_{i,t}, a_{i,t}) + \gamma \hat{V}_\phi^\pi(s_{i,t+1}) - \hat{V}_\phi^\pi(s_{i,t}) \right)$$

So what are the pros and cons of the two approaches? In policy gradient with baselines, we have shown that there is no bias in our estimation, but there might be high variance due to our single-sample estimation of the cost-to-go function. On the other hand, in the actor-critic algorithm, we have shown that we have lower variance due to the critic, but we end up having a biased estimation because of the possibly bad critic as we are bootstrapping. So can we somehow keep the estimator unbiased while lowering the variance with the critic $\hat{V}_\phi^\pi$?

The solution is obvious and straightforward, we can just use $\hat{V}_\phi^\pi$ in place of b:

$$\nabla_\theta J(\theta) \simeq \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \right) \left(\sum_{t'=1}^{T} r(s_{i,t'}, a_{i,t'}) - \hat{V}_\phi^\pi(s_{i,t}) \right)$$

In this way, we obtain an unbiased estimator with lower variance.

4.9 Eligibility Traces and n-Step Returns

In the above comparison of the two methods, we have seen that in the actor-critic advantage function, we have lower variance but higher bias, while in the Monte Carlo policy gradient,

the advantage function has lower bias but higher variance. The reason why this tradeoff exists is that as we go further in our trajectory into the future, the variance increases due to the fact that the current single sample approximation is not representative enough for the future. Therefore, the Monte Carlo advantage function is good for getting accurate values in the near term, but not the long term. In contrast, in actor-critic advantage, the bias potentially skews the values in the near term, but the fact that the bias incorporates a lot of states will likely make it a better approximator in the long run. Therefore, it would be better if we could use the actor-critic based advantage for further in the future, and use the Monte Carlo based one for the near term in order to control the bias-variance tradeoff.

As a result, we can cut the trajectory before the variance gets too big. Mathematically, we can estimate the advantage function by combining the two approaches: use the Monte Carlo approach only for the first n steps:

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

here we applied an n-step estimator, which sums the reward from now to n steps from now, and $n > 1$ often gives us better performance.

Furthermore, if we don't want to choose just one n, we can use a weighted combination of different n-steps returns, which we can define as the General Advantage Estimation(GAE):

$$\hat{A}_{GAE}(s_t, a_t) = \sum_{n=1}^{\infty} w_n \hat{A}_n^\pi(s_t, a_t)$$

To choose the weights, we should prefer cutting earlier, so we can assign the weights accordingly: $w_n \propto \lambda^{n-1}$, where we call λ the chance of getting cut.

Chapter 5: Value Function Methods

In the last two chapters, we discussed some policy gradient-based algorithms. We have also seen the fact that the policy gradient methods have high variance. Therefore, it would be nice if we could completely omit the gradient step. To achieve this, we are going to talk about the value function methods in this chapter.

5.1 An Implicit Policy

To omit the policy gradient, one still has to generate a policy function so that it takes in a state and outputs an action. Recall in actor-critic algorithms, we use the advantage function $A^\pi(s_t, a_t)$ to gauge how much better is the action a_t than the average action according to π. Provided that we have a somehow accurate representation of this advantage function, we can just forget about generating a policy π, and just do this:

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

which means we take the best action from s_t, if we follow π. Even though we have no knowledge of what the policy π actually is, by doing the $\arg\max$, we can guarantee that the action produced is at least as good as the action from the policy function that we do not know. Therefore, as long as we have a accurate representation of the advantage function $A^\pi(s_t, a_t)$, we can implicitly generate a parameter-free policy function:

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

and as we have shown above, this implicit policy is at least as good as the unknown policy π.

5.2 Policy Iteration

Having omitted the policy, we can then proceed to introduce the policy iteration algorithm.

5.2.1 High Level Idea

The basic idea of policy iteration algorithms is very simple: we evaluate the advantage function $A^\pi(s, a)$ and then update the policy using the update rule as defined above in the implicit policy, and then we can loop to constantly improve our policy.

The problem here is how to evaluate $A^\pi(s, a)$. In other words, how does one find an accurate representation of the advantage function in order to accurately update the policy. As before, we have seen that the advantage function an be equivalently defined as follows:

$$A^\pi(s, a) = r(s, a) + \gamma \mathbb{E}\left[V^\pi(s')\right] - V^\pi(s)$$

Algorithm 6 Policy Iteration via DP

1: **while** true **do**
2: Evaluate $V^\pi(s, a)$
3: Set $\pi \leftarrow \pi'$

Algorithm 7 Value Iteration via DP

1: **while** true **do**
2: Set $Q(s, a) \leftarrow r(s, a) + \gamma \mathbb{E}\left[V(s')\right]$
3: Set $V(s) \leftarrow \max_a Q(s, a)$

Therefore, if we can evaluate the value function $V^\pi(s)$ then we can also evaluate $A^\pi(s, a)$. So in the high-level policy iteration algorithm, we can just use the value function in place of the advantage function.

5.2.2 Dynamic Programming

Now let us make a simple assumption. Suppose we know a priori the transition probability $p(s'|s, a)$ and both states s and action a are discrete. Then a very natural dynamic programming update is the bootstrapped update, as we have seen before:

$$V^\pi(s) \leftarrow \mathbb{E}_{a\sim\pi(a|s)}\left[r(s, a) + \gamma \mathbb{E}_{s'\sim p(s'|s,a)}\left[V^\pi(s')\right]\right]$$

and we can just use the current estimate inside the nested expectation for simplicity.

According to our definition of the implicit policy function π', the policy is actually deterministic. Therefore, we can completely get rid of the outside expectation, and the value function update can be further simplified as:

$$V^\pi(s) \leftarrow r(s, \pi(s)) + \gamma \mathbb{E}_{s'\sim p(s'|s,a)}\left[V^\pi(s')\right]$$

This version of policy iteration is sketched in Algorithm 6

We can even further simplify the dynamic programming update. Note that we in each iteration, we are updating the policy first in order to update the value function. Thus, it would be faster if we could skip the policy part and directly improve the value estimation. Meanwhile, the arg max operation that we apply on the advantage function itself is an implicit policy. We also know that $\arg\max_{a_t} A^\pi(s, a) = \arg\max_{a_t} Q^\pi(s, a)$, because the two values only differ by the subtraction term V^π, which does not depend on action:

$$A^\pi(s, a) = r(s, a) + \gamma \mathbb{E}\left[V^\pi(s')\right]$$

Having this, we can simplify the policy iteration algorithm further, as illustrated in Alg. 7. As we skipped the policy update part, we call this new, simplified algorithm "value iteration algorithm".

5.3 Fitted Value Iteration

The policy iteration and value iteration algorithm we discussed above are heavily based on an impractical assumption: the total number of states is finite and small, because we are trying

Algorithm 8 Fitted Value Iteration

while true **do**
  set $y_i \leftarrow \max_{a_i} (r(s_i, a_i) + \gamma \mathbb{E}[V_\phi(s_i')])$
  set $\phi \leftarrow \arg\min_\phi \frac{1}{2} \Sigma_i ||V_\phi(s_i) - y_i||^2$

Algorithm 9 Fitted Q-Iteration Algorithm

Require: Some base policy for data collection; hyperparameter K
while true **do**
  Collect dataset $\{(s_i, a_i, s_i', r_i)\}$ using some policy
  for K times **do**
    Set $y_i \leftarrow r(s_i, a_i) + \gamma \max_{a_i'} Q_\phi(s_i', a_i')$
    Set $\phi \leftarrow \arg\min_\phi \frac{1}{2} \Sigma_i ||Q_\phi(s_i, a_i) - y_i||^2$

to construct a tabular expression of the value function and the Q function. Apparently, the tables are going to explode in dimensions if there are a lot of states. We call this the Curse of Dimensionality. To resolve this problem, we can use a neural network to approximate the functions instead of constructing a tabular expression of the function.

5.3.1 Fitted Supervised Value Iteration Algorithm

Since we know that the value function is defined as $\max_a Q^\pi(s, a)$, we can use this definition as the labels for the value function in order to define a L2 loss function:

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

Then we can sketch out a simple fitted value iteration algorithm using this loss function in Algorithm 8. Note that when setting the label, the ideal step to take is to enumerate all the states and find the corresponding label. However, when it is impractical, one could just use some samples and enumerate all the actions to find the labels. Moreover, when we take the maximum over all the actions from a state, we implicitly assume that the transition dynamics are known. Why? Because we want to take an action, record the value of that action, and then roll back to the previous state in order to check the values of other actions. Thus, without the transition dynamics, we cannot easily take the maximum.

5.3.2 Fitted Q-Iteration Algorithm

To address this problem, we can apply the same "max" trick in policy iteration. In policy iteration, we skip the policy update and calculate the values directly. Here in fitted value iteration, we can get around the transition dynamics by looking up the Q function table, because $V_\phi(s) \simeq \max_a Q_\phi(s, a)$, and this max operation is merely a table lookup from the Q value table. Consequently, we are now iterating on the Q values. Such a method works for off-policy samples (unlike actor-critic), and it only needs one network, so it does not have any high-variance policy gradient. However, as we shall see in later sections, such methods do not have convergence guarantees on non-linear functions, which could potentially be problematic.

The full fitted Q-iteration algorithm is shown in Algorithm 9.

Algorithm 10 Online Q-Iteration Algorithm

1: **while** true **do**
2: Take some action a_i and observe (s_i, a_i, s_i', r_i)
3: $y_i = r(s_i, a_i) + \gamma \max_{a'} Q_\phi(s_i, a_i, s_i', r_i)$
4: $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(s_i, a_i)(Q_\phi(s_i, a_i) - y_i)$

5.3.3 A Closer Look at Q-Iteration Algorithm

Let us take a closer look at the fitted Q-learning algorithm. First, let us discuss why the algorithm is fully off-policy. In step 2 of Alg. 9, we are not collecting a lot of transition data, and we do not care about the trajectories. Furthermore, in step 4, we are taking the step off-policy in that we do not care about which state we are going to, because we only care about the value of the transition. In other words, the tansition we take is independent of the unknown policy π. Therefore, the fitted Q-iteration algorithm is fully off-policy.

Another question we can ask is that what is fitted Q-iteration actually optimizing? In step 5 of Alg. 9, we are minimizing the difference between the Q function value and the label we approximated. In fact, we call this difference the Bellman Error, defined as follows:

$$\epsilon = \frac{1}{2}\mathbb{E}_{(s,a)\sim\beta}\left[\left(Q_\phi(s,a) - \left[r(s,a) + \gamma \max_{a'} Q_\phi(s', a')\right]\right)^2\right]$$

So in this particular step, we are optimizing the Bellman Error, and if $\epsilon = 0$, we have optimal Q-function, corresponding to optimal policy π, which can be recovered by the $\arg\max$ operation. However, rather ironically, we do not know what we are optimizing in the previous steps, and this is a potential problem of the fitted Q-learning algorithm, and most convergence guarantees are lost when we do not have the tabular case.

5.3.4 Online Q-Iteration Algorithm

We can also make the samples more efficient by making the Q-iteration algorithm completely online. By online we mean that we do not store any transition. Instead, we take one transition and immediately apply the transition to our value update. The online version of Q-Iteration Algorithm is sketched in Alg. 10. As we see in step 2 of the algorithm, we are taking an action off-policy, so we have a lot of choices to make.

5.4 Value Function Learning Theory

One question that one might ask after seeing the variety of algorithm as shown above is does the value function method converge? If so, it converges to what? To take a closer look in order to answer the question, let us define a Bellman backup operator $\mathcal{B}$:

$$\mathcal{B}V = \max_a r_a + \gamma \mathcal{T}_a V$$

where r_a is a stacked vector of rewards at all states for action a. $\mathcal{T}_a$ is a matrix of transitions for action a such that $\mathcal{T}_{a,i,j} = p(s' = i | s = j, a)$. We also define a fixed point of the Bellman backup operator $\mathcal{B}$, denoted as V^*:

$$V^*(s) = \max_a r(s,a) + \gamma \mathbb{E}[V^*(s')]$$

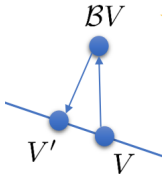

Figure 5.1: Bellman Backup Projection

so it is similar to the notion of the stationary distribution in MDP, $V^* = \mathcal{B}V^*$. One can prove that such fixed point always exists, and it corresponds to the optimal policy, but the online question is: will we reach it?

In the **tabular representation** case, we can prove that value iteration always reaches the fixed point V^* because mathematically, the Bellman backup operator is a **contraction**. A contraction in our scenario is defined as follows: for any V and $\bar{V}$, we have $||\mathcal{B}V - \mathcal{B}\bar{V}||_\infty \leq ||V - \bar{V}||_\infty$. In other words, after applying the Bellman backup operator, the gap always gets smaller by γ with respect to the l-∞ norm.

Now let us proceed to analyze the **non-tabular representation** case. In this scenario, unfortunately, we have lost a lot of convergence guarantees. Recall that in normal value iteration (tabular case), we use the Bellman backup operator $\mathcal{B}$ to update V: $V \leftarrow \mathcal{B}V$. In fitted (non-tabular) value iteration, we use the Bellman backup operator, $\mathcal{B}$, together with another operator Π. The operator Π is defined as: $\Pi V = \arg\min_{V' \in \Omega} \frac{1}{2} \sum ||V'(s) - V(s)||^2$. So Π is a projection onto Ω in terms of l2 norm. This projection is illustrated in Fig. 5.1. The set that V and V' lie in can be thought as a representation of all value functions. Therefore, the set Ω can be represented by neural networks.

Now we have the two operators defined, we can see that $\mathcal{B}$ is a contraction with respect to the l-∞ norm, and the operator Π is a contraction with respect to the l2 norm. But what if we impose one operator to another? Is the compound operator also a contraction? The answer is no. Therefore, such non-tabular Q-iterations do not have any convergence guarantee as the operator is not a contraction.

What about fitted Q-iteration? Concisely, the fitted Q-iteration algorithm can be defined as $Q \leftarrow \Pi\mathcal{B}Q$. Therefore, the same reasoning can be applied to the fitted Q-learning: since the compound operator is no longer a contraction, we do not have any guarantee for convergence. We can say the same thing in online Q-iteration as well.

However, one might ask, in step 4 of Alg. 10, aren't we just doing gradient descent, which definitely converges? As a matter of fact, this is not real gradient descent in that the target value is constantly changing due to the off-policy nature of this algorithm. So we have this sad corollary: in general cases, fitted bootstrapped policy evaluation does not converge.

Chapter 6: Q-Function Methods

Recall the algorithms that we discussed in last chapter: Alg. 9 and Alg. 10, where we devised a fitted Q-learning algorithm and a fully online version of it. We have also shown that Q-learning is fully off-policy, meaning that we do not care about the trajectory we are taking, we only care about the current transition and the next state we land in. So what is the problem with the above Q-learning algorithms? To see this, let us carefully look at step 4 of Alg. 10. The gradient step that we are taking is equivalently written as:

$$\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(s_i, a_i) \left(Q_\phi(s_i, a_i) - \left[r(s_i, a_i) + \gamma \max_{a'} Q_\phi(s_i, a_i, s_i', r_i) \right] \right)$$

This is not gradient descent! Because the "target" value y_i is not constant and depends on our parameter ϕ, and we are not taking gradient from y_i. Therefore, this is not the gradient descent step that we used to see before. Moreover, in step 2, we are only taking one sample of transition. This sampling scheme brings us two problems: the first one is that one sample is really hard to train the network (recall in online actor-critic, we would use parallel workers to obtain multiple online samples), and the second problem is the samples we are drawing are correlated in that the transitions are dependent on each other. As you may know, Stochastic Gradient Descent converges only if we are taking the correct gradient, and when the samples are IID. We violated both requirements, so the Q-learning algorithms in Alg. 9 and Alg. 10 do not have any convergence guarantees.

6.1 Replay Buffers

One of the solutions that we can implement to resolve the above correlation issue is to use a **replay buffer**. A replay buffer $\mathcal{B}$ is a buffer of transition data that contains the single samples that we have drawn so far. Therefore, in Q-learning, if we sample a batch from $\mathcal{B}$, the samples are no longer correlated, and we also keep updating the replay buffer by adding in real-world transitions. One can view this interaction in the image illustrated in Fig. 6.1.

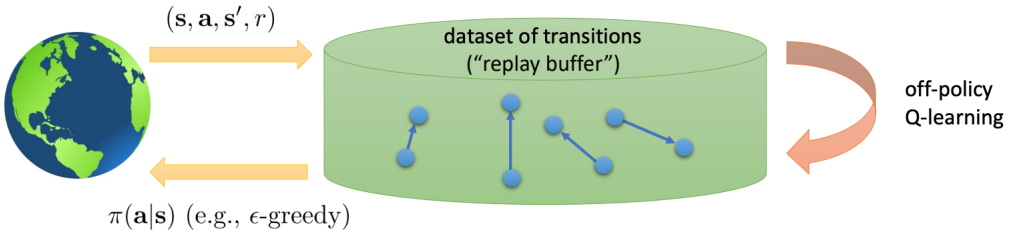

Figure 6.1: Q-learning with replay buffers

Algorithm 11 Q-Learning with Replay Buffer

Require: Some base policy for data collection; hyperparameter K

1: **while** true **do**
2: Collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy, add it to replay buffer $\mathcal{B}$
3: **for** K times **do**
4: Sample a batch (s_i, a_i, s'_i, r_i) from $\mathcal{B}$
5: Set $y_i \leftarrow r(s_i, a_i) + \gamma \max_{a'_i} Q_\phi(s'_i, a'_i)$
6: Set $\phi \leftarrow \phi - \alpha \Sigma_i \frac{dQ_\phi}{d\phi}(s_i, a_i)(Q_\phi(s_i, a_i) - y_i)$

Algorithm 12 Q-Learning with Replay Buffer and Target Network

Require: Some base policy for data collection; hyperparameter K and N

1: **while** true **do**
2: Save target network parameters: $\phi' \leftarrow \phi$
3: **for** N times **do**
4: Collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy, add it to replay buffer $\mathcal{B}$
5: **for** K times **do**
6: Sample a batch (s_i, a_i, s'_i, r_i) from $\mathcal{B}$
7: Set $y_i \leftarrow r(s_i, a_i) + \gamma \max_{a'_i} Q_{\phi'}(s'_i, a'_i)$
8: Set $\phi \leftarrow \phi - \alpha \Sigma_i \frac{dQ_\phi}{d\phi}(s_i, a_i)(Q_\phi(s_i, a_i) - y_i)$

Putting it all together, we sketch out the full Q-learning algorithm with replay buffer in Alg. 11. What have changed here in Alg. 11 compared with Alg. 9? In step 2, we are not only collecting dataset, but also adding the dataset to replay buffer $\mathcal{B}$. Inside the for loop, we are now sampling a batch of transitions from $\mathcal{B}$, which will bring us lower variance when we take the gradient step on the batch. We also periodically update $\mathcal{B}$.

The above solution solves the correlation problem, but we still need to address the wrong gradient problem.

6.2 Target Networks

To solve the wrong gradient problem, we are essentially trying to solve the problem that the target value y_i is heavily dependent on our gradient parameter ϕ. Therefore, one way to improve this is to use a separate Q-function with another parameter ϕ' in order to decorrelate the two values. We should also use a well-defined ϕ', which can be set as the ϕ parameter 1000 steps ago. When setting y_i, we set it from the Q function with another parameter, thus decorrelating the two values. Together with the use of replay buffer, we alleviated the wrong policy and the correlation samples problems, as shown in Alg. 12. Here we are frequently sampling batches from $\mathcal{B}$ and taking the gradient steps. We are less frequently updating the replay buffer as discussed in Alg. 11. We are even less frequently updating the network parameters, since we said one good choice for parameter ϕ' is to use ϕ 1000 steps ago.

Using target networks and replay buffers, we can sketch out a classic deep Q-learning algorithm (DQN), proposed by Minh et al. in [4]. The pseudocode is in 13. Here we choose N to be a large number because our intention is to infrequently update the parameters. To further optimize the algorithm, it might feel weird abruptly update ϕ' after N steps.

Algorithm 13 Classic Deep Q-Learning Algorithm (DQN)

1: **while** true **do**
2: Take some action a_i and observe (s_i, a_i, s'_i, r_i) and add it to $\mathcal{B}$
3: Sample mini-batch $\{s_j, a_j, s'_j, r_j\}$
4: Compute $y_j = r_j + \gamma \max_{a'_j} Q_{\phi'}(s'_j, a'_j)$ using target network $Q_{\phi'}$
5: $\phi \leftarrow \phi - \alpha \Sigma_j \frac{dQ_\phi}{d\phi}(s_j, a_j)(Q_\phi(s_j, a_j) - y_J)$
6: update ϕ': copy ϕ every N steps.

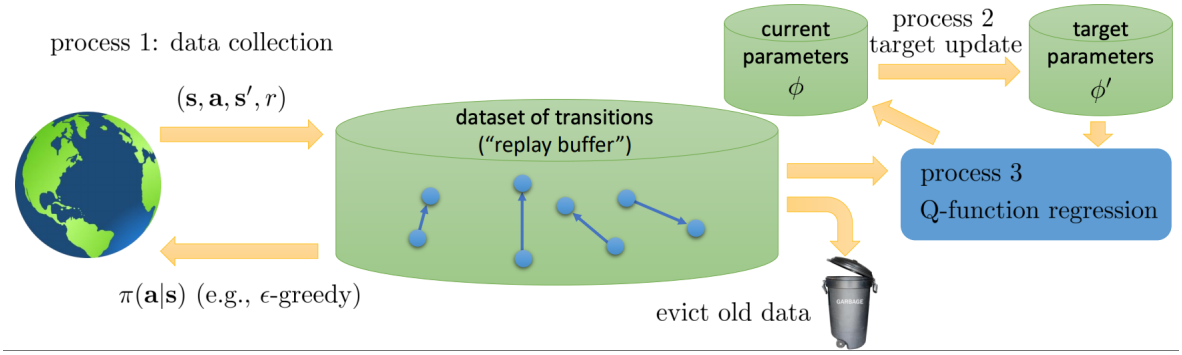

Figure 6.2: Q-learning in a more general view.

Therefore, to alleviate this "abruptness", we can use **Polyak Averaging**: in step 6 of Alg. 13, instead of copying ϕ every N steps, we do $\phi' \leftarrow \tau\phi' + (1-\tau)\phi$. We also call such update a damped update.

Now let us view the three different Q-learning algorithms in a more general way. As shown in Fig. 6.2, there are three different steps in the algorithm. The first step is to collect data, the second step is to update the target in the target network, and the third step is to regress onto the Q-function. In the simplest, regression-based fitted Q-learning algorithm, process 3 is in the inner loop of process 2, which is in the inner loop of process 1. In online Q-learning, we evict the old transitions immediately, and process 1, 2, and 3 run at the same speed. In DQN, process 1 and 3 run at the same speed, but process 2 runs slower.

6.3 Inaccuracy in Q-Learning

Q-values are not necessarily accurate. The reason lies in the target value. Recall that the target value y is defined as $y_j = r_j + \gamma \max_{a'_j} Q_{\phi'}(s'_j, a'_j)$. The max operation in the target is the main problem, because for two random variables X_1 and X_2, $\mathbb{E}[\max(X_1, X_2)] \geq \max(\mathbb{E}[X_1], \mathbb{E}[X_2])$. Therefore, when $Q_{\phi'}(s'_j, a'_j)$ is noisy, the max operation is going to overestimate the next Q-value.

6.3.1 Double Q-Learning

One might notice that $\max_{a'} Q_{\phi'}(s', a') = Q_{\phi'}(s', \arg\max_{a'} Q_{\phi'}(s', a'))$. Thus, if we somehow managed to decorrelate the error from the selected action and the error from the Q-function, we could eliminate the erroneous overestimation. To achieve this, we can use two different

networks to choose actions and evaluate the Q-function values.

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

Essentially we are using one network's parameter to update the value, while using the other's to select the action. Using the two separate networks, we are decorrelating the action selection and value evaluation errors, thus decreasing the overestimation in the Q-values.

In practice, we can just use the actual and target networks for the two separate networks. Therefore, instead of setting target y as $y = r + \gamma Q_{\phi'}\left(s', \arg\max_{a'} Q_{\phi'}(s',a')\right)$, we use the current network to select action, and use the target network to evaluate value: $y = r + \gamma Q_{\phi'}\left(s', \arg\max_{a'} Q_{\phi}(s',a')\right)$.

6.3.2 N-Step Return Estimator

In the definition of our target y, $y_{i,t} = r_{i,t} + \max_{a_{i,t+1}} Q_{\phi'}(s_{i,t+1}, a_{i,t+1})$, the Q-value only matters if it is a good estimate. If the Q-value estimate is bad, the only values that matter are from the reward term, so we are not learning much about the Q-function. To resolve this problem, let us recall the N-step cut trick we did in the actor-critic algorithm. In actor-critic algorithm, to leverage the bias and variance tradeoff in policy gradient, we can end the trajectory earlier, and only count the reward summed up to N steps from now. Specifically, we can define the target as:

$$y_{i,t} = \sum_{t'=t}^{t+N-1} \gamma^{t-t'} r_{i,t} + \gamma^N \max_{a_{i,t+1}} Q_{\phi'}(s_{i,t+1}, a_{i,t+1})$$

One subtle problem with this solution is that the learning process suddenly becomes on-policy, so we cannot efficiently make use of the off-policy data. Why is it on-policy? If we look at the summation of the rewards, we are collecting the rewards data using a certain trajectory, which is generated by a specific policy. Therefore, we end up having less biased target values when the Q-values are inaccurate, and in practice, it is faster in early stages of learning. However, it is only correct when we are learning on-policy. To fix this problem, one could ignore this mismatch, which somehow works very well in practice. Or one could cut the trace by dynamically adapting N to get only on-policy data. Also, one could use importance sampling as we discussed before. For more details, please refer to this paper by Munos et al. [5].

6.3.3 Q-Learning with Continuous Actions

Recall the implicit policy that we define using Q-learning:

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

One problem with this definition is that the arg max operation cannot be easily applied if the actions are continuous. How are we going to address such an issue?

Algorithm 14 Deep Deterministic Policy Gradient (DDPG)

1: **while** true **do**
2: Take some action a_i and observe (s_i, a_i, s_i', r_i) and add it to $\mathcal{B}$
3: Sample mini-batch $\{s_j, a_j, s_j', r_j\}$
4: Compute $y_j = r_j + \gamma \max_{a_j'} Q_{\phi'}(s_j', a_j')$ using target networks $Q_{\phi'}$ and $\mu_{\theta'}$
5: $\phi \leftarrow \phi - \alpha \Sigma_j \frac{dQ_\phi}{d\phi}(s_j, a_j)(Q_\phi(s_j, a_j) - y_J)$
6: $\theta \leftarrow \theta + \beta \sum_j \frac{da}{d\theta}(s_j) \frac{dQ_\phi}{da}(s_j, a)$
7: update ϕ' and θ'

One option is to use various optimization techniques, as one may have seen in UC Berkeley's EE 127. Specifically, one could use SGD on the action space to produce an optimal a_t by solving an optimization problem. Another simple approach is to stochastically optimize the Q-values by using some samples of the values from some pre-defined distribution (e.g. uniform): $\max_a Q(s, a) \simeq \max\{Q(s, a_1), ..., Q(s, a_N)\}$. One could also improve the accuracy by using some more sophisticated optimization techniques such as Cross-Entropy Method (CEM).

Option no. 2 is to use function classes that are easy to maximize. For example, one could use the Normalized Advantage Functions (NAF) proposed by Gu et al. in [6].

Another rather fancier option is to learn an approximate optimizer, which was originally proposed by Lillicrap et al. in [7]. The idea of Deep Deterministic Policy Gradient (which is actually a Q-learning in disguise) is to train another network $\mu_\theta(s)$ such that $\mu_\theta(s) \simeq \arg\max_a Q_\phi(s, a)$. To train the network, one can see that the optimization of Q-function with respect to θ can be solved by $\theta \leftarrow \arg\max_\theta Q_\phi(s, \mu_\theta(s))$ because by chain rule, $\frac{dQ_\phi}{d\theta} = \frac{da}{d\theta}\frac{dQ_\theta}{da}$. Then the new target becomes:

$$y_j = r_j +_{\phi'} (s_j', \mu_\theta(s_j')) \simeq r_j +_{\phi'} (s_j', \arg\max_{a'} Q_{\phi'}(s_j', a_j'))$$

The sketch of DDPG is in Alg. 14. In step 5, we are updating the Q-function, and in step 6, we are updating the argmax-er. Therefore, DDPG is essentially DQN with argmax-er.

Chapter 7: Policy Gradients Theory

Why does Policy Gradient algorithm work? Recall our generic policy gradient algorithm: we are essentially looping to constantly estimate the advantage function $\hat{A}^{\pi}(s_t, a_t)$ for the current policy π, and then we use this estimate to improve the policy by taking a gradient step on the policy parameter θ, as shown in Alg. 2. This is very similar to the policy iteration algorithm that we discussed in last chapter; the idea of policy iteration is to constantly evaluate the advantage function $A^{\pi}(s, a)$ and update the policy accordingly using the arg max implicit policy. In this chapter, we are going to dive deeper into the policy gradient algorithm, and we will show that the policy gradient algorithm can be reduced to our policy iteration algorithm, which we will prove mathematically.

7.1 Policy Gradient as Policy Iteration

The sum of rewards of RL was defined as $J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}\left[\sum_t \gamma^t r(s_t, a_t)\right]$, which is an expectation taken under the trajectory distribution. We claim that given a new parameter θ', $J(\theta') - J(\theta) = \mathbb{E}_{\tau \sim p_{\theta'}(\tau)}\left[\sum_t \gamma^t A^{\pi_\theta}(s_t, a_t)\right]$, which is an expectation taken under the new policy's trajectory distribution. The difference of the two sums of rewards is the improvement of applying the new policy compared to using the old policy. We claim that this improvement is equal to the expected value of the old policy's advantage function value taken under the new policy's trajectory distribution. The proof is as follows:

$$
\begin{aligned}
J(\theta') - J(\theta) &= J(\theta') - \mathbb{E}_{s_0 \sim p(s_0)}\left[V^{\pi_\theta}(s_0)\right] \\
&= J(\theta') - \mathbb{E}_{\tau \sim p_{\theta'}(\tau)}\left[V^{\pi_\theta}(s_0)\right] \\
&= J(\theta') - \mathbb{E}_{\tau \sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty} \gamma^t V^{\pi_\theta}(s_t) - \sum_{t=1}^{\infty} \gamma^t V^{\pi_\theta}(s_t)\right] \\
&= J(\theta') + \mathbb{E}_{\tau \sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty} \gamma^t \left({}^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t)\right)\right] \\
&= \mathbb{E}_{\tau \sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right] + \mathbb{E}_{\tau \sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty} \gamma^t \left({}^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t)\right)\right] \\
&= \mathbb{E}_{\tau \sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty} \gamma^t \left(r(s_t, a_t) + \gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t)\right)\right] \\
&= \mathbb{E}_{\tau \sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty} \gamma^t A^{\pi_\theta}(s_t, a_t)\right]
\end{aligned}
$$

In the first two steps, we swapped out the initial states distribution in the expectation. This might seem weird at the first sight, but the intuition is that the initial state marginal is the same for any policy. Therefore, the expectation taken under the initial state marginal can be equivalently written as any policy's trajectory distribution, and for simplicity, we choose the policy of interest π', with corresponding parameter θ'.

Now we have proved our claim, but we see the result has a distribution mismatch: the expectation we take is under $\pi_{\theta'}$, but the advantage function A is under π_θ. It would be nice if we could make the two distributions the same. Therefore, we make use of our powerful statistical tool: **importance sampling**:

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

Now the outer expectation is still under θ' state marginal. Can we simply ignore the distribution mismatch and say that it is approximately equal to $\sum_t \mathbb{E}_{s_t\sim p_\theta(s_t)}\left[\mathbb{E}_{a_t\sim\pi_\theta(a_t|s_t)}\left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\gamma^t A^{\pi_\theta}(s_t,a_t)\right]\right]$, which we define as $\bar{A}(\theta')$? We would be all set if the approximation holds, because if so, then $J(\theta')-J(\theta)\simeq\bar{A}(\theta')$, which means we can calculate $\nabla_{\theta'}\bar{A}(\theta')$ without generating new samples and calculating any new advantage functions because the only term that depends on θ' in $\bar{A}(\theta')$ is the policy term in the numerator of importance sampling ratio. Thus, we can just use the current samples from π_θ.

7.2 Distribution Mismatch Bound

As we discussed above, if we could ignore the distribution mismatch, then we would solve a number of problems. So when can we indeed ignore the distribution mismatch? We claim that $p_\theta(s_t)$ is close to $p_{\theta'}(s_t)$ when π_θ is close $\pi_{\theta'}$. This claim sounds rather silly, but in light of this claim, we could quantify the mismatch and bound the distribution change.

7.2.1 A Simple ϵ Bound

First, let us assume that π_θ is deterministic, which means $a_t=\pi_\theta(s_t)$. Then as we have seen in imitation learning, $\pi_{\theta'}$ is close to π_θ if $\pi_{\theta'}(a_t\neq\pi_\theta(s_t)|s_t)\leq\epsilon$. Using the same probability bound we defined in imitation learning, we have the new policy's state marginal defined as:

$$p_{\theta'}(s_t)=(1-\epsilon)^t p_\theta(s_t)+(1-(1-\epsilon)^t)p_{mistake}(s_t)$$

and we can bound the prior distribution mismatch by using:

$$\begin{aligned}|p_{\theta'}(s_t)-p_\theta(s_t)| &= (1-(1-\epsilon)^t)|p_{mistake}(s_t)-p_\theta(s_t)| \\ &\leq 2(1-(1-\epsilon)^t) \\ &\leq 2\epsilon t\end{aligned}$$

This is not a good bound, but it is a bound.

Now let's focus on the more general case, that π_θ is am arbitrary distribution. Then we can try to quantify the notion of "close" by saying π_θ is close to $\pi_{\theta'}$ if:

$$|\pi_{\theta'}(a_t|s_t) - \pi_\theta(a_t|s_t)| \leq \epsilon \; \forall s_t$$

Here is a useful lemma that we will use later: if $|p_X(x) - p_Y(y)| = \epsilon$, then there exists a joint distribution of x, y, which we call $p(x, y)$ such that $p(x) = p_X(x)$ and $p(y) = p_Y(y)$ and $p(x = y) = 1 - \epsilon$. Equivalently, this means that under these circumstances, $p_X(x)$ agrees with $p_Y(y)$ with probability ϵ. If we plug in our π_θ and $\pi_{\theta'}$, then we can show that $\pi_{\theta'}(a_t|s_t)$ takes a different action than $\pi_\theta(a_t|s_t)$ with probability less than ϵ. Using this lemma, we have the same bound as in the deterministic case:

$$\begin{aligned}
|p_{\theta'}(s_t) - p_\theta(s_t)| &= (1 - (1-\epsilon)^t)|p_{mistake}(s_t) - p_\theta(s_t)| \\
&\leq 2(1 - (1-\epsilon)^t) \\
&\leq 2\epsilon t
\end{aligned}$$

Now let us first focus on a more general case where we have a generic function of state $f(s_t)$:

$$\begin{aligned}
\mathbb{E}_{p'_\theta(s_t)}[f(s_t)] &= \sum_{s_t} p_{\theta'}(s_t) f(s_t) \\
&\geq \sum_{s_t} p_\theta(s_t) f(s_t) - |p_\theta(s_t) - p_{\theta'}(s_t)| * \max_{s_t} f(s_t) \\
&\geq \mathbb{E}_{p_\theta(s_t)}[f(s_t)] - 2\epsilon t \max_{s_t} f(s_t)
\end{aligned}$$

Now, putting it all together, let us plug in the term inside the expectation taken under the mismatched distribution:

$$\begin{aligned}
&\sum_t \mathbb{E}_{s_t \sim p_{\theta'}}(s_t) \left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} \left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t) \right] \right] \\
&\qquad \geq \sum_t \mathbb{E}_{s_t \sim p_\theta}(s_t) \left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} \left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t) \right] \right] - \sum_t 2\epsilon C \quad (7.1)
\end{aligned}$$

where the constant term C is a constant depending on the maximum reward, so in the finite horizon case, it should be of $O(Tr_{max})$, and in infinite horizon case, it should be of $O(r_{max}\gamma^t$, which can be simplified by convergence theory to $O(\frac{r_{max}}{1-\gamma})$ Therefore, for small ϵ, we can simply ignore the mismatch.

What have we proved? We have proved that we can update the policy parameter θ' by

$$\theta' \leftarrow \arg\max_{\theta'} \sum_t \mathbb{E}_{s_t \sim p_\theta}(s_t) \left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} \left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t) \right] \right]$$

such that $|\pi_{\theta'}(a_t|s_t) - \pi_\theta(a_t|s_t)| \leq \epsilon$ if ϵ is small.

7.2.2 A More Convenient Bound - KL Divergence

We now use a better, more convenient bound, which is the KL-divergence. We claim that we can apply the aforementioned update if the total variational divergence is bounded by the KL-divergence as follows:

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

where KL-divergence is defined as:

$$D_{KL}(p_1(x)||p_2(x)) = \mathbb{E}_{x\sim p_1(x)}\left[\log \frac{p_1(x)}{p_2(x)}\right]$$

Then the update rule of the policy parameter becomes:

$$\theta' \leftarrow \arg\max_{\theta'} \sum_t \mathbb{E}_{s_t\sim p_\theta}(s_t)\left[\mathbb{E}_{a_t\sim\pi_\theta(a_t|s_t)}\left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\gamma^t A^{\pi_\theta}(s_t,a_t)\right]\right]$$

such that $D_{KL}(\pi_{\theta'}(a_t|s_t)||\pi_\theta(a_t|s_t)) \leq \epsilon$. We have guaranteed improvement if we have small ϵ.

7.2.3 Enforcing the Distribution Mismatch Constraint

Now how do we incorporate the constraint on the distribution mismatch with our objective? One way to do it is to introduce a Lagrangian because we have the following optimization problem:

$$\begin{aligned}\theta' \leftarrow \arg\max_{\theta'} \sum_t \mathbb{E}_{s_t\sim p_\theta}(s_t)\left[\mathbb{E}_{a_t\sim\pi_\theta(a_t|s_t)}\left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\gamma^t A^{\pi_\theta}(s_t,a_t)\right]\right] \\ \text{s.t. } D_{KL}(\pi_{\theta'}(a_t|s_t)||\pi_\theta(a_t|s_t)) \leq \epsilon\end{aligned} \tag{7.2}$$

Then we can have our Lagrangian $\mathcal{L}(\theta', \lambda)$ as

$$\mathcal{L}(\theta',\lambda) = \sum_t \mathbb{E}_{s_t\sim p_\theta}(s_t)\left[\mathbb{E}_{a_t\sim\pi_\theta(a_t|s_t)}\left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\gamma^t A^{\pi_\theta}(s_t,a_t)\right]\right] - \lambda(D_{KL}(\pi_{\theta'}(a_t|s_t)||\pi_\theta(a_t|s_t)) - \epsilon)$$

Then we optimize in terms of the Lagrangian by first maximizing $\mathcal{L}(\theta', \lambda)$ with respect to θ', which we can just do incompletely for a few gradient steps, then we update the dual variable by $\lambda \leftarrow \lambda + \alpha(D_{KL}(\pi_{\theta'}(a_t|s_t)||\pi_\theta(a_t|s_t)) - \epsilon)$. This technique is an instance of dual gradient descent, and we will talk about it more in depth in a later chapter. Essentially, the intuition is that we raise λ if the constraint is violated too much, and else lower it.

7.2.4 Other Optimization Techniques

There are also some other ways to optimize based on the distribution mismatch bound. One way to do it is by using 1st order Taylor expansion. Since $\theta' \leftarrow \arg\max_{\theta'} \bar{A}(\theta')$, we can apply first order Taylor expansion by

$$\begin{aligned}\theta' \leftarrow \arg\max_{\theta'} \nabla_\theta \bar{A}(\theta)(\theta' - \theta) \\ \text{s.t. } D_{KL}(\pi_{\theta'}(a_t|s_t)||\pi_\theta(a_t|s_t)) \leq \epsilon\end{aligned} \tag{7.3}$$

From what we have learned in policy gradients, we can derive the gradient of $\bar{A}$ as follows:

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

and if we have $\pi_\theta \simeq \pi_{\theta'}$, then we can effectively cancel out the importance ratio:

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

just like normal policy gradient.

Consequently, our original optimization problem can be equivalently written as:

$$\begin{aligned} \theta' &\leftarrow \arg\max_{\theta'} \nabla_\theta J(\theta)^T (\theta' - \theta) \\ \text{s.t. } & D_{KL}(\pi_{\theta'}(a_t|s_t)||\pi_\theta(a_t|s_t)) \leq \epsilon \end{aligned} \tag{7.4}$$

We now have the RL objective in our optimization, then can we just use gradient ascent just like what we did in policy gradient? Well, it turns out gradient ascent is enforcing some other constraint:

$$\begin{aligned} \theta' &\leftarrow \arg\max_{\theta'} \nabla_\theta J(\theta)^T (\theta' - \theta) \\ & \text{s.t. } ||\theta' - \theta||^2 \leq \epsilon \end{aligned} \tag{7.5}$$

whose update rule can be written as $\theta' \leftarrow \theta + \sqrt{\frac{\epsilon}{||\nabla_\theta J(\theta)||^2}} \nabla_\theta J(\theta)$, and the square root term is just our learning rate, which depends on ϵ.

Since the two optimization problems are not the same, we will tweak the KL-divergence constraint a little bit using Taylor expansion one more time. If the two policies are very similar to each other, one could approximate KL-divergence by $D_{KL}(\pi_(\theta')||\pi_\theta) \simeq \frac{1}{2}(\theta' - \theta)^T F (\theta' - \theta)$, where F is called the Fisher-information matrix, and it is defined as $\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s)^T\right]$. This matrix F can be estimated using samples, and it gives us a convenient quadratic bound. Using a technique similar to Newton-Raphson, we can update the parameter by $\theta' \leftarrow \theta + \alpha F^{-1} \nabla_\theta J(\theta)$, and the learning rate α can be chosen as $\alpha = \sqrt{\frac{2\epsilon}{\nabla_\theta J(\theta)^T F \nabla_\theta}}$. Now our update rule is a lot more similar to gradient descent, except that in gradient descent, the l2 norm constrains the update step into a circle, while in our 2nd order approximation of KL-divergence, it constrains the update step into a ellipse.

Chapter 8: Model-Based Reinforcement Learning

What we have covered so far can be categorized as "model-free" reinforcement learning. The reason why it is called model-free is that the transition probabilities are unknown and we did not even attempt to learn the transition probabilities. Recall the RL objective:

$$\begin{aligned}\pi_\theta(\tau) &= p_\theta(s_1, a_1, ..., s_T, a_t) \\ &= p(s_1)\prod_{t=1}^{T}\pi_\theta(a_t|s_t)p(s_{t+1}|s_t, a_t) \\ \theta^* &= \arg\max_\theta \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_t r(s_t, a_t)\right]\end{aligned}$$

The transition probabilities $p(s_{t+1}|s_t, a_t)$ is not known in all the model-free RL algorithms that we have learned such as Q-learning and policy gradients. But what if we know the transition dynamics? Recall that at the very beginning of the notes we drew an analogy of RL and control theory; in many cases, we do know the system's internal transition. For example, in games, easily modeled systems, and simulated environments, the transitions are given to us. Moreover, it is not uncommon to learn the transition models: in classic robotics, system identification fits unknown parameters of a known model to learn how the system evolves, and one could also imagine a deep learning approach where we could potentially fit a general-purpose model to observed transition data for later use. In fact, the latter case is the essence of Model-based RL, where we learn the transition dynamics first, and then figure out how to choose actions. To learn about model-based RL, we shall start from a simpler case, where we know the transitions and determine how we control the system optimally based on the transitions. After this, we can apply our optimal control theory to the more general case, where we actually learn the transitions first.

8.1 Optimal Control

Optimal control is a task that we come across when we are well aware of the transition probabilities and we try to learn how to control the system optimally. In optimal control, there are two different categories of controller design: the first one is **open-loop** control, where we do not have any state feedbacks, and we roll out a sequence of actions based on the current state that we observe. The second one is called **closed-loop** control, where we determine the action at each time step based on the current state, and how we determine the action to apply is based on state feedbacks. In some cases, our transition functions are deterministic, while in others, the transition functions are stochastic.

In a open loop controller, if we have a deterministic transition in our system such that $a_{t+1} = f(s_t, a_t)$, then our action sequence should be determined by choosing those that can

return the maximum rewards:

$$a_1, ..., a_T = \arg\max_{a1...a_T} \sum_{t=1}^{T} r(s_t, a_t)$$
$$\text{s.t. } a_{t+1} = f(s_t, a_t)$$

In stochastic scenarios, the transition function is a probabilistic distribution, where we have $p(s_{t+1}|s_t, a_t)$, and the action sequence should be chosen based on expectation of the rewards:

$$p_\theta(s_1, ..., s_T) = p(s_1) \prod_{t=1}^{T} p(s_{t+1}|s_t, a_t)$$
$$a_1, ..., a_T = \arg\max_{a_1,...,a_T} \mathbb{E}\left[\sum_t r(s_t, a_t)|a_1, ..., a_T\right]$$

Note that we roll out all actions to apply only based on the initial state marginal, so we do not consider any state-feedback in this case.

In a closed-loop controller, however, we keep interacting with the world, so we need a policy function that can tell us the action to apply if we input the current state: $a_t \sim \pi(a_t|s_t)$, which we call a state-feedback. We choose our policy function as follows:

$$p(s_1, a_1, ..., s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi(a_t|s_t) p(s_{t+1}|s_t, a_t)$$
$$\pi = \arg\max_{\pi} \mathbb{E}_{\tau\sim p(\tau)}\left[\sum_t r(s_t, a_t)\right]$$

Generally, π could take many forms, such as a neural net or time-variant linear controller $K_t s_t + k_t$.

8.2 Open-loop Planning

For now, let us focus on a simple, open-loop controller, and see how we choose actions using such controller. In open-loop scenarios, we roll out a sequence of actions by doing arg max on the sum of rewards: $a_1, ..., a_T = \arg\max_{a_1,...,a_T} J(a_1, ..., a_T)$ and compactly, we can say that $A = \arg\max_A J(A)$.

8.2.1 Random Shooting

Perhaps the easiest and most intuitive stochastic optimization method in open-loop control is the random shooting method. In such method, we first sample some different action sequences $A_1, ..., A_N$ from some known distribution (such as uniform), and then we choose A_i based on $\arg\max_i J(A_i)$. This is highly efficient in that we are not improving what we sample, so we might get stuck in some mediocre action sequence. Therefore, we can keep improving the samples we choose from a Gaussian distribution based on some elites sequences. This is the basic idea of Cross Entropy Method (CEM).

Algorithm 15 Cross Entropy Method with Continuous-valued Input

Require: Some base distribution for action sequence $p(A)$

while true **do**
  Sample $A_1, ..., A_N$ from $p(A)$
  Evaluate $J(A_1), ..., J(A_N)$
  Pick elites $A_{i_1}, ..., A_{i_M}$ with the highest value, where $M < N$
  Refit $p(A)$ to elites $A_{i_1}, ..., A_{i_M}$

Algorithm 16 Generic Monte Carlo Tree Search (MCTS)

Require: Some base tree policy for expanding nodes, some base default policy to simulate a trajectory from a leaf

while true **do**
  Find a leaf s_l using TreePolicy(s_1)
  Evaluate the leaf using DefaultPolicy(s_l)
  Update all values in tree between s_1 and s_l
Take best action from s_1

8.2.2 Cross Entropy Method

CEM improves upon the random shooting's guess and check scheme by choosing some elites sequences which give us higher rewards and refit the distribution to the high rewards. Intuitively, we are getting closer to higher rewards as we refit the distribution. Here is a sketch of CEM, as shown in Alg. 15. This algorithm is extremely simple and very fast if parallelized. However, it suffers from very harsh dimensionality limit and it only works for open-loop scenarios.

8.2.3 Monte Carlo Tree Search

Now imagine our action space is discrete, we can apply a stochastic optimization technique called Monte Carlo Tree Search (MCTS), which is very popular in planning in stochastic games. The gist of this method is that in discrete action space, we are essentially expanding out a tree. However, the tree might be too big to expand out due to computational cost. Therefore, one way to save the computational cost is to partially expand the tree and use a policy to simulate a trajectory from the last expanded node. A generic sketch of MCTS is shown in Alg. 16. Note that we the tree policy is not an actual policy, because it is just a method to traverse through our tree in order to select a node to expand. The default policy is an actual policy that is able to simulate the system. Since simulations are involved here, we have to be able to roll back to the original state.

8.2.4 UCT Tree Policy

In MCTS, how do we choose the nodes to expand? Intuitively, we need to keep choosing the nodes with high rewards so far, and simultaneously pick the ones that have not been chosen in order to explore. Therefore, one way to do it is to use the UCT tree policy. In this policy, we gauge the performance of each node by assigning a score function where $\text{Score}(s_t) = \frac{Q(s_t)}{N(s_t)} + 2C\sqrt{\frac{2\ln N(s_{t-1})}{N(s_t)}}$. If the current node s_t is not fully expanded, meaning

that there is action that we never took before, then we choose new a_t; else, we choose the child with best Score(s_{t+1}).

More details about MCTS can be found in [8] and [9].

8.2.5 Using Derivatives

Let us consider the control theory counterpart of the RL objective. Essentially, we have a constrained optimization problem defined as follows:

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

if we plug in the transition constraint, we have:

$$\min_{u_1,...,u_T} c(x_1, u_1) + c(f(x_1, u_1), u_2) + ... + c(f(f(...)...), u_T)$$

which becomes an unconstrained optimization problem. Since it is unconstrained now, one might ask, can we do gradient descent on it? The usual answer is yes, but only if we use some more powerful optimization technique such as 2nd-order Newton method. Because optimization problems such as shooting methods are hard and often ill-conditioned via 1st order gradient descent.

8.2.6 Shooting Methods and Collocation Methods

There are two different classes of gradient descent based method: shooting method and collocation method. In shooting methods, we optimize only on action sequences, so the actions are the only optimization variables. We only optimize upon the actions and apply the action to the state and see where it shoots to, so the states are the consequences of the actions that we optimize. The optimization problem can be written as:

$$\min_{u_1,...,u_T} c(x_1, u_1) + c(f(x_1, u_1), u_2) + ... + c(f(f(...)...), u_T)$$

However, in collocation method, we optimize upon both actions and states, with constraints, and the optimization problem is written as:

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

8.2.7 Linear Quadratic Regulator (LQR)

Let us start with a simple case of shooting method, where we apply a 2nd order-style optimization technique to achieve optimal control. The simple case assumes that we have a linear system, where the transition function is affine, and we have a quadratic cost function.

Thus, the transition function should be of the form:

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

and the cost function should be of the form:

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

What we are doing right now is to solve for a closed-form solution for an optimal LQR controller. The idea is to use backward recursion. Since we are doing shooting method, we have

$$\min_{u_1,...,u_T} c(x_1, u_1) + c(f(x_1, u_1), u_2) + ... + c(f(f(...)...), u_T)$$

and the last item is the only term that depends on u_T. Therefore, as a base case, we can try to solve for u_T first. In order to simplify our computation, let us define some blocks in the matrices that we defined above. Specifically, let us assume that

$$C_T = \begin{bmatrix} C_{x_T,x_T} & C_{x_T,u_T} \\ C_{u_t,x_T} & C_{u_T,u_T} \end{bmatrix}$$

and

$$c_T = \begin{bmatrix} c_{x_T} \\ c_{u_T} \end{bmatrix}$$

Since our cost function is

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

by setting gradient to 0, we will have

$$\nabla_{u_T} Q(x_T, u_T) = C_{u_t,x_T} x_T + C_{u_T,u_T} u_T + c_{u_T}^T = 0$$

By solving this equation, we have solved u_T, in terms of known constants and x_T:

$$u_T = -C_{u_T,u_T}^{-1}(C_{u_t,x_T} x_T + c_{u_T})$$

and to make notations more compact, let us denote u_T as $u_T = K_T x_T + k_T$, and $K_T = -C_{u_T,u_T}^{-1} C_{u_t,x_T}$, $k_T = -C_{u_T,u_T}^{-1} c_{u_T}$.

Now having solved our terminal control input u_T, which is fully determined by our terminal state x_T, we can eliminate u_T in $Q(x_T, u_T)$. Plugging in, we have

$$\begin{aligned} V(x_T) &= \text{const} + \frac{1}{2}\begin{bmatrix} x_T \\ K_T x_T + k_T \end{bmatrix}^T C_T \begin{bmatrix} x_T \\ K_T x_T + k_T \end{bmatrix} + \begin{bmatrix} x_T \\ K_T x_T + k_T \end{bmatrix}^T c_T \\ &= \frac{1}{2} x_T^T C_{x_T,x_T} x_T + \frac{1}{2} x_T^T C_{x_T,u_T} K_T x_T + \frac{1}{2} x_T^T K_T^T C_{u_T,x_T} x_T + \frac{1}{2} x_T^T K_T^T C_{u_T,u_T} K_T x_T \\ &\quad + x_T^T K_T^T C_{u_T,u_T} k_T + \frac{1}{2} x_T^T C_{x_T,u_T} k_T + x_T^T c_{x_T} + x_T^T K_T^T c_{u_T} + \text{const} \\ &= \text{const} + \frac{1}{2} x_T^T V_T x_T + x_T^T v_T \end{aligned}$$

where we define v_T and V_T to make the notation more compact as follows:

$$V_T = C_{x_T,x_T} + C_{x_T,u_T}K_T + K_T^T C_{u_T,x_T} + K_T^T C_{u_T,u_T}K_T^T$$
$$v_T = c_{x_T} + C_{x_T,u_T}k_T + K_T^T c_{u_T} + K_T^T C_{u_T,u_T}k_T$$

Having solved the base case, we solve solve for other optimal control inputs backwards. Let us first proceed to solve for u_{T-1} in terms of x_{T-1}. Now note that u_{T-1} not only affects state x_{T-1}, but it also affects x_T because of the system dynamics:

$$f(x_{T-1}, u_{T-1}) = x_T = F_{T-1}\begin{bmatrix}x_{T-1}\\u_{T-1}\end{bmatrix} + f_{T-1}$$

Therefore, the cost function from $T-1$ can be calculated as:

$$Q(x_{T-1}, u_{T-1}) = \text{const} + \frac{1}{2}\begin{bmatrix}x_{T-1}\\u_{T-1}\end{bmatrix}^T C_{T-1}\begin{bmatrix}x_{T-1}\\u_{T-1}\end{bmatrix} + \begin{bmatrix}x_{T-1}\\u_{T-1}\end{bmatrix}^T c_{T-1} + V(f(x_{T-1}, u_{T-1}))$$

and if we plug the transition dynamics function into $V(x_T)$, we will have:

$$V(x_T) = \text{const} + \frac{1}{2}\begin{bmatrix}x_{T-1}\\u_{T-1}\end{bmatrix}^T F_{T-1}^T V_T F_{T-1}\begin{bmatrix}x_{T-1}\\u_{T-1}\end{bmatrix} + \begin{bmatrix}x_{T-1}\\u_{T-1}\end{bmatrix}^T F_{T-1}^T V_T f_{T-1} + \begin{bmatrix}x_{T-1}\\u_{T-1}\end{bmatrix}^T F_{T-1}^T v_T$$

More compactly, we write the cost function as:

$$Q(x_{T-1}, u_{T-1}) = \text{const} + \frac{1}{2}\begin{bmatrix}x_{T-1}\\u_{T-1}\end{bmatrix} Q_{T-1}\begin{bmatrix}x_{T-1}\\u_{T-1}\end{bmatrix} + \begin{bmatrix}x_{T-1}\\u_{T-1}\end{bmatrix}^T q_{T-1}$$

where $Q_{T-1} = C_{T-1} + F_{T-1}^T V_T F_{T-1}$, and $q_{T-1} = c_{T-1} + F_{T-1}^T V_T f_{T-1} + F_{T-1}^T v_T$. To solve the optimization problem, we set the gradient to 0:

$$\nabla_{u_{T-1}} Q(x_{T-1}, u_{T-1}) Q_{u_{T-1},x_{T-1}} x_{T-1} + Q_{u_{T-1},u_{T-1}} u_{T-1} + q_{u_{T-1}}^T = 0$$

solving the equation, we have the following expression for u_{T-1}:

$$u_{T-1} = K_{T-1}x_{T-1} + k_{T-1}$$
$$K_{T-1} = -Q_{u_{T-1},u_{T-1}}^{-1} Q_{u_{T-1},x_{T-1}}$$
$$k_{T-1} = -Q_{u_{T-1},u_{T-1}} q_{u_{T-1}}$$

Applying the same technique backwards, we can solve for the states and inputs at each time step, as illustrated in Alg. 17. In step 5 of Alg. 17, Q-function represents the total cost from now until end if we take u_t from state x_t, and in step 11, the V-function represents the total cost from now until end from state x_t, so $V(x_t) = \min_{u_t} Q_{x_t,u_t}$, which we call the cost-to-go function.

What we have analyzed above is based on deterministic dynamics. What if the transition (dynamics) is stochastic? Specifically, consider the following setup:

$$x_{t+1} \sim p(x_{t+1}|x_t, u_t)$$
$$p(x_{t+1}|x_t, u_t) = \mathcal{N}\left(F_t\begin{bmatrix}x_t\\u_t\end{bmatrix} + f_t, \Sigma_t\right)$$

where our transition is actually a Gaussian distribution with constant covariance. It turns out that we can apply the exact same algorithm, choosing actions according to $u_t = K_t x_t + k_t$, and we can ignore Σ_t due to symmetry of Gaussians.

Algorithm 17 Solving for Linear Quadratic Regulator (LQR)

Backward Recursion
for $t = T$ to 1 **do**
  $Q_t = C_t + F_t^T V_{t+1} F_t$
  $q_t = c_t + F_t^T V_T f_t + F_t^T v_{t+1}$
  $Q(x_t, u_t) = \text{const} + \frac{1}{2} \begin{bmatrix} x_t \\ u_t \end{bmatrix} Q_t \begin{bmatrix} x_t \\ u_t \end{bmatrix} + \begin{bmatrix} x_t \\ u_t \end{bmatrix}^T q_t$
  $u_t \leftarrow \arg\min_{u_t} Q(x_t, u_t) = K_t x_t + k_t$
  $K_t = -Q_{u_t,u_t}^{-1} Q_{u_t,x_t}$
  $k_t = -Q_{u_t,u_t} q_{u_t}$
  $V_t = Q_{x_t,x_t} + Q_{x_t,u_t} K_t + K_t^T Q_{u_t,x_t} + K_t^T Q_{u_t,u_t} K_t$
  $v_t = q_{x_t} + Q_{x_t,u_t} k_t + K_t^T Q_{u_t} + K_t^T Q_{u_t,u_t} k_t$
  $V(x_t) = \text{const} + \frac{1}{2} x_t^T V_t x_t + x_t^T v_t$
Forward Recursion
for $t = 1$ to T **do**
  $u_t = K_t x_t + k_t$
  $x_{t+1} = f(x_t, u_t)$

8.2.8 Iterative LQR (iLQR)

In LQR, we assumed that the dynamics are linear. In non-linear cases, however, we can apply a similar approach called iterative LQR. Specifically, we can iteratively apply Jacobian linearization to locally linearize the system with respect to an equilibrium point. Consequently, we approximate a non-linear system as a linear-quadratic system:

$$f(x_t, u_t) \simeq f(\hat{x_t}, \hat{u_t}) + \nabla_{x_t,u_t} f(\hat{x_t}, \hat{u_t}) \begin{bmatrix} x_t - \hat{x_t} \\ u_t - \hat{u_t} \end{bmatrix}$$

$$c(x_t, u_t) \simeq c(\hat{x_t}, \hat{u_t}) + \nabla_{x_t,u_t} c(\hat{x_t}, \hat{u_t}) \begin{bmatrix} x_t - \hat{x_t} \\ u_t - \hat{u_t} \end{bmatrix} + \frac{1}{2} \begin{bmatrix} x_t - \hat{x_t} \\ u_t - \hat{u_t} \end{bmatrix}^T \nabla^2_{x_t,u_t} c(\hat{x_t}, \hat{u_t}) \begin{bmatrix} x_t - \hat{x_t} \\ u_t - \hat{u_t} \end{bmatrix}$$

Now we have an LQR system with respect to the divergence from the action space and state space's equilibrium points:

$$\bar{f}(\delta x_t, \delta u_t) = F_t \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix}$$

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

where

$$F_t = \nabla_{x_t,u_t} f(\delta x_t, \delta u_t)$$
$$C_t = \nabla^2_{x_t,u_t} c(\delta x_t, \delta u_t)$$
$$c_t = \nabla_{x_t,u_t} c(\delta x_t, \delta u_t)$$

Then we can iteratively run LQR with dynamics $\bar{f}$, cost $\bar{c}$, state δx_t, and action δu_t.

A sketch of iLQR is shown in Alg. 18. In essence, iLQR is an approximation of Newton's method for solving $\min_{u_1,\ldots,u_T} c(x_1, u_1) + c(f(x_1, u_1), u_2) + \ldots + c(f(f(\ldots)\ldots), u_T)$.

Algorithm 18 Iterative LQR (iLQR)

while until convergence **do**
  $F_t = \nabla_{x_t,u_t} f(\delta x_t, \delta u_t)$
  $C_t = \nabla^2_{x_t,u_t} c(\delta x_t, \delta u_t)$
  $c_t = \nabla_{x_t,u_t} c(\delta x_t, \delta u_t)$
  Run LQR backward recursion on state $\delta x_t = x_t - \hat{x_t}$ and action $\delta u_t = u_t - \hat{u_t}$
  Run forward pass with real nonlinear dynamics and $u_t = K_t(x_t - \hat{x_t}) + k_t + \hat{u_t}$
  Update $\hat{x_t}$ and $\hat{u_t}$ based on states and actions in forward pass

Algorithm 19 Model-based Reinforcement Learning Version 0.5

Require: Some base policy for data collection π_0
Run base policy $\pi(a_t|s_t)$ (e.g. random policy) to collect $\mathcal{D} = \{(s, a, s')_i\}$
Learn dynamics model $f(s, a)$ to minimize $\sum_i ||f(s_i, a_i) - s'_i||^2$
Plan through $f(s, a)$ to choose actions

8.3 Model-based RL

In this section, we are going to cover a rather simpler case of model-based RL. Specifically, we are going to talk about a technique to learn a model of the system first, and then use the optimal control technique we covered last time to improve the model. Furthermore, we will learn to address uncertainty in the model such as model mismatch and imperfection.

8.3.1 Basics

Why do we learn the model? Because when the model is unknown, we can learn the model so that we know $f(s_t, a_t) = s_{t+1}$ or $p(s_{t+1}|s_t, a|t)$ in stochastic case, we could use the tools from optimal control to maximize our rewards.

Our first attempt is naive, we learn $f(s_t, a_t)$ from data, and then plan through it. We call this approach model-based RL version 0.5, or vanilla model-based RL, as shown in algorithm 19. This is essentially what people do in system identification, which is a technique used in classic robotics, and it is effective when we can hand-engineer a dynamics representation using our knowledge of physics, and fit just a few parameters. However, it does not work generally because of distribution mismatch: when the model is imperfect, we might suffer from false learning. Furthermore, since we are blindly following a trajectory, the mismatch exacerbates as we use more expressive model classes, when $p_{\pi_0}(s_t) \neq p_{\pi_f}(s_t)$.

Acknowledging this disadvantage, we could improve the vanilla model-based RL by making $p_{\pi_0}(s_t) = p_{\pi_f}(s_t)$. As we have seen in Alg. 1, we can keep aggregating data into our dataset in order to make our model converge to demonstration model. Applying the same approach, we keep updating the dataset by running the current model, and then update the model accordingly. Take a look at the updated model-based RL algorithm in Alg. 20. Version 1.0 addresses the model mismatch issue and drives the current model as close as possible to the true dynamics model. However, we are still blindly following a trajectory in step 5 of Alg. 20, and if we made a mistake, we would follow the wrong step which makes the mistake exacerbate. Therefore, we need to somehow adjust our plan as time goes on. One way to do this is to borrow some ideas from modern control theory: Model Predictive

Algorithm 20 Model-based Reinforcement Learning Version 1.0

Require: Some base policy for data collection π_0

1: Run base policy $\pi(a_t|s_t)$ (e.g. random policy) to collect $\mathcal{D} = \{(s, a, s')_i\}$
2: **while** True **do**
3: Learn dynamics model $f(s, a)$ to minimize $\sum_i ||f(s_i, a_i) - s'_i||^2$
4: Plan through $f(s, a)$ to choose actions
5: Execute those actions and add the resulting data $\{(s, a, s')_j\}$ to $\mathcal{D}$

Algorithm 21 Model-based Reinforcement Learning Version 1.5

Require: Some base policy for data collection π_0, hyperparameter N

1: Run base policy $\pi(a_t|s_t)$ (e.g. random policy) to collect $\mathcal{D} = \{(s, a, s')_i\}$
2: **for** every N steps **do**
3: **while** True **do**
4: Learn dynamics model $f(s, a)$ to minimize $\sum_i ||f(s_i, a_i) - s'_i||^2$
5: Plan through $f(s, a)$ to choose actions
6: Execute the first planned action, observe resulting state s' (MPC)
7: Append (s, a, s') to dataset $\mathcal{D}$

Control (MPC).

In MPC, we are given the system's dynamics model, and we are trying to design an adaptive controller by solving a finite time constrained optimal control problem at each time step, and take only the first action in the generated sequence of actions. Then we replan based on the new state. For sake of simplicity, we will skip the discussion about safe set and terminal set in MPC in this chapter. But the "replan" idea in MPC is exactly what we need to improve our model-based RL version 1.0. We essentially are aiming to take one action in the planned sequence and only observe one new state, and then append the observed transition to our dataset $\mathcal{D}$. The improvement is shown in Alg. 21. The while loop in algorithm 21 refers to replanning in MPC, which is solving for an optimization problem at each time step after we take the first action planned. The for loop, however, means that we are periodically retraining the model in order to make it closer to the true underlying transition model. Intuitively, the more frequently the agent replans, the less perfect each individual plan needs to be, because since we are frequently replanning, we are able to correct our mistakes made in previous plans more easily. Consequently, one is able to correct the plans as one increase the replanning frequency. Therefore, if we are frequently replanning, we could use shorter horizons in the CFTOC problem that MPC is solving.

8.3.2 Performance Gaps in Model-based RL

Believe it or not, sometimes model-based RL performs worse than model-free RL. The problem is from step 5 of algorithm 21. In this step, we plan through the model to choose actions, which means we are solving an optimization problem based on the data we collect. One could imagine that if we overfit the data, the agent might have some wrong belief about the model, thus generating wrong actions. Pictorially, this phenomenon is illustrated in Fig. 8.1.

Therefore, we need to explore to get better, more representative data of the model, thus preventing overfitting and false belief. The expected value of the reward is not the same as

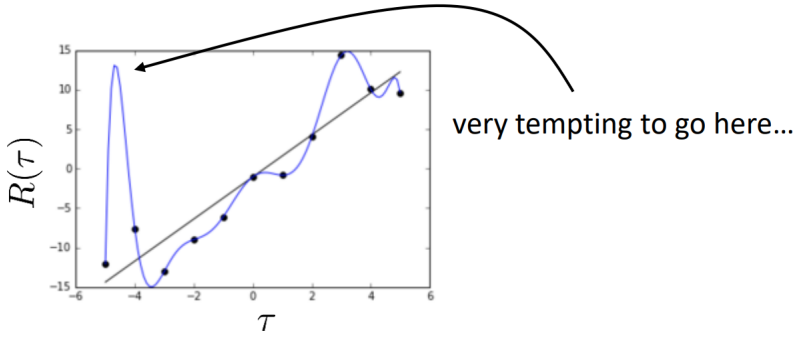

Figure 8.1: False belief about the model from overfitting

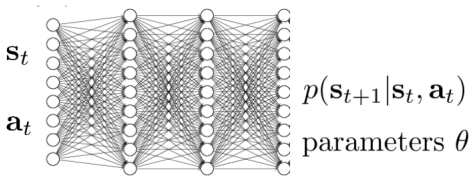

Figure 8.2: Estimating the model using a neural net

optimistic or pessimistic. In step 5, when we choose actions, we only take actions for which we think we will get high reward in expectation, with respect to uncertain dynamics, which avoids exploiting the model too much.

8.3.3 Uncertainty-aware Models

Under imperfect models and model mismatch, one might expect wrong actions planned. Therefore, one way to deal with this problem is to construct an uncertainty-aware model, where we can quantitatively estimate the uncertainty in the model, so that we can assess the accuracy of the model and the planned actions.

The first idea is to use entropy of output distribution, and as we know, higher entropy means higher uncertainty. We can estimate the entropy of $p(s_{t+1}|s_t, a_t)$. However, this is not enough because when the model is wrong, we might still have low variance, thus low entropy. Even though in some regions the model is highly uncertain, the output entropy is still low.

The reason why entropy of the output distribution alone is not expressive enough is that there are two types of uncertainty:

- aleatoric (statistical) uncertainty, where the data itself is noisy.
- epistemic (model) uncertainty, where the model is certain about data, but we are not certain about model.

These two types of uncertainty are not the same. We cannot gauge the correctness of the second model based on the output entropy, and the entropy of the first model might be higher even though it is potentially a very "good" model.

The second idea is to estimate the model uncertainty, where we essentially estimate how uncertainty we are about the model. Usually, we use maximum likelihood estimation, where

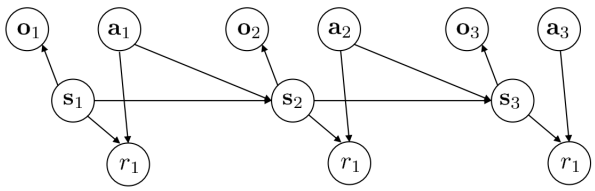

Figure 8.3: Latent space model

$$\arg\max_{\theta} \log p(\theta|\mathcal{D}) = \arg\max_{\theta} \log p(\mathcal{D}|\theta)$$

Instead if we estimate the posterior of data $p(\theta|\mathcal{D})$ instead of argmax, the entropy of the distribution gives us the model uncertainty from the data. Moreover, we can predict using $\int p(s_{t+1}|s_t, a_t, \theta)p(\theta|\mathcal{D})d\theta$.

To learn the posterior distribution, we can apply bootstrap ensembles, where we use multiple networks to learn the same distribution. Formally, say we have N networks, each with a parameter θ_i to learn $p(s_{t+1}|s_t, a_t)$, we can then estimate the posterior by:

$$p(\theta|\mathcal{D}) \simeq \frac{1}{N}\sum_i \delta(\theta_i)$$

where $\delta(\cdot)$ is the direc-delta function. To train it, we need to generate independent datasets to get independent models. One way to do this is to train θ_i on $\mathcal{D}_i$ sampled with replacement from $\mathcal{D}$. This method is simple, but it is a very crude approximation.

With this ensemble of networks, we choose actions a little differently. Before, we choose actions by $J(a_1, \ldots, a_H) = \sum_{t=1}^{H} r(s_t, a_t)$, where $s_{t+1} = f(s_t, a_t)$, and now we average over the ensemble by $J(a_1, \ldots, a_H) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{H} r(s_{t,i}, a_{t,i})$, where $s_{t+1,i} = f(s_{t,i}, a_{t,i})$

In general, for candidate action sequence $a_1, \ldots, a_H$, we first sample $\theta \sim p(\theta|\mathcal{D})$, then at each time step t, we sample $s_{t+1} \sim p(s_{t+1}|s_t, a_t, \theta)$, then we calculate the reward $R = \sum_t r(s_t, a_t)$, and we repeat the aforementioned steps and accumulate the average reward.

8.3.4 Latent Space Model

In many cases, we are given very complex observations of the states such as pixel-based images, where we do not have full access to the states. To learn the dynamics using observations, we need to learn from the latent space and infer the states from observations. From Fig. 8.3, we can see that we need to learn the following models:

- $p(o_t|s_t)$, the observation model
- $p(s_{t+1}|s_t, a_t)$, the dynamics model
- $p(r_t|s_t, a_t)$, the reward model

Recall that in high level, model-based RL algorithms are basically doing a maximum likelihood estimation in training given fully observed states:

$$\max_{\phi} \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T} \log p_\phi(s_{t+1,i}|s_{t,i}, a_{t,i})$$

Algorithm 22 Model-based Reinforcement Learning with Latent States

Require: Some base policy for data collection π_0, hyperparameter N

1: Run base policy $\pi(a_t|s_t)$ (e.g. random policy) to collect $\mathcal{D} = \{(s, a, s')_i\}$
2: **for** every N steps **do**
3: **while** True **do**
4: Learn dynamics model $p_\phi(s_{t+1}|s_t, a_t), p_\phi(r_t|s_t), p(o_t|s_t), g_\psi(o_t)$
5: Plan through $f(s, a)$ to choose actions
6: Execute the first planned action, observe resulting state o' (MPC)
7: Append (o, a, o') to dataset $\mathcal{D}$

then with latent models, we are not sure about the actual state, so we take the expected value:

$$\max_\phi \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \mathbb{E} \left[\log p_\phi(s_{t+1,i}|s_{t,i}, a_{t,i}) + \log p_\phi(o_{t,i}|s_{t,i}) \right]$$

where the expectation is with respect to the distribution of $(s_t, s_{t+1}) \sim p(s_t, s_{t+1}|o_{1:T}, a_{1:T})$

However, the posterior distribution $p(s_t, s_{t+1}|o_{1:T}, a_{1:T})$ is usually intractable if we have very complex dynamics. As a result, we could instead try to learn an approximate posterior, which we call $q_\psi(s_t|o_{1:t}, a_{1;t})$. We could also learn $q_\psi(s_t, s_{t+1}|o_{1:t}, a_{1;t})$ and $q_\psi(s_t|o_t)$. We call this technique learning an **encoder**. Learning the distribution $q_\psi(s_t|o_t)$ is crude, but it is the simplest to implement. If we just decide to learn this distribution for now, then the expectation becomes:

$$\max_\phi \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \mathbb{E} \left[\log p_\phi(s_{t+1,i}|s_{t,i}, a_{t,i}) + \log p_\phi(o_{t,i}|s_{t,i}) \right]$$

such that the expectation is with respect to $s_t \sim q_\psi(s_t|o_t)$, $s_{t+1} \sim q_\psi(s_{t+1}|o_{t+1})$

For now, let us focus on a simple case where $q(s_t|o_t)$ is deterministic, because the stochastic requires variational inference, which will be covered in-depth in a later chapter. In deterministic case, we are training a neural net $g_\psi(o_t) = s_t$ using a direc-delta function such that $q_\psi(s_t|o_t) = \delta(s_t = g_\psi(o_t))$. Then the expectation can be simplified as

$$\max_\phi \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \log p_\phi(g_\psi(o_{t+1,i})|g_\psi(o_{t,i}), a_{t,i}) + \log p_\phi(o_{t,i}|g_\psi(o_{t,i}))$$

Now everything is differentiable, we can train using backpropagation.

Thus, we can slightly modify Alg. 21 so that we can deal with observations and latent space. We show the sketch of this slightly modified algorithm in Alg. 22. In step 4, we are respectively learning the dynamics, reward model, observation model, and encoder.

Interested readers can refer to [10] and [11] for more information on learning from pixel-based images as latent states.

Chapter 9: Model-based Policy Learning

So far we have covered the basics of model-based RL that we first learn a model and use a model for control. We have seen that this approach does not work well in general because of the effect of distributional shift in model-based RL. We have also seen the method to quantify uncertainty in our model in order to alleviate this issue. The methods we covered so far do not involve learning policies. In this chapter, we will cover model-based reinforcement learning of policies. Specifically, we will learn global policies and local policies, and combine local policies into global policies using guided policy search and policy distillation. We shall understand how and why we should use models to learn policies, global and local policy learning, and how local policies can be merged via supervised learning into a global policy.

We have seen the difference between a closed-loop and open-loop controller. We also discussed why an open-loop controller is suboptimal because we are rolling out a whole sequence of actions solely based on one state observation. Therefore, it would be more ideal if we could design a closed-loop controller where state feedbacks can help us correct the mistakes we make. Recall in a stochastic environment, we are optimizing over the policy as follows:

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

$$\pi = \arg\max_{\pi} \mathbb{E}_{\tau \sim p(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

and π could take several forms: π can be a neural net, which we call a **global** policy, and it can also be a time-varying linear controller $K_t s_t + k_t$ as we saw in LQR, which we call a **local** policy.

9.1 Back-propagate into the Policy

Let us start with a simple solution for model-based policy learning. Ideally, we could build a computational graph in Tensorflow, and calculate the partial derivatives step by step so that we can backpropage into policy and optimize the policy, illustrated in Fig. 9.1. Then we can modify our model-based policy-free RL algorithm to accomodate this new policy learning process in Alg. 23.

9.1.1 Vanishing and Exploding Gradients

One problem with Alg. 23, or general gradient-based optimization is that as we progress into the time steps, we might encounter vanishing or exploding gradients. Because as we apply chain rule, the gradients get multiplied by each other, so we the product may get extremely

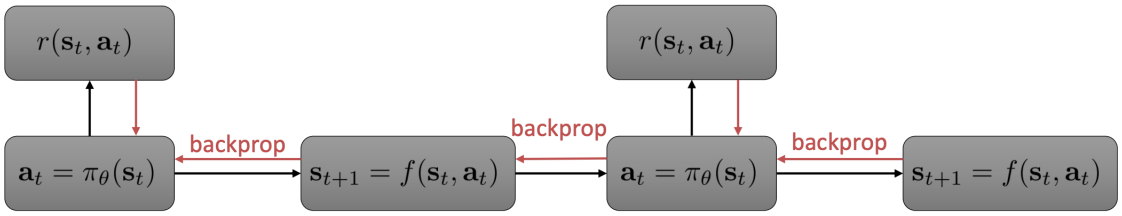

Figure 9.1: Back-propagate into policies

Algorithm 23 Model-based Reinforcement Learning Version 1.5

Require: Some base policy for data collection π_0

1: Run base policy $\pi_0(a_t|s_t)$ (e.g. random policy) to collect $\mathcal{D} = \{(s, a, s')_i\}$
2: **while** True **do**
3: Learn dynamics model $f(s, a)$ to minimize $\sum_i ||f(s_i, a_i) - s'_i||^2$
4: Backpropagate through $f(s, a)$ into the policy to optimize $\pi_\theta(a_t|s_t)$
5: Run $\pi_\theta(a_t|s_t)$, appending the visited tuples (s, a, s') to $\mathcal{D}$.

big (exploding) or extremely small (vanishing), making optimization a lot harder. Furthermore, we have similar parameter sensitivity problems as shooting methods, but we no longer have convenient second order LQR-like method, because the policy function is extremely complicated and policy parameters couple all the time steps, so no dynamic programming.

So what can we do about it? First, we can use model-free RL algorithms with synthetic samples generated by the model. Essentially, we are using models to accelerate model-free RL. Second, we cam use simpler policies than neural nets such as LQR, and train local policies to solve simple tasks, and then combine them into global policies via supervised learning.

9.2 Model-free Optimization with a Model

Recall the equation from policy gradients:

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

Note that we are not doing any backprop through time in policy gradient because we are calculating the gradient with respect to an expectation, so we can just take the derivative of the probability of the samples instead of the actual dynamics function.

Then we look at the regular backprop (pathwise) gradient, we see a more chain rule-like gradient:

$$\nabla_\theta J(\theta) = \sum_{t=1}^{T} \frac{dr_t}{ds_t} \prod_{t'=2}^{t} \frac{ds_{t'}}{da_{t'-1}} \frac{da_{t'-1}}{ds_{t'-1}}$$

The two gradients are different, because the policy gradient is for stochastic systems while the backprop policy is for deterministic systems. But using variational inference, we can prove that they are calculating the same gradient differently, this having different tradeoffs. We will talk about variational inference more in-depth in the next chapter.

Algorithm 24 Dyna

Require: Some exploration policy for data collection π_0

1: Given state s, pick action a using exploration policy
2: Observe s' and r, to get transition (s, a, s', r)
3: Update model $\hat{p}(s'|s, a)$ and $\hat{r}(s, a)$ using (s, a, s')
4: Q-update: $Q(s, a) \leftarrow Q(s, a) + \alpha \mathbb{E}_{s',r} [r + \max_{a'} Q(s', a') - Q(s, a)]$
5: **for** K times **do**
6: Sample $(s, a) \sim \mathcal{B}$ from buffer of past states and actions
7: Q-update: $Q(s, a) \leftarrow Q(s, a) + \alpha \mathbb{E}_{s',r} [r + \max_{a'} Q(s', a') - Q(s, a)]$

Algorithm 25 General Dyna

Require: Some exploration policy for data collection π_0

1: Collect some data, consisting of transitions (s, a, s', r)
2: Learn model $\hat{p}(s'|s, a)$ (and optionally, $\hat{r}(s, a)$
3: **for** K times **do**
4: Sample $s \sim \mathcal{B}$ from buffer
5: Choose action a (from $\mathcal{B}$, from π, or random)
6: Simulate $s' \sim \hat{p}(s'|s, a)$ (and $r = \hat{r}(s, a)$)
7: Train on (s, a, s', r) with model-free RL
8: (optional) take N more model-based steps

Actually, given more samples to reduce variance, policy gradient is more stable because it does not require multiplying many Jacobians. However, if our models are inaccurate, the samples we use from the wrong model will be incorrect, and the mistakes are likely to exacerbate as time goes on. So it would be nice to use such model-free optimizer and keep the rolled out samples' trajectory short. This is essentially what Dyna algorithm does.

9.2.1 Dyna

Dyna is an online Q-learning algorithm that performs model-free RL with a model. In step 3 of Alg. 24, we are updating the model and reward function using the observed transition. Then in step 6, we will sample some old state and action pairs and apply the model onto the sampled pair. Intuitively, as the models get better, the expectation estimate in step 7 also gets more accurate. This algorithm seems arbitrary in many aspects, but the gist is to keep improving models and use models to improve Q-function estimation by taking expectations.

We can also generalize Dyna to see how this kind of general Dyna-style model-based RL algorithms work. The generalized algorithm is shown in Alg. 25 25. As shown in Fig. 25, we choose some states (orange dots) from the buffer, simulate the next states using the learned model, and then train model-free RL with synthetic data (s, a, s', r) where s is from the experience buffer, s' is from the learned model. One could also take more than one step if one believes that the model is good enough for more steps.

This algorithm only requires very short (as few as one step) rollouts from model, so the mistakes will not exacerbate and accumulate much. Moreover, we explore well with a lot of samples because we still see diverse states.

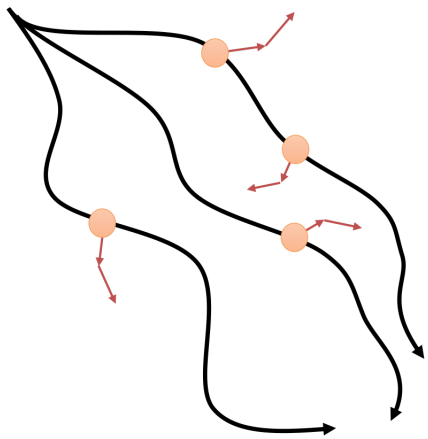

Figure 9.2: General Dyna training

9.3 Local and Global Models

Recall that in LQR, we can turn a constrianed optimization problem into an unconstrained problem:

$$\min_{u_1,\dots,u_T} c(x_1,u_1)+c(f(x_1,u_1),u_2)+\cdots+c(f(f(\dots)\dots),u_T)$$

Backpropagation is indeed a possible solution to solve this optimization problem, and we need $\frac{df}{dx_t}$, $\frac{df}{du_t}$, $\frac{dc}{dx_t}$, $\frac{dc}{du_t}$

9.3.1 Local Models

Since LQR gives us a state-feedback controller for a linear system, we can keep linearizing the system and iteratively apply LQR to generate local models. We fit $\frac{df}{dx_t}$, $\frac{df}{du_t}$ around the current trajectory or policy. Say the model is a Gaussian $p(x_{t+1}|x_t,u_t)=\mathcal{N}(f(x_t,u_t),\Sigma)$, then we can approximate the model as a linear function $f(x_t,u_t)\simeq A_tx_t+B_tu_t$, and we can use $\frac{df}{dx_t}$ as A_t, and $\frac{df}{du_t}$ as B_t.

Iterative LQR produces $\hat{x}_t,\hat{u}_t,K_t,k_t$, where $u_t=K_t(x_t-\hat{x}_t)+k_t+\hat{u}_t$. We can execute the controller using a Gaussian $p(u_t|x_t)=\mathcal{N}(K_t(x_t-\hat{x}_t)+k_t+\hat{u}_t,\Sigma_t)$ because we can add noise to the iLQR controller so that all samples do not look the same. Practically, we can set $\Sigma_t=Q^{-1}_{u_t,u_t}$. We can fit the model $p(s_{t+1}|s_t,a_t)$ using Bayesian linear regression, and use the global model as prior.

We also need to stay close to old controller if we go too far. If trajectory distribution is close, then dynamics will be close too. Close here means the KL-divergence is small $D_{KL}(p(\tau)||p(\bar{\tau}))\le\epsilon$.

9.3.2 Guided Policy Search

The high level idea of guided policy search is to use some simpler local policy such as local LQR controller to help and guide the learning process of more complex global policy learner. Essentially, we would use the local models trajectories as the training data for a supervised learning neural net that can solve all the tasks.

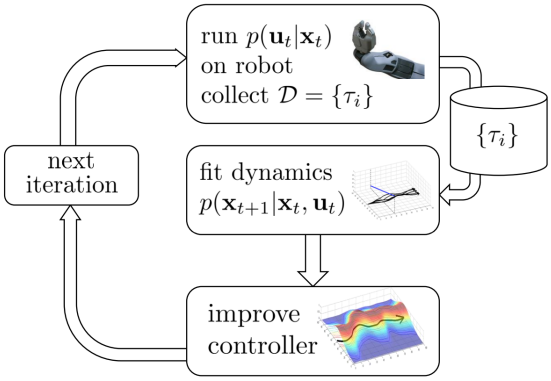

Figure 9.3: Local models fitting

Algorithm 26 Guided Policy Search

1: **while** True **do**
2: Optimize each local policy $\pi_{LQR,i}(u_t|x_t)$ on initial state $x_{0,i}$ with respect to $\tilde{c}_{k,i}(x_t, u_t)$
3: Use samples from the previous step to train $\pi_\theta(u_t|x_t)$ to mimic each $\pi_{LQR,i}(u_t|x_t)$
4: Update cost function $\tilde{c}_{k+1,i}(x_t, u_t) = c(x_t, u_t) + \lambda_{k+1} \log \pi_\theta(u_t|x_t)$

However, one problem is that the local policies might not be able to be reproduced using a single neural net. Therefore, after training the global policy with supervised learning, we need to reoptimize the local policies using the global policy so that the policies are consistent with each other. The sketch of guided policy search is shown in Alg. 26. Note that the cost function $\tilde{c}_{k,i}$ is the modified cost function to keep π_{LQR} close to π_θ.

In Divide and Conquer RL, the idea is similar, except that we are replacing the local LQR controllers with local neural net.

Chapter 10: Variational Inference and Generative Models

In this chapter we are going to explore some techniques that allow us to infer latent variables in latent space. We will try to understand the role of latent probabilistic models in deep learning and how to use them.

In RL, we are mostly concerned with conditional distributions $p(x|y)$ because we are trying to fit a policy function $\pi_\theta(a|s)$ which is a probabilistic model of action conditioned on state.

So what are latent variable models? Consider that we have a very complicated distribution $p(x)$, which cannot be easily modeled by a mixture of Gaussians. By Bayes' rule, this complicated prior can be modeled by two other easier distributions:

$$p(x) = \int p(x|z)p(z)dz$$

$p(x|z)$ and $p(z)$ could be modeled by a conditional Gaussian and a Gaussian respectively. Since any function could be represented by a big enough neural network to an arbitrary precision, we can then use a neural net to represent $p(x|z)$ as $p(x|z) = \mathcal{N}(\mu_{nn}(z), \mu_{nn}(z))$. This sample distribution is a easy distribution with complicated parameters. Often in practice, we won't even learn $p(z)$, because we could just model it as a Gaussian distribution and transform it to any nonlinear distribution using the integral. The challenge of this approach, however, is to efficiently approximate the integral, which is quite hard.

In RL, we mainly use latent variable models in the following scenarios. First, we could use conditional latent variable models for multi-modal policies, as we discussed in imitation learning. Specifically, we could train a network with Gaussian noise to infer the state from image-based observations. Another scenario is that we could use latent variable models for model-based RL. Essentially, we learn a conditional distribution $p(o_t|x_t)$ and prior $p(x_t)$.

10.1 Training Latent Variable Models

The model we are trying to fit is $p_\theta(x)$. We train the model using data $\mathcal{D} = \{x_1, x_2, \ldots, x_N\}$. We use maximum likelihood fit: $\theta \leftarrow \arg\max_\theta \frac{1}{N}\sum_i \log p_\theta(x_i)$. Using latent variables, we have $\theta \leftarrow \arg\max_\theta \frac{1}{N}\sum_i \log\left(\int p_\theta(x_i|z)p(z)dz\right)$. And as we have shown above, the integral is completely intractable.

Alternatively, we could use the expected log-likelihhod:

$$theta \leftarrow \arg\max_\theta \frac{1}{N}\sum_i \mathbb{E}_{z\sim p(z|x_i)} \log p_\theta(x_i)$$

However, the conditional distribution $p(z|x_i)$ is unknown. Therefore, we can approximate this distribution with a simpler distribution $q_i(z) = \mathcal{N}(\mu_i, \sigma_i)$.

10.1.1 Variational Approximation

It turns out that if we approximate the distribution using $q_i(z)$, we can bound the distribution of interest $\log p(x_i)$. Therefore, by maximizing this lower bound, we are maximizing the log likelihood. We use $q_i(z)$ to approximate $\log p(x_i)$ by:

$$\begin{aligned}
\log p(x_i) &= \log \int_z p(x_i|z)p(z) \\
&= \log \int_z p(x_i|z)p(z)\frac{q_i(z)}{q_i(z)} \\
&= \log \mathbb{E}_{z\sim q_i(z)}\left[\frac{p(x_i|z)p(z)}{q_i(z)}\right] \\
&\geq \mathbb{E}_{z\sim q_i(z)}\left[\log \frac{p(x_i|z)p(z)}{q_i(z)}\right] \\
&= \mathbb{E}_{z\sim q_i(z)}\left[\log p(x_i|z) + \log p(z)\right] - \mathbb{E}_{z\sim q_i(z)}\left[\log q_i(z)\right] \\
&= \mathbb{E}_{z\sim q_i(z)}\left[\log p(x_i|z) + \log p(z)\right] + \mathcal{H}(q_i)
\end{aligned}$$

where we applied Jensen's inequality in the second to last step. Jensen's inequality states that:

$$\log \mathbb{E}[y] \geq \mathbb{E}[\log y]$$

If we maximize $\log p(x_i|z)$, we will maximize $\log p(x_i)$. Also, intuitively, if we maximize $\log p(x_i|z)$, we are maximizing the peak of the distribution, and since we are maximizing the entropy $\mathcal{H}(q_i)$ too, we are also making the distribution as wide as possible, which is how we drive the approximated distribution $q_i(z)$ as close as possible to the target distribution $p(x_i, z)$.

Let us take a closer look at this lower bound. Define $\mathbb{E}_{z\sim q_i(z)}\left[\log p(x_i|z) + \log p(z)\right] + \mathcal{H}(q_i)$ as $\mathcal{L}_i(p, q_i)$. In tuitively, this term measures the likelihood. For a $q_i(z)$ to approximate $p(z|x_i)$ well, we need to minimize the KL-divergence between the two distributions. By definition, the KL divergence of the two distributions is written as:

$$\begin{aligned}
D_{KL}(q_i(z)||p(z|x_i)) &= \mathbb{E}_{z\sim q_i(z)}\left[\log \frac{q_i(z)}{p(z|x_i)}\right] \\
&= \mathbb{E}_{z\sim q_i(z)}\left[\log \frac{q_i(z)p(x_i)}{p(x_i, z)}\right] \\
&= -\mathbb{E}_{z\sim q_i(z)}\left[\log p(x_i|z) + \log p(z)\right] + \mathbb{E}_{z\sim q_i(z)}\left[\log q_i(z)\right] + \mathbb{E}_{z\sim q_i(z)}\left[\log p(x_i)\right] \\
&= -\mathbb{E}_{z\sim q_i(z)}\left[\log p(x_i|z) + \log p(z)\right] - \mathcal{H}(q_i) + \log p(x_i) \\
&= -\mathcal{L}_i(p, q_i) + \log p(x_i)
\end{aligned}$$

Therefore,

$$\begin{aligned}
\log p(x_i) &= D_{KL}(q_i(z)||p(z|x_i)) + \mathcal{L}_i(p, q_i) \\
\log p(x_i) &\geq \mathcal{L}_i(p, q_i)
\end{aligned}$$

Note that we eliminated the expectation $\mathbb{E}_{z\sim q_i(z)}[\log p(x_i)]$ because $p(x_i)$ does not depend z.

Since $D_{KL}(q_i(x_i)||p(z|x_i)) = -\mathcal{L}_i(p,q_i)+\log p(x_i)$, maximizing $\mathcal{L}_i(p,q_i)$ with respect to q_i minimizes the KL-divergence. Now in our maximum likelihood training, instead of doing $\theta \leftarrow \arg\max_\theta \frac{1}{N}\sum_i \log p_\theta(x_i)$, we can use the lower bound and do $\theta \leftarrow \arg\max_\theta \frac{1}{N}\sum_i \mathcal{L}_i(p,q_i)$ to approximate it. To optimize, for each x_i, we calculate $\nabla_\theta \mathcal{L}_i(p,q_i)$ by sampling $z\sim q_i(z)$ and the gradient of the likelihood term can be approximated using $\nabla_\theta \mathcal{L}_i(p,q_i) \simeq \nabla_\theta \log p_\theta(x_i|z)$ because $\log p_\theta(x_i|z)$ is the only term in the likelihood that depends on θ. Then we apply gradient ascent on the parameter θ by $\theta \leftarrow \theta + \alpha\nabla_\theta \mathcal{L}_i(p,q_i)$.

However, we also need to update q_i to maximize $\mathcal{L}_i(p,q_i)$ because it also depends on $\mathcal{H}(q_i)$. Let's say $q_i(z) = \mathcal{N}(\mu_i,\sigma_i)$, then we can apply gradient ascent on both parameters μ_i, σ_i to update this distribution. The problem here is the above update rule is for each data point. Therefore, the number of parameters is $|\theta| + (|\mu_i| + |\sigma_i|) * N$, where N is the number of data points. Thus, we can modify the distribution we are learning so that we use a more general neural network to approximate $q(z|x_i)$ such that $q(z|x_i) = q_i(z) \simeq p(z|x_i)$. Now the number of the network parameter does not scale with the number of data points.

10.1.2 Amortized Variational Inference

The above idea is called amortized variational inference. When we maximize the likelihood, instead of using q_i for each data point, we use a general neural net q_ϕ, parameterized by ϕ. Then when we update q_ϕ, we can just apply gradient ascent on ϕ by $\phi \leftarrow \phi + \alpha\nabla_\phi \mathcal{L}$. The likelihood can be denoted as $\mathcal{L}_i(p_\theta(x_i|z), q_\phi(z|x_i))$.

How do we calculate $\nabla_\phi \mathcal{L}$? Note that

$$\mathcal{L}_i = \mathbb{E}_{z\sim q_\phi(z|x_i)}[\log p_\theta(x_i|z) + \log p(z)] + \mathcal{H}(q_\phi(z|x_i))$$

to calculate the gradient of the likelihood with respect to ϕ, we can calculate the entropy term's gradient easily using textbook formula. However, the first term is harder because the expectation is taken under a distribution depending on ϕ, but the term inside the expectation is independent of ϕ. Where have we seen this before? Where have seen the same type of gradient in policy gradient, and by applying the convenient identity, we can get the same form of gradient. If we call $\log p_\theta(x_i|z)+\log p(z)$ as $r(x_i,z)$, and $\mathbb{E}_{z\sim q_\phi(z|x_i)}$ as $J(\phi)$. Applying the same trick as in policy gradient, we can calculate $\nabla J(\phi)$ as:

$$\nabla J(\phi) \simeq \frac{1}{M}\sum_j \nabla_\phi \log q_\phi(z_j|x_i) r(x_i,z_j)$$

10.1.3 The Reparameterization Trick

Consider $q_\phi(z|x)$ as a Gaussian distribution $\mathcal{N}(\mu_\phi(x),\sigma_\phi(x))$, then for every z in this distribution, it can be expressed as $z = \mu_\phi(x) + \epsilon\sigma_\phi(x)$, where ϵ is some type of a Gaussian noise $\epsilon\sim\mathcal{N}(0,1)$, and the noise is independent of ϕ. Thus, we have:

$$\begin{aligned} J(\phi) &= \mathbb{E}_{z\sim q_\phi(z|x_i)}[r(x_i,z)] \\ &= \mathbb{E}_{\epsilon\sim\mathcal{N}(0,1)}[r(x_i,\mu_\phi(x)+\epsilon\sigma_\phi(x))] \end{aligned}$$

To estimate $\nabla_\phi J(\phi)$, we can just sample M samples of ϵ from a Gaussian $\mathcal{N}(0,1)$.

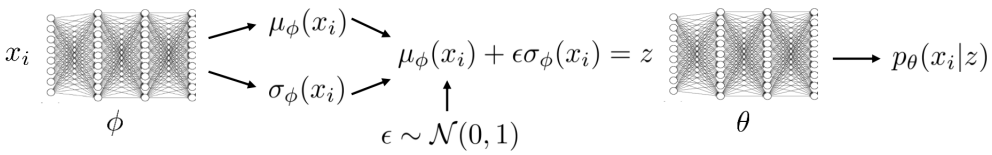

Figure 10.1: Variational inference

Using this reparameterization trick, we can derive the expression of $\mathcal{L}_i$ in another way:

$$\begin{aligned}
\mathcal{L}_i &= \mathbb{E}_{z\sim q_\phi(z|x_i)}\left[\log p_\theta(x_i|z) + \log p(z)\right] + \mathcal{H}(q_\phi(z|x_i)) \\
&= \mathbb{E}_{z\sim q_\phi(z|x_i)}\left[\log p_\theta(x_i|z)\right] + \mathbb{E}_{z\sim q_\phi(z|x_i)}\left[\log p(z)\right] + \mathcal{H}(q_\phi(z|x_i)) \\
&= \mathbb{E}_{z\sim q_\phi(z|x_i)}\left[\log p_\theta(x_i|z)\right] - D_{KL}(q_\phi(z|x_i)||p(z)) \\
&= \mathbb{E}_{\epsilon\sim\mathcal{N}(0,1)}\left[\log p_\theta(x_i|\mu_\phi(x_i) + \epsilon\sigma_\phi(x_i))\right] - D_{KL}(q_\phi(z|x_i)||p(z)) \\
&\simeq \log p_\theta(x_i|\mu_\phi(x_i) + \epsilon\sigma_\phi(x_i)) - D_{KL}(q_\phi(z|x_i)||p(z))
\end{aligned}$$

The complete computational graph for variational inference is shown in Fig. 10.1.

Compared with policy gradient, the reparameterization trick is easy to implement and as low variance, but it only works for continuous latent variables. Policy gradient can Can handle both discrete and continuous latent variables, but it is subject to high variance, rand equires multiple samples and small learning rates.

10.2 Variational Autoencoder (VAE)

The variational autoencoder (VAE) consists of two parts: an encoder and a decoder. The encoder $q_\phi(z|x) = \mathcal{N}(\mu_\phi(x), \sigma_\phi(x))$ parameterized by ϕ gives us a latent variable z, and the decoder $p_\theta(x|z) = \mathcal{N}(\mu_\theta(x), \sigma_\theta(x))$ is parameterized by θ.

When we are inferring $p(x)$ by $p(x) = \int p(x|z)p(z)dz$, we sample z from the distribution $p(z)$, and sample x from the distribution $p(x|z)$. Why does this work? Recall the evidence lower bound $\mathcal{L}_i$ is defined as:

$$\mathcal{L}_i = \mathbb{E}_{z\sim q_\phi(z|x_i)}\left[\log p_\theta(x_i|z)\right] - D_{KL}(q_\phi(z|x_i)||p(z))$$

q_ϕ should embed your observations x_i into z, into a distribution that is closer to the prior. So if the training data is embedded into the distribution that is similar to the prior, it makes sense that the samples from the prior will give you things that look like the data.

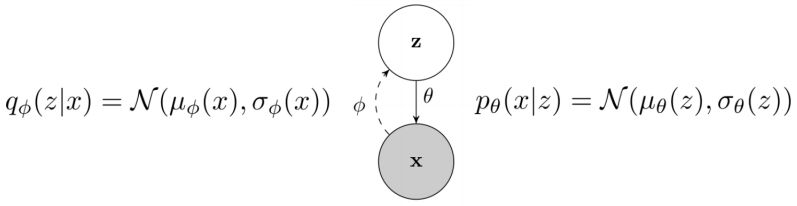

Figure 10.2: VAE

Chapter 11: Control as Inference

In this chapter, we will talk about how we derive optimal control, reinforcement learning, and planning as probabilistic inference. In a lot of scenarios that, say, involve biological behaviors, the data is not optimal. The behavior of the agent might be stochastic, but good behaviors are still more likely.

11.1 Probabilistic Graphical Model of Decision Making

When we do not make any assumption of optimal behavior, we cannot ensure that the actions are chosen optimally. In other words, we cannot assume the following relation:

$$a_1, \dots, a_T = \arg\max_{a_1,\dots,a_T} \sum_{t=1}^{T} r(s_t, a_t)$$

Instead, we should model the probability distribution of seeing a trajectory $p(\tau) = p(s_{1:T}, a_{1:T})$. We also introduce a binary optimality variable $\mathcal{O}_t$, which represents if the agent if behaving optimally at time step t. Then we are interested in $p(\tau|\mathcal{O}_{1:T})$, where we infer the probability of the given trajectory given the agent is optimal at every time step. Now we will model the optimality variable as follows: we model that the probability that variable is true given state and action is an exponential of the reward:

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

this might seem an arbitrary choice at the first sight, but we shall see later that this gives us an elegant mathematical expression in our derivation. We also assume for now that the reward function is always negative, but we can always take any reward function and

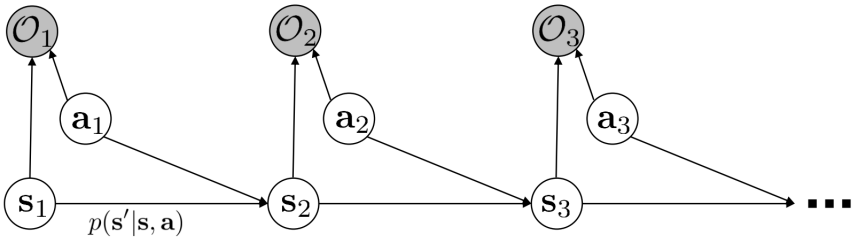

Figure 11.1: Optimalility in stochastic behaviors

normalize it by subtracting the max reward. Then by Bayes' Rule, we have:

$$\begin{aligned} p(\tau|\mathcal{O}_{1:T}) &= \frac{p(\tau, \mathcal{O}_{1:T})}{p(\mathcal{O}_{1:T})} \\ &\propto p(\tau) \prod_t \exp r(s_t, a_t) \\ &= p(\tau) \exp \sum_t r(s_t, a_t) \end{aligned}$$

What does the above expression imply? Well, let us pretend that the dynamics are deterministic, then the first term $p(\tau)$ just means if this trajectory is possible. If not, then the probability is 0. If the trajectory is indeed possible, since we are multiplying by the exponent of the sum of rewards, then the probability of a trajectory given the agent is acting optimally is big with high rewards, but small with low rewards.

Let us take a look at the optimality model in Fig. 11.1. Why is this model important? Because the model is able to model suboptimal behavior, which is important for inverse RL that will be covered later. We then can apply inference algorithms to solve control and planning problems. It also provides an explanation for why stochastic behavior might be preferred, which is useful for exploration and transfer learning.

11.1.1 Inference in the Optimality Model

The first inference we will do is to compute the backward message $\beta_t(s_t, a_t) = p(\mathcal{O}_{t:T}|s_t, a_t)$, which means the probability of the agent being optimal from the current time step to the end given state and action. Another inference we will do is the policy $p(a_t|s_t, \mathcal{O}_{1:T})$. Note that we are inferring the possible actions taken given optimality. The last inference we do is the forward message $\alpha_t(s_t) = p(s_t|\mathcal{O}_{1:t-1})$, which is the probability of landing in a particular state given that the agent is acting optimally up to the current time step.

11.1.2 Inferring the Backward Messages

The backward messages we are inferring is $\beta_t(s_t, a_t) = p(\mathcal{O}_{t:T}|s_t, a_t)$, which we will try to express in terms of transition probability $p(s_{t+1}|s_t, a_t)$ and optimality probability $p(\mathcal{O}_t|s_t, a_t)$. Mathematically, we can calculate $\beta_t(s_t, a_t)$ as:

$$\begin{aligned} \beta_t(s_t, a_t) &= p(\mathcal{O}_{t:T}|s_t, a_t) \\ &= \int p(\mathcal{O}_{t:T}, s_{t+1}|s_t, a_t) ds_{t+1} \\ &= \int p(\mathcal{O}_{t+1:T}|s_{t+1}) p(s_{t+1}|s_t, a_t) p(\mathcal{O}_t|s_t, a_t) ds_{t+1} \end{aligned}$$

The second and the third terms in the product are known, so let us now focus on the first term:

$$\begin{aligned} p(\mathcal{O}_{t+1:T}|s_{t+1}) &= \int p(\mathcal{O}_{t+1:T}|s_{t+1}, a_{t+1}) p(a_{t+1}|s_{t+1}) da_{t+1} \\ &= \int \beta(s_{t+1}, a_{t+1}) da_{t+1} \end{aligned}$$

we ignored $p(a_{t+1}|s_{t+1})$ it means which actions are likely a priori, and we assume it is uniform (constant) for now.

Therefore, to calculate the backward message, we have a recursive relation. For $t = T-1$ to 1:

$$\beta_t(s_t, a_t) = p(\mathcal{O}_t|s_t, a_t)\mathbb{E}_{s_{t+1}\sim p(s_{t+1}|s_t,a_t)}[\beta_{t+1}(s_{t+1})]$$
$$\beta_t(s_t) = \mathbb{E}_{a_t\sim p(a_t|s_t)}[\beta_t(s_t, a_t)]$$

11.1.3 A Closer Look

Let us take a closer look at the backward pass. Let $V_t(s_t) = \log \beta_t(s_t)$, and let $Q_t(s_t, a_t) = \log \beta_t(s_t, a_t)$. Then

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

As $Q_t(s_t, a_t)$ gets bigger $V_t(s_t) \to \max_{a_t} Q_t(s_t, a_t)$. Using the expression of $\beta_t(s_t, a_t)$, we will have

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

Recall in value iteration, we set $Q(s, a) \leftarrow r(s, a) + \gamma\mathbb{E}[V(s')]$. When the transition is deterministic, we have $Q_t(s_t, a_t) = r(s_t, a_t) + V_{t+1}(s_{t+1})$, which is similar to value iteration. However, when the transition is stochastic, then the log exp term is like a maximum operation, so we have a biased optimistic estimation of the Q-function.

11.1.4 Aside: The Action Prior

Recall that we assumed $p(a_t|s_t)$ to be uniform, so it became constant in our integral. However, we shall see that it does not change much if the action prior is not uniform. Our V function now becomes $V_t(s_t) = \log \int \exp(Q_t(s_t, a_t) + \log p(a_t|s_t))da_t$, and our Q-function becomes $Q(s_t, a_t) = r(s_t, a_t) + \log p(a_t|s_t) + \log \mathbb{E}[\exp(V_{t+1}(s_{t+1}))]$ We can put the extra $p(a_t|s_t)$ into the reward term, then we will have the same expression of the Q-funtion, thus the V function. Therefore, uniform action prior can be assumed without loss of generality because it can always be folded into the reward.

11.1.5 Inferring the Policy

Now with backward messages available to us, we can then proceed to infer the policy $p(a_t|s_t, \mathcal{O}_{1:T})$. We derive the policy as follows:

$$\begin{aligned}
p(a_t|s_t, \mathcal{O}_{1:T}) &= \pi(a_t|s_t) \\
&= p(a_t|s_t, \mathcal{O}_{t:T}) \\
&= \frac{p(a_t, s_t|\mathcal{O}_{t:T})}{p(s_t|\mathcal{O}_{t:T})} \\
&= \frac{p(\mathcal{O}_{t:T}|a_t, s_t)p(a_t, s_t)/p(\mathcal{O}_{t:T})}{p(\mathcal{O}_{t:T}|s_t)p(s_t)/p(\mathcal{O}_{t:T})} \\
&= \frac{p(\mathcal{O}_{t:T}|a_t, s_t)}{p(\mathcal{O}_{t:T}|s_t)}\frac{p(a_t, s_t)}{p(s_t)} \\
&= \frac{\beta_t(s_t, a_t)}{\beta_t(s_t)}p(a_t|s_t)
\end{aligned}$$

we discard the optimality variables $1, \dots t-1$ because then are conditionally independent of s_t. We also discard $p(a_t|s_t)$ since we can assume it as uniform. Using the definition of V, Q, we have

$$\begin{aligned}\pi(a_t|s_t) &= \frac{\beta_t(s_t, a_t)}{\beta_t(s_t)} \\ &= \exp(Q_t(s_t, a_t) - V_t(s_t)) \\ &= \exp(A_t(s_t, a_t))\end{aligned}$$

This result makes sense, because when we have large advantage function values, the action is more likely to be taken.

11.1.6 Inferring the Forward Messages

We now can infer our third task, the forward message $\alpha_t(s_t) = p(s_t|\mathcal{O}_{1:t-1})$. The derivation is as follows:

$$\begin{aligned}\alpha_t(s_t) &= p(s_t|\mathcal{O}_{1:t-1}) \\ &= \int p(s_t, s_{t-1}, a_{t-1}|\mathcal{O}_{1:t-1}) ds_{t-1} da_{t-1} \\ &= \int p(s_t|s_{t-1}, a_{t-1}, \mathcal{O}_{1:t-1}) p(a_{t-1}|s_{t-1}, \mathcal{O}_{1:t-1}) p(s_{t-1}|\mathcal{O}_{1:t-1}) ds_{t-1} da_{t-1} \\ &= \int p(s_t|s_{t-1}, a_{t-1}) p(a_{t-1}|s_{t-1}, \mathcal{O}_{t-1}) p(s_{t-1}|\mathcal{O}_{1:t-1}) ds_{t-1} da_{t-1}\end{aligned}$$

here we used the fact that the current state is conditionally independent of the previous optimality variables given the previous state, and we also used the fact that the current action is conditionally independent of the previous optimality variables given the current state. The first term is just the dynamics, so we need to figure out what the second and the third terms by Bayes' rule:

$$\begin{aligned}p(a_{t-1}|s_{t-1}, \mathcal{O}_{t-1}) p(s_{t-1}|\mathcal{O}_{1:t-1}) &= \frac{p(\mathcal{O}_{t-1}|s_{t-1}, a_{t-1}) p(a_{t-1}|s_{t-1})}{p(\mathcal{O}_{t-1}|s_{t-1})} \frac{p(\mathcal{O}_{t-1}|s_{t-1}) p(s_{t-1}|p(\mathcal{O}_{1:t-2})}{p(\mathcal{O}_{t-1}|\mathcal{O}_{1:t-2})} \\ &= \frac{p(\mathcal{O}_{t-1}|s_{t-1}, a_{t-1}) p(a_{t-1}|s_{t-1})}{p(\mathcal{O}_{t-1}|\mathcal{O}_{1:t-2})} \alpha_{t-1}(s_{t-1})\end{aligned}$$

so now we have a recursive relation, and $\alpha_a(s_1) = p(s_1)$ is usually known.

Another byproduct of having this forward message is that we can combine it with the backward message to calculate the probability of landing in a particular state given optimality variables:

$$p(s_t|\mathcal{O}_{1:T}) = \frac{p(s_t, \mathcal{O}_{1:T})}{p(\mathcal{O}_{1:T})} = \frac{p(\mathcal{O}_{t:T}|s_t) p(s_t, \mathcal{O}_{1:t-1})}{p(\mathcal{O}_{1:T})} \propto \beta_t(s_t) p(s_t|\mathcal{O}_{1:t-1}) p(\mathcal{O}_{1:t-1}) \propto \beta_t(s_t) \alpha_t(s_t)$$

Geometrically, the relation between the state marginal and the product of backward and forward messages is shown in Fig. 11.2. Here the backward messages is a backward cone, and the forward message is a forward cone. When we take the product of the two, we are essentially finding the intersection of the two cones. Intuitively, for a state in a trajectory, the state marginals are tighter near the beginning and the end, but looser near the center because the state marginals need to close in at the beginning and the end of a trajectory.

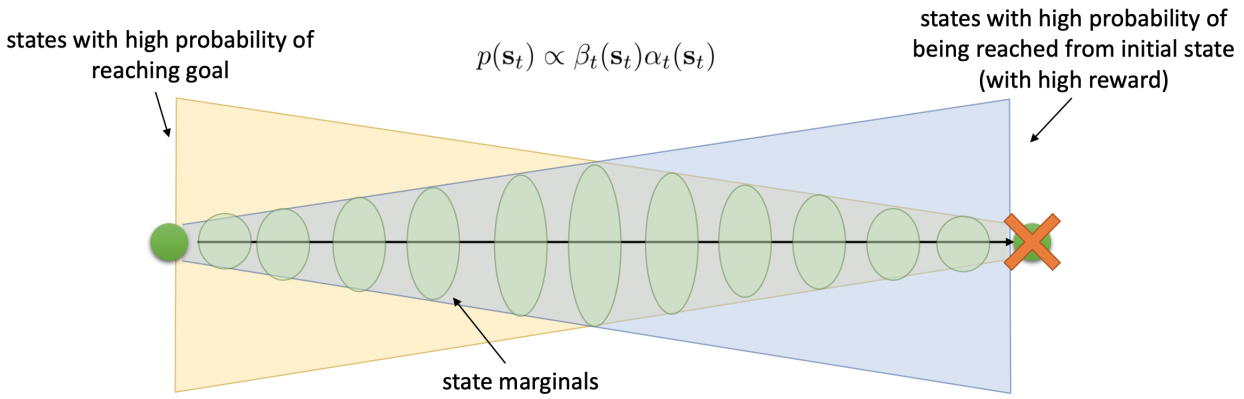

Figure 11.2: Forward/backward messages intersection

11.2 The Optimism Problem

Recall in the dynamic programming view of our backward message inference, the Q-function can be written as:

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

We have shown that $\log \mathbb{E} \exp$ behaves like a max, thus bringing us bias in the estimate of Q-function. Marginalizing and conditioning the backward message $\beta_t(s_t, a_t) = p(\mathcal{O}_{t:T}|s_t, a_t)$, we can have two different distributions to infer: first, we can have the policy $p(a_t|s_t, \mathcal{O}_{1:T})$, which means given that you had a high reward (optimal), what was your action probability? Second, we can have the transition $p(s_{t+1}|s_t, a_t, \mathcal{O}_{1:T})$, and we should notice that this is not equal to the transition probability $p(s_{t+1}|s_t, a_t)$ because now we are asking given that you obtained high rewards, what was your transition probability? To address the optimism problem, we need to ask the first question: given that you obtained high reward, what was your action probability, assuming that we have the same transition probability, such that we are no luckier than we usually are.

It turns out the first question is a difficult one. To answer that question, we can find another distribution $q(s_{1:T}, a_{1:T})$ that is close to $p(s_t, a_t|\mathcal{O}_{1:T})$, but have the same $p(s_{t+1}|s_t, a_t)$. So let's us try variational inference. Let our evidence x, what we have observed, be the optimality variables $\mathcal{O}_{1:T}$, and the latent variable z, what we have not observed, be the trajectory $s_{1:T}, a_{1:T}$. Using variational inference, we find a $q(z)$ to approximate $p(z|x)$.

Let $q(s_{1:T}, a_{1:T}) = p(s_1) \prod_t p(s_{t+1}|s_t, a_t) q(a_t|s_t)$ since we are keeping the same initial state distribution and the same transition. Recall that the variational lower bound of the likelihood approximation is:

$$\log p(x) \geq \mathbb{E}_{z \sim q(z)}[\log p(x, z) - \log q(z)]$$

plugging in our previous definition of $q(z)$, we have

$$\begin{aligned}
\log p(\mathcal{O}_{1:T}) &\geq \mathbb{E}_{(s_{1:T},a_{1:T})\sim q}\Bigg[\log p(s_1) + \sum_{t=1}^{T} \log p(s_{t+1}|s_t,a_t) + \log p(\mathcal{O}_T|s_t,a_t) \\
&\quad - \log p(s_1) - \sum_{t=1}^{T} \log p(s_{t+1}|s_t,a_t) - \sum_{t=1}^{T} \log q(a_t|s_t)\Bigg] \\
&= \mathbb{E}_{(s_{1:T},a_{1:T})\sim q}\left[\sum_t r(s_t,a_t) - \log q(a_t|s_t)\right] \\
&= \mathbb{E}_{(s_{1:T},a_{1:T})\sim q}\left[r(s_t,a_t) + \mathcal{H}(q(a_t|s_t))\right]
\end{aligned}$$

Therefore, to maximize the lower bound, we maximize the reward and the entropy.

Using dynamic programming, we can get rid of the optimism max in the Bellman backup term.

Chapter 12: Inverse Reinforcement Learning

So far in our RL algorithms, we have been assuming that the reward function is known a priori, or it is manually designed to define a task. What if we want to learn the reward function from observing an expert, and then use reinforcement learning? This is the idea of inverse RL, where we first figure out the reward function and then apply RL.

Why should we worry about learning rewards at all? From the imitation learning perspective, the agent learns via imitation by copying the actions performed by the expert, without any reasoning about outcomes of actions. However, the natural way that human learn through imitation is that human copy the intent of the expert, and thus might take very different actions. In RL, it is often the case that the reward function is ambiguous in the environment. For example, it is hard to hand-design a reward function for autonomous driving.

The inverse RL problem definition is as follows: we try to infer the reward functions from demonstrations, and then learn to maximize the inferred reward using any RL algorithm that was covered so far. Formally, in inverse RL, we learn $r_\psi(s, a)$, and then use it to learn $\pi^*(a|s)$. However, this is an underspecified problem, because many reward function can explain the same behavior. The reward function can take many forms. One potential form is the linear reward function, which is a weighted sum of features:

$$r_\psi(s, a) = \sum_i \psi_i f_i(s, a) = \psi^T f(s, a)$$

or it could be a neural net with parameters ψ.

12.1 Feature Matching Inverse RL

Let us focus on the linear reward function design for now. Since it is a weighted sum of features, one natural interpretation to match the features is to match the expectation of important features. Let π^{r_ψ} be the optimal policy for reward functionr_ψ, then we to design the reward, we are picking ψ such that

$$\mathbb{E}_{\pi^{r_\psi}}[f(s, a)] = \mathbb{E}_{\pi^*}[f(s, a)]$$

The right hand side expectation can be estimated using samples from expert: take N samples of features, and get the average. The left hand side expectation is a little involved. One way to do it is to use any RL algorithm to maximize r_ψ, which is defined using the right hand side samples, and then produce π^{r_ψ}, and then we can use this policy to generate more samples. Another way is to use dynamic programming if we are given the transitions. To ensure the equality holds, we borrow some ideas from the support vector machine classifier,

where we maximize the margin between the optimal policy's rewards and that of any other policy:

$$\max_{\psi,m} m \text{ s.t. } \psi^T \mathbb{E}_{\pi^*}[f(s,a)] \geq \max_{\pi \in \Pi} \psi^T \mathbb{E}_{\pi}[f(s,a)] + m$$

but we also need to address the similarity between π and π^* so that similar policies do not need to abide by the m margin requirement.

Using the SVM trick (with the use of Lagrangian dual), we can transform the above optimization into the following which also contains a function that measures the similarity between policies:

$$\min_{\psi} \frac{1}{2}||\psi||^2 \text{ s.t. } \psi^T \mathbb{E}_{\pi^*}[f(s,a)] \geq \max_{\pi \in \Pi} \psi^T \mathbb{E}_{\pi}[f(s,a)] + D(\pi, \pi^*)$$

where $D(\pi, \pi^*)$ measures the difference in feature expectations. However, such approaches have some issues: maximizing the margin is a bit arbitrary, and there is no clear model of expert suboptimality (can add slack variables). Furthermore, now we have a messy constrained optimization problem, which is not great for deep learning!

12.2 Learning the Optimality Variable

Recall that in last chapter, we introduced the optimality variable $\mathcal{O}_t$ to indicate if the agent is acting optimally. It turns out that as we learn the reward function, we are also learning the optimality variable. The optimality variable is defined as $p(\mathcal{O}_t|s_t, a_t) = \exp(r_\psi(s_t, a_t))$. Since the reward parameter ψ is unknown, the optimality distribution should also depend on ψ: $p(\mathcal{O}_t|s_t, a_t, \psi)$. Recall that

$$p(\tau|\mathcal{O}_{1:T}, \psi) \propto \exp\left(\sum_t r_\psi(s_t, a_t)\right)$$

Note that we can ignore $p(\tau)$ in our optimiztion since it does not depend on ψ. We are given sample trajectories $\{\tau_i\}$ sampled from expert policy $\pi^*(\tau)$, so the maximum likelihood training can be done using:

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

where Z is the **partition function** needed to make the sum of probability with respect to τ 1.

12.2.1 Inverse RL Partition Function

In our maximum likelihood training, to make the probability with respect to τ sum to 1, we introduced the IRL partition function Z. Mathematically, Z is the integral of all possible trajectories:

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

Then we take the gradient of the likelihood with respect to ψ after plugging in Z:

$$\begin{aligned}\nabla_\psi \mathcal{L} &= \frac{1}{N}\sum_{i=1}^{N}\nabla_\psi r_\psi(\tau_i) - \frac{1}{Z}\int p(\tau)\exp(r_\psi(\tau))\nabla_\psi r_\psi(\tau)d\tau \\ &= \mathbb{E}_{\tau\sim\pi^*(\tau)}[\nabla_\psi r_\psi(\tau_i)] - \mathbb{E}_{\tau\sim p(\tau|\mathcal{O}_{1:T},\psi)}[\nabla_\psi r_\psi(\tau)]\end{aligned}$$

The first expectation is estimated with expert samples, and the second expectation is the soft optimal policy under current reward. To increase the gradient, we want more expert trajectory and less current agent trajectory.

12.2.2 Estimating the Expectation

In the above derivation of the gradient of the likelihood, the first expectation is easy to calculate, but the second one is hard. To calculate the second expectation, we need to do some messaging:

$$\begin{aligned}\mathbb{E}_{\tau\sim p(\tau|\mathcal{O}_{1:T},\psi)}[\nabla_\psi r_\psi(\tau)] &= \mathbb{E}_{\tau\sim p(\tau|\mathcal{O}_{1:T},\psi)}\left[\nabla_\psi \sum_{t=1}^{T} r_\psi(s_t,a_t)\right] \\ &= \sum_{t=1}^{T}\mathbb{E}_{(s_t,a_t)\sim p(s_t,a_t|\mathcal{O}_{1:T},\psi)}[\nabla_\psi r_\psi(s_t,a_t)]\end{aligned}$$

Note that the distribution $p(s_t,a_t|\mathcal{O}_{1:T},\psi)$ can be rewritten using chain rule as:

$$p(s_t,a_t|\mathcal{O}_{1:T},\psi) = p(a_t|s_t,\mathcal{O}_{1:T},\psi)p(s_t|\mathcal{O}_{1:T},\psi)$$

where

$$\begin{aligned}p(a_t|s_t,\mathcal{O}_{1:T},\psi) &= \frac{\beta(s_t,a_t)}{\beta(s_t)} \\ p(s_t|\mathcal{O}_{1:T},\psi) &\propto \alpha(s_t)\beta(s_t)\end{aligned}$$

Therefore, the distribution is directly proportional to the product of the backward message and the forward message:

$$p(a_t|s_t,\mathcal{O}_{1:T},\psi)p(s_t|\mathcal{O}_{1:T},\psi) \propto \beta(s_t,a_t)\alpha(s_t)$$

If we let $\mu_t(s_t,a_t) \propto \beta(s_t,a_t)\alpha(s_t)$, then the second expectation can be written as:

$$\begin{aligned}\mathbb{E}_{\tau\sim p(\tau|\mathcal{O}_{1:T},\psi)}[\nabla_\psi r_\psi(\tau)] &= \sum_{t=1}^{T}\int\int \mu_t(s_t,a_t)\nabla_\psi r_\psi(\tau)ds_t da_t \\ &= \sum_{t=1}^{T}\mu_t^T\nabla_\psi r_\psi\end{aligned}$$

where μ_t is the state-action visitation probability for each (s_t,a_t).

Now we are ready to sketch out our MaxEnt Inverse RL algorithm in Alg. 27. We can use this to learn the reward function. Why is it called maximum entropy (MaxEnt)? Because in cases where $r_\psi(s_t,a_t) = \psi^T f(s_t,a_t)$, we can show that Alg. 27 oprimizes

$$\max_\psi \mathcal{H}(\pi^{r_\psi}) \text{ s.t. } \quad \mathbb{E}_{\pi^{r_\psi}}[f] = \mathbb{E}_{\pi^*}[f]$$

Algorithm 27 MaxEnt Inverse RL

Require: Some random reward parameter ψ

1: **while** True **do**
2: Given ψ, compute backward message $\beta(s_t, a_t)$
3: Given ψ, compute forward message $\alpha(s_t)$
4: Compute $\mu_t(s_t, a_t) \propto \beta(s_t, a_t)\alpha(s_t)$
5: Evaluate $\nabla_\psi \mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\psi r_\psi(s_{i,t}, a_{i,t}) - \sum_{t=1}^{T}\int\int \mu_t(s_t, a_t)\nabla_\psi r_\psi(\tau) ds_t da_t$
6: $\psi \leftarrow \psi + \eta\nabla_\psi \mathcal{L}$

12.3 Unknown Dynamics and Large State/Action Spaces

So far, MaxEnt inverse RL requires us to solve for a soft optimal policy in the inner loop, and it enumerates all state-action tuples for visitation frequency and gradient. To apply the IRL algorithms in practical problem settings, we need to handle large and continuous state and action spaces and unknown dynamics.

Recall the gradient of likelihood is calculated as

$$\nabla_\psi \mathcal{L} = \mathbb{E}_{\tau\sim\pi^*(\tau)}[\nabla_\psi r_\psi(\tau_i)] - \mathbb{E}_{\tau\sim p(\tau|\mathcal{O}_{1:T},\psi)}[\nabla_\psi r_\psi(\tau)]$$

We know that the first expectation is easy to calculate by sampling expert data, but the second expectation which is taken under the soft optimal policy under current reward is hard to calculate. One idea to calculate it is to learn the entire soft optimal policy $p(a_t|s_t, \mathcal{O}_{1:T}, \psi)$ using any max-ent RL algorithm and then run this policy to sample $\{\tau_j\}$ such that:

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

where we estimate the second expectation using the current policy samples. However, this is highly impractical because this requires us to run an RL algorithm to convergence in every gradient step.

12.3.1 More Efficient Updates

As mentioned above, learning $p(a_t|s_t, \mathcal{O}_{1:T}, \psi)$ in the inner loop in each time step is expensive. Therefore, we can relax this objective a little to make it more efficient: instead of learning the policy at each time step, we could improve the policy a little in each time step such that if the policy keeps getting better, we can generate good samples eventually. Now sampling from this improved distribution is not actually sampling from the distribution we want, which is $p(\tau|\mathcal{O}_{1:T}, \psi)$, we are actually getting a biased estimate of the distribution. Therefore, to resolve this issue, we use importance sampling:

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

$$w_j = \frac{p(\tau)\exp(r_\psi(\tau_j))}{\pi(\tau_j)}$$

And if we take a closer look at the importance ratio w_j:

$$\begin{aligned} w_j &= \frac{p(\tau)\exp(r_\psi(\tau_j))}{\pi(\tau_j)} \\ &= \frac{p(s_1)\prod_t p(s_{t+1}|s_t, a_t \exp(r_\psi(s_t, a_t))}{p(s_1)\prod_t p(s_{t+1}|s_t, a_t \pi(a_t|s_t)} \\ &= \frac{\exp(\sum_t r_\psi(s_t, a_t))}{\prod_t \pi(a_t|s_t)} \end{aligned}$$

With the importance ratio, each policy update with respect to r_ψ brings us closer to the target distribution.

12.4 Inverse RL as a Generative Adversarial Network

The idea of inverse RL looks like a game. Specifically, we have an initial policy π_θ, and expert demonstrations π^*. We sample trajectories τ_j from the initial policy, and τ_i from the expert policy. Then our gradient step looks like:

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

where demos are made more likely and samples are made less likely. Then we update the initial policy π_θ with respect to r_ψ:

$$\nabla_\theta \mathcal{L} \simeq \frac{1}{M}\sum_{j=1}^{M} \nabla_\theta \log \pi_\theta(\tau_j) r_\psi(\tau_j)$$

which in turn changes the policy to make it harder to distinguish from demos.

This looks a lot like a GAN. In a GAN, we have a generator that takes in some noise z, and outputs a distribution $p_\theta(x|z)$. We sample from the generator distribution $p_\theta(x)$. There is also demonstration data, for example, the real images, which we sample from its distribution $p^*(x)$. There is a discriminator parameterized by ψ that determines if the data generated by the generator is real: $D(x) = p_\psi(\text{real}|x)$. We update the discriminator parameter by maximizing the binary log likelihood:

$$\psi = \arg\max_\psi \frac{1}{N}\sum_{x\sim p^*} \log D_\psi(x) + \frac{1}{M}\sum_{x\sim p_\theta} \log(1 - D_\psi(x))$$

where the log likelihood of the data is from demonstration is maximized and that of the data is from generator is minimized. We also update the generator parameter θ:

$$\theta \leftarrow \arg\max_\theta \mathbb{E}_{x\sim p_\theta} \log D_\psi(x)$$

so as to make it harder to distinguish from demos.

Therefore, interestingly, we can frame the IRL problem as a GAN. In a GAN, the optimal discriminator can be defined as:

$$D^*(x) = \frac{p^*(x)}{p_\theta(x) + p^*(x)}$$

For inverse RL, the optimal policy approaches $\pi_\theta(\tau) \propto p(\tau)\exp(r_\psi(\tau))$. Choosing the above optimal parameterization of the discriminator:

$$\begin{aligned} D_\psi(\tau) &= \frac{p(\tau)\frac{1}{Z}\exp(r(\tau))}{p_\theta(\tau) + p(\tau)\frac{1}{Z}\exp(r(\tau))} \\ &= \frac{p(\tau)\frac{1}{Z}\exp(r(\tau))}{p(\tau)\prod_t \pi_\theta(a_t|s_t) + p(\tau)\frac{1}{Z}\exp(r(\tau))} \\ &= \frac{\frac{1}{Z}\exp(r(\tau))}{\prod_t \pi_\theta(a_t|s_t) + \frac{1}{Z}\exp(r(\tau))} \end{aligned}$$

then we optimize the discriminator with respect to ψ such that:

$$\psi \leftarrow \arg\max_\psi \mathbb{E}_{\tau\sim p^*}[\log D_\psi(\tau)] + \mathbb{E}_{\tau\sim\pi_\theta}[\log(1 - D_\psi(\tau))]$$

Now we don't need the importance ratio anymore, because it is subsumed into Z.

We could also use a general discriminator, where D_ψ is just a normal binary neural net classifier. It is often simpler to set up optimization, because we have fewer moving parts. However, the discriminator knows nothing at convergence generally cannot reoptimize the reward.

Chapter 13: Transfer Learning

This chapter is a high-level overview of transfer learning and multi-task learning techniques. More to be filled in later.

Chapter 14: Exploration

In reinforcement learning, we aim to balance exploitation and exploration. In a lot of setting, exploring with random behaviors does not give us satisfactory result due to the highly complex environment. Explorations mainly concerns with two different questions: how can an agent discover high-reward strategies that require a temporally extended sequence of complex behaviors that, individually, are not rewarding? How can an agent decide whether to attempt new behaviors (to discover ones with higher reward) or continue to do the best thing it knows so far? Long story short, we can define exploitation as doing what you know will yield highest reward, and exploration as doing things you haven't done before, in the hopes of getting even higher reward. In order to explore well, we need to come up with some smart strategies to discover some better way to gain rewards. To illustrate, let us look at a classic example of exploration and exploitation trade-off. Say you want to eat in a restaurant, to exploit, you would eat at your favorite restaurant, but to explore, you would go to a new restaurant and see if it is better than your favorite one.

14.1 Multi-arm Bandits

A bandit problem is a type of simple exploration problem. The term comes from the name of a popular slot machine. We are interested in multi-arm bandits where each arm gives us different reward, and we are mainly concerned with the question of pulling which arm gives us the best reward among all arms.

Mathematically, our actions to choose are

$$\mathcal{A} = \{\text{pull}_1, \text{pull}_2, \ldots, \text{pull}_n\}$$

since we have n arms, and assume there is a true distribution of which arm gives us a higher reward, which we do not know a priori:

$$r(a_n) \sim p(r|a_n)$$

14.1.1 Defining a Bandit

Assume our reward for each arm r_i comes from a distribution $r(a_i) \sim p_{\theta_i}(r_i)$. For example, if our reward is a binary variable then we can define $p_{\theta_i}(r_i)$ as

$$p(r_i = 1) = \theta_i, \;\; p(r_i = 0) = 1 - \theta_i$$

We also know that $\theta_i \sim p(\theta)$, but we do not know anything else about the distribution.

This actually defines a meta-level POMDP. Our latent state is actually $s = [\theta_1, \ldots, \theta_n]$, which is the true parameterization of the arms' reward distribution. We also have a belief

state, which is our observation in some sense. The belief state is an estimate of the probability of getting high reward of each arm:

$$\hat{p}(\theta_1, \ldots, \theta_n)$$

To measure the goodness of exploration algorithm, we define the regret of exploration. The regret of exploration is the difference from optimal policy at time step T:

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

where the first term means in hindsight, how much reward could I have got if I had taken the best action all the way until the end, and the second term means the actual reward I have got. The difference of these two gives us the reward.

14.1.2 Optimistic Exploration

One simple way to explore is to use an optimistic exploration strategy, where we keep track of average reward $\hat{\mu}_a$ for each action a. Naturally, one way to pick an action is the greedy exploitation:

$$a = \arg\max \hat{\mu}_a$$

then to explore better, we need to add bonus to new actions too:

$$a = \arg\max \hat{\mu}_a + C\sigma_a$$

where σ_a is some quantification of uncertainty about action a.

The intuition behind this strategy is to try each arm until you are sure that it is not great. Therefore, if you think an action might be good, go on and try it for a few more times, and try something else if you are sure it is not good.

One popular method to gauge the uncertainty about an action is to use the upper confidence bound (UCB). Specifically,

$$a = \arg\max \hat{\mu}_a + \sqrt{\frac{2\ln T}{N(a)}}$$

where $N(a)$ counts the number of times that we have applied action a. Using this strategy, we can get a bound of regret $O(\log T)$.

14.1.3 Probability Matching

Recall we have a belief state model that represents our own estimate of each arm's reward parameterization:

$$\hat{p}(\theta_1, \ldots, \theta_n)$$

We can improve our belief state by keep updating it. The idea is to sample $(\theta_1, \ldots, \theta_n)$ from the distribution, and pretend that the model $(\theta_1, \ldots, \theta_n)$ is correct, then we take the optimal action and update the belief model. Then we take the action by $a = \arg\max_a \mathbb{E}_{\theta_a}[r(a)]$.

This is called posterior sampling or Thompson sampling. This method is harder to analyze theoretically, but can work very well empirically.

14.1.4 Information Gain

Say we want to determine some latent variable z, one way to decide which action to take is look at the entropy of the estimate of the prior $\mathcal{H}(\hat{p}(z))$. Intuitively, the entropy should be high if our estimate is off, and low if our estimate is accurate. By the same token, we can incorporate evidence y with this entropy so that we look at the entropy $\mathcal{H}(\hat{p}(z|y))$ in order to see if the entropy changes after y is known to us. For example, y could be the reward $r(a)$.

The **information gain** of observing y is defined as

$$IG(z, y) = \mathbb{E}_y[\mathcal{H}(\hat{p}(z)) - \mathcal{H}(\hat{p}(z|y))]$$

which is the expected decrease of entropy after observing y. Note that we do not know what y actually is, but we have some knowledge of what it might be. This is why we are taking the expected value. Typically, the information gain also depends on the action so we can have $IG(z, y|a)$. Hence,

$$IG(z, y) = \mathbb{E}_y[\mathcal{H}(\hat{p}(z)) - \mathcal{H}(\hat{p}(z|y))|a]$$

it measures how much we learn from z from action a, given the current beliefs.

In our exploration setting, the observation is the observed reward:

$$y = r(a)$$

the latent state is the parameters for model $p(r_a)$:

$$z = \theta_a$$

then the information gain of a is calculated as:

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

we also define another quantity $\Delta(a)$ that measures the expected suboptimality of a:

$$\Delta(a) = \mathbb{E}[r(a^*) - r(a)]$$

As a result we take action according to the rule

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

This rule intuitively means that we do not take an action if we are sure if it is optimal (large $\Delta(a)$), or if we cannot learn anything from applying that action (small $g(a)$).

We talked about bandits models because bandits are easier to analyze and understand. We can derive foundations for exploration methods, ane then apply these methods to more complex MDPs. Most exploration strategies require some kind of uncertainty estimation (even if it's naïve). We usually assumes some value to new information. For example, we assume unknown means good (optimism), sample means truth, and information gain means good.

Algorithm 28 Exploring with Pseudo-count

Require: Some base model $p_\theta(s)$

1: **while** not done **do**
2: Fit model $p_\theta(s)$ to all the states $\mathcal{D}$ seen so far
3: Take a step i and observe s_i
4: Fit model $p_{\theta'}(s)$ to $\mathcal{D} \cup s_i$
5: Use $p_\theta(s_i)$ and $p_{\theta'}(s')$ to estimate $\hat{N}(s)$
6: Set $r^+(s,a) = r(s,a) + \mathcal{B}(\hat{N}(s))$

14.2 Exploration in MDPs

Recall our UCB exploration policy to choose action:

$$a = \arg\max \hat{\mu}_a + \sqrt{\frac{2 \ln T}{N(a)}}$$

here $N(a)$ is the exploration bonus.

Can we apply the same idea in MDPs, which are what we work with in RL? Essentially, we can do the same exploration bonus $N(s,a)$ or $N(s)$ and add it to the reward:

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

where the bonus $N(s)$ decreases with the increase of visitation frequency, and then we use $r^+(s,a)$ instead of $r(s,a)$ in any model-free algorithm. This is a simple addition to any RL algorithm, but we need to tune the bonus weight.

14.2.1 Counting the Exploration Bonus

We count the number of times that we have encountered the state s using $N(s)$. However, in many situations such as video games or autonomous driving, we never actually see the exact same state twice. Therefore, we need to take the notion of similarity into account: we count the number of times we have encountered similar states, instead of the same states.

The idea is to fit a density model $p_\theta(s)$ or $p_\theta(s,a)$ to the states. $p_\theta(s)$ is low for very novel states, and high for states that are very similar to the states we have seen, even if it might be completely new. To design this density model, we can seek some inspirations from a simple small MDP. If we have a small MDP, then the density of visiting a state s is modeled as:

$$P(s) = \frac{N(s)}{n}$$

and if we see the same state again, this density becomes:

$$P'(s) = \frac{N(s)+1}{n+1}$$

we design our neural net density model obeying the same rule.

We devise a deep pseudo-count procedure to count the states as shown in Alg. 28 In

step 5, we solve for $\hat{N}$ using the following equations:

$$p_\theta(s_i) = \frac{\hat{N}(s_i)}{\hat{n}}$$
$$p_{\theta'}(s_i) = \frac{\hat{N}(s_i)+1}{\hat{n}+1}$$

this two equations with two unknowns, and we solve for $\hat{N}, \hat{n}$ as follows:

$$\hat{N} = \hat{n} p_\theta(s_i)$$
$$\hat{n} = \frac{1 - p_{\theta'}(s_i)}{p_{\theta'}(s_i) - p_\theta(s_i)} p_\theta(s_i)$$

These counters are able to count similar states.

14.3 Exploration with Q-functions

Recall in earlier chapters, we covered epsilon-greedy exploration strategy. This strategy is essentially taking random actions with a probability of ϵ. However, in many cases, exploring by taking random actions might not be ideal. For example, in the Atari game called SeaQuest, taking random actions by going back and forth might make the submarine run out of oxygen quickly without exploring meaningful states. Therefore, we need to explore more efficiently by sticking to one general strategy for an extended period of time.

Thus, we introduce exloring with Q-functions. Exploring with random actions (e.g., epsilon-greedy) is not efficient enough because we oscillate back and forth, so we might not go to a coherent or interesting place. However, exploring with random Q-functions make us commit to a randomized but internally consistent strategy for an entire episode, so we act coherently in the same episode.

To do this, we maintain a distribution of Q-functions. This distribution could be any Q-function with artificial Gaussian noise. Then we sample a Q-function from this distribution $p(Q)$, and act according to this function for the entire episode. Then we update $p(Q)$ and repeat. Since Q-learning is off-policy, we don't care which Q-function was used to collect data.

Using this method, we don't need to modify the original reward function since we are just doing off-policy Q-learning, but in many cases good exploration bonus functions tend to do better.

14.4 Revisiting Information Gain in MDP Exploration

In MDPs, we can also use information gain $IG(z, y|a)$ which was introduced in the multi-arm bandit problem. However, we need to figure out what information gain we are looking for exactly. First, we can find information gain about reward $r(s, a)$. However, this is not useful when the reward is sparse, so nothing meaningful could come out from this. We could also find information gain about state density $p(s)$, where we can find how much the state visitation changes after we know something. This is useful because $p(s)$ changes drastically if the information gain is big enough. We could also learn the information gain about the transition model $p(s'|s, a)$, which is good for learning the MDP itself. However, none of the

above three different settings can be calculated exactly. Thus, we need a proxy to estimate the information gain.

14.4.1 Prediction Gain

One way to approximate the information gain is to use the prediction gain:

$$\log p_{\theta'}(s) - \log_{\theta}(s)$$

which is used on state densities. This quantity takes the difference between the density before and after seeing the state s. Therefore, if the prediction gain is big, then the state s is novel.

14.4.2 Variational Information Maximization for Exploration (VIME)

This method to approximate the information gain was first introduced in [12] by Houthooft et al.. Mathematically, the information gain can be equivalently written in terms of KL divergence as:

$$D_{KL}(p(z|y)||p(z))$$

There is some quantity about the MDP we want to learn about, which in this case, without loss of generality, is the transition

$$p_{\theta}(s_{t+1}|s_t, a_t)$$

then in the parametrization of the information gain, what we want to learn about is the parameter of the quantity of interest θ. z could also be some other distributions that involve θ such as $p_{\theta}(s)$ and $p_{\theta}(r|s, a)$. The evidence y we observe is the transition. Therefore:

$$\begin{aligned} z &= \theta \\ y &= (s_t, a_t, s_{t+1}) \end{aligned}$$

Our information gain in terms of KL-divergence can be set up as

$$D_{KL}(p(\theta|h, s_{t+1}, s_t, a_t)||p(\theta|h))$$

where h is the history of all prior transitions. Therefore, intuitively, the transition we observe is more intuitive if it causes the belief over θ to change.

The idea of VIME is to use variational inference to approximate $p(\theta|h)$ since maintaining the whole history h is not feasible. So we use a distribution to approximate the history:

$$q(\theta|\phi) \simeq p(\theta|h)$$

the new distribution is parameterized by ϕ, so when we observe a new transition, we update ϕ to get ϕ'.

As you recall, we update the parameters by optimizing the variational lower bound

$$D_{KL}(q(\theta|\phi)||p(h|\theta)p(\theta)$$

and we represent $q(\theta|\phi)$ as a product of independent Gaussians of parameter distributions with mean ϕ.

After updating ϕ', we use $D_{KL}(q(\theta|\phi')||q(\theta|\phi))$ as the approximate information gain.

Algorithm 29 Pretrain and Fine Tune

Require: Demonstrations data $\mathcal{D}$

1: Collect demonstration data $\mathcal{D} = \{(s_i, a_i)\}$
2: Initialize π_θ as $\max_\theta \sum_i \log \pi_\theta(a_i|s_i)$
3: **while** not done **do**
4: Run π_θ to collect experience
5: Improve π_θ with any RL algorithm

14.5 Improving RL with Imitation

Imitation learning is simple, stable supervised learning, but it requires demonstrations, and it must address distributional shift. Furthermore, the state of the art imitation learning can only be as good as the demo. In contrast, reinforcement learning can become arbitrarily good, but it requires reward function, and must address exploration. Also, it often does not have convergence guarantees.

14.5.1 Pretrain and Finetune

Can we somehow combine the two such that we have both demonstrations data and rewards? The answer is yes. The simplest idea that is practical and works is to pretrain with imitation learning and fine tune with RL. This idea is shown in Alg. 29.

The problem with Alg. 29 is that in step 4, we might collect very bad experience due to distribution shift, so the first batch of data might be bad, thus destroying the initialization.

14.5.2 Off-policy RL

One way to mitigate the issue of forgetting the demonstrations is to use off-policy RL because off-policy RL can use any data, and if we let it use demonstrations as off-policy samples, since demonstrations are provided as data in every iteration, they are never forgotten. Furthermore, the policy can still become better than the demos, since it is not forced to mimic them. To achieve this, we could use off-policy policy gradients and off-policy Q-learning.

Recall policy gradients with importance sampling:

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

The trick here is when we collect the sum of samples, in our samples, we include both our experience and demonstration. However, it seems a little weird because in policy gradients we actually want on-policy data, so why are we including off-policy data. To answer this question, let us build up some intuition by looking at the optimal importance sampling distribution. Say we want to estimate $\mathbb{E}_{p(x)}[f(x)]$ using importance sampling:

$$\mathbb{E}_{p(x)}[f(x)] \simeq \frac{1}{N} \sum_i \frac{p(x_i)}{q(x_i)} f(x_i)$$

and it can be proven that

$$q \propto p(x)|f(x)|$$

gives us the smallest variance. Therefore, by taking off-policy demonstration samples, we are motivating importance sampling to use distributions that have higher reward than current policy in order to get closer to the optimal distribution. To construct the sampling distribution, first we need to figure out which distribution the demonstrations come from. First, we could use supervised behavioral cloning to learn π_{demo}. If the demonstration is from multiple distribution, we could instead use **fusion distribution** by

$$q(x) = \frac{1}{M} \sum_i q_i(x)$$

14.5.3 Q-learning with Demonstrations

Since Q-learning is already off-policy, there is actually no need to bother with importance weights like we did in policy gradients. Therefore, one simple solution is just drop demonstrations into the replay buffer.

We can modify Alg. 12 slightly such that we initialize $\mathcal{B}$ with some demonstrations data, and then do the same things as before.

14.5.4 Imitation as an Auxiliary Loss Function

Recall the imitation learning maximum likelihood training objective is

$$\sum_{(s,a)\sim\mathcal{D}_{demo}} \log \pi_\theta(a|s)$$

and the RL objective is

$$\mathbb{E}_{\pi_\theta}[r(s,a)]$$

to combine the two, we can come up with a hybrid objective:

$$\mathbb{E}_{\pi_\theta}[r(s,a)] + \lambda \sum_{(s,a)\sim\mathcal{D}_{demo}} \log \pi_\theta(a|s)$$

Bibliography

[1] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 627–635.

[2] P. de Haan, D. Jayaraman, and S. Levine, "Causal confusion in imitation learning," *arXiv preprint arXiv:1905.11979*, 2019.

[3] P. Thomas, "Bias in natural actor-critic algorithms," in *International conference on machine learning*, 2014, pp. 441–448.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[5] R. Munos, T. Stepleton, A. Harutyunyan, and M. Bellemare, "Safe and efficient off-policy reinforcement learning," in *Advances in Neural Information Processing Systems*, 2016, pp. 1054–1062.

[6] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine, "Continuous deep q-learning with model-based acceleration," in *International Conference on Machine Learning*, 2016, pp. 2829–2838.

[7] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[8] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[9] X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang, "Deep learning for real-time atari game play using offline monte-carlo tree search planning," in *Advances in neural information processing systems*, 2014, pp. 3338–3346.

[10] M. Watter, J. Springenberg, J. Boedecker, and M. Riedmiller, "Embed to control: A locally linear latent dynamics model for control from raw images," in *Advances in neural information processing systems*, 2015, pp. 2746–2754.

[11] M. Zhang, S. Vikram, L. Smith, P. Abbeel, M. J. Johnson, and S. Levine, "Solar: Deep structured latent representations for model-based reinforcement learning," *arXiv preprint arXiv:1808.09105*, 2018.

[12] R. Houthooft, X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel, "Vime: Variational information maximizing exploration," in *Advances in Neural Information Processing Systems*, 2016, pp. 1109–1117.