

35 Reinforcement learning

This chapter is co-authored with Lihong Li.

35.1 Introduction

Reinforcement learning or **RL** is a paradigm of learning where an agent sequentially interacts with an initially unknown environment. The interaction typically results in a **trajectory**, or multiple trajectories. Let $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, \dots, s_T)$ be a trajectory of length T , consisting of a sequence of states s_t , actions a_t , and rewards r_t .¹ The goal of the agent is to optimize her action-selection policy, so that the discounted cumulative reward, $G_0 \triangleq \sum_{t=0}^{T-1} \gamma^t r_t$, is maximized for some given **discount factor** $\gamma \in [0, 1]$.

In general, G_0 is a random variable. We will focus on maximizing its expectation, inspired by the maximum expected utility principle (Section 34.1.3), but note other possibilities such as **conditional value at risk**² that can be more appropriate in risk-sensitive applications.

We will focus on the Markov decision process, where the generative model for the trajectory τ can be factored into single-step models. When these model parameters are known, solving for an optimal policy is called **planning** (see Section 34.6); otherwise, RL algorithms may be used to obtain an optimal policy from trajectories, a process called **learning**.

In **model-free RL**, we try to learn the policy without explicitly representing and learning the models, but directly from the trajectories. In **model-based RL**, we first learn a model from the trajectories, and then use a planning algorithm on the learned model to solve for the policy. See Figure 35.1 for an overview. This chapter will introduce some of the key concepts and techniques, and will mostly follow the notation from [SB18]. More details can be found in textbooks such as [Sze10; SB18; Ber19; Aga+21a; Mey22; Aga+22], and reviews such as [WO12; Aru+17; FL+18; Li18].

35.1.1 Overview of methods

In this section, we give a brief overview of how to compute optimal policies when the MDP model is not known. Instead, the agent interacts with the environment and learns from the observed

1. Note that the time starts at 0 here, while it starts at 1 when we discuss bandits (Section 34.4). Our choices of notation are to be consistent with conventions in respective literature.

2. The conditional value at risk, or CVaR, is the expected reward conditioned on being in the worst 5% (say) of samples. See [Cho+15] for an example application in RL.

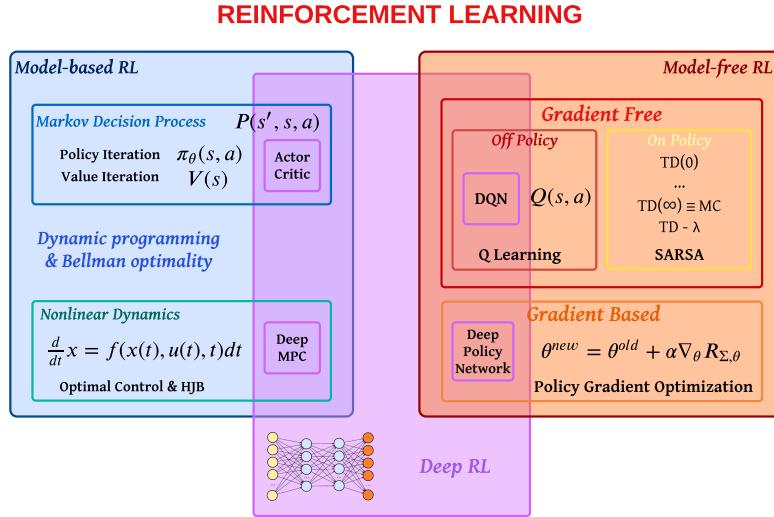


Figure 35.1: Overview of RL methods. Abbreviations: DQN = Deep Q network (Section 35.2.6); MPC = Model Predictive Control (Section 35.4); HJB = Hamilton-Jacobi-Bellman equation; TD = temporal difference learning (Section 35.2.2). Adapted from a slide by Steve Brunton.

Method	Functions learned	On/Off	Section
SARSA	$Q(s, a)$	On	Section 35.2.4
Q -learning	$Q(s, a)$	Off	Section 35.2.5
REINFORCE	$\pi(a s)$	On	Section 35.3.2
A2C	$\pi(a s)$, $V(s)$	On	Section 35.3.3.1
TRPO/PPO	$\pi(a s)$, $A(s, a)$	On	Section 35.3.4
DDPG	$a = \pi(s)$, $Q(s, a)$	Off	Section 35.3.5
Soft actor-critic	$\pi(a s)$, $Q(s, a)$	Off	Section 35.6.1
Model-based RL	$p(s' s, a)$	Off	Section 35.4

Table 35.1: Summary of some popular methods for RL. On/off refers to on-policy vs off-policy methods.

trajectories. This is the core focus of RL. We will go into more details into later sections, but first provide this roadmap.

We may categorize RL methods by the quantity the agent represents and learns: value function, policy, and model; or by how actions are selected: on-policy (actions must be selected by the agent's current policy), and off-policy. Table 35.1 lists a few representative examples. More details are given in the subsequent sections. We will also discuss at greater depth two important topics of off-policy learning and inference-based control in Sections 35.5 and 35.6.

35.1.2 Value-based methods

In a value-based method, we often try to learn the optimal Q -function from experience, and then derive a policy from it using Equation (34.84). Typically, a function approximator (e.g., a neural network), Q_w , is used to represent the Q -function, which is trained iteratively. Given a transition (s, a, r, s') , we define the **temporal difference** (also called the **TD error**) as

$$r + \gamma \max_{a'} Q_w(s', a') - Q_w(s, a)$$

Clearly, the expected TD error is the Bellman error evaluated at (s, a) . Therefore, if $Q_w = Q_*$, the TD error is 0 on average by Bellman's optimality equation. Otherwise, the error provides a signal for the agent to change w to make $Q_w(s, a)$ closer to $R(s, a) + \gamma \max_{a'} Q_w(s', a')$. The update on Q_w is based on a target that is computed using Q_w . This kind of update is known as **bootstrapping** in RL, and should not be confused with the statistical bootstrap (Section 3.3.2). Value based methods such as **Q-learning** and **SARSA** are discussed in Section 35.2.

35.1.3 Policy search methods

In **policy search**, we try to directly maximize $J(\pi_\theta)$ wrt the policy parameter θ . If $J(\pi_\theta)$ is differentiable wrt θ , we can use stochastic gradient ascent to optimize θ , which is known as **policy gradient**, as described in Section 35.3.1. The basic idea is to perform **Monte Carlo rollouts**, in which we sample trajectories by interacting with the environment, and then use the score function estimator (Section 6.3.4) to estimate $\nabla_\theta J(\pi_\theta)$. Here, $J(\pi_\theta)$ is defined as an expectation whose distribution depends on θ , so it is invalid to swap ∇ and \mathbb{E} in computing the gradient, and the score function estimator can be used instead. An example of policy gradient is **REINFORCE**.

Policy gradient methods have the advantage that they provably converge to a local optimum for many common policy classes, whereas Q -learning may diverge when approximation is used (Section 35.5.3). In addition, policy gradient methods can easily be applied to continuous action spaces, since they do not need to compute $\text{argmax}_a Q(s, a)$. Unfortunately, the score function estimator for $\nabla_\theta J(\pi_\theta)$ can have a very high variance, so the resulting method can converge slowly.

One way to reduce the variance is to learn an approximate value function, $V_w(s)$, and to use it as a baseline in the score function estimator. We can learn $V_w(s)$ using one of the value function methods similar to Q -learning. Alternatively, we can learn an advantage function, $A_w(s, a)$, and use it to estimate the gradient. These policy gradient variants are called **actor critic** methods, where the actor refers to the policy π_θ and the critic refers to V_w or A_w . See Section 35.3.3 for details.

35.1.4 Model-based RL

Value-based methods, such as Q-learning, and policy search methods, such as policy gradient, can be very **sample inefficient**, which means they may need to interact with the environment many times before finding a good policy. If an agent has prior knowledge of the MDP model, it can be more sample efficient to first learn the model, and then compute an optimal (or near-optimal) policy of the model without having to interact with the environment any more.

This approach is called **model-based RL**. The first step is to learn the MDP model including the $p_T(s'|s, a)$ and $R(s, a)$ functions, e.g., using DNNs. Given a collection of (s, a, r, s') tuples, such a model can be learned using standard supervised learning methods. The second step can be done

by running an RL algorithm on synthetic experiences generated from the model, or by running a planning algorithm on the model directly (Section 34.6). In practice, we often interleave the model learning and planning phases, so we can use the partially learned policy to decide what data to collect. We discuss model-based RL in more detail in Section 35.4.

35.1.5 Exploration-exploitation tradeoff

A fundamental problem in RL with unknown transition and reward models is to decide between choosing actions that the agent knows will yield high reward, or choosing actions whose reward is uncertain, but which may yield information that helps the agent get to parts of state-action space with even higher reward. This is called the **exploration-exploitation tradeoff**, which has been discussed in the simpler contextual bandit setting in Section 34.4. The literature on efficient exploration is huge. In this section, we briefly describe several representative techniques.

35.1.5.1 ϵ -greedy

A common heuristic is to use an **ϵ -greedy** policy π_ϵ , parameterized by $\epsilon \in [0, 1]$. In this case, we pick the greedy action wrt the current model, $a_t = \operatorname{argmax}_a \hat{R}_t(s_t, a)$ with probability $1 - \epsilon$, and a random action with probability ϵ . This rule ensures the agent's continual exploration of all state-action combinations. Unfortunately, this heuristic can be shown to be suboptimal, since it explores every action with at least a constant probability $\epsilon/|\mathcal{A}|$.

35.1.5.2 Boltzmann exploration

A source of inefficiency in the ϵ -greedy rule is that exploration occurs uniformly over all actions. The **Boltzmann policy** can be more efficient, by assigning higher probabilities to explore more promising actions:

$$\pi_\tau(a|s) = \frac{\exp(\hat{R}_t(s_t, a)/\tau)}{\sum_{a'} \exp(\hat{R}_t(s_t, a')/\tau)} \quad (35.1)$$

where $\tau > 0$ is a temperature parameter that controls how entropic the distribution is. As τ gets close to 0, π_τ becomes close to a greedy policy. On the other hand, higher values of τ will make $\pi(a|s)$ more uniform, and encourage more exploration. Its action selection probabilities can be much “smoother” with respect to changes in the reward estimates than ϵ -greedy, as illustrated in Table 35.2.

35.1.5.3 Upper confidence bounds and Thompson sampling

The upper confidence bound (UCB) (Section 34.4.5) and Thompson sampling (Section 34.4.6) approaches may also be extended to MDPs. In contrast to the contextual bandit case, where the only uncertainty is in the reward function, here we must also take into account uncertainty in the transition probabilities.

As in the bandit case, the UCB approach requires to estimate an upper confidence bound for all actions' Q -values in the current state, and then take the action with the highest UCB score. One way to obtain UCBs of the Q -values is to use **count-based exploration**, where we learn the optimal

$\hat{R}(s, a_1)$	$\hat{R}(s, a_2)$	$\pi_\epsilon(a s_1)$	$\pi_\epsilon(a s_2)$	$\pi_\tau(a s_1)$	$\pi_\tau(a s_2)$
1.00	9.00	0.05	0.95	0.00	1.00
4.00	6.00	0.05	0.95	0.12	0.88
4.90	5.10	0.05	0.95	0.45	0.55
5.05	4.95	0.95	0.05	0.53	0.48
7.00	3.00	0.95	0.05	0.98	0.02
8.00	2.00	0.95	0.05	1.00	0.00

Table 35.2: Comparison of ϵ -greedy policy (with $\epsilon = 0.1$) and Boltzmann policy (with $\tau = 1$) for a simple MDP with 6 states and 2 actions. Adapted from Table 4.1 of [GK19].

Q -function with an **exploration bonus** added to the reward in a transition (s, a, r, s') :

$$\tilde{r} = r + \alpha / \sqrt{N_{s,a}} \quad (35.2)$$

where $N_{s,a}$ is the number of times action a has been taken in state s , and $\alpha \geq 0$ is a weighting term that controls the degree of exploration. This is the approach taken by the **MBIE-EB** method [SL08] for finite-state MDPs, and in the generalization to continuous-state MDPs through the use of hashing [Bel+16]. Other approaches also explicitly maintain uncertainty in state transition probabilities, and use that information to obtain UCBs. Examples are **MBIE** [SL08], **UCRL2** [JOA10], and **UCBVI** [AOM17], among many others.

Thompson sampling can be similarly adapted, by maintaining the posterior distribution of the reward and transition model parameters. In finite-state MDPs, for example, the transition model is a categorical distribution conditioned on the state. We may use the conjugate prior of Dirichlet distributions (Section 3.4) for the transition model, so that the posterior distribution can be conveniently computed and sampled from. More details on this approach are found in [Rus+18].

Both UCB and Thompson sampling methods have been shown to yield efficient exploration with provably strong regret bounds (Section 34.4.7) [JOA10], or related PAC bounds [SLL09; DLB17], often under necessary assumptions such as finiteness of the MDPs. In practice, these methods may be combined with function approximation like neural networks and implemented approximately.

35.1.5.4 Optimal solution using Bayes-adaptive MDPs

The Bayes optimal solution to the exploration-exploitation tradeoff can be computed by formulating the problem as a special kind of POMDP known as a **Bayes-adaptive MDP** or **BAMDP** [Duf02]. This extends the Gittins index approach in Section 34.4.4 to the MDP setting.

In particular, a BAMDP has a **belief state** space, \mathcal{B} , representing uncertainty about the reward model $p_R(r|s, a, s')$ and transition model $p_T(s'|s, a)$. The transition model on this augmented MDP can be written as follows:

$$T^+(s_{t+1}, b_{t+1}|s_t, b_t, a_t, r_t) = T^+(s_{t+1}|s_t, a_t, b_t)T^+(b_{t+1}|s_t, a_t, r_t, s_{t+1}) \quad (35.3)$$

$$= \mathbb{E}_{b_t}[T(s_{t+1}|s_t, a_t)] \times \mathbb{I}(b_{t+1} = p(R, T|\mathbf{h}_{t+1})) \quad (35.4)$$

where $\mathbb{E}_{b_t}[T(s_{t+1}|s_t, a_t)]$ is the posterior predictive distribution over next states, and $p(R, T|\mathbf{h}_{t+1})$ is the new belief state given $\mathbf{h}_{t+1} = (s_{1:t+1}, a_{1:t+1}, r_{1:t+1})$, which can be computed using Bayes' rule.

Similarly, the reward function for the augmented MDP is given by

$$R^+(r|s_t, b_t, a_t, s_{t+1}, b_{t+1}) = \mathbb{E}_{b_{t+1}} [R(s_t, a_t, s_{t+1})] \quad (35.5)$$

For small problems, we can solve the resulting augmented MDP optimally. However, in general this is computationally intractable. [Gha+15] surveys many methods to solve it more efficiently. For example, [KN09] develop an algorithm that behaves similarly to Bayes optimal policies, except in a provably small number of steps; [GSD13] propose an approximate method based on Monte Carlo rollouts. More recently, [Zin+20] propose an approximate method based on meta-learning (Section 19.6.4), in which they train a (model-free) policy for multiple related tasks. Each task is represented by a task embedding vector m , which is inferred from \mathbf{h}_t using a VAE (Section 21.2). The posterior $p(m|\mathbf{h}_t)$ is used as a proxy for the belief state b_t , and the policy is trained to perform well given s_t and b_t . At test time, the policy is applied to the incrementally computed belief state; this allows the method to infer what kind of task this is, and then to use a pre-trained policy to quickly solve it.

35.2 Value-based RL

In this section, we assume the agent has access to samples from p_T and p_R by interacting with the environment. We will show how to use these samples to learn optimal Q -functions from which we can derive optimal policies.

35.2.1 Monte Carlo RL

Recall that $Q_\pi(s, a) = \mathbb{E}[G_t | s_t = s, a_t = a]$ for any t . A simple way to estimate this is to take action a , and then sample the rest of the trajectory according to π , and then compute the average sum of discounted rewards. The trajectory ends when we reach a terminal state, if the task is episodic, or when the discount factor γ^t becomes negligibly small, whichever occurs first. This is the **Monte Carlo estimation** of the value function.

We can use this technique together with policy iteration (Section 34.6.2) to learn an optimal policy. Specifically, at iteration k , we compute a new, improved policy using $\pi_{k+1}(s) = \text{argmax}_a Q_k(s, a)$, where Q_k is approximated using MC estimation. This update can be applied to all the states visited on the sampled trajectory. This overall technique is called **Monte Carlo control**.

To ensure this method converges to the optimal policy, we need to collect data for every (state, action) pair, at least in the tabular case, since there is no generalization across different values of $Q(s, a)$. One way to achieve this is to use an ϵ -greedy policy. Since this is an on-policy algorithm, the resulting method will converge to the optimal ϵ -soft policy, as opposed to the optimal policy. It is possible to use importance sampling to estimate the value function for the optimal policy, even if actions are chosen according to the ϵ -greedy policy. However, it is simpler to just gradually reduce ϵ .

35.2.2 Temporal difference (TD) learning

The Monte Carlo (MC) method in Section 35.2.1 results in an estimator for $Q_\pi(s, a)$ with very high variance, since it has to unroll many trajectories, whose returns are a sum of many random rewards generated by stochastic state transitions. In addition, it is limited to episodic tasks (or finite horizon

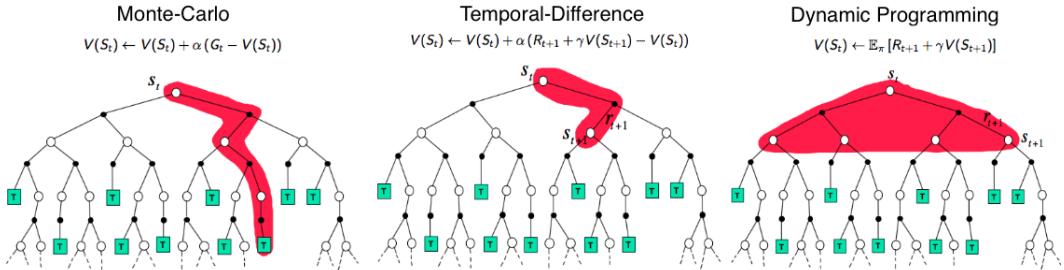


Figure 35.2: Backup diagrams of $V(s_t)$ for Monte Carlo, temporal difference, and dynamic programming updates of the state-value function. Used with kind permission of Andy Barto.

truncation of continuing tasks), since it must unroll to the end of the episode before each update step, to ensure it reliably estimates the long term return.

In this section, we discuss a more efficient technique called **temporal difference** or **TD** learning [Sut88]. The basic idea is to incrementally reduce the Bellman error for sampled states or state-actions, based on transitions instead of a long trajectory. More precisely, suppose we are to learn the value function V_π for a fixed policy π . Given a state transition (s, a, r, s') where $a \sim \pi(s)$, we change the estimate $V(s)$ so that it moves towards the bootstrapping target (Section 35.1.2)

$$V(s_t) \leftarrow V(s_t) + \eta [r_t + \gamma V(s_{t+1}) - V(s_t)] \quad (35.6)$$

where η is the learning rate. The term multiplied by η above is known as the **TD error**. A more general form of TD update for parametric value function representations is

$$\mathbf{w} \leftarrow \mathbf{w} + \eta [r_t + \gamma V_{\mathbf{w}}(s_{t+1}) - V_{\mathbf{w}}(s_t)] \nabla_{\mathbf{w}} V_{\mathbf{w}}(s_t) \quad (35.7)$$

of which Equation (35.6) is a special case. The TD update rule for learning Q_π is similar.

It can be shown that TD learning in the tabular case, Equation (35.6), converges to the correct value function, under proper conditions [Ber19]. However, it may diverge when approximation is used (Equation (35.7)), an issue we will discuss further in Section 35.5.3.

The potential divergence of TD is also consistent with the fact that Equation (35.7) is not SGD (Section 6.3.1) on any objective function, despite a very similar form. Instead, it is an example of **bootstrapping**, in which the estimate, $V_{\mathbf{w}}(s_t)$, is updated to approach a target, $r_t + \gamma V_{\mathbf{w}}(s_{t+1})$, which is defined by the value function estimate itself. This idea is shared by DP methods like value iteration, although they rely on the complete MDP model to compute an exact Bellman backup. In contrast, TD learning can be viewed as using sampled transitions to approximate such backups. An example of non-bootstrapping approach is the Monte Carlo estimation in the previous section. It samples a complete trajectory, rather than individual transitions, to perform an update, and is often much less efficient. Figure 35.2 illustrates the difference between MC, TD, and DP.

35.2.3 TD learning with eligibility traces

A key difference between TD and MC is the way they estimate returns. Given a trajectory $\tau = (s_0, a_0, r_0, s_1, \dots, s_T)$, TD estimates the return from state s_t by one-step lookahead, $G_{t:t+1} = r_t +$

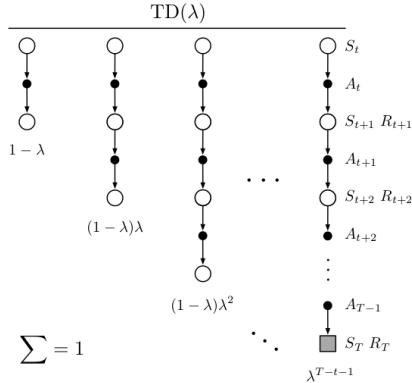


Figure 35.3: The backup diagram for $TD(\lambda)$. Standard TD learning corresponds to $\lambda = 0$, and standard MC learning corresponds to $\lambda = 1$. From Figure 12.1 of [SB18]. Used with kind permission of Richard Sutton.

$\gamma V(s_{t+1})$, where the return from time $t + 1$ is replaced by its value function estimate. In contrast, MC waits until the end of the episode or until T is large enough, then uses the estimate $G_{t:T} = r_t + \gamma r_{t+1} + \dots + \gamma^{T-t-1} r_{T-1}$. It is possible to interpolate between these by performing an n -step rollout, and then using the value function to approximate the return for the rest of the trajectory, similar to heuristic search (Section 35.4.1.1). That is, we can use the n -step estimate

$$G_{t:t+n} = r_t + \gamma r_{t+1} + \dots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(s_{t+n}) \quad (35.8)$$

The corresponding n -step version of the TD update becomes

$$V(s_t) \leftarrow V(s_t) + \eta [G_{t:t+n} - V(s_t)] \quad (35.9)$$

Rather than picking a specific lookahead value, n , we can take a weighted average of all possible values, with a single parameter $\lambda \in [0, 1]$, by using

$$G_t^\lambda \triangleq (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} \quad (35.10)$$

This is called the **λ -return**. The coefficient of $1 - \lambda = (1 + \lambda + \lambda^2 + \dots)^{-1}$ in the front ensures this is a convex combination of n -step returns. See Figure 35.3 for an illustration.

An important benefit of using the geometric weighting in Equation (35.10) is that the corresponding TD learning update can be efficiently implemented, through the use of **eligibility traces**, even though G_t^λ is a sum of infinitely many terms. The method is called **TD(λ)**, and can be combined with many algorithms to be studied in the rest of the chapter. See [SB18] for a detailed discussion.

35.2.4 SARSA: on-policy TD control

TD learning is for policy evaluation, as it estimates the value function for a fixed policy. In order to find an optimal policy, we may use the algorithm as a building block inside generalized policy

iteration (Section 34.6.2). In this case, it is more convenient to work with the action-value function, Q , and a policy π that is greedy with respect to Q . The agent follows π in every step to choose actions, and upon a transition (s, a, r, s') the TD update rule is

$$Q(s, a) \leftarrow Q(s, a) + \eta [r + \gamma Q(s', a') - Q(s, a)] \quad (35.11)$$

where $a' \sim \pi(s')$ is the action the agent will take in state s' . After Q is updated (for policy evaluation), π also changes accordingly as it is greedy with respect to Q (for policy improvement). This algorithm, first proposed by [RN94], was further studied and renamed to **SARSA** by [Sut96]; the name comes from its update rule that involves an augmented transition (s, a, r, s', a') .

In order for SARSA to converge to Q_* , every state-action pair must be visited infinitely often, at least in the tabular case, since the algorithm only updates $Q(s, a)$ for (s, a) that it visits. One way to ensure this condition is to use a “greedy in the limit with infinite exploration” (**GLIE**) policy. An example is the ϵ -greedy policy, with ϵ vanishing to 0 gradually. It can be shown that SARSA with a GLIE policy will converge to Q_* and π_* [Sin+00].

35.2.5 Q-learning: off-policy TD control

SARSA is an **on-policy** algorithm, which means it learns the Q -function for the policy it is currently using, which is typically not the optimal policy (except in the limit for a GLIE policy). However, with a simple modification, we can convert this to an **off-policy** algorithm that learns Q_* , even if a suboptimal policy is used to choose actions.

The idea is to replace the sampled next action $a' \sim \pi(s')$ in Equation (35.11) with a greedy action in s' : $a' = \operatorname{argmax}_b Q(s', b)$. This results in the following update when a transition (s, a, r, s') happens

$$Q(s, a) \leftarrow Q(s, a) + \eta \left[r + \gamma \max_b Q(s', b) - Q(s, a) \right] \quad (35.12)$$

This is the update rule of **Q-learning** for the tabular case [WD92]. The extension to work with function approximation can be done in a way similar to Equation (35.7). Since it is off-policy, the method can use (s, a, r, s') triples coming from any data source, such as older versions of the policy, or log data from an existing (non-RL) system. If every state-action pair is visited infinitely often, the algorithm provably converges to Q_* in the tabular case, with properly decayed learning rates [Ber19]. Algorithm 35.1 gives a vanilla implementation of Q-learning with ϵ -greedy exploration.

35.2.5.1 Example

Figure 35.4 gives an example of Q-learning applied to the simple 1d grid world from Figure 34.13, using $\gamma = 0.9$. We show the Q -function at the start and end of each episode, after performing actions chosen by an ϵ -greedy policy. We initialize $Q(s, a) = 0$ for all entries, and use a step size of $\eta = 1$, so the update becomes $Q_*(s, a) = r + \gamma Q_*(s', a_*)$, where $a_* = \downarrow$ for all states.

35.2.5.2 Double Q-learning

Standard Q-learning suffers from a problem known as the **optimizer’s curse** [SW06], or the **maximization bias**. The problem refers to the simple statistical inequality, $\mathbb{E} [\max_a X_a] \geq \max_a \mathbb{E} [X_a]$,

Algorithm 35.1: Q-learning with ϵ -greedy exploration

```

1 Initialize value function parameters  $w$ 
2 repeat
3   Sample starting state  $s$  of new episode
4   repeat
5     Sample action  $a = \begin{cases} \text{argmax}_b Q_w(s, b), & \text{with probability } 1 - \epsilon \\ \text{random action}, & \text{with probability } \epsilon \end{cases}$ 
6     Observe state  $s'$ , reward  $r$ 
7     Compute the TD error:  $\delta = r + \gamma \max_{a'} Q_w(s', a') - Q_w(s, a)$ 
8      $w \leftarrow w + \eta \delta \nabla_w Q_w(s, a)$ 
9      $s \leftarrow s'$ 
10    until state  $s$  is terminal
11 until converged

```

for a set of random variables $\{X_a\}$. Thus, if we pick actions greedily according to their random scores $\{X_a\}$, we might pick a wrong action just because random noise makes it appealing.

Figure 35.5 gives a simple example of how this can happen in an MDP. The start state is A. The right action gives a reward 0 and terminates the episode. The left action also gives a reward of 0, but then enters state B, from which there are many possible actions, with rewards drawn from $\mathcal{N}(-0.1, 1.0)$. Thus the expected return for any trajectory starting with the left action is -0.1 , making it suboptimal. Nevertheless, the RL algorithm may pick the left action due to the maximization bias making B appear to have a positive value.

One solution to avoid the maximization bias is to use two separate Q -functions, Q_1 and Q_2 , one for selecting the greedy action, and the other for estimating the corresponding Q -value. In particular, upon seeing a transition (s, a, r, s') , we perform the following update

$$Q_1(s, a) \leftarrow Q_1(s, a) + \eta \left[r + \gamma Q_2(s', \underset{a'}{\text{argmax}} Q_1(s', a')) - Q_1(s, a) \right] \quad (35.13)$$

and may repeat the same update but with the roles of Q_1 and Q_2 swapped. This technique is called **double Q-learning** [Has10]. Figure 35.5 shows the benefits of the algorithm over standard Q-learning in a toy problem.

35.2.6 Deep Q-network (DQN)

When function approximation is used, Q-learning may be hard to use in practice due to instability problems. Here, we will describe two important heuristics, popularized by the **deep Q-network** or **DQN** work [Mni+15], which was able to train agents to outperform humans at playing Atari games, using CNN-structured Q -networks.

The first technique, originally proposed in [Lin92], is to leverage an **experience replay** buffer, which stores the most recent (s, a, r, s') transition tuples. In contrast to standard Q-learning which updates the Q -function when a new transition occurs, the DQN agent also performs additional updates using transitions sampled from the buffer. This modification has two advantages. First, it

	Q-function episode start		Episode	Time Step	Action	(s, a, r, s')	$r + \gamma Q^*(s', a)$	Q-function episode end
Q_1	UP DOWN		1	1	\downarrow	$(S_1, D, 0, S_2)$	$0 + 0.9 \times 0 = 0$	S_1
	S_1		1	2	\uparrow	$(S_2, U, 0, S_1)$	$0 + 0.9 \times 0 = 0$	S_2
	S_2		1	3	\downarrow	$(S_1, D, 0, S_1)$	$0 + 0.9 \times 0 = 0$	S_1
	S_1		1	4	\downarrow	$(S_2, U, 0, S_1)$	$0 + 0.9 \times 0 = 0$	S_2
	S_3		1	5	\downarrow	$(S_3, D, 1, S_{T2})$	1	S_3
Q_2	UP DOWN		2	1	\downarrow	$(S_1, D, 0, S_2)$	$0 + 0.9 \times 0 = 0$	S_1
	S_1		2	2	\downarrow	$(S_2, D, 0, S_3)$	$0 + 0.9 \times 1 = 0.9$	S_2
	S_2		2	3	\downarrow	$(S_3, D, 0, S_{T2})$	1	S_3
Q_3	UP DOWN		3	1	\downarrow	$(S_1, D, 0, S_2)$	$0 + 0.9 \times 0.9 = 0.81$	S_1
	S_1		3	2	\downarrow	$(S_2, D, 0, S_3)$	$0 + 0.9 \times 1 = 0.9$	S_2
	S_2		3	3	\uparrow	$(S_3, D, 0, S_2)$	$0 + 0.9 \times 0.9 = 0.81$	S_3
	S_3		3	4	\downarrow	$(S_2, D, 0, S_3)$	$0 + 0.9 \times 1 = 0.9$	
	S_1		3	5	\downarrow	$(S_3, D, 0, S_{T2})$	1	
Q_4	UP DOWN		4	1	\downarrow	$(S_1, D, 0, S_2)$	$0 + 0.9 \times 0.9 = 0.81$	S_1
	S_1		4	2	\uparrow	$(S_2, U, 0, S_1)$	$0 + 0.9 \times 0.81 = 0.73$	S_2
	S_2		4	3	\downarrow	$(S_1, D, 0, S_2)$	$0 + 0.9 \times 0.9 = 0.81$	S_1
	S_1		4	4	\uparrow	$(S_2, U, 0, S_1)$	$0 + 0.9 \times 0.81 = 0.73$	S_2
	S_2		4	5	\downarrow	$(S_1, D, 0, S_2)$	$0 + 0.9 \times 0.9 = 0.81$	S_1
	S_1		4	6	\downarrow	$(S_2, D, 0, S_3)$	$0 + 0.9 \times 1 = 0.9$	
	S_2		4	7	\downarrow	$(S_2, D, 0, S_3)$	1	
Q_5	UP DOWN		5	1	\uparrow	$(S_1, U, 0, S_{T2})$	0	S_1
	S_1		5	2	\downarrow	$(S_2, D, 0, S_3)$	$0 + 0.9 \times 0.9 = 0.81$	S_2
	S_2		5	3	\downarrow	$(S_3, D, 1, S_{T2})$	1	S_3

Figure 35.4: Illustration of Q learning for the 1d grid world in Figure 34.13 using ϵ -greedy exploration. At the end of episode 1, we make a transition from S_3 to S_{T2} and get a reward of $r = 1$, so we estimate $Q(S_3, \downarrow) = 1$. In episode 2, we make a transition from S_2 to S_3 , so S_2 gets incremented by $\gamma Q(S_3, \downarrow) = 0.9$. Adapted from Figure 3.3 of [GK19].

improves data efficiency as every transition can be used multiple times. Second, it improves stability in training, by reducing the correlation of the data samples that the network is trained on.

The second idea to improve stability is to regress the Q -network to a “frozen” **target network** computed at an earlier iteration, rather than trying to chase a constantly moving target. Specifically, we maintain an extra, frozen copy of the Q -network, $Q_{\mathbf{w}-}$, of the same structure as $Q_{\mathbf{w}}$. This new Q -network is to compute bootstrapping targets for training $Q_{\mathbf{w}}$, in which the loss function is

$$\mathcal{L}^{\text{DQN}}(\mathbf{w}) = \mathbb{E}_{(s, a, r, s') \sim U(\mathcal{D})} \left[(r + \gamma \max_{a'} Q_{\mathbf{w}-}(s', a') - Q_{\mathbf{w}}(s, a))^2 \right] \quad (35.14)$$

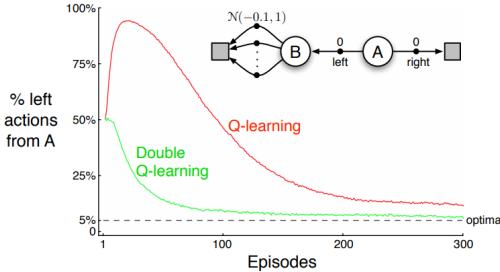


Figure 35.5: Comparison of Q -learning and double Q -learning on a simple episodic MDP using ϵ -greedy action selection with $\epsilon = 0.1$. The initial state is A , and squares denote absorbing states. The data are averaged over 10,000 runs. From Figure 6.5 of [SB18]. Used with kind permission of Richard Sutton.

where $U(\mathcal{D})$ is a uniform distribution over the replay buffer \mathcal{D} . We then periodically set $\mathbf{w}^- \leftarrow \mathbf{w}$, usually after a few episodes. This approach is an instance of **fitted value iteration** [SB18].

Various improvements to DQN have been proposed. One is **double DQN** [HGS16], which uses the double learning technique (Section 35.2.5.2) to remove the maximization bias. The second is to replace the uniform distribution in Equation (35.14) with one that favors more important transition tuples, resulting in the use of **prioritized experience replay** [Sch+16a]. For example, we can sample transitions from \mathcal{D} with probability $p(s, a, r, s') \propto (|\delta| + \varepsilon)^\eta$, where δ is the corresponding TD error (under the current Q -function), $\varepsilon > 0$ a hyperparameter to ensure every experience is chosen with nonzero probability, and $\eta \geq 0$ controls the “inverse temperature” of the distribution (so $\eta = 0$ corresponds to uniform sampling). The third is to learn a value function $V_{\mathbf{w}}$ and an advantage function $A_{\mathbf{w}}$, with shared parameter \mathbf{w} , instead of learning $Q_{\mathbf{w}}$. The resulting **dueling DQN** [Wan+16] is shown to be more sample efficient, especially when there are many actions with similar Q -values.

The **rainbow** method [Hes+18] combines all three improvements, as well as others, including multi-step returns (Section 35.2.3), distributional RL [BDM17] (which predicts the distribution of returns, not just the expected return), and noisy nets [For+18b] (which adds random noise to the network weights to encourage exploration). It produces state-of-the-art results on the Atari benchmark.

35.3 Policy-based RL

In the previous section, we considered methods that estimate the action-value function, $Q(s, a)$, from which we derive a policy, which may be greedy or softmax. However, these methods have three main disadvantages: (1) they can be difficult to apply to continuous action spaces; (2) they may diverge if function approximation is used; and (3) the training of Q , often based on TD-style updates, is not directly related to the expected return garnered by the learned policy.

In this section, we discuss **policy search** methods, which directly optimize the parameters of the policy so as to maximize its expected return. However, we will see that these methods often benefit from estimating a value or advantage function to reduce the variance in the policy search process.

35.3.1 The policy gradient theorem

We start by defining the objective function for policy learning, and then derive its gradient. We consider the episodic case. A similar result can be derived for the continuing case with the average reward criterion [SB18, Sec 13.6].

We define the objective to be the expected return of a policy, which we aim to maximize:

$$J(\pi) \triangleq \mathbb{E}_{p_0, \pi} [G_0] = \mathbb{E}_{p_0(s_0)} [V_\pi(s_0)] = \mathbb{E}_{p_0(s_0)\pi(a_0|s_0)} [Q_\pi(s_0, a_0)] \quad (35.15)$$

We consider policies π_θ parameterized by θ , and compute the gradient of Equation (35.15) wrt θ :

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{p_0(s_0)} \left[\nabla_\theta \left(\sum_{a_0} \pi_\theta(a_0|s_0) Q_{\pi_\theta}(s_0, a_0) \right) \right] \quad (35.16)$$

$$= \mathbb{E}_{p_0(s_0)} \left[\sum_{a_0} \nabla \pi_\theta(a_0|s_0) Q_{\pi_\theta}(s_0, a_0) \right] + \mathbb{E}_{p_0(s_0)\pi_\theta(a_0|s_0)} [\nabla_\theta Q_{\pi_\theta}(s_0, a_0)] \quad (35.17)$$

Now we calculate the term $\nabla_\theta Q_{\pi_\theta}(s_0, a_0)$:

$$\nabla_\theta Q_{\pi_\theta}(s_0, a_0) = \nabla_\theta [R(s_0, a_0) + \gamma \mathbb{E}_{p_T(s_1|s_0, a_0)} [V_{\pi_\theta}(s_1)]] = \gamma \nabla_\theta \mathbb{E}_{p_T(s_1|s_0, a_0)} [V_{\pi_\theta}(s_1)] \quad (35.18)$$

The right-hand side above is in a form similar to $\nabla_\theta J(\pi_\theta)$. Repeating the same steps as before gives

$$\nabla_\theta J(\pi_\theta) = \sum_{t=0}^{\infty} \gamma^t \mathbb{E}_{p_t(s)} \left[\sum_a \nabla_\theta \pi_\theta(a|s) Q_{\pi_\theta}(s, a) \right] \quad (35.19)$$

$$= \frac{1}{1-\gamma} \mathbb{E}_{p_{\pi_\theta}^\infty(s)} \left[\sum_a \nabla_\theta \pi_\theta(a|s) Q_{\pi_\theta}(s, a) \right] \quad (35.20)$$

$$= \frac{1}{1-\gamma} \mathbb{E}_{p_{\pi_\theta}^\infty(s)\pi_\theta(a|s)} [\nabla_\theta \log \pi_\theta(a|s) Q_{\pi_\theta}(s, a)] \quad (35.21)$$

where $p_t(s)$ is the probability of visiting s in time t if we start with $s_0 \sim p_0$ and follow π_θ , and $p_{\pi_\theta}^\infty(s) = (1-\gamma) \sum_{t=0}^{\infty} \gamma^t p_t(s)$ is the normalized discounted state visitation distribution. Equation (35.21) is known as the **policy gradient theorem** [Sut+99].

In practice, estimating the policy gradient using Equation (35.21) can have a high variance. A baseline $b(s)$ can be used for variance reduction (Section 6.3.4.1):

$$\nabla_\theta J(\pi_\theta) = \frac{1}{1-\gamma} \mathbb{E}_{p_{\pi_\theta}^\infty(s)\pi_\theta(a|s)} [\nabla_\theta \log \pi_\theta(a|s) (Q_{\pi_\theta}(s, a) - b(s))] \quad (35.22)$$

A common choice for the baseline is $b(s) = V_{\pi_\theta}(s)$. We will discuss how to estimate it below.

35.3.2 REINFORCE

One way to apply the policy gradient theorem to optimize a policy is to use stochastic gradient ascent. Suppose $\tau = (s_0, a_0, r_0, s_1, \dots, s_T)$ is a trajectory with $s_0 \sim p_0$ and π_θ . Then,

$$\nabla_\theta J(\pi_\theta) = \frac{1}{1-\gamma} \mathbb{E}_{p_{\pi_\theta}^\infty(s)\pi_\theta(a|s)} [\nabla_\theta \log \pi_\theta(a|s) Q_{\pi_\theta}(s, a)] \quad (35.23)$$

$$\approx \sum_{t=0}^{T-1} \gamma^t G_t \nabla_\theta \log \pi_\theta(a_t|s_t) \quad (35.24)$$

where the return G_t is defined in Equation (34.76), and the factor γ^t is due to the definition of $p_{\pi_\theta}^\infty$ where the state at time t is discounted.

We can use a baseline in the gradient estimate to get the following update rule:

$$\theta \leftarrow \theta + \eta \sum_{t=0}^{T-1} \gamma^t (G_t - b(s_t)) \nabla_\theta \log \pi_\theta(a_t|s_t) \quad (35.25)$$

This is called the **REINFORCE** algorithm [Wil92].³ The update equation can be interpreted as follows: we compute the sum of discounted future rewards induced by a trajectory, compared to a baseline, and if this is positive, we increase θ so as to make this trajectory more likely, otherwise we decrease θ . Thus, we reinforce good behaviors, and reduce the chances of generating bad ones.

We can use a constant (state-independent) baseline, or we can use a state-dependent baseline, $b(s_t)$ to further lower the variance. A natural choice is to use an estimated value function, $V_w(s)$, which can be learned, e.g., with MC. Algorithm 35.2 gives the pseudocode where stochastic gradient updates are used with separate learning rates.

Algorithm 35.2: REINFORCE with value function baseline

```

1 Initialize policy parameters  $\theta$ , baseline parameters  $w$ 
2 repeat
3   Sample an episode  $\tau = (s_0, a_0, r_0, s_1, \dots, s_T)$  using  $\pi_\theta$ 
4   Compute  $G_t$  for all  $t \in \{0, 1, \dots, T-1\}$  using Equation (34.76)
5   for  $t = 0, 1, \dots, T-1$  do
6      $\delta = G_t - V_w(s_t)$  // scalar error
7      $w \leftarrow w + \eta_w \delta \nabla_w V_w(s_t)$ 
8      $\theta \leftarrow \theta + \eta_\theta \gamma^t \delta \nabla_\theta \log \pi_\theta(a_t|s_t)$ 
9 until converged

```

35.3.3 Actor-critic methods

An **actor-critic** method [BSA83] uses the policy gradient method, but where the expected return is estimated using temporal difference learning of a value function instead of MC rollouts. The term

³ The term “REINFORCE” is an acronym for “REward Increment = nonnegative Factor x Offset Reinforcement x Characteristic Eligibility”. The phrase “characteristic eligibility” refers to the $\nabla \log \pi_\theta(a_t|s_t)$ term; the phrase “offset reinforcement” refers to the $G_t - b(s_t)$ term; and the phrase “nonnegative factor” refers to the learning rate η of SGD.

“actor” refers to the policy, and the term “critic” refers to the value function. The use of bootstrapping in TD updates allows more efficient learning of the value function compared to MC. In addition, it allows us to develop a fully online, incremental algorithm, that does not need to wait until the end of the trajectory before updating the parameters (as in Algorithm 35.2).

Concretely, consider the use of the one-step TD(0) method to estimate the return in the episodic case, i.e., we replace G_t with $G_{t:t+1} = r_t + \gamma V_w(s_{t+1})$. If we use $V_w(s_t)$ as a baseline, the REINFORCE update in Equation (35.25) becomes

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \sum_{t=0}^{T-1} \gamma^t (G_{t:t+1} - V_w(s_t)) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t | s_t) \quad (35.26)$$

$$= \boldsymbol{\theta} + \eta \sum_{t=0}^{T-1} \gamma^t (r_t + \gamma V_w(s_{t+1}) - V_w(s_t)) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t | s_t) \quad (35.27)$$

35.3.3.1 A2C and A3C

Note that $r_{t+1} + \gamma V_w(s_{t+1}) - V_w(s_t)$ is a single sample approximation to the advantage function $A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$. This method is therefore called **advantage actor critic** or **A2C** (Algorithm 35.3). If we run the actors in parallel and asynchronously update their shared parameters, the method is called **asynchronous advantage actor critic** or **A3C** [Mni+16].

Algorithm 35.3: Advantage actor critic (A2C) algorithm

```

1 Initialize actor parameters  $\boldsymbol{\theta}$ , critic parameters  $\mathbf{w}$ 
2 repeat
3   Sample starting state  $s_0$  of a new episode
4   for  $t = 0, 1, 2, \dots$  do
5     Sample action  $a_t \sim \pi_{\boldsymbol{\theta}}(\cdot | s_t)$ 
6     Observe next state  $s_{t+1}$  and reward  $r_t$ 
7      $\delta = r_t + \gamma V_w(s_{t+1}) - V_w(s_t)$ 
8      $\mathbf{w} \leftarrow \mathbf{w} + \eta_{\mathbf{w}} \delta \nabla_{\mathbf{w}} V_w(s_t)$ 
9      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_{\boldsymbol{\theta}} \gamma^t \delta \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t | s_t)$ 
10 until converged

```

35.3.3.2 Eligibility traces

In A2C, we use a single step rollout, and then use the value function in order to approximate the expected return for the trajectory. More generally, we can use the n -step estimate

$$G_{t:t+n} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n V_w(s_{t+n}) \quad (35.28)$$

and obtain an n -step advantage estimate as follows:

$$A_{\pi_{\boldsymbol{\theta}}}^{(n)}(s_t, a_t) = G_{t:t+n} - V_w(s_t) \quad (35.29)$$

The n steps of actual rewards are an unbiased sample, but have high variance. By contrast, $V_w(s_{t+n+1})$ has lower variance, but is biased. By changing n , we can control the bias-variance tradeoff. Instead of using a single value of n , we can take an weighted average, with weight proportional to λ^n for $A_{\pi_\theta}^{(n)}(s_t, a_t)$, as in $\text{TD}(\lambda)$. The average can be shown to be equivalent to

$$A_{\pi_\theta}^{(\lambda)}(s_t, a_t) \triangleq \sum_{\ell=0}^{\infty} (\gamma \lambda)^\ell \delta_{t+\ell} \quad (35.30)$$

where $\delta_t = r_t + \gamma V_w(s_{t+1}) - V_w(s_t)$ is the TD error at time t . Here, $\lambda \in [0, 1]$ is a parameter that controls the bias-variance tradeoff: larger values decrease the bias but increase the variance, as in $\text{TD}(\lambda)$. We can implement Equation (35.30) efficiently using eligibility traces, as shown in Algorithm 35.4, as an example of **generalized advantage estimation (GAE)** [Sch+16b]. See [SB18, Ch.12] for further details.

Algorithm 35.4: Actor critic with eligibility traces

```

1 Initialize actor parameters  $\theta$ , critic parameters  $w$ 
2 repeat
3   Initialize eligibility trace vectors:  $z_\theta \leftarrow \mathbf{0}$ ,  $z_w \leftarrow \mathbf{0}$ 
4   Sample starting state  $s_0$  of a new episode
5   for  $t = 0, 1, 2, \dots$  do
6     Sample action  $a_t \sim \pi_\theta(\cdot | s_t)$ 
7     Observe state  $s_{t+1}$  and reward  $r_t$ 
8     Compute the TD error:  $\delta = r_t + \gamma V_w(s_{t+1}) - V_w(s_t)$ 
9      $z_w \leftarrow \gamma \lambda_w z_w + \nabla_w V_w(s)$ 
10     $z_\theta \leftarrow \gamma \lambda_\theta z_\theta + \gamma^t \nabla_\theta \log \pi_\theta(a_t | s_t)$ 
11     $w \leftarrow w + \eta_w \delta z_w$ 
12     $\theta \leftarrow \theta + \eta_\theta \delta z_\theta$ 
13 until converged

```

35.3.4 Bound optimization methods

In policy gradient methods, the objective $J(\theta)$ does not necessarily increase monotonically, but rather can collapse especially if the learning rate is not small enough. We now describe methods that guarantee monotonic improvement, similar to bound optimization algorithms (Section 6.5).

We start with a useful fact that relate the policy values of two arbitrary policies [KL02]:

$$J(\pi') - J(\pi) = \frac{1}{1-\gamma} \mathbb{E}_{p_{\pi'}^\infty(s)} [\mathbb{E}_{\pi'(a|s)} [A_\pi(s, a)]] \quad (35.31)$$

where π can be interpreted as the current policy during policy optimization, and π' a candidate new policy (such as the greedy policy wrt Q_π). As in the policy improvement theorem (Section 34.6.2), if $\mathbb{E}_{\pi'(a|s)} [A_\pi(s, a)] \geq 0$ for all s , then $J(\pi') \geq J(\pi)$. However, we cannot ensure this condition to hold when function approximation is used, as such a uniformly improving policy π' may not be

representable by our parametric family, $\{\pi_\theta\}_{\theta \in \Theta}$. Therefore, nonnegativity of Equation (35.31) is not easy to ensure, when we do not have a direct way to sample states from $p_{\pi'}^\infty$.

One way to ensure monotonic improvement of J is to improve the policy conservatively. Define $\pi_\theta = \theta\pi' + (1 - \theta)\pi$ for $\theta \in [0, 1]$. It follows from the policy gradient theorem (Equation (35.21), with $\theta = [\theta]$) that $J(\pi_\theta) - J(\pi) = \theta L(\pi') + O(\theta^2)$, where

$$L(\pi') \triangleq \frac{1}{1 - \gamma} \mathbb{E}_{p_{\pi'}^\infty(s)} [\mathbb{E}_{\pi'(a|s)} [A_\pi(s, a)]] = \frac{1}{1 - \gamma} \mathbb{E}_{p_{\pi'}^\infty(s)\pi(a|s)} \left[\frac{\pi'(a|s)}{\pi(a|s)} A_\pi(s, a) \right] \quad (35.32)$$

In the above, we have switched the state distribution from $p_{\pi'}^\infty$ in Equation (35.31) to p_π^∞ , while at the same time introducing a higher order residual term of $O(\theta^2)$. The linear term, $\theta L(\pi')$, can be estimated and optimized based on episodes sampled by π . The higher order term can be bounded in various ways, resulting in different lower bounds of $J(\pi_\theta) - J(\pi)$. We can then optimize θ to make sure this lower bound is positive, which would imply $J(\pi_\theta) - J(\pi) > 0$. In **conservative policy iteration** [KL02], the following (slightly simplified) lower bound is used

$$J^{\text{CPI}}(\pi_\theta) \triangleq J(\pi) + \theta L(\pi') - \frac{2\varepsilon\gamma}{(1 - \gamma)^2} \theta^2 \quad (35.33)$$

where $\varepsilon = \max_s |\mathbb{E}_{\pi'(a|s)} [A_\pi(s, a)]|$.

This idea can be generalized to policies beyond those in the form of π_θ , where the condition of a small enough θ is replaced by a small enough divergence between π' and π . In **safe policy iteration** [Pir+13], the divergence is the maximum total variation, while in **trust region policy optimization (TRPO)** [Sch+15b], the divergence is the maximum KL-divergence. In the latter case, π' may be found by optimizing the following lower bound

$$J^{\text{TRPO}}(\pi') \triangleq J(\pi) + L(\pi') - \frac{\varepsilon\gamma}{(1 - \gamma)^2} \max_s D_{\text{KL}}(\pi(s) \parallel \pi'(s)) \quad (35.34)$$

where $\varepsilon = \max_{s,a} |A_\pi(s, a)|$.

In practice, the above update rule can be overly conservative, and approximations are used. [Sch+15b] propose a version that implements two ideas: one is to replace the point-wise maximum KL-divergence by some average KL-divergence (usually averaged over $p_{\pi_\theta}^\infty$); the second is to maximize the first two terms in Equation (35.34), with π' lying in a KL-ball centered at π . That is, we solve

$$\underset{\pi'}{\operatorname{argmax}} L(\pi') \quad \text{s.t.} \quad \mathbb{E}_{p_{\pi}^\infty(s)} [D_{\text{KL}}(\pi(s) \parallel \pi'(s))] \leq \delta \quad (35.35)$$

for some threshold $\delta > 0$.

In Section 6.4.2.1, we show that the trust region method, using a KL penalty at each step, is equivalent to natural gradient descent (see e.g., [Kak02; PS08b]). This is important, because a step of size η in parameter space does not always correspond to a step of size η in the policy space:

$$d_\theta(\theta_1, \theta_2) = d_\theta(\theta_2, \theta_3) \not\Rightarrow d_\pi(\pi_{\theta_1}, \pi_{\theta_2}) = d_\pi(\pi_{\theta_2}, \pi_{\theta_3}) \quad (35.36)$$

where $d_\theta(\theta_1, \theta_2) = \|\theta_1 - \theta_2\|$ is the Euclidean distance, and $d_\pi(\pi_1, \pi_2) = D_{\text{KL}}(\pi_1 \parallel \pi_2)$ the KL distance. In other words, the effect on the policy of any given change to the parameters depends on where we are in parameter space. This is taken into account by the natural gradient method,

resulting in faster and more robust optimization. The natural policy gradient can be approximated using the KFAC method (Section 6.4.4), as done in [Wu+17].

Other than TRPO, another approach inspired by Equation (35.34) is to use the KL-divergence as a penalty term, replacing the factor $2\varepsilon\gamma/(1-\gamma)^2$ by a tuning parameter. However, it often works better, and is simpler, by using the following clipped objective, which results in the **proximal policy optimization** or **PPO** method [Sch+17]:

$$J^{\text{PPO}}(\pi') \triangleq \frac{1}{1-\gamma} \mathbb{E}_{p_\pi^\infty(s)\pi(a|s)} \left[\kappa_\epsilon \left(\frac{\pi'(a|s)}{\pi(a|s)} \right) A_\pi(s, a) \right] \quad (35.37)$$

where $\kappa_\epsilon(x) \triangleq \text{clip}(x, 1-\epsilon, 1+\epsilon)$ ensures $|\kappa(x) - 1| \leq \epsilon$. This method can be modified to ensure monotonic improvement as discussed in [WHT19], making it a true bound optimization method.

35.3.5 Deterministic policy gradient methods

In this section, we consider the case of a deterministic policy, that predicts a unique action for each state, so $a_t = \mu_\theta(s_t)$, rather than $a_t \sim \pi_\theta(s_t)$. We assume the states and actions are continuous, and define the objective as

$$J(\mu_\theta) \triangleq \frac{1}{1-\gamma} \mathbb{E}_{p_{\mu_\theta}^\infty(s)} [R(s, \mu_\theta(s))] \quad (35.38)$$

The **deterministic policy gradient theorem** [Sil+14] provides a way to compute the gradient:

$$\nabla_\theta J(\mu_\theta) = \frac{1}{1-\gamma} \mathbb{E}_{p_{\mu_\theta}^\infty(s)} [\nabla_\theta Q_{\mu_\theta}(s, \mu_\theta(s))] \quad (35.39)$$

$$= \frac{1}{1-\gamma} \mathbb{E}_{p_{\mu_\theta}^\infty(s)} [\nabla_\theta \mu_\theta(s) \nabla_a Q_{\mu_\theta}(s, a)|_{a=\mu_\theta(s)}] \quad (35.40)$$

where $\nabla_\theta \mu_\theta(s)$ is the $M \times N$ Jacobian matrix, and M and N are the dimensions of \mathcal{A} and θ , respectively. For stochastic policies of the form $\pi_\theta(a|s) = \mu_\theta(s) + \text{noise}$, the standard policy gradient theorem reduces to the above form as the noise level goes to zero.

Note that the gradient estimate in Equation (35.40) integrates over the states but not over the actions, which helps reduce the variance in gradient estimation from sampled trajectories. However, since the deterministic policy does not do any exploration, we need to use an off-policy method, that collects data from a stochastic behavior policy β , whose stationary state distribution is p_β^∞ . The original objective, $J(\mu_\theta)$, is approximated by the following:

$$J_b(\mu_\theta) \triangleq \mathbb{E}_{p_\beta^\infty(s)} [V_{\mu_\theta}(s)] = \mathbb{E}_{p_\beta^\infty(s)} [Q_{\mu_\theta}(s, \mu_\theta(s))] \quad (35.41)$$

with the off-policy deterministic policy gradient approximated by (see also Section 35.5.1.2)

$$\nabla_\theta J_b(\mu_\theta) \approx \mathbb{E}_{p_\beta^\infty(s)} [\nabla_\theta [Q_{\mu_\theta}(s, \mu_\theta(s))]] = \mathbb{E}_{p_\beta^\infty(s)} [\nabla_\theta \mu_\theta(s) \nabla_a Q_{\mu_\theta}(s, a)|_{a=\mu_\theta(s)} ds] \quad (35.42)$$

where we have dropped a term that depends on $\nabla_\theta Q_{\mu_\theta}(s, a)$ and is hard to estimate [Sil+14].

To apply Equation (35.42), we may learn $Q_w \approx Q_{\mu_\theta}$ with TD, giving rise to the following updates:

$$\delta = r_t + \gamma Q_w(s_{t+1}, \mu_\theta(s_{t+1})) - Q_w(s_t, a_t) \quad (35.43)$$

$$w_{t+1} \leftarrow w_t + \eta_w \delta \nabla_w Q_w(s_t, a_t) \quad (35.44)$$

$$\theta_{t+1} \leftarrow \theta_t + \eta_\theta \nabla_\theta \mu_\theta(s_t) \nabla_a Q_w(s_t, a)|_{a=\mu_\theta(s_t)} \quad (35.45)$$

This avoids importance sampling in the actor update because of the deterministic policy gradient, and avoids importance sampling in the critic update because of the use of Q-learning.

If Q_w is linear in w , and uses features of the form $\phi(s, a) = a^\top \nabla_\theta \mu_\theta(s)$, where a is the vector representation of a , then we say the function approximator for the critic is **compatible** with the actor; in this case, one can show that the above approximation does not bias the overall gradient. The method has been extended in various ways. The **DDPG** algorithm of [Lil+16], which stands for “deep deterministic policy gradient”, uses the DQN method (Section 35.2.6) to update Q that is represented by deep neural networks. The **TD3** algorithm [FHM18], which stands for “twin delayed DDPG”, extends DDPG by using double DQN (Section 35.2.5.2) and other heuristics to further improve performance. Finally, the **D4PG** algorithm [BM+18], which stands for “distributed distributional DDPG”, extends DDPG to handle distributed training, and to handle **distributional RL** (i.e., working with distributions of rewards instead of expected rewards [BDM17]).

35.3.6 Gradient-free methods

The policy gradient estimator computes a “zeroth order” gradient, which essentially evaluates the function with randomly sampled trajectories. Sometimes it can be more efficient to use a derivative-free optimizer (Section 6.7), that does not even attempt to estimate the gradient. For example, [MGR18] obtain good results by training linear policies with random search, and [Sal+17b] show how to use evolutionary strategies to optimize the policy of a robotic controller.

35.4 Model-based RL

Model-free approaches to RL typically need a lot of interactions with the environment to achieve good performance. For example, state of the art methods for the Atari benchmark, such as rainbow (Section 35.2.6), use millions of frames, equivalent to many days of playing at the standard frame rate. By contrast, humans can achieve the same performance in minutes [Tsi+17]. Similarly, OpenAI’s robot hand controller [And+20] learns to manipulate a cube using 100 years of simulated data.

One promising approach to greater sample efficiency is **model-based RL (MBRL)**. In this approach, we first learn the transition model and reward function, $p_T(s'|s, a)$ and $R(s, a)$, then use them to compute a near-optimal policy. This approach can significantly reduce the amount of real-world data that the agent needs to collect, since it can “try things out” in its imagination (i.e., the models), rather than having to try them out empirically.

There are several ways we can use a model, and many different kinds of model we can create. Some of the algorithms mentioned earlier, such as MBIE and UCLR2 for provably efficient exploration (Section 35.1.5.3), are examples of model-based methods. MBRL also provides a natural connection between RL and planning (Section 34.6) [Sut90]. We discuss some examples in the sections below, and refer to [MBJ20; PKP21; MH20] for more detailed reviews.

35.4.1 Model predictive control (MPC)

So far in this chapter, we have focused on trying to learn an optimal policy $\pi_*(s)$, which can then be used at run time to quickly pick the best action for any given state s . However, we can also avoid performing all this work in advance, and wait until we know what state we are in, call it s_t , and then use a model to predict future states and rewards that might follow for each possible sequence of

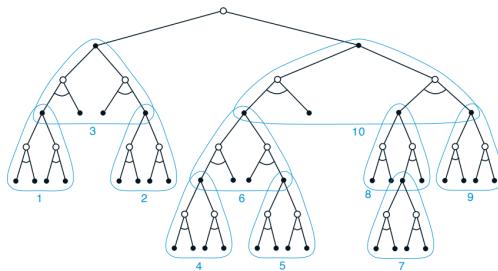


Figure 35.6: Illustration of heuristic search. In this figure, the subtrees are ordered according to a depth-first search procedure. From Figure 8.9 of [SB18]. Used with kind permission of Richard Sutton.

future actions we might pursue. We then take the action that looks most promising, and repeat the process at the next step. More precisely, we compute

$$a_{t:t+H-1}^* = \underset{\mathbf{a}_{t:t+H-1}}{\operatorname{argmax}} \mathbb{E} \left[\sum_{h=0}^{H-1} R(s_{t+h}, a_{t+h}) + \hat{V}(s_{t+H}) \right] \quad (35.46)$$

where the expectation is over state sequences that might result from executing $a_{t:t+H-1}$ from state s_t . Here, H is called the **planning horizon**, and $\hat{V}(s_{t+H})$ is an estimate of the reward-to-go at the end of this H -step look-ahead process. This is known as **receding horizon control** or **model predictive control (MPC)** [MM90; CA13]. We discuss some special cases of this below.

35.4.1.1 Heuristic search

If the state and action spaces are finite, we can solve Equation (35.46) exactly, although the time complexity will typically be exponential in H . However, in many situations, we can prune off unpromising trajectories, thus making the approach feasible in large scale problems.

In particular, consider a discrete, deterministic MDP where reward maximization corresponds to finding a shortest path to a goal state. We can expand the successors of the current state according to all possible actions, trying to find the goal state. Since the search tree grows exponentially with depth, we can use a **heuristic function** to prioritize which nodes to expand; this is called **best-first search**, as illustrated in Figure 35.6.

If the heuristic function is an optimistic lower bound on the true distance to the goal, it is called **admissible**; If we aim to maximize total rewards, admissibility means the heuristic function is an upper bound of the true value function. Admissibility ensures we will never incorrectly prune off parts of the search space. In this case, the resulting algorithm is known as **A^* search**, and is optimal. For more details on classical AI **heuristic search** methods, see [Pea84; RN19].

35.4.1.2 Monte Carlo tree search (MCTS)

Monte Carlo tree search or **MCTS** is similar to heuristic search, but learns a value function for each encountered state, rather than relying on a manually designed heuristic (see e.g., [Mun14] for

details). MCTS is inspired by UCB for bandits (Section 34.4.5), but applies to general sequential decision making problems including MDPs [KS06].

The MCTS method forms the basis of the famous **AlphaGo** and **AlphaZero** programs [Sil+16; Sil+18], which can play expert-level Go, chess, and shogi (Japanese chess), using a known model of the environment. The **MuZero** method of [Sch+20] and the **Stochastic MuZero** method of [Ant+22] extend this to the case where the world model is also learned. The action-value functions for the intermediate nodes in the search tree are represented by deep neural networks, and updated using temporal difference methods that we discuss in Section 35.2. MCTS can also be applied to many other kinds of sequential decision problems, such as experiment design for sequentially creating molecules [SPW18].

35.4.1.3 Trajectory optimization for continuous actions

For continuous actions, we cannot enumerate all possible branches in the search tree. Instead, Equation (35.46) can be viewed as a nonlinear program, where $a_{t:t+H-1}$ are the real-valued variables to be optimized. If the system dynamics are linear and the reward function corresponds to negative quadratic cost, the optimal action sequence can be solved mathematically, as in the **linear-quadratic-Gaussian (LQG)** controller (see e.g., [AM89; HR17]). However, this problem is hard in general and often solved by numerical methods such as **shooting** and **collocation** [Die+07; Rao10; Kal+11]. Many of them work in an iterative fashion, starting with an initial action sequence followed by a step to improve it. This process repeats until convergence of the cost.

An example is **differential dynamic programming (DDP)** [JM70; TL05]. In each iteration, DDP starts with a reference trajectory, and linearizes the system dynamics around states on the trajectory to form a locally quadratic approximation of the reward function. This system can be solved using LQG, whose optimal solution results in a new trajectory. The algorithm then moves to the next iteration, with the new trajectory as the reference trajectory.

Other alternatives are possible, including black-box (gradient-free) optimization methods like the cross-entropy method. (see Section 6.7.5).

35.4.2 Combining model-based and model-free

In Section 35.4.1, we discussed MPC, which uses the model to decide which action to take at each step. However, this can be slow, and can suffer from problems when the model is inaccurate. An alternative is to use the learned model to help reduce the sample complexity of policy learning.

There are many ways to do this. One approach is to generate rollouts from the model, and then train a policy or Q -function on the “hallucinated” data. This is the basis of the famous **dyna** method [Sut90]. In [Jan+19], they propose a similar method, but generate short rollouts from previously visited real states; this ensures the model only has to extrapolate locally.

In [Web+17], they train a model to predict future states and rewards, but then use the hidden states of this model as additional context for a policy-based learning method. This can help overcome partial observability. They call their method **imagination-augmented agents**. A related method appears in [Jad+17], who propose to train a model to jointly predict future rewards and other auxiliary signals, such as future states. This can help in situations when rewards are sparse or absent.

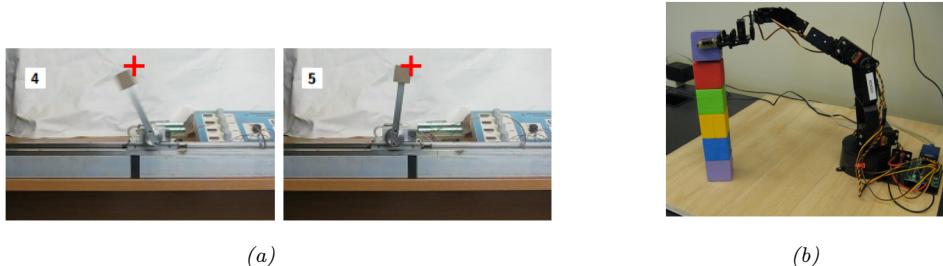


Figure 35.7: (a) A cart-pole system being controlled by a policy learned by PILCO using just 17.5 seconds of real-world interaction. The goal state is marked by the red cross. The initial state is where the cart is stationary on the right edge of the workspace, and the pendulum is horizontal. For a video of the system learning, see <https://bit.ly/35fpLmR>. (b) A low-quality robot arm being controlled by a block-stacking policy learned by PILCO using just 230 seconds of real-world interaction. From Figures 11, 12 from [DFR15]. Used with kind permission of Marc Deisenroth.

35.4.3 MBRL using Gaussian processes

This section gives some examples of dynamics models that have been learned for low-dimensional continuous control problems. Such problems frequently arise in robotics. Since the dynamics are often nonlinear, it is useful to use a flexible and sample-efficient model family, such as Gaussian processes (Section 18.1). We will use notation like s and a for states and actions to emphasize they are vectors.

35.4.3.1 PILCO

We first describe the **PILCO** method [DR11; DFR15], which stands for ‘‘probabilistic inference for learning control’’. It is extremely data efficient for continuous control problems, enabling learning from scratch on real physical robots in a matter of minutes.

PILCO assumes the world model has the form $s_{t+1} = f(s_t, a_t) + \epsilon_t$, where $\epsilon_t \sim \mathcal{N}(\mathbf{0}, \Sigma)$, and f is an unknown, continuous function.⁴ The basic idea is to learn a Gaussian process (Section 18.1)) approximation of f based on some initial random trajectories, and then to use this model to generate ‘‘fantasy’’ rollout trajectories of length T , that can be used to evaluate the expected cost of the current policy, $J(\pi_\theta) = \sum_{t=1}^T \mathbb{E}_{a_t \sim \pi_\theta} [c(s_t)]$, where $s_0 \sim p_0$. This function and its gradients wrt θ can be computed deterministically, if a Gaussian assumption about the state distribution at each step is made, because the Gaussian belief state can be propagated deterministically through the GP model. Therefore, we can use deterministic batch optimization methods, such as Levenberg-Marquardt, to optimize the policy parameters θ , instead of applying SGD to sampled trajectories. (See <https://github.com/mathDR/jax-pilco> for some JAX code.)

Due to its data efficiency, it is possible to apply PILCO to real robots. Figure 35.7a shows the results of applying it to solve a **cart-pole swing-up** task, where the goal is to make the inverted pendulum swing up by applying a horizontal force to move the cart back and forth. The state of the system $s \in \mathbb{R}^4$ consists of the position x of the cart (with $x = 0$ being the center of the track), the

4. An alternative, which often works better, is to use f to model the residual, so that $s_{t+1} = s_t + f(s_t, a_t) + \epsilon_t$.

velocity \dot{x} , the angle θ of the pendulum (measured from hanging downward), and the angular velocity $\dot{\theta}$. The control signal $a \in \mathbb{R}$ is the force applied to the cart. The target state is $s_* = (0, \star, \pi, \star)$, corresponding to the cart being in the middle and the pendulum being vertical, with velocities unspecified. The authors used an RBF controller with 50 basis functions, amounting to a total of 305 policy parameters. The controller was successfully trained using just 7 real world trials.⁵

Figure 35.7b shows the results of applying PILCO to solve a **block stacking** task using a low-quality robot arm with 6 degrees of freedom. A separate controller was trained for each block. The state space $s \in \mathbb{R}^3$ is the 3d location of the center of the block in the arm’s gripper (derived from an RGBD sensor), and the control $a \in \mathbb{R}^4$ corresponds to the pulse widths of four servo motors. A linear policy was successfully trained using as few as 10 real world trials.

35.4.3.2 GP-MPC

[KD18a] have proposed an extension to PILCO that they call **GP-MPC**, since it combines a GP dynamics model with model predictive control (Section 35.4.1). In particular, they use an open-loop control policy to propose a sequence of actions, $a_{t:t+H-1}$, as opposed to sampling them from a policy. They compute a Gaussian approximation to the future state trajectory, $p(s_{t+1:t+H}|a_{t:t+H-1}, s_t)$, by moment matching, and use this to deterministically compute the expected reward and its gradient wrt $a_{t:t+H-1}$ (as opposed to the policy parameters θ). Using this, they can solve Equation (35.46) to find $a_{t:t+H-1}^*$; finally, they execute the first step of this plan, a_t^* , and repeat the whole process.

The advantage of GP-MPC over policy-based PILCO is that it can handle constraints more easily, and it can be more data efficient, since it continually updates the GP model after every step (instead of at the end of a trajectory).

35.4.4 MBRL using DNNs

Gaussian processes do not scale well to large sample sizes and high dimensional data. Deep neural networks (DNNs) work much better in this regime. However, they do not naturally model uncertainty, which can cause MPC methods to fail. We discuss various methods for representing uncertainty with DNNs in Section 17.1. Here, we mention a few approaches that have been used for MBRL.

The **deep PILCO** method uses DNNs together with Monte Carlo dropout (Section 17.3.1) to represent uncertainty [GMAR16]. [Chu+18] proposed **probabilistic ensembles with trajectory sampling** or **PETS**, which represents uncertainty using an ensemble of DNNs (Section 17.3.9). Many other approaches are possible, depending on the details of the problem being tackled.

Since these are all stochastic methods (as opposed to the GP methods above), they can suffer from a high variance in the predicted returns, which can make it difficult for the MPC controller to pick the best action. We can reduce variance with the **common random number** trick [KSN99], where all rollouts share the same random seed, so differences in $J(\pi_\theta)$ can be attributed to changes in θ but not other factors. This technique was used in **PEGASUS** [NJ00]⁶ and in [HMD18].

5. 2 random initial trials, each 5 seconds, and then 5 policy-generated trials, each 2.5 seconds, totalling 17.5 seconds.

6. PEGASUS stands for “Policy Evaluation-of-Goodness And Search Using Scenarios”, where the term “scenario” refers to one of the shared random samples.

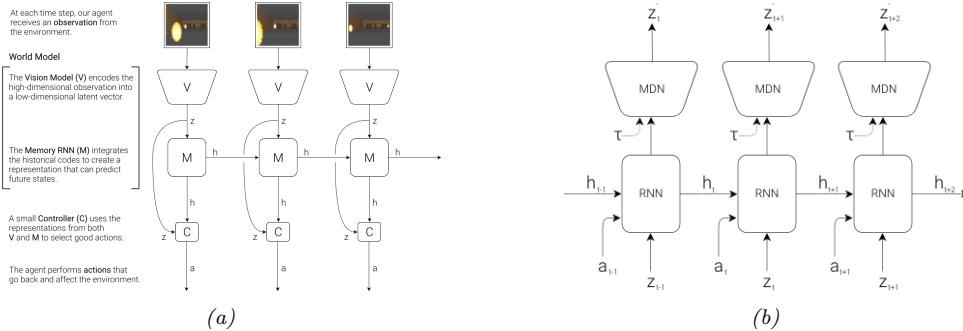


Figure 35.8: (a) Illustration of an agent interacting with the VizDoom environment. (The yellow blobs represent fireballs being thrown towards the agent by various enemies.) The agent has a world model, composed of a vision system V and a memory RNN M , and has a controller C . (b) Detailed representation of the memory model. Here h_t is the deterministic hidden state of the RNN at time t , which is used to predict the next latent of the VAE, z_{t+1} , using a mixture density network (MDN). Here τ is a temperature parameter used to increase the variance of the predictions, to prevent the controller from exploiting model inaccuracies. From Figures 4, 6 of [HS18]. Used with kind permission of David Ha.

35.4.5 MBRL using latent-variable models

In this section, we describe some methods that learn latent variable models, rather than trying to predict dynamics directly in the observed space, which is hard to do when the states are images.

35.4.5.1 World models

The ‘‘world models’’ paper [HS18] showed how to learn a generative model of two simple video games (CarRacing and a VizDoom-like environment), such that the model can be used to train a policy entirely in simulation. The basic idea is shown in Figure 35.8. First, we collect some random experience, and use this to fit a VAE model (Section 21.2) to reduce the dimensionality of the images, $\mathbf{x}_t \in \mathbb{R}^{64 \times 64 \times 3}$, to a latent $\mathbf{z}_t \in \mathbb{R}^{64}$. Next, we train an RNN to predict $p(\mathbf{z}_{t+1} | \mathbf{z}_t, \mathbf{a}_t, \mathbf{h}_t)$, where \mathbf{h}_t is the deterministic RNN state, and \mathbf{a}_t is the continuous action vector (3-dimensional in both cases). The emission model for the RNN is a mixture density network, in order to model multi-modal futures. Finally, we train the controller using \mathbf{z}_t and \mathbf{h}_t as inputs; here \mathbf{z}_t is a compact representation of the current frame, and \mathbf{h}_t is a compact representation of the predicted distribution over \mathbf{z}_{t+1} .

The authors of [HS18] trained the controller using a derivative free optimizer called **CMA-ES** (covariance matrix adaptation evolutionary strategy, see Section 6.7.6.2). It can work better than policy gradient methods, as discussed in Section 35.3.6. However, it does not scale to high dimensions. To tackle this, the authors use a linear controller, which has only 867 parameters.⁷ By contrast, VAE has 4.3M parameters and MDN-RNN 422k. Fortunately, these two models can be trained in an unsupervised way from random rollouts, so sample efficiency is less critical than when training the policy.

7. The input is a 32-dimensional \mathbf{z}_t plus a 256-dimensional \mathbf{h}_t , and there are 3 outputs. So the number of parameters is $(32 + 256) \times 3 + 3 = 867$, to account for the weights and biases.

So far, we have described how to use the representation learned by the generative model as informative features for the controller, but the controller is still learned by interacting with the real world. Surprisingly, we can also train the controller entirely in “dream mode”, in which the generated images from the VAE decoder at time t are fed as input to the VAE encoder at time $t + 1$, and the MDN-RNN is trained to predict the next reward r_{t+1} as well as \mathbf{z}_{t+1} . Unfortunately, this method does not always work, since the model (which is trained in an unsupervised way) may fail to capture task-relevant features (due to underfitting) and may memorize task-irrelevant features (due to overfitting). The controller can learn to exploit weaknesses in the model (similar to an adversarial attack) and achieve high simulated reward, but such a controller may not work well when transferred to the real world.

One approach to combat this is to artificially increase the variance of the MDN model (by using a temperature parameter τ), in order to make the generated samples more stochastic. This forces the controller to be robust to large variations; the controller will then treat the real world as just another kind of noise. This is similar to the technique of domain randomization, which is sometimes used for sim-to-real applications; see e.g., [MAZA18].

35.4.5.2 PlaNet and Dreamer

In [HS18], they first learn the world model on random rollouts, and then train a controller. On harder problems, it is necessary to iterate these two steps, so the model can be trained on data collected by the controller, in an iterative fashion.

In this section, we describe one method of this kind, known as **PlaNet** [Haf+19]. PlaNet uses a POMDP model, where \mathbf{z}_t are the latent states, \mathbf{s}_t are the observations, \mathbf{a}_t are the actions, and r_t are the rewards. It fits a recurrent state space model (Section 29.13.2) of the form $p(\mathbf{z}_t|\mathbf{z}_{t-1}, \mathbf{a}_{t-1})p(\mathbf{s}_t|\mathbf{z}_t)p(r_t|\mathbf{z}_t)$ using variational inference, where the posterior is approximated by $q(\mathbf{z}_t|\mathbf{s}_{1:t}, \mathbf{a}_{1:t-1})$. After fitting the model to some random trajectories, the system uses the inference model to compute the current belief state, and then uses the cross entropy method to find an action sequence for the next H steps to maximize expected reward, by optimizing in latent space. The system then executes \mathbf{a}_t^* , updates the model, and repeats the whole process. To encourage the dynamics model to capture long term trajectories, they use the “latent overshooting” training method described in Section 29.13.3. The PlaNet method outperforms model-free methods, such as A3C (Section 35.3.3.1) and D4PG (Section 35.3.5), on various image-based continuous control tasks, illustrated in Figure 35.9.

Although PlaNet is sample efficient, it is not computationally efficient. For example, they use CEM with 1000 samples and 10 iterations to optimize trajectories with a horizon of length 12, which requires 120,000 evaluations of the transition dynamics to choose a single action. [AY19] improve this by replacing CEM with differentiable CEM, and then optimize in a latent space of action sequences. This is much faster, but the results are not quite as good. However, since the whole policy is now differentiable, it can be fine-tuned using PPO (Section 35.3.4), which closes the performance gap at negligible cost.

A recent extension of the PlaNet paper, known as **Dreamer**, was proposed in [Haf+20]. In this paper, the online MPC planner is replaced by a policy network, $\pi(\mathbf{a}_t|\mathbf{z}_t)$, which is learned using gradient-based actor-critic in latent space. The inference and generative models are trained by maximizing the ELBO, as in PlaNet. The policy is trained by SGD to maximize expected total reward as predicted by the value function, and the value function is trained by SGD to minimize

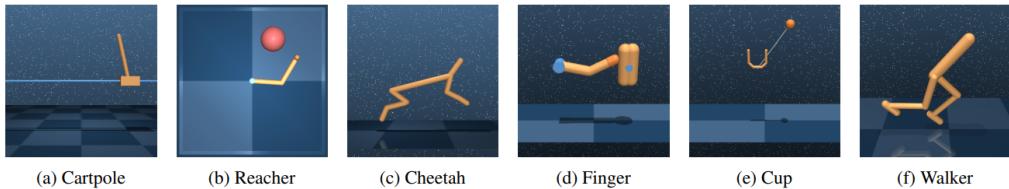


Figure 35.9: Illustration of some image-based control problems used in the PlaNet paper. Inputs are $64 \times 64 \times 3$. (a) The cartpole swingup task has a fixed camera so the cart can move out of sight, making this a partially observable problem. (b) The reacher task has a sparse reward. (c) The cheetah running task includes both contacts and a larger number of joints. (d) The finger spinning task includes contacts between the finger and the object. (e) The cup task has a sparse reward that is only given once the ball is caught. (f) The walker task requires balance and predicting difficult interactions with the ground when the robot is lying down. From Figure 1 of [Haf+19]. Used with kind permission of Danijar Hafner.

MSE between predicted future reward and the TD- λ estimate (Section 35.2.2). They show that Dreamer gives better results than PlaNet, presumably because they learn a policy to optimize the long term reward (as estimated by the value function), rather than relying on MPC based on short-term rollouts.

35.4.6 Robustness to model errors

The main challenge with MBRL is that errors in the model can result in poor performance of the resulting policy, due to the distribution shift problem (Section 19.2). That is, the model is trained to predict states and rewards that it has seen using some behavior policy (e.g., the current policy), and then is used to compute an optimal policy under the learned model. When the latter policy is followed, the agent will experience a different distribution of states, under which the learned model may not be a good approximation of the real environment.

We require the model to generalize in a robust way to new states and actions. (This is related to the off-policy learning problem that we discuss in Section 35.5.) Failing that, the model should at least be able to quantify its uncertainty (Section 19.3). These topics are the focus of much recent research (see e.g., [Luo+19; Kur+19; Jan+19; Isl+19; Man+19; WB20; Eys+21]).

35.5 Off-policy learning

We have seen examples of off-policy methods such as Q-learning. They do not require that training data be generated by the policy it tries to evaluate or improve. Therefore, they tend to have greater data efficiency than their on-policy counterparts, by taking advantage of data generated by other policies. They are also easier to be applied in practice, especially in domains where costs and risks of following a new policy must be considered. This section covers this important topic.

A key challenge in off-policy learning is that the data distribution is typically different from the desired one, and this mismatch must be dealt with. For example, the probability of visiting a state s at time t in a trajectory depends not only on the MDP's transition model, but also on the policy that is being followed. If we are to estimate $J(\pi)$, as defined in Equation (35.15), but the trajectories

are generated by a different policy π' , simply averaging rewards in the data gives us $J(\pi')$, not $J(\pi)$. We have to somehow correct for the gap, or ‘‘bias’’. Another challenge is that off-policy data can also make an algorithm unstable and divergent, which we will discuss in Section 35.5.3.

Removing distribution mismatches is not unique in off-policy learning, and is also needed in supervised learning to handle covariate shift (Section 19.2.3.1), and in causal effect estimation (Chapter 36), among others. Off-policy learning is also closely related to **offline reinforcement learning** (also called **batch reinforcement learning**): the former emphasizes the distributional mismatch between data and the agent’s policy, and the latter emphasizes that the data is static and no further online interaction with the environment is allowed [LP03; EGW05; Lev+20]. Clearly, in the offline scenario with fixed data, off-policy learning is typically a critical technical component. Recently, several datasets have been prepared to facilitate empirical comparisons of offline RL methods (see e.g., [Gul+20; Fu+20]).

Finally, while this section focuses on MDPs, most methods can be simplified and adapted to the special case of contextual bandits (Section 34.4). In fact, off-policy methods have been successfully used in numerous industrial bandit applications (see e.g., [Li+10; Bot+13; SJ15; HLR16]).

35.5.1 Basic techniques

We start with four basic techniques, and will consider more sophisticated ones in subsequent sections. The off-policy data is assumed to be a collection of trajectories: $\mathcal{D} = \{\tau^{(i)}\}_{1 \leq i \leq n}$, where each trajectory is a sequence as before: $\tau^{(i)} = (s_0^{(i)}, a_0^{(i)}, r_0^{(i)}, s_1^{(i)}, \dots)$. Here, the reward and next states are sampled according to the reward and transition models; the actions are chosen by a **behavior policy**, denoted π_b , which is different from the **target policy**, π_e , that the agent is evaluating or improving. When π_b is unknown, we are in a **behavior-agnostic off-policy** setting.

35.5.1.1 Direct method

A natural approach to off-policy learning starts with estimating the unknown reward and transition models of the MDP from off-policy data. This can be done using regression and density estimation methods on the reward and transition models, respectively, to obtain \hat{R} and \hat{P} ; see Section 35.4 for further discussions. These estimated models then give us an inexpensive way to (approximately) simulate the original MDP, and we can apply on-policy methods on the simulated data. This method directly models the outcome of taking an action in a state, thus the name **direct method**, and is sometimes known as **regression estimator** and **plug-in estimator**.

While the direct method is natural and sometimes effective, it has a few limitations. First, a small estimation error in the simulator has a compounding effect in long-horizon problems (or equivalently, when the discount factor γ is close to 1). Therefore, an agent that is optimized against an MDP simulator may overfit the estimation errors. Unfortunately, learning the MDP model, especially the transition model, is generally difficult, making the method limited in domains where \hat{R} and \hat{P} can be learned to high fidelity. See Section 35.4.6 for a related discussion.

35.5.1.2 Importance sampling

The second approach relies on importance sampling (IS) (Section 11.5) to correct for distributional mismatches in the off-policy data. To demonstrate the idea, consider the problem of estimating the

target policy value $J(\pi_e)$ with a fixed horizon T . Correspondingly, the trajectories in \mathcal{D} are also of length T . Then, the IS off-policy estimator, first adopted by [PSS00], is given by

$$\hat{J}_{\text{IS}}(\pi_e) \triangleq \frac{1}{n} \sum_{i=1}^n \frac{p(\boldsymbol{\tau}^{(i)}|\pi_e)}{p(\boldsymbol{\tau}^{(i)}|\pi_b)} \sum_{t=0}^{T-1} \gamma^t r_t^{(i)} \quad (35.47)$$

It can be verified that $\mathbb{E}_{\pi_b} [\hat{J}_{\text{IS}}(\pi_e)] = J(\pi_e)$, that is, $\hat{J}_{\text{IS}}(\pi_e)$ is **unbiased**, provided that $p(\boldsymbol{\tau}|\pi_b) > 0$ whenever $p(\boldsymbol{\tau}|\pi_e) > 0$. The **importance ratio**, $\frac{p(\boldsymbol{\tau}^{(i)}|\pi_e)}{p(\boldsymbol{\tau}^{(i)}|\pi_b)}$, is used to compensate for the fact that the data is sampled from π_b and not π_e . Furthermore, this ratio does *not* depend on the MDP models, because for any trajectory $\boldsymbol{\tau} = (s_0, a_0, r_0, s_1, \dots, s_T)$, we have from Equation (34.74) that

$$\frac{p(\boldsymbol{\tau}|\pi_e)}{p(\boldsymbol{\tau}|\pi_b)} = \frac{p(s_0) \prod_{t=0}^{T-1} \pi_e(a_t|s_t) p_T(s_{t+1}|s_t, a_t) p_R(r_t|s_t, a_t, s_{t+1})}{p(s_0) \prod_{t=0}^{T-1} \pi_b(a_t|s_t) p_T(s_{t+1}|s_t, a_t) p_R(r_t|s_t, a_t, s_{t+1})} = \prod_{t=0}^{T-1} \frac{\pi_e(a_t|s_t)}{\pi_b(a_t|s_t)} \quad (35.48)$$

This simplification makes it easy to apply IS, as long as the target and behavior policies are known. If the behavior policy is unknown, we can estimate it from \mathcal{D} (using, e.g., logistic regression or DNNs), and replace π_b by its estimate $\hat{\pi}_b$ in Equation (35.48). For convenience, define the **per-step importance ratio** at time t by $\rho_t(\boldsymbol{\tau}) \triangleq \pi_e(a_t|s_t)/\pi_b(a_t|s_t)$, and similarly, $\hat{\rho}_t(\boldsymbol{\tau}) \triangleq \pi_e(a_t|s_t)/\hat{\pi}_b(a_t|s_t)$.

Although IS can in principle eliminate distributional mismatches, in practice its usability is often limited by its potentially high variance. Indeed, the importance ratio in Equation (35.47) can be arbitrarily large if $p(\boldsymbol{\tau}^{(i)}|\pi_e) \gg p(\boldsymbol{\tau}^{(i)}|\pi_b)$. There are many improvements to the basic IS estimator. One improvement is based on the observation that the reward r_t is independent of the trajectory beyond time t . This leads to a **per-decision importance sampling** variant that often yields lower variance (see Section 11.6.2 for a statistical motivation, and [LBB20] for a further discussion):

$$\hat{J}_{\text{PDIS}}(\pi_e) \triangleq \frac{1}{n} \sum_{i=1}^n \sum_{t=0}^{T-1} \prod_{t' \leq t} \rho_{t'}(\boldsymbol{\tau}^{(i)}) \gamma^t r_t^{(i)} \quad (35.49)$$

There are many other variants such as self-normalized IS and truncated IS, both of which aim to reduce variance possibly at the cost of a small bias; precise expressions of these alternatives are found, e.g., in [Liu+18b]. In the next subsection, we will discuss another systematic way to improve IS.

IS may also be applied to improve a policy against the policy value given in Equation (35.15). However, directly applying the calculation of Equation (35.48) runs into a fundamental issue with IS, which we will discuss in Section 35.5.2. For now, we may consider the following approximation of policy value, averaging over the state distribution of the behavior policy:

$$J_b(\pi_{\boldsymbol{\theta}}) \triangleq \mathbb{E}_{p_{\beta}^{\infty}(s)} [V_{\pi}(s)] = \mathbb{E}_{p_{\beta}^{\infty}(s)} \left[\sum_a \pi_{\boldsymbol{\theta}}(a|s) Q_{\pi}(s, a) \right] \quad (35.50)$$

Differentiating this and ignoring the term $\nabla_{\boldsymbol{\theta}} Q_{\pi}(s, a)$, as suggested by [DWS12], gives a way to (approximately) estimate the **off-policy policy-gradient** using a one-step IS correction ratio:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J_b(\pi_{\boldsymbol{\theta}}) &\approx \mathbb{E}_{p_{\beta}^{\infty}(s)} \left[\sum_a \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s) Q_{\pi}(s, a) \right] \\ &= \mathbb{E}_{p_{\beta}^{\infty}(s) \beta(a|s)} \left[\frac{\pi_{\boldsymbol{\theta}}(a|s)}{\beta(a|s)} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s) Q_{\pi}(s, a) \right] \end{aligned}$$

Finally, we note that in the tabular MDP case, there exists a policy π_* that is optimal in all states (Section 34.5.5). This policy maximizes J and J_b simultaneously, so Equation (35.50) can be a good proxy for Equation (35.15) as long as all states are “covered” by the behavior policy π_b . The situation is similar when the set of value functions or policies under consideration is sufficiently expressive: an example is a Q-learning like algorithm called Retrace [Mun+16; ASN20]. Unfortunately, in general when we work with parametric families of value functions or policies, such a uniform optimality is lost, and the distribution of states has a direct impact on the solution found by the algorithm. We will revisit this problem in Section 35.5.2.

35.5.1.3 Doubly robust

It is possible to combine the direct and importance sampling methods discussed previously. To develop intuition, consider the problem of estimating $J(\pi_e)$ in a contextual bandit (Section 34.4), that is, when $T = 1$ in \mathcal{D} . The **doubly robust** (DR) estimator is given by

$$\hat{J}_{\text{DR}}(\pi_e) \triangleq \frac{1}{n} \sum_{i=1}^n \left(\frac{\pi_e(a_0^{(i)}|s_0^{(i)})}{\hat{\pi}_b(a_0^{(i)}|s_0^{(i)})} \left(r_0^{(i)} - \hat{Q}(s_0^{(i)}, a_0^{(i)}) \right) + \hat{V}(s_0^{(i)}) \right) \quad (35.51)$$

where \hat{Q} is an estimate of Q_{π_e} , which can be obtained using methods discussed in Section 35.2, and $\hat{V}(s) = \mathbb{E}_{\pi_e(a|s)} [\hat{Q}(s, a)]$. If $\hat{\pi}_b = \pi_b$, the term \hat{Q} is canceled by \hat{V} on average, and we get the IS estimate that is unbiased; if $\hat{Q} = Q_{\pi_e}$, the term \hat{Q} is canceled by the reward on average, and we get the estimator as in the direct method that is also unbiased. In other words, the estimator Equation (35.51) is unbiased, as long as one of the estimates, $\hat{\pi}_b$ and \hat{Q} , is right. This observation justifies the name doubly robust, which has its origin in causal inference (see e.g., [BR05]).

The above DR estimator may be extended to MDPs recursively, starting from the last step. Given a length- T trajectory $\boldsymbol{\tau}$, define $\hat{J}_{\text{DR}}[T] \triangleq 0$, and for $t < T$,

$$\hat{J}_{\text{DR}}[t] \triangleq \hat{V}(s_t) + \hat{\rho}_t(\boldsymbol{\tau}) \left(r_t + \gamma \hat{J}_{\text{DR}}[t+1] - \hat{Q}(s_t, a_t) \right) \quad (35.52)$$

where $\hat{Q}(s_t, a_t)$ is the estimated cumulative reward for the remaining $T - t$ steps. The DR estimator of $J(\pi_e)$, denoted $\hat{J}_{\text{DR}}(\pi_e)$, is the average of $\hat{J}_{\text{DR}}[0]$ over all n trajectories in \mathcal{D} [JL16]. It can be verified (as an exercise) that the recursive definition is equivalent to

$$\hat{J}_{\text{DR}}[0] = \hat{V}(s_0) + \sum_{t=0}^{T-1} \left(\prod_{t'=0}^t \hat{\rho}_{t'}(\boldsymbol{\tau}) \right) \gamma^t \left(r_t + \gamma \hat{V}(s_{t+1}) - \hat{Q}(s_t, a_t) \right) \quad (35.53)$$

This form can be easily generalized to the infinite-horizon setting by letting $T \rightarrow \infty$ [TB16]. Other than double robustness, the estimator is also shown to result in minimum variance under certain conditions [JL16]. Finally, the DR estimator can be incorporated into policy gradient for policy optimization, to reduce gradient estimation variance [HJ20].

35.5.1.4 Behavior regularized method

The three methods discussed previously do not impose any constraint on the target policy π_e . Typically, the more different π_e is from π_b , the less accurate our off-policy estimation can be.

Therefore, when we optimize a policy in offline RL, a natural strategy is to favor target policies that are “close” to the behavior policy. Similar ideas are discussed in the context of conservative policy gradient (Section 35.3.4).

One approach is to impose a hard constraint on the proximity between the two policies. For example, we may modify the loss function of DQN (Equation (35.14)) as follows

$$\mathcal{L}_1^{\text{DQN}}(\mathbf{w}) \triangleq \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}} \left[(r + \gamma \max_{\pi: D(\pi, \pi_b) \leq \varepsilon} \mathbb{E}_{\pi(a'|s')} [Q_{\mathbf{w}}(s', a')] - Q_{\mathbf{w}}(s, a))^2 \right] \quad (35.54)$$

In the above, we replace the $\max_{a'}$ operation by an expectation over a policy that stays close enough to the behavior policy, measured by some distance function D . For various instantiations and further details, see e.g., [FMP19; Kum+19a].

We may also impose a soft constraint on the proximity, by penalizing target policies that are too different. The DQN loss function can be adapted accordingly:

$$\mathcal{L}_2^{\text{DQN}}(\mathbf{w}) \triangleq \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}} \left[(r + \gamma \max_{\pi} \mathbb{E}_{\pi(a'|s')} [Q_{\mathbf{w}}(s', a')] - \alpha \gamma D(\pi(s'), \pi_b(s')) - Q_{\mathbf{w}}(s, a))^2 \right] \quad (35.55)$$

This idea has been used in contextual bandits [SJ15] and empirically studied in MDPs by [WTN19].

There are many choices for the function D , such as the KL-divergence, for both hard and soft constraints. More detailed discussions and examples can be found in [Lev+20].

Finally, behavior regularization and previous methods like IS can be combined, where the former ensures lower variance and greater generalization of the latter (e.g., [SJ15]). Furthermore, most proposed behavior regularized methods consider one-step difference in D , comparing $\pi(s)$ and $\pi_b(s)$ conditioned on s . In many cases, it is desired to consider the difference between the long-term distributions, p_{β}^{∞} and p^{∞} , which we will discuss next.

35.5.2 The curse of horizon

The IS and DR approaches presented in the previous section all rely on an importance ratio to correct distributional mismatches. The ratio depends on the entire trajectory, and its variance grows exponentially in the trajectory length T . Correspondingly, the off-policy estimate of either the policy value or policy gradient can suffer an exponentially large variance (and thus very low accuracy), a challenge called the **curse of horizon** [Liu+18b]. Policies found by approximate algorithms like Q-learning and off-policy actor-critic often have hard-to-control error due to distribution mismatches.

This section discusses an approach to tackling this challenge, by considering corrections in the state-action distribution, rather than in the trajectory distribution. This change is critical: [Liu+18b] describes an example, where the state-action distributions under the behavior and target policies are identical, but the importance ratio of a trajectory grows exponentially large. It is now more convenient to assume the off-policy data consists of a set of transitions: $\mathcal{D} = \{(s_i, a_i, r_i, s'_i)\}_{1 \leq i \leq m}$, where $(s_i, a_i) \sim p_{\mathcal{D}}$ (some fixed but unknown sampling distribution, such as p_{β}^{∞}), and r_i and s'_i are sampled from the MDP’s reward and transition models. Given a policy π , we aim to estimate the correction ratio $\zeta_*(s, a) = p_{\pi}^{\infty}(s, a) / p_{\mathcal{D}}(s, a)$, as it allows us to rewrite the policy value (Equation (35.15)) as

$$J(\pi) = \frac{1}{1 - \gamma} \mathbb{E}_{p_{\pi}^{\infty}(s,a)} [R(s, a)] = \frac{1}{1 - \gamma} \mathbb{E}_{p_{\beta}^{\infty}(s,a)} [\zeta_*(s, a) R(s, a)] \quad (35.56)$$

For simplicity, we assume the initial state distribution p_0 is known, or can be easily sampled from. This assumption is often easy to satisfy in practice.

The starting point is the following linear program formulation for any given π :

$$\max_{d \geq 0} -D_f(d \| p_D) \quad \text{s.t.} \quad d(s, a) = (1 - \gamma)\mu_0(s)\pi(a|s) + \gamma \sum_{\bar{s}, \bar{a}} p(s|\bar{s}, \bar{a})d(\bar{s}, \bar{a})\pi(a|\bar{s}) \quad \forall(s, a) \quad (35.57)$$

where D_f is the f -divergence (Section 2.7.1). The constraint is a variant of Equation (34.93), giving similar flow conditions in the space of $\mathcal{S} \times \mathcal{A}$ under policy π . Under mild conditions, p_π^∞ is only solution that satisfies the flow constraints, so the objective does not affect the solution, but will facilitate the derivation below. We can now obtain the Lagrangian, with multipliers $\{\nu(s, a)\}$, and use the change-of-variables $\zeta(s, a) = d(s, a)/p_D(s, a)$ to obtain the following optimization problem:

$$\begin{aligned} \max_{\zeta \geq 0} \min_{\nu} \mathcal{L}(\zeta, \nu) = & \mathbb{E}_{p_D(s, a)} [-f(\zeta(s, a))] + (1 - \gamma)\mathbb{E}_{p_0(s)\pi(a|s)} [\nu(s, a)] \\ & + \mathbb{E}_{\pi(a'|s')p(s'|s, a)p_D(s, a)} [\zeta(s, a)(\gamma\nu(s', a') - \nu(s, a))] \end{aligned} \quad (35.58)$$

It can be shown that the saddle point to Equation (35.58) must coincide with the desired correction ratio ζ_* . In practice, we may parameterize ζ and ν , and apply two-timescales stochastic gradient descent/ascent on the off-policy data \mathcal{D} to solve for an approximate saddle-point. This is the **DualDICE** method [Nac+19a], which is extended to **GenDICE** [Zha+20c].

Compared to the IS or DR approaches, Equation (35.58) does not compute the importance ratio of a trajectory, thus generally has a lower variance. Furthermore, it is behavior-agnostic, without having to estimate the behavior policy, or even to assume data consists of a collection of trajectories. Finally, this approach can be extended to be doubly robust (e.g., [UHJ20]), and to optimize a policy [Nac+19b] against the true policy value $J(\pi)$ (as opposed to approximations like Equation (35.50)). For more examples along this line of approach, see [ND20] and the references therein.

35.5.3 The deadly triad

Other than introducing bias, off-policy data may also make a value-based RL method unstable and even divergent. Consider the simple MDP depicted in Figure 35.10a, due to [Bai95]. It has 7 states and 2 actions. Taking the dashed action takes the environment to the 6 upper states uniformly at random, while the solid action takes it to the bottom state. The reward is 0 in all transitions, and $\gamma = 0.99$. The value function V_w uses a linear parameterization indicated by the expressions shown inside the states, with $w \in \mathbb{R}^8$. The target policies π always chooses the solid action in every state. Clearly, the true value function, $V_\pi(s) = 0$, can be exactly represented by setting $w = \mathbf{0}$.

Suppose we use a behavior policy b to generate a trajectory, which chooses the dashed and solid actions with probabilities 6/7 and 1/7, respectively, in every state. If we apply TD(0) on this trajectory, the parameters diverge to ∞ (Figure 35.10b), even though the problem appears simple! In contrast, with on-policy data (that is, when b is the same as π), TD(0) with linear approximation can be guaranteed to converge to a good value function approximate [TR97].

The divergence behavior is demonstrated in many value-based bootstrapping methods, including TD, Q-learning, and related approximate dynamic programming algorithms, where the value function is represented either linearly (like the example above) or nonlinearly [Gor95; Ber19]. The root cause

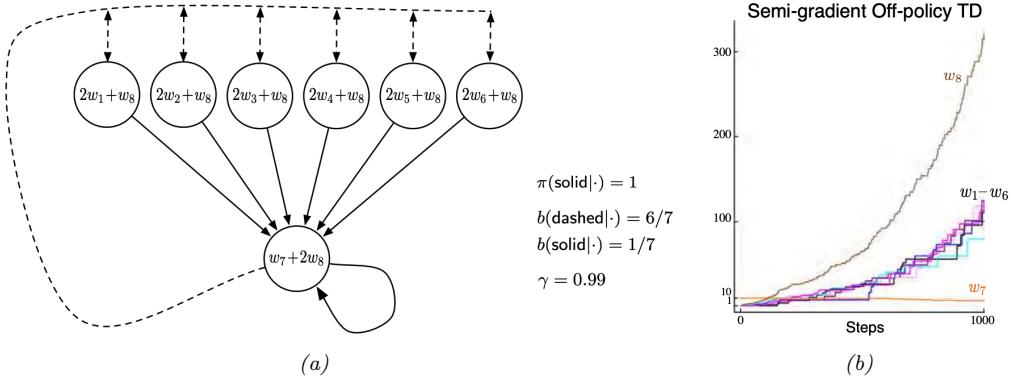


Figure 35.10: (a) A simple MDP. (b) Parameters of the policy diverge over time. From Figures 11.1 and 11.2 of [SB18]. Used with kind permission of Richard Sutton.

of these divergence phenomena is that the contraction property in the tabular case (Equation (34.87)) may no longer hold when V is approximated by V_w . An RL algorithm can become unstable when it has these three components: off-policy learning, bootstrapping (for faster learning, compared to MC), function approximation (for generalization in large scale MDPs). This combination is known as **the deadly triad** [SB18]. It highlights another important challenge introduced by off-policy learning, and is a subject of ongoing research (e.g., [van+18; Kum+19a]).

A general way to ensure convergence in off-policy learning is to construct an objective function function, the minimization of which leads to a good value function approximation; see [SB18, Ch. 11] for more background. A natural candidate is the discrepancy between the left and right hand sides of the Bellman optimality Equation (34.82), whose unique solution is V_* . However, the “max” operator is not friendly to optimization. Instead, we may introduce an entropy term to smooth the greedy policy, resulting in a differential square loss in **path consistency learning (PCL)** [Nac+17]:

$$\min_{V, \pi} \mathcal{L}^{\text{PCL}}(V, \pi) \triangleq \mathbb{E} \left[\frac{1}{2} (r + \gamma V(s') - \lambda \log \pi(a|s) - V(s))^2 \right] \quad (35.59)$$

where the expectation is over (s, a, r, s') tuples drawn from some off-policy distribution (e.g., uniform over \mathcal{D}). Minimizing this loss, however, does not result in the optimal value function and policy in general, due to an issue known as “double sampling” [SB18, Sec. 11.5].

This problem can be mitigated by introducing a dual function in the optimization [Dai+18]

$$\min_{V, \pi} \max_{\nu} \mathcal{L}^{\text{SBEED}}(V, \pi; \nu) \triangleq \mathbb{E} \left[\nu(s, a) (r + \gamma V(s') - \lambda \log \pi(a|s) - V(s))^2 - \nu(s, a)^2 / 2 \right] \quad (35.60)$$

where ν belongs to some function class (e.g., a DNN [Dai+18] or RKHS [FLL19]). It can be shown that optimizing Equation (35.60) forces ν to model the Bellman error. So this approach is called **smoothed Bellman error embedding**, or **SBEED**. In both PCL and SBEED, the objective can be optimized by gradient-based methods on parameterized value functions and policies.

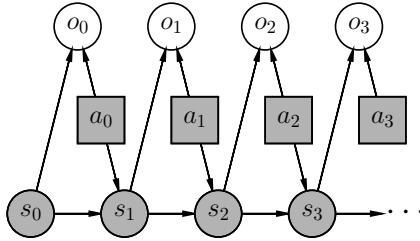


Figure 35.11: A graphical model for optimal control. States and actions are observed, while optimality variables are not. Adapted from Figure 1b of [Lev18].

35.6 Control as inference

In this section, we will discuss another approach to policy optimization, by reducing it to probabilistic inference. This is called **control as inference**, see e.g., [Att03; TS06; Tou09; BT12; KGO12; HR17; Lev18]. This approach allows one to incorporate domain knowledge in modeling, and apply powerful tools from approximate inference (see e.g., Chapter 7), in a consistent and flexible framework.

35.6.1 Maximum entropy reinforcement learning

We now describe a graphical model that exemplifies such a reduction, which results in RL algorithms that are closely related to some discussed previously. The model allows a trade-off between reward and entropy maximization, and recovers the standard RL setting when the entropy part vanishes in the trade-off. Our discussion mostly follows the approach of [Lev18].

Figure 35.11 gives a probabilistic model, which not only captures state transitions as before, but also introduces a new variable, o_t . This variable is binary, indicating whether the action at time t is optimal or not, and has the following probability distribution:

$$p(o_t = 1|s_t, a_t) = \exp(\lambda^{-1} R(s_t, a_t)) \quad (35.61)$$

for some temperature parameter $\lambda > 0$ whose role will be clear soon. In the above, we have assumed without much loss of generality that $R(s, a) < 0$, so that Equation (35.61) gives a valid probability. Furthermore, we can assume a non-informative, uniform action prior, $p(a_t|s_t)$, to simplify the exposition, for we can always push $p(a_t|s_t)$ into Equation (35.61). Under these assumptions, the likelihood of observing a length- T trajectory τ , when optimality achieved in every step, is:

$$\begin{aligned} p(\tau|\mathbf{o}_{0:T-1} = \mathbf{1}) &\propto p(\tau, \mathbf{o}_{0:T-1} = \mathbf{1}) \propto p(s_0) \prod_{t=0}^{T-1} p(o_t = 1|s_t, a_t) p_T(s_{t+1}|s_t, a_t) \\ &= p(s_0) \prod_{t=0}^{T-1} p_T(s_{t+1}|s_t, a_t) \exp\left(\frac{1}{\lambda} \sum_{t=0}^{T-1} R(s_t, a_t)\right) \end{aligned} \quad (35.62)$$

The intuition of Equation (35.62) is clearest when the state transitions are deterministic. In this case, $p_T(s_{t+1}|s_t, a_t)$ is either 1 or 0, depending on whether the transition is dynamically feasible or

not. Hence, $p(\boldsymbol{\tau} | \mathbf{o}_{0:T-1} = \mathbf{1})$ is either proportional to $\exp(\lambda^{-1} \sum_{t=0}^{T-1} R(s_t, a_t))$ if $\boldsymbol{\tau}$ is feasible, or 0 otherwise. Maximizing reward is equivalent to inferring a trajectory with maximum $p(\boldsymbol{\tau} | \mathbf{o}_{0:T-1} = \mathbf{1})$.

The optimal policy in this probabilistic model is given by

$$\begin{aligned} p(a_t | s_t, \mathbf{o}_{t:T-1} = \mathbf{1}) &= \frac{p(s_t, a_t | \mathbf{o}_{t:T-1} = \mathbf{1})}{p(s_t | \mathbf{o}_{t:T-1} = \mathbf{1})} = \frac{p(\mathbf{o}_{t:T-1} = \mathbf{1} | s_t, a_t) p(a_t | s_t) p(s_t)}{p(\mathbf{o}_{t:T-1} = \mathbf{1} | s_t) p(s_t)} \\ &\propto \frac{p(\mathbf{o}_{t:T-1} = \mathbf{1} | s_t, a_t)}{p(\mathbf{o}_{t:T-1} = \mathbf{1} | s_t)} \end{aligned} \quad (35.63)$$

The two probabilities in Equation (35.63) can be computed as follows, starting with $p(o_{T-1} = 1 | s_{T-1}, a_{T-1}) = \exp(\lambda^{-1} R(s_{T-1}, a_{T-1}))$,

$$p(\mathbf{o}_{t:T-1} = \mathbf{1} | s_t, a_t) = \int_S p(\mathbf{o}_{t+1:T-1} = \mathbf{1} | s_{t+1}) p_T(s_{t+1} | s_t, a_t) \exp(\lambda^{-1} R(s_t, a_t)) ds_{t+1} \quad (35.64)$$

$$p(\mathbf{o}_{t:T-1} = \mathbf{1} | s_t) = \int_A p(\mathbf{o}_{t:T-1} = \mathbf{1} | s_t, a_t) p(a_t | s_t) da_t \quad (35.65)$$

The calculation above is expensive. In practice, we can approximate the optimal policy using a parametric form, $\pi_\theta(a_t | s_t)$. The resulted probability of trajectory $\boldsymbol{\tau}$ now becomes

$$p_\theta(\boldsymbol{\tau}) = p(s_1) \prod_{t=0}^{T-1} p_T(s_{t+1} | s_t, a_t) \pi_\theta(a_t | s_t) \quad (35.66)$$

If we optimize θ so that $D_{\text{KL}}(p_\theta(\boldsymbol{\tau}) \| p(\boldsymbol{\tau} | \mathbf{o}_{0:T-1} = \mathbf{1}))$ is minimized, which can be simplified to

$$D_{\text{KL}}(p_\theta(\boldsymbol{\tau}) \| p(\boldsymbol{\tau} | \mathbf{o}_{0:T-1} = \mathbf{1})) = -\mathbb{E}_{p_\theta} \left[\sum_{t=0}^{T-1} \lambda^{-1} R(s_t, a_t) + \mathbb{H}(\pi_\theta(s_t)) \right] + \text{const} \quad (35.67)$$

where the constant term only depends on the uniform action prior $p(a_t | s_t)$, but not θ . In other words, the objective is to maximize total reward, with an entropy regularization favoring more uniform policies. Thus this approach is called **maximum entropy RL**, or **MERL**. If π_θ can represent all stochastic policies, a softmax version of the Bellman equation can be obtained for Equation (35.67):

$$Q_*(s_t, a_t) = \lambda^{-1} R(s_t, a_t) + \mathbb{E}_{p_T(s_{t+1} | s_t, a_t)} \left[\log \int_A \exp(Q_*(s_{t+1}, a_{t+1})) da \right] \quad (35.68)$$

with the convention that $Q_*(s_T, a) = 0$ for all a , and the optimal policy has a softmax form: $\pi_*(a_t | s_t) \propto \exp(Q_*(s_t, a_t))$. Note that the Q_* above is different from the usual optimal Q -function (Equation (34.83)), due to the introduction of the entropy term. However, as $\lambda \rightarrow 0$, their difference vanishes, and the softmax policy becomes greedy, recovering the standard RL setting.

The **soft actor-critic (SAC)** algorithm [Haa+18b; Haa+18c] is an off-policy actor-critic method whose objective function is equivalent to Equation (35.67) (by taking T to ∞):

$$J^{\text{SAC}}(\theta) \triangleq \mathbb{E}_{p_{\pi_\theta}^\infty(s) \pi_\theta(a|s)} [R(s, a) + \lambda \mathbb{H}(\pi_\theta(s))] \quad (35.69)$$

Note that the entropy term has also the added benefit of encouraging exploration.

To compute the optimal policy, similar to other actor-critic algorithms, we will work with the “soft” state- and action-function approximations, parameterized by \mathbf{w} and \mathbf{u} , respectively:

$$Q_{\mathbf{w}}(s, a) = R(s, a) + \gamma \mathbb{E}_{p_T(s'|s, a)} [V_{\mathbf{u}}(s', a') - \lambda \log \pi_{\boldsymbol{\theta}}(a'|s')] \quad (35.70)$$

$$V_{\mathbf{u}}(s, a) = \lambda \log \sum_a \exp(\lambda^{-1} Q_{\mathbf{w}}(s, a)) \quad (35.71)$$

This induces an improved policy (with entropy regularization): $\pi_{\mathbf{w}}(a|s) = \exp(\lambda^{-1} Q_{\mathbf{w}}(s, a))/Z_{\mathbf{w}}(s)$, where $Z_{\mathbf{w}}(s) = \sum_a \exp(\lambda^{-1} Q_{\mathbf{w}}(s, a))$ is the normalization constant. We then perform a soft policy improvement step to update $\boldsymbol{\theta}$ by minimizing $\mathbb{E}[D_{\text{KL}}(\pi_{\boldsymbol{\theta}}(s) \| \pi_{\mathbf{w}}(s))]$ where the expectation may be approximated by sampling s from a replay buffer D .

In [Haa+18c; Haa+18b], they show that the SAC method outperforms the off-policy DDPG algorithm (Section 35.3.5) and the on-policy PPO algorithm (Section 35.3.4) by a wide margin on various continuous control tasks. For more details, see [Haa+18c].

There is a variant of soft actor-critic, which only requires to model the action-value function. It is based on the observation that both the policy and soft value function can be induced by the soft action-value function as follows:

$$V_{\mathbf{w}}(s) = \lambda \log \sum_a \exp(\lambda^{-1} Q_{\mathbf{w}}(s, a)) \quad (35.72)$$

$$\pi_{\mathbf{w}}(a|s) = \exp(\lambda^{-1}(Q_{\mathbf{w}}(s, a) - V_{\mathbf{w}}(s))) \quad (35.73)$$

We then only need to learn \mathbf{w} , using approaches similar to DQN (Section 35.2.6). The resulting algorithm, **soft Q-learning** [SAC17], is convenient if the number of actions is small (when \mathcal{A} is discrete), or if the integral in obtaining $V_{\mathbf{w}}$ from $Q_{\mathbf{w}}$ is easy to compute (when \mathcal{A} is continuous).

It is interesting to see that algorithms derived in the maximum entropy RL framework bears a resemblance to PCL and SBEED in Section 35.5.3, both of which were to minimize an objective function resulting from the entropy-smoothed Bellman equation.

35.6.2 Other approaches

VIREL is an alternative model to maximum entropy RL [Fel+19]. Similar to soft actor-critic, it uses an approximate action-value function, $Q_{\mathbf{w}}$, a stochastic policy, $\pi_{\boldsymbol{\theta}}$, and a binary optimality random variable o_t at time t . A different probability model for o_t is used

$$p(o_t = 1|s_t, a_t) = \exp\left(\frac{Q_{\mathbf{w}}(s_t, a_t) - \max_a Q_{\mathbf{w}}(s_t, a)}{\lambda_{\mathbf{w}}}\right) \quad (35.74)$$

The temperature parameter $\lambda_{\mathbf{w}}$ is also part of the parameterization, and can be updated from data.

An EM method can be used to maximize the objective

$$\mathcal{L}(\mathbf{w}, \boldsymbol{\theta}) = \mathbb{E}_{p(s)} \left[\mathbb{E}_{\pi_{\boldsymbol{\theta}}(a|s)} \left[\frac{Q_{\mathbf{w}}(s, a)}{\lambda_{\mathbf{w}}} \right] + \mathbb{H}(\pi_{\boldsymbol{\theta}}(s)) \right] \quad (35.75)$$

for some distribution p that can be conveniently sampled from (e.g., in a replay buffer). The algorithm may be interpreted as an instance of actor-critic. In the E-step, the critic parameter \mathbf{w} is fixed, and the actor parameter $\boldsymbol{\theta}$ is updated using gradient ascent with stepsize $\eta_{\boldsymbol{\theta}}$ (for policy improvement):

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\mathbf{w}, \boldsymbol{\theta}) \quad (35.76)$$

In the M-step, the actor parameter is fixed, and the critic parameter is updated (for policy evaluation):

$$\mathbf{w} \leftarrow \mathbf{w} + \eta_{\mathbf{w}} \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}, \boldsymbol{\theta}) \quad (35.77)$$

Finally, there are other possibilities of reducing optimal control to probabilistic inference, in addition to MERL and VIREL. For example, we may aim to maximize the expectation of the trajectory return G , by optimizing the policy parameter $\boldsymbol{\theta}$:

$$J(\pi_{\boldsymbol{\theta}}) = \int G(\boldsymbol{\tau}) p(\boldsymbol{\tau} | \boldsymbol{\theta}) d\boldsymbol{\tau} \quad (35.78)$$

It can be interpreted as a pseudo-likelihood function, when the $G(\boldsymbol{\tau})$ is treated as probability density, and solved (approximately) by a range of algorithms (see e.g., [PS07; Neu11; Abd+18]). Interestingly, some of these methods have a similar objective as MERL (Equation (35.67)), although the distribution involving $\boldsymbol{\theta}$ appears in the second argument of D_{KL} . As discussed in Section 2.7.1, this forwards KL-divergence is mode-covering, which in the context of RL is argued to be less preferred than the mode-seeking, reverse KL-divergence used by MERL. For more details and references, see [Lev18].

Control as inference is also closely related to **active inference**; this is based on the **free energy principle** which is popular in neuroscience (see e.g., [Fri09; Buc+17; SKM18; Ger19; Maz+22]). The FEP is equivalent to using variational inference (see Section 10.1) to perform state estimation (perception) and parameter estimation (learning). In particular, consider a latent variable model with hidden states \mathbf{s} , observations \mathbf{y} , and parameters $\boldsymbol{\theta}$. Following Section 10.1.1.1, we define the variational free energy to be $\mathcal{F}[p, q](\mathbf{y}) = D_{\text{KL}}(q(\mathbf{s}, \boldsymbol{\theta} | \mathbf{y}) \| p(\mathbf{s}, \boldsymbol{\theta} | \mathbf{y})) - \log p(\mathbf{y})$. State estimation corresponds to solving $\min_{q(\mathbf{s} | \mathbf{y}, \boldsymbol{\theta})} \mathcal{F}(\mathbf{y})$, and parameter estimation corresponds to solving $\min_{q(\boldsymbol{\theta} | \mathbf{y})} \mathcal{F}(\mathbf{y})$, just as in variational Bayes EM (Section 10.3.5).

If $p(\mathbf{s}, \mathbf{y} | \boldsymbol{\theta})$ is a nonlinear hierarchical Gaussian model, and $q(\mathbf{s} | \mathbf{y}, \boldsymbol{\theta})$ is a Gaussian mean field approximation — where $q(\mathbf{s} | \mathbf{y}, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{s} | \hat{\mathbf{s}}, \mathbf{H})$ is a Laplace approximation, with the mode $\hat{\mathbf{s}}$, being computed using gradient descent, and \mathbf{H} being the Hessian at the mode — then we recover the method known as **predictive coding** (see e.g., [RB99; Fri03; Spr17; HM20; MSB21; Mar21; OK22; Sal+23]). This can be considered a non-amortized version of a VAE (Section 21.2), where inference (E step) is done with iterated gradient descent, and parameter estimation (M step) is also done with gradient descent. (A more efficient incremental EM version of predictive coding, which updates $\{\hat{\mathbf{s}}_n : n = 1 : N\}$ and $\boldsymbol{\theta}$ in parallel, was recently presented in [Sal+24].) For more details on predictive coding, see [Supplementary](#) Section 8.1.4.)

To extend the above method to decision making problems we define the **expected free energy** as $\bar{\mathcal{F}}(a) = \mathbb{E}_{q(\mathbf{y}|a)} [\mathcal{F}(\mathbf{y})]$, where $q(\mathbf{y}|a)$ is the posterior predictive distribution over observations given actions sequence a . We then define the policy to be $\pi(a) = \text{softmax}(\bar{\mathcal{F}}(a))$. To guide the agent towards preferred outcomes, we define the prior over states as $p(\mathbf{s}) \propto e^{R(s)}$, where R is the reward function. Alternatively, we can define the prior over observations as $p(\mathbf{y}) \propto e^{R(y)}$. Either way, the generative model is defined in terms of what the agent wants to achieve, rather than being an “objective” model of reality. The advantage of this approach is that it automatically induces *goal-directed* information-seeking behavior, rather than than the maxent approach which models uncertainty in a goal-independent way. Despite this difference, the technique of active inference is very similar to control as inference, as explained in [Mil+20; WIP20; LÖW21].

35.6.3 Imitation learning

In previous sections, an RL agent is to learn an optimal sequential decision making policy so that the total reward is maximized. **Imitation learning** (IL), also known as **apprenticeship learning** and **learning from demonstration** (LfD), is a different setting, in which the agent does not observe rewards, but has access to a collection \mathcal{D}_{exp} of trajectories generated by an expert policy π_{exp} ; that is, $\tau = (s_0, a_0, s_1, a_1, \dots, s_T)$ and $a_t \sim \pi_{\text{exp}}(s_t)$ for $\tau \in \mathcal{D}_{\text{exp}}$. The goal is to learn a good policy by imitating the expert, in the absence of reward signals. IL finds many applications in scenarios where we have demonstrations of experts (often humans) but designing a good reward function is not easy, such as car driving and conversational systems. See [Osa+18] for a survey up to 2018.

35.6.3.1 Imitation learning by behavior cloning

A natural method is **behavior cloning**, which reduces IL to supervised learning; see [Pom89] for an early application to autonomous driving. It interprets a policy as a classifier that maps states (inputs) to actions (labels), and finds a policy by minimizing the imitation error, such as

$$\min_{\pi} \mathbb{E}_{p_{\pi_{\text{exp}}}^{\infty}(s)} [D_{\text{KL}}(\pi_{\text{exp}}(s) \parallel \pi(s))] \quad (35.79)$$

where the expectation wrt $p_{\pi_{\text{exp}}}^{\infty}$ may be approximated by averaging over states in \mathcal{D}_{exp} . A challenge with this method is that the loss does not consider the sequential nature of IL: future state distribution is not fixed but instead depends on earlier actions. Therefore, if we learn a policy $\hat{\pi}$ that has a low imitation error under distribution $p_{\pi_{\text{exp}}}^{\infty}$, as defined in Equation (35.79), it may still incur a large error under distribution $p_{\hat{\pi}}^{\infty}$ (when the policy $\hat{\pi}$ is actually run). Further expert demonstrations or algorithmic augmentations are often needed to handle the distribution mismatch (see e.g., [DLM09; RGB11]).

35.6.3.2 Imitation learning by inverse reinforcement learning

An effective approach to IL is **inverse reinforcement learning** (IRL) or **inverse optimal control** (IOC). Here, we first infer a reward function that “explains” the observed expert trajectories, and then compute a (near-)optimal policy against this learned reward using any standard RL algorithms studied in earlier sections. The key step of reward learning (from expert trajectories) is the opposite of standard RL, thus called inverse RL [NR00a].

It is clear that there are infinitely many reward functions for which the expert policy is optimal, for example by several optimality-preserving transformations [NHR99]. To address this challenge, we can follow the maximum entropy principle (Section 2.4.7), and use an energy-based probability model to capture how expert trajectories are generated [Zie+08]:

$$p(\tau) \propto \exp \left(\sum_{t=0}^{T-1} R_{\theta}(s_t, a_t) \right) \quad (35.80)$$

where R_{θ} is an unknown reward function with parameter θ . Abusing notation slightly, we denote by $R_{\theta}(\tau) = \sum_{t=0}^{T-1} R_{\theta}(s_t, a_t)$ the cumulative reward along the trajectory τ . This model assigns exponentially small probabilities to trajectories with lower cumulative rewards. The partition function, $Z_{\theta} \triangleq \int_{\tau} \exp(R_{\theta}(\tau))$, is in general intractable to compute, and must be approximated. Here, we can

take a sample-based approach. Let \mathcal{D}_{exp} and \mathcal{D} be the sets of trajectories generated by an expert, and by some known distribution q , respectively. We may infer $\boldsymbol{\theta}$ by maximizing the likelihood, $p(\mathcal{D}_{\text{exp}}|\boldsymbol{\theta})$, or equivalently, minimizing the negative log-likelihood loss

$$\mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{|\mathcal{D}_{\text{exp}}|} \sum_{\tau \in \mathcal{D}_{\text{exp}}} R_{\boldsymbol{\theta}}(\tau) + \log \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\exp(R_{\boldsymbol{\theta}}(\tau))}{q(\tau)} \quad (35.81)$$

The term inside the log of the loss is an importance sampling estimate of Z that is unbiased as long as $q(\tau) > 0$ for all τ . However, in order to reduce the variance, we can choose q adaptively as $\boldsymbol{\theta}$ is being updated. The optimal sampling distribution (Section 11.5), $q_*(\tau) \propto \exp(R_{\boldsymbol{\theta}}(\tau))$, is hard to obtain. Instead, we may find a policy $\hat{\pi}$ which induces a distribution that is close to q_* , for instance, using methods of maximum entropy RL discussed in Section 35.6.1. Interestingly, the process above produces the inferred reward $R_{\boldsymbol{\theta}}$ as well as an approximate optimal policy $\hat{\pi}$. This approach is used by **guided cost learning** [FLA16], and found effective in robotics applications.

35.6.3.3 Imitation learning by divergence minimization

We now discuss a different, but related, approach to IL. Recall that the reward function depends only on the state and action in an MDP. It implies that if we can find a policy π , so that $p_{\pi}^{\infty}(s, a)$ and $p_{\pi_{\text{exp}}}^{\infty}(s, a)$ are close, then π receives similar long-term reward as π_{exp} , and is a good imitation of π_{exp} in this regard. A number of IL algorithms find π by minimizing the divergence between p_{π}^{∞} and $p_{\pi_{\text{exp}}}^{\infty}$. We will largely follow the exposition of [GZG19]; see [Ke+19b] for a similar derivation.

Let f be a convex function, and D_f the f -divergence (Section 2.7.1). From the above intuition, we want to minimize $D_f(p_{\pi_{\text{exp}}}^{\infty} \| p_{\pi}^{\infty})$. Then, using a variational approximation of D_f [NWJ10a], we can solve the following optimization problem for π :

$$\min_{\pi} \max_{\mathbf{w}} \mathbb{E}_{p_{\pi_{\text{exp}}}^{\infty}(s, a)} [T_{\mathbf{w}}(s, a)] - \mathbb{E}_{p_{\pi}^{\infty}(s, a)} [f^*(T_{\mathbf{w}}(s, a))] \quad (35.82)$$

where $T_{\mathbf{w}} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is a function parameterized by \mathbf{w} . The first expectation can be estimated using \mathcal{D}_{exp} , as in behavior cloning, and the second can be estimated using trajectories generated by policy π . Furthermore, to implement this algorithm, we often use a parametric policy representation π_{θ} , and then perform stochastic gradient updates to find a saddle-point to Equation (35.82).

With different choices of the convex function f , we can obtain many existing IL algorithms, such as **generative adversarial imitation learning (GAIL)** [HE16b] and **adversarial inverse RL (AIRL)** [FLL18], as well as new algorithms like **f-divergence max-ent IRL (f-MAX)** and **forward adversarial inverse RL (FAIRL)** [GZG19; Ke+19b].

Finally, the algorithms above typically require running the learned policy π to approximate the second expectation in Equation (35.82). In risk- or cost-sensitive scenarios, collecting more data is not always possible. Instead, we are in the off-policy IL setting, working with trajectories collected by some policy other than π . Hence, we need to correct the mismatch between p_{π}^{∞} and the off-policy trajectory distribution, for which techniques from Section 35.5 can be used. An example is **ValueDICE** [KNT20], which uses a similar distribution correction method of DualDICE (Section 35.5.2).

36 Causality

This chapter is written by Victor Veitch and Alex D'Amour.

36.1 Introduction

The bulk of machine learning considers relationships between observed variables with the goal of summarizing these relationships in a manner that allows predictions on similar data. However, for many problems, our main interest is to predict how system would change if it were observed under different conditions. For instance, in healthcare, we are interested in whether a patient will recover if given a certain treatment (as opposed to whether treatment and recovery are associated in the observed data). **Causal inference** addresses how to formalize such problems, determine whether they can be solved, and, if so, how to solve them. This chapter covers the fundamentals of this subject. Code examples for the discussed methods are available at <https://github.com/vveitch/causality-tutorials>. For more information on the connections between ML and causal inference, see e.g., [Kad+22; Xia+21a].

To make the gap between observed data modeling and causal inference concrete, consider the relationships depicted in Figure 36.1a and Figure 36.1b. Figure 36.1a shows the relationship between deaths by drowning and ice cream production in the United States in 1931 (the pattern holds across most years). Figure 36.1b shows the relationship between smoking and lung cancer across various countries. In each case, there is a strong positive association. Faced with this association, we might ask: could we reduce drowning deaths by banning ice cream? Could we reduce lung cancer by banning cigarettes? We intuitively understand that these interventional questions have different answers, despite the fact that the observed associations are similar. Determining the causal effect of some intervention in the world requires some such causal hypothesis about the world.

For concreteness, consider three possible explanations for the association between ice cream and drowning. Perhaps eating ice cream does cause people to drown — due to stomach cramps or similar. Or, perhaps, drownings increase demand for ice cream — the survivors eat huge quantities of ice cream to handle their grief. Or, the association may be due (at least in part) to a common cause: warm weather makes people more likely to eat ice cream and more likely to go swimming (and, hence, to drown). Under all three scenarios, we can observe exactly the same data, but the implications for an ice cream ban are very different. Hence, answering questions about what will happen under an intervention requires us to incorporate some causal knowledge of the world — e.g., which of these scenarios is plausible?

Our goal in this chapter to introduce the essentials of estimating causal effects. The high-level