Short course on RL theory

Gaurav Mahajan

May 2025

Contents

Chapter 1

Basics of Reinforcement Learning

In this chapter, we begin with motivating examples and then introduce the mathematical framework used to model decision-making problems in reinforcement learning. Let's start with a few real-world scenarios that illustrate the types of problems reinforcement learning aims to solve. These examples are described informally, without invoking RL-specific terminology. Our goal is to revisit them later and show how they can be modeled using the RL framework.

1.1 The Slot Machine Problem: Exploration vs Exploitation.

Imagine walking into a casino and finding 3 slot machines lined up in front of you. Each machine, when played, gives a random reward between 0 and 100. The machines differ in how they are programmed — some machines may tend to give higher rewards on average, while others may be worse.

Figure 1.1: An example of 3 slot machines. Some slot machines may give higher rewards on average.

You only have time to play a total of 1000 times, across all the machines combined, and your goal is to maximize the total reward you earn. This scenario raises an important question: how should you decide which machines to play?

- On the one hand, you would like to *explore* the different machines, by trying each of them enough times to estimate how good they are.
- On the other hand, you would also like to *exploit* your current knowledge by repeatedly playing the machines that seem to offer the best rewards so far.

Balancing this trade-off between *exploration* (gathering information) and *exploitation* (maximizing reward based on current knowledge) is one of the core challenges in reinforcement learning.

1.2 Learning to Play Chess: Credit Assignment

Let's consider another scenario. Suppose you are an AI agent learning how to play chess against a human opponent. At each turn, the agent observes the current *state* of the game — that is, the configuration of all pieces on the board — and decides which move to make.

Figure 1.2: An example chess board configuration. The agent observes the state of the board and decides on a move.

In principle, if the agent could determine the best move for every possible board configuration, it would be able to play optimally. However, learning what the best move is in each state is a highly challenging task for several reasons.

- First, the agent only receives feedback at the end of each game: it either wins, loses, or draws. The consequence of individual moves is not immediately known.
- Secondly, since chess involves rich strategic interactions between pieces, it can be hard to determine which specific actions, or sequences of actions, are responsible for this final outcome.

1.3 Reinforcement Learning Formulation

Both of the examples above, as well as many other real-world problems, can be cast in the reinforcement learning (RL) framework. In this framework, we model the interaction between two entities: **The agent**, which makes decisions by observing its current situation and choosing actions. **The environment**, which responds to the agent's actions by providing feedback in the form of rewards and updating the situation accordingly. The interaction proceeds as follows:

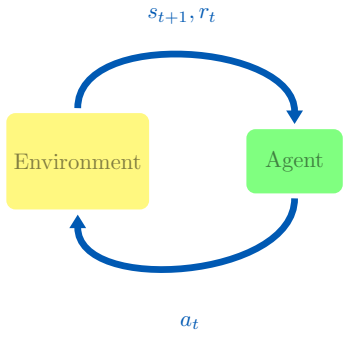

Figure 1.3: Interaction loop between the agent and environment in reinforcement learning. At each time step t, the agent observes the state s_t, takes an action a_t, and receives a new state s_{t+1} and reward r_t.

1. The agent begins in some *state* representing the current situation.
2. Based on the observed state, the agent chooses an *action* from the set of allowed actions.
3. The environment responds by:
   - Providing the agent with a *reward*, which reflects the immediate quality of the chosen action in the current state.
   - Updating the *state*, potentially in a stochastic (random) manner, based on the agent's action.
4. The agent observes the new state and repeats the process.

This sequence continues over a series of steps, often called an *episode*, which may end after a fixed number of steps or upon reaching a special terminal state. The goal of the agent is to learn a strategy that maximizes its expected cumulative reward over time. We now formalize this setup mathematically.

1.4 Markov Decision Process

The standard framework for modeling reinforcement learning problems is the *Markov Decision Process (MDP)*. This framework captures the sequential nature of decision-making under uncertainty.

Definition 1.1 (Markov Decision Process)**.** *A **finite-horizon Markov Decision Process (MDP)** is described by:*

1. State space S*: the set of all possible states.*
2. Action space A*: the set of all actions available to the agent.*
3. Transition function $T : S \times A \to \Delta(S)$[1]*: The transition function describes the next state given the current state and the action chosen by the agent. If the current state is s, and the agent takes action a, the next state s' is drawn from the distribution $T(s, a)$.*
4. Reward function $R : S \times A \to \Delta([0, 1])$*: The reward function describes the immediate reward given the current state and the action chosen by the agent. If the current state is s, and the agent takes action a, then the reward r is drawn from the distribution $R(s, a)$.*
5. Initial state $s_0 \in S$*: the starting state of each episode.*
6. Horizon $H \in \mathbb{N}$*: the number of time steps over which the agent interacts with the environment in an episode, before restarting from initial state s_0.*

Remark 1.2. *Instead of a fixed starting state s_0, one can assume an initial state distribution $\mu \in \Delta(S)$. This is a more general model, but assuming a fixed starting state simplifies analysis and is without loss of generality: a random initial state can be simulated by adding a dummy initial state and increasing the horizon by 1.*

We now formally define how an agent interacts with the environment in a given MDP. The interaction follows the following protocol:

1. The agent starts in state s_0.
2. For each time step $t = 0, 1, \ldots, H - 1$
   (a) The agent observes the state s_t
   (b) The agent selects an action $a_t \in A$ based on its policy.
   (c) The environment provides a reward $r_t \sim R(s_t, a_t)$.
   (d) The environment transitions to the next state $s_{t+1} \sim T(s_t, a_t)$.

[1] Given any finite set K, define $\Delta(K)$ as the set of all probability distributions over K, i.e.,

$$\Delta(K) := \{p : K \to [0, 1] : \sum_{k \in K} p(k) = 1\}.$$

The decisions made by the agent are specified by a *policy*. Formally, a policy is a function $\pi : S \to \Delta(A)$, where $\pi(s)$ is a distribution over actions given the current state s. That is, at each time t, the agent samples action $a_t \sim \pi(s_t)$.

Definition 1.3 (Goal)**.** *The goal of the agent is to find a policy that maximizes the expected cumulative reward over the horizon. In other words, the objective is to solve the optimization problem:*

$$\max_{\pi} \mathbb{E}_{T,R,\pi} \left[\sum_{t=0}^{H-1} r_t \right],$$

where the expectation is over the randomness in the transition function, reward function, and the policy.

In reality (similar to standard PAC learning settings), we only expect to find an approximate optimal policy with some constant probability.

1.5 Unrolling the MDP

In general, the transition function is stochastic, but to build intuition, it will also be useful to consider deterministic transition function. When the transitions are deterministic, we can visualize the unrolling of the MDP, into a tree (Figure 1.4).

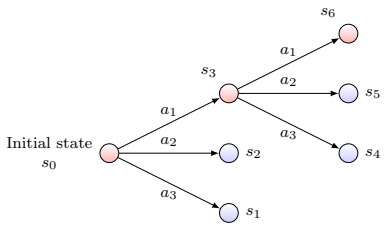

Figure 1.4: Unrolling of an MDP with states $S = \{s_0, s_1, s_2, \ldots, s_6\}$ and actions $A = \{a_1, a_2, a_3\}$. Each node corresponds to some state $s \in S$; edges correspond to some action $a \in A$, and transition to some state $s' \in S$ given by $T(s, a)$. The root of the tree is the initial state s_0.

1.6 Value Functions

To reason about how good a policy is, we define two central objects in reinforcement learning: the state value function and the state-action value function.

Definition 1.4 (Value Functions)**.** *Let $\pi : S \to \Delta(A)$ be a policy, and let H be the horizon. For each time step $h \in \{0, \ldots, H-1\}$, we define:*

- *The* state value function $V_h^\pi : S \to \mathbb{R}$ *as:*

$$V_h^\pi(s) := \mathbb{E}_{\pi,T,R}\left[\sum_{t=h}^{H-1} r_t \;\middle|\; s_h = s\right],$$

i.e., the expected total reward collected from time h to $H-1$, starting in state s and following policy π.

- *The* state-action value function $Q_h^\pi : S \times A \to \mathbb{R}$ *as:*

$$Q_h^\pi(s,a) := \mathbb{E}_{\pi,T,R}\left[\sum_{t=h}^{H-1} r_t \;\middle|\; s_h = s, a_h = a\right],$$

i.e., the expected total reward obtained by taking action a in state s at time h, and then following policy π until the end of the episode.

To simplify notation, we embed the time step h into the state (e.g., by replacing s with the pair (s,h)), and will often use $V^\pi(s)$ and $Q^\pi(s,a)$ to denote the value functions. We use V^* and Q^* to denote the value functions $V^*(s) = \max_\pi V^\pi(s)$ and $Q^*(s,a) = \max_\pi Q^\pi(s,a)$ and refer to the corresponding policy as the optimal policy π^*[2].

[2]even though optimal policy π^* is not unique, $V^*(s)$ and $Q^*(s,a)$ defined as $\max_\pi V^\pi(s)$ and $\max_\pi Q^\pi(s,a)$ are unique.

Chapter 2

Policy Gradient Methods

Recall that the goal of reinforcement learning is to solve the following optimization problem:

$$\max_{\pi} V^{\pi}(s_0) := \max_{\pi} \mathbb{E}_{T,R,\pi}\left[\sum_{t=0}^{H-1} r_t\right],$$

where s_0 is a fixed initial state. Policy gradient methods approach this problem by iteratively improving the policy using local gradient-based updates.

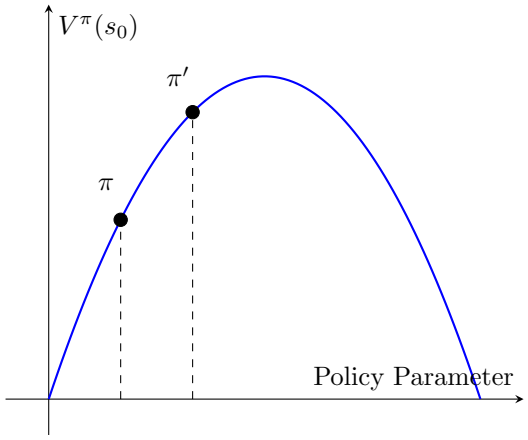

Figure 2.1: Gradient ascent step improves the policy value $V^{\pi}(s_0)$.

Suppose we have a policy π with value $V^{\pi}(s_0)$. Unless π is optimal, there always exists a small perturbation of the policy that leads to a higher expected return. Specifically, for a small change in policy from π to π', we can approximate the change in value as:

$$V^{\pi'}(s_0) \approx V^{\pi}(s_0) + \nabla_{\pi} V^{\pi}(s_0) \cdot (\pi' - \pi),$$

where $\nabla_\pi V^\pi(s_0)$ is the gradient of the value function with respect to the policy parameters. This suggests that we can improve the policy by moving in the direction of the gradient:

$$\pi' = \pi + \alpha \nabla_\pi V^\pi(s_0),$$

where $\alpha > 0$ is a learning rate. This update rule is the foundation of policy gradient methods, which directly optimize the expected return by estimating and following the gradient (see Figure 2.1 for an illustration).

In the rest of this chapter, we introduce a simple gradient-based algorithm, and implement it to solve two environments: CartPole and Pong.

2.1 Environments: CartPole and Pong

In this chapter, we focus on implementing policy gradient methods to solve two benchmark environments in reinforcement learning: **CartPole** and **Pong**. These environments differ in terms of state representation, action space, and the challenges they present for learning-based algorithms. We begin by introducing each environment in detail.

2.1.1 CartPole Environment

The CartPole environment is a classic control problem where the objective is to prevent a pole, hinged on a cart, from falling over. The agent can apply a force of fixed magnitude to the left or right of the cart to balance the pole.

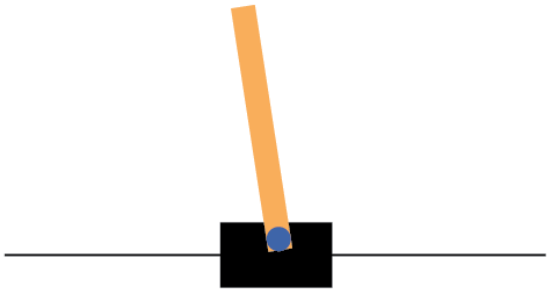

Figure 2.2: Screenshot of the CartPole environment. The agent applies left/right forces to the cart to keep the pole balanced.

- **Observation space:** A 4-dimensional continuous vector:

  $$\text{obs} = (\text{cart position}, \text{cart velocity}, \text{pole angle}, \text{pole angular velocity})$$

  The observation limits are summarized in the table below:

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	$-\infty$	∞
2	Pole Angle	~ -0.418 rad ($-24°$)	~ 0.418 rad ($24°$)
3	Pole Angular Velocity	$-\infty$	∞

- **Action space:** A discrete set $\{0, 1\}$, where 0 represents pushing the cart to the left, and 1 to the right.
- **Reward:** The agent receives a reward of $+1$ for every timestep that the pole remains upright.
- **Episode termination:** An episode ends when the pole angle exceeds $\pm 24°$ or the cart moves beyond ± 4.8 units from the center or 500 timesteps.

We use the implementation available in the Gymnasium library [1] under the environment name `CartPole-v1`.

2.1.2 Pong Environment

Pong is a visually rich environment based on the Atari 2600 game. The agent controls a paddle and aims to win a game of Pong against a built-in opponent. The objective of the game is to keep deflecting the ball away from your goal and into the opponent's goal.

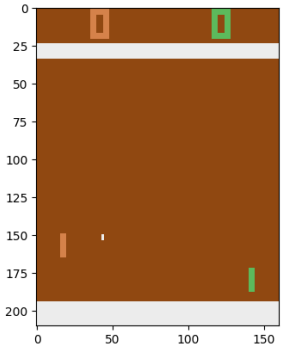

Figure 2.3: Frame from the Pong environment. Green paddle is the agent, orange is the opponent.

- **Observation space:** An RGB image of shape $(210, 160, 3)$ representing the full screen.

- **Action space:** A discrete set of 6 actions, of which we restrict to 3 meaningful ones: `NOOP`, `UP`, and `DOWN`.
- **Reward:** +1 when the agent scores a point, -1 when the agent loses a point, and 0 otherwise.
- **Episode termination:** A game ends when one of the players reaches 21 points.

We use the implementation from the ALE (Arcade Learning Environment) interface in Gymnasium [2], under the environment name `ALE/Pong-v5`.

2.2 Vanilla Policy Gradient Algorithm

In this section, we describe how to solve CartPole and Pong using the policy gradient algorithm with a linear policy. We first formalize the policy representation, then present the algorithm and explain the gradient derivation used to update the policy parameters.

2.2.1 Policy Parameterization

Let the state space be $S \subseteq \mathbb{R}^d$ and the action space be finite, $A = \{1, 2, \ldots, K\}$, where d is the dimension of the observation vector and $K = |A|$ is the number of discrete actions. For CartPole and Pong, (d, K) are $(4, 2)$ and $(210 * 160 * 3 = 100800, 3)$ respectively. We define a policy $\pi_\theta : S \to \Delta(A)$ using a softmax function over a linear function of the state:

$$\pi_\theta(a \mid s) = \frac{\exp(\langle \theta_a, s \rangle)}{\sum_{a' \in A} \exp(\langle \theta_{a'}, s \rangle)},$$

where $\theta = (\theta_1, \ldots, \theta_K)$ is a matrix in $\mathbb{R}^{K \times d}$, and $\theta_a \in \mathbb{R}^d$ is the parameter vector associated with action a. The policy assigns a probability distribution over actions based on the current state.

2.2.2 Algorithm

Our goal is to find a policy π_θ that maximizes expected return from a fixed initial state s_0:

$$\max_\theta V(\theta) \quad \text{where } V(\theta) = \mathbb{E}_{T,R,\pi_\theta}\left[\sum_{t=0}^{H-1} r_t\right].$$

The policy gradient algorithm performs gradient ascent to improve the policy, using the update

$$\theta \to \theta + \nabla_\theta V(\theta)$$

Since, we can only get an estimator for the function $V(\theta)$, we can get an unbiased estimator for the policy gradient: Let $\tau = (s_0, a_0, r_0, s_1, \ldots, s_H)$ be a trajectory generated by following policy π_θ,

$$\nabla_\theta V(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{H-1} \left(\sum_{t=0}^{H-1} r_t \right) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right].$$

We now present the algorithm using this gradient estimate:

Algorithm 1: Vanilla Policy Gradient with Linear Policy

Input: Learning rate α, number of episodes N
Output: Learned policy parameters $\theta \in \mathbb{R}^{K \times d}$

{1} Initialize $\theta \leftarrow 0$
{2} **for** $i = 1$ *to* N **do**
{3} Sample a trajectory $\tau = (s_0, a_0, r_0, \ldots, s_H)$ using policy π_θ
{4} Compute return: $R = \sum_{t=0}^{H-1} r_t$
{5} Update parameters: $\theta \leftarrow \theta + \alpha \cdot \sum_{t=0}^{H-1} R \cdot \nabla_\theta \log \pi_\theta(a_t \mid s_t)$
{6} **return** θ

We can replace the total return term $\sum_{t=0}^{H-1} r_t$ with other functions that depend on π_θ, s_t, a_t, such as the state-action value function $Q^{\pi_\theta}(s_t, a_t)$ or the advantage function $A^{\pi_\theta}(s_t, a_t) = Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)$. Algorithms like TRPO [3] and PPO [4] use these modifications and typically achieve better performance than the vanilla policy gradient method in practice.

2.3 Implementing in Python

We now describe how to implement the vanilla policy gradient algorithm from Algorithm 1 in Python, using PyTorch and Gymnasium.

2.3.1 Environment Setup and Dependencies

We recommend using a virtual environment to keep dependencies isolated.

Step 1: Create a virtual environment

```
python3 -m venv .venv
source .venv/bin/activate  # on Linux/macOS
.venv\Scripts\activate.bat # on Windows
```

Running the above code for the first time will lead to an error requiring Xcode developer tools, followed by a prompt to install the necessary tools. Since this installation requires a stable internet connection, it's best to complete it a day in advance.

Step 2: Install required packages

```
pip install --upgrade pip
pip install torch
pip install "gymnasium[classic-control]"
```

Step 3: Verify installation Check that PyTorch and Gymnasium work correctly:

```
python -c "import torch; print(torch.__version__)"
python -c "import gymnasium as gym; env = gym.make('CartPole-v1'); print(env)"
```

2.3.2 Training on CartPole

Below is a full implementation using a linear softmax policy for CartPole:

```python
# Libraries to help construct neural networks
# and train them.
import torch
import torch.nn as nn
import torch.optim as optim

# Library to create and interact with environments.
import gymnasium as gym

# Define the policy network
# It first applies a linear mapping from the state space to hidden layer,
# and then applies a softmax activation to obtain the action probabilities.

class LinearPolicy(nn.Module):
    def __init__(self, obs_dim, n_actions):
        super().__init__()
        # This has obs_dim x n_actions many parameters.
        # Useful to inutively see how much data you need to train this.
        self.linear = nn.Linear(obs_dim, n_actions)

    def forward(self, x):
        # Basically if self.linear(x) = [2, 3, -4]
        # softmax(self.linear(x)) will normalize it to  [0.4, 0.5, 0.1]
        # And if we do batch, then x is [batch_size, obs_dim]
        # self.linear(x) will be [batch_size, n_actions]
        # softmax(self.linear(x)) will be [batch_size, n_actions]
        return torch.softmax(self.linear(x), dim=-1)

# Next, we create our Cartpole environment
env = gym.make("CartPole-v1")

# Here is how the observation and action space looks like
obs_dim = env.observation_space.shape[0]
```

```
n_actions = env.action_space.n
print(f"S = R^{obs_dim} and A = {list(range(n_actions))}")

policy = LinearPolicy(obs_dim, n_actions)
optimizer = optim.Adam(policy.parameters(), lr=1e-2)

# We train for 1000 episodes
for episode in range(1000):
    # We get our starting state from the environment
    obs, _ = env.reset()

    # We need to store log probabilities and rewards for computing gradients
    log_probs = []
    rewards = []
    done = False

    # We run our current policy until the end of episode
    while not done:
        # Turn our state into a tensor (because nn uses tensors)
        obs_tensor = torch.tensor(obs, dtype=torch.float32)

        # Compute action probabilities under our policy
        # Sample an action.
        action_probs = policy(obs_tensor)
        dist = torch.distributions.Categorical(action_probs)
        action = dist.sample()

        # Execute the action in the environment, collect reward, new state
        obs, reward, terminated, truncated, _ = env.step(action.item())

        # We need log probabilities and rewards to compute gradient, so lets save that
        log_probs.append(dist.log_prob(action))
        rewards.append(reward)

        done = terminated or truncated

    # We need to turn rewards into tensors before using them.
    rewards = torch.tensor(rewards, dtype=torch.float32)

    # Compute the loss to propagate the gradients
    loss = -sum(log_probs)*sum(rewards)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if episode % 50 == 0:
        print(f"Episode {episode}, return = {sum(rewards)}")
```

2.3.3 Visualizing the Learned CartPole Policy

To play back the trained policy:

```
env = gym.make("CartPole-v1", render_mode="human")

for episode in range(5):
    obs, _ = env.reset()
    done = False
    while not done:
        obs_tensor = torch.tensor(obs, dtype=torch.float32)
        action_probs = policy(obs_tensor)
        action = torch.argmax(action_probs).item()
        obs, reward, terminated, truncated, _ = env.step(action)
        done = terminated or truncated

env.close()
```

Chapter 3

Optimism and Bellman Equations

In the previous chapter, we ran the policy gradient algorithm on two environments: CartPole and Pong. We observed that finding the optimal policy was much easier in CartPole than in Pong. This naturally raises the question: what makes some RL problems easier than others?

By "easy," we mean problems for which we can design algorithms that are efficient in both sample complexity (the amount of interaction required) and computational complexity (the amount of computation, e.g., GPU hours, needed).

This chapter will focus on: when can we design sample efficient RL algorithms? This chapter introduces two core algorithmic ideas in reinforcement learning theory: *optimism* and *bellman equations*. These principles underlie many sample-efficient algorithms across various settings, including Linear Bellman-Complete MDPs [5], Linear MDPs [6], Deterministic linear Q* [7], and more. The first attempt to generalize these ideas was made in [8]. A broader overview can be found in [9].

In this chapter, we focus on the simplest such setting—the *linear setting*—where the optimal value functions V^* and Q^* lie in a known linear function class.

Definition 3.1 (Linear V^* and Q^*)**.** *We assume that the optimal value functions admit a known linear feature representation:*

- ***Linear*** Q^****:*** *There exists an unknown parameter vector $w^* \in \mathbb{R}^d$ and a known feature map $\phi : S \times A \to \mathbb{R}^d$ such that*

$$Q^*(s,a) = \langle w^*, \phi(s,a) \rangle \quad \text{for all } (s,a) \in S \times A.$$

- ***Linear*** V^****:*** *There exists an unknown parameter vector $\theta^* \in \mathbb{R}^d$ and a known feature map $\phi : S \to \mathbb{R}^d$ such that*

$$V^*(s) = \langle \theta^*, \phi(s) \rangle \quad \text{for all } s \in S.$$

We use the same notation ϕ for both feature maps, with the meaning clear from context.

We study the problem of learning a near-optimal policy in the finite-horizon linear setting, where the agent interacts with the environment and seeks a *sample-efficient* algorithm: one that learns a good policy using as few episodes as possible.

Definition 3.2 (Goal: Sample-Efficient Algorithm for Linear Setting)**.** *Let $\varepsilon, \delta \in (0,1)$ be accuracy and confidence parameters. The goal is to design an algorithm which, with probability at least $1-\delta$, outputs a policy $\hat{\pi}$ such that*

$$V^{\hat{\pi}}(s_0) \geq V^*(s_0) - \varepsilon,$$

where V^* denotes the value of an optimal policy and s_0 is the starting state. The algorithm is sample efficient *if the number of environment interactions required is polynomial in the feature dimension d, number of actions $|A|$, horizon H, and parameters $\varepsilon^{-1}, \log \delta^{-1}$.*

3.1 Bellman Equation

A key observation behind our algorithm is the Bellman equation, which holds for all state-action pairs (s_h, a_h):

$$Q_h^*(s_h, a_h) = \mathop{\mathbb{E}}_{\substack{s_{h+1} \sim T(s_h, a_h) \\ r_h \sim R(s_h, a_h)}} \left[r_h + V_{h+1}^*(s_{h+1}) \right]. \tag{3.1}$$

This immediately implies:

$$\mathop{\mathbb{E}}_{\substack{s_{h+1} \sim T(s_h, a_h) \\ r_h \sim R(s_h, a_h)}} \left[Q_h^*(s_h, a_h) - r_h - V_{h+1}^*(s_{h+1}) \right] = 0. \tag{3.2}$$

We are interested in analyzing this identity under the distribution induced by a policy, which we define next.

Definition 3.3 (Trajectory Distribution d^π)**.** *Let π be a policy. The distribution d_h^π denotes the marginal over (s_h, a_h, r_h, s_{h+1}) at time h under policy π, with the following sampling process:*

- *s_0 is the known starting state,*
- *$a_t \sim \pi(s_t)$, $s_{t+1} \sim T(s_t, a_t)$, and $r_t \sim R(s_t, a_t)$.*

We write $d^\pi = \{d_0^\pi, \ldots, d_{H-1}^\pi\}$ to denote the full set of marginals over the trajectory.

Taking expectations in Equation (3.2) over any distribution on (s_h, a_h), including the trajectory distribution d^π induced by a policy π, preserves the identity:

$$\mathop{\mathbb{E}}_{(s_h, a_h, r_h, s_{h+1}) \sim d_h^\pi} \left[Q^*(s_h, a_h) - r_h - V_{h+1}^*(s_{h+1}) \right] = 0. \tag{3.3}$$

For brevity, henceforth, we will use $\mathbb{E}_{d_h^\pi}$ to denote the above expectation. Finally, note that given a policy π, it is straightforward to sample from d^π by interacting with the environment: simply run π for multiple episodes and collect the tuples (s_h, a_h, r_h, s_{h+1}) at each time step h.

3.2 Algorithm

We now present a sample-efficient algorithm that iteratively constructs policies that are (a) *optimistic*, and (b) *approximately satisfy Bellman equation on past data*. At each iteration t, the algorithm uses data collected under previous policies $\pi_1, \ldots, \pi_t$ to construct a new policy π_{t+1} that maximizes estimated value while ensuring that its Bellman residual (Equation (3.3)) is small under the empirical trajectory distributions from earlier policies.

Let $\hat{d}_h^{\pi_k}$ denote the empirical distribution over $n_{\text{emp}} = \text{poly}(d, H)$ transitions (s_h, a_h, r_h, s_{h+1}) collected from executing policy π_k. The number of samples collected per policy n_{emp} and the constraint parameter $\varepsilon_{\text{cons}}$ will be chosen later in the analysis.

Algorithm 2: Optimistic Algorithm

for $t = 1, 2, \ldots, n_{rounds}$ **do**

Solve the following optimization problem:

$$\max_{\theta, w} \langle \theta, \phi(s_0) \rangle$$

$$\text{s.t.} \quad \sum_{k=1}^{t-1} \mathop{\mathbb{E}}_{\hat{d}_h^{\pi_k}} \Big[\langle w, \phi(s_h, a_h) \rangle - r_h - \langle \theta, \phi(s_{h+1}) \rangle \Big]^2 \leq \frac{\varepsilon^2}{\text{poly}(d, H)}$$

Here, $\|\theta\|_2$, $\|w\|_2 \leq 1$ are norm constrained and satisfy $\langle \theta, \phi(s) \rangle = \max_a \langle w, \phi(s, a) \rangle$ for all states s.

Let θ_t, w_t be the resulting parameters. Define π_t as the resulting policy defined by $\pi_t(s) = \text{argmax}_a \langle w_t, \phi(s, a) \rangle$.

Execute π_t for n_{emp} episodes, and use the collected data to construct empirical distributions $\hat{d}^{\pi_t} = \{\hat{d}_h^{\pi_t}\}_{h=0}^{H-1}$.;

return the best policy π_t observed so far.;

3.3 Optimization Constraint in Linear Form

We begin by rewriting the constraint in our optimization program using the linear representations of the value functions. Recall that we denote the time-indexed value functions as Q_h^π and V_h^π, where $h \in \{0, \ldots, H-1\}$. However, to keep notation light, we will suppress the dependence on h in the remainder of

this proof, with the understanding that the analysis applies separately at each time step.

At round t, let θ_t, w_t be the resulting parameters, and define π_t as the resulting policy defined by $\pi_t(s) = \text{argmax}_a \langle w_t, \phi(s,a)\rangle$. Define the concatenated parameter and feature vectors:

$$W(\pi_t) := [w_t, \theta_t] \in \mathbb{R}^{2d},$$
$$X(\pi_t) := \mathop{\mathbb{E}}_{d_h^{\pi_t}} [\phi(s_h, a_h), -\phi(s_{h+1})] \in \mathbb{R}^{2d}.$$

Here, recall the notation $\mathbb{E}_{d_h^{\pi_t}}$ is shorthand for the expectation over samples $(s_h, a_h, s_{h+1}) \sim d_h^{\pi_t}$. The expected Bellman optimality equation for a candidate policy π_t at time step h is:

$$\begin{aligned} &\mathop{\mathbb{E}}_{d^{\pi_t}} [\langle w_t, \phi(s_h, a_h)\rangle - r_h - \langle \theta_t, \phi(s_{h+1})\rangle] && (3.4)\\ =&\mathop{\mathbb{E}}_{d^{\pi_t}} [\langle w_t, \phi(s_h, a_h)\rangle - \langle \theta_t, \phi(s_{h+1})\rangle - Q^*(s_h, a_h) + V^*(s_{h+1})] \\ =&\langle W(\pi_t) - W(\pi^*), X(\pi_t)\rangle, && (3.5)\end{aligned}$$

where the first equality uses the Bellman identity for the optimal policy (Equation (3.3)) to replace r_h with $Q^*(s_h, a_h) - V^*(s_{h+1})$ in expectation.

3.4 Optimism: Bounding Regret

To analyze the regret, we relate the suboptimality of the current policy π_t to the difference between its parameters and those of the optimal value functions. The key idea is optimism: the algorithm selects θ_t, w_t to maximize $\langle \theta_t, \phi(s_0)\rangle$, which serves as an optimistic estimate of $V^{\pi_t}(s_0)$. Although this estimate may be inaccurate, the algorithm proceeds by acting according to π_t.

Lemma 3.4 (Regret Decomposition via Optimism)**.** *Let the policy π_t, and vectors $W(\pi_t), X(\pi_t) \in \mathbb{R}^{2d}$ be as defined above. Then:*

$$V^*(s_0) - V^{\pi_t}(s_0) \le \sum_{h=0}^{H-1} \left|\langle W(\pi_t) - W(\pi^*), X_h(\pi_t)\rangle\right|.$$

Proof. Throughout, we use that $a_h = \text{argmax}_a \langle w_t, \phi(s_h, a)\rangle$. The claim follows

from:

$$
\begin{aligned}
&V^*(s_0) - V^{\pi_t}(s_0) \\
&\leq \langle \theta_t, \phi(s_0) \rangle - V^{\pi_t}(s_0) && \text{(follows from optimism)} \\
&= \langle w_t, \phi(s_0, a_0) \rangle - \mathop{\mathbb{E}}_{d^{\pi_t}} \left[\sum_{h=0}^{H-1} r_h \right] && \text{(since } \langle \theta_t, \phi(s_0) \rangle = \langle w_t, \phi(s_0, a_0) \rangle\text{)} \\
&= \sum_{h=0}^{H-1} \mathop{\mathbb{E}}_{d_h^{\pi_t}} \left[\langle w_t, \phi(s_h, a_h) \rangle - r_h - \langle w_t, \phi(s_{h+1}, a_{h+1}) \rangle \right] && \text{(by telescoping sum)} \\
&= \sum_{h=0}^{H-1} \mathop{\mathbb{E}}_{d_h^{\pi_t}} \left[\langle w_t, \phi(s_h, a_h) \rangle - r_h - \langle \theta_t, \phi(s_{h+1}) \rangle \right] \\
& && \text{(since } \langle \theta_t, \phi(s_{h+1}) \rangle = \langle w_t, \phi(s_h, a_h) \rangle\text{)} \\
&\leq \sum_{h=0}^{H-1} \left| \langle W(\pi_t) - W(\pi^*), X_h(\pi_t) \rangle \right|. && \text{(follows from Equation (3.5))}
\end{aligned}
$$

□

3.5 Exploration: Bounding the Number of Rounds

The final step is to bound how many rounds n_{rounds} are needed before the algorithm identifies a policy π_t with small suboptimality. We provide a simplified argument, which can be improved to obtain tighter bounds.

The key idea is that for large enough n_{emp}, the empirical distribution $\hat{d}_h^{\pi_k}$ concentrates around the true distribution $d_h^{\pi_k}$. Then, from the regret decomposition lemma above (Lemma 3.4), we know that if the residual term $\langle W(\pi_k) - W(\pi^*), X_h(\pi_k) \rangle$ is zero for all h, then the policy π_k is optimal. Otherwise, a nonzero residual indicates that π_k adds a new constraint that helps eliminate a portion of the hypothesis space.

More precisely, these constraints are linear in the difference vector $W(\pi) - W(\pi^*)$, and each non-redundant policy introduces a constraint that is at least ε-independent (Definition 3.5) of the previous ones. Since the vector space has finite dimension $2d$, only a bounded number of such ε-independent constraints can exist. This implies that the algorithm can only generate a finite number of meaningfully distinct (and suboptimal) policies before it must identify the optimal one.

Thus, it suffices to analyze the following situation: how many times can it happen that

$$
\begin{aligned}
\sum_{k=1}^{t-1} \left(\langle W(\pi_t) - W(\pi^*), X_h(\pi_k) \rangle \right)^2 &\leq \varepsilon^2 \\
\left(\langle W(\pi_t) - W(\pi^*), X_h(\pi_t) \rangle \right)^2 &> \varepsilon^2
\end{aligned}
$$

This question—how many times an ε-independent constraint can arise—was first analyzed by Russo and Van Roy [10].

Definition 3.5 (ε-independence)**.** *Let $S_d = \{\theta \in R^d : \|\theta\|_2 \leq 1\}$. We say $x_n \in S_d$ is ε-independent of sequence $\{x_1, x_2, \ldots, x_{n-1}\}$ if there exists $\theta, \theta^* \in S_d$ such that*

$$\sum_{i=1}^{n-1} \left(\langle \theta, x_i \rangle - \langle \theta^*, x_i \rangle\right)^2 \leq \varepsilon^2$$
$$\left(\langle \theta, x_n \rangle - \langle \theta^*, x_n \rangle\right)^2 > \varepsilon^2$$

We now bound the number of such ε-independent vectors that can appear in a sequence.

Lemma 3.6 (Bound on Number of ε-Independent Constraints)**.** *Let $\{x_1, \ldots, x_n\} \subseteq S_d$ be a sequence such that each x_k is ε-independent of the previous vectors in the sense of Definition 3.5. Then, the number n of such vectors is at most*

$$n = O\left(d \cdot \log\left(\frac{1}{\varepsilon}\right)\right).$$

Proof. More generally, assume that $\|x_k\|_2 \leq B_\phi$ for all k, and that $\|\theta - \theta^*\|_2 \leq 2B_\theta$. In our case, both B_ϕ and B_θ are 1.

Let $V_n := \sum_{i=1}^{n-1} x_i x_i^\top + \lambda I$ for some regularization $\lambda = \varepsilon^2/(2B_\theta)^2$. If x_n is ε-independent of the previous $x_1, \ldots, x_{n-1}$, then there exists θ, θ^* with $\|\theta - \theta^*\|_2 \leq 2B_\theta$ such that:

$$\sum_{i=1}^{n-1} \left(\langle \theta - \theta^*, x_i \rangle\right)^2 \leq \varepsilon^2, \quad \text{but} \quad \left(\langle \theta - \theta^*, x_n \rangle\right)^2 > \varepsilon^2.$$

This implies:

$$\begin{aligned}
\varepsilon &\leq \left\{ (\theta - \theta^*)^\top x_n : (\theta - \theta^*)^\top \left(\sum_{i=1}^{n-1} x_i x_i^\top\right)(\theta - \theta^*) \leq \varepsilon^2 \quad \text{and} \quad (\theta - \theta^*)^\top I\,(\theta - \theta^*) \leq (2B_\theta)^2 \right\} \\
&\leq \max_\rho \left\{ \rho^\top x_n : \rho^\top \left(\sum_{i=1}^{n-1} x_i x_i^\top\right)\rho \leq \varepsilon^2 \quad \text{and} \quad \rho^\top I \rho \leq (2B_\theta)^2 \right\} \\
&\leq \max_\rho \left\{ \rho^\top x_n : \rho^\top \left(\sum_{i=1}^{n-1} x_i x_i^\top + \lambda I\right)\rho \leq 2\varepsilon^2 \right\} \\
&= \sqrt{2\varepsilon^2}\|x_n\|_{V_n^{-1}}
\end{aligned}$$

And therefore,

$$\|x_n\|^2_{V_n^{-1}} = x_n^\top V_n^{-1} x_n \geq \frac{1}{2}.$$

We now analyze the growth of $\det(V_n)$ using the matrix determinant lemma:

$$\begin{aligned}\det(V_n) &= \det(V_{n-1})\left(1 + x_{n-1}^\top V_{n-1}^{-1} x_{n-1}\right) \\ &\geq \det(V_{n-1}) \cdot \frac{3}{2}.\end{aligned}$$

Iterating gives:

$$\det(V_n) \geq \det(\lambda I) \cdot \left(\frac{3}{2}\right)^{n-1} = \lambda^d \left(\frac{3}{2}\right)^{n-1}.$$

On the other hand, since $\operatorname{trace}(V_n) \leq nB_\phi^2 + d\lambda$, we apply the AM–GM inequality:

$$\det(V_n) \leq \left(\frac{\operatorname{trace}(V_n)}{d}\right)^d \leq \left(\frac{nB_\phi^2}{d} + \lambda\right)^d.$$

Equating the lower and upper bounds on $\det(V_n)$ gives:

$$\lambda^d \left(\frac{3}{2}\right)^{n-1} \leq \left(\frac{nB_\phi^2}{d} + \lambda\right)^d.$$

Solving this inequality yields the desired bound:

$$n = O\left(d \cdot \log\left(\frac{B_\theta^2 B_\phi^2}{\varepsilon}\right)\right).$$

□

Chapter 4

Computational Complexity

In the previous chapter, we showed how to find an approximately optimal policy in a sample-efficient manner—that is, by interacting with the transition and reward functions at most polynomially many times in the feature dimension d and horizon H. In this chapter, we show that despite the statistical problem being easy, the computational problem is hard: no efficient algorithm exists for solving the same problem under standard complexity assumptions.

4.1 Complexity Problems

Our proof is based on a reduction from the classical 3-SAT problem:

Definition 4.1 (3-SAT)**.** *Given a Boolean formula φ in conjunctive normal form (CNF) with v variables and $O(v)$ clauses, the goal is to determine whether φ is satisfiable—that is, whether there exists an assignment $w \in \{0,1\}^v$ such that every clause in φ evaluates to* `True`*.*

For example, the formula $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee x_4)$ is satisfiable. We now formally define the interaction model for the linear reinforcement learning (RL) problem:

Definition 4.2 (Linear RL and Interaction Model)**.** *An algorithm is given access to a deterministic finite-horizon MDP M with horizon H, and the following oracles:*

- ***Reward oracle:*** *Given a state s and action a, returns a sample from $R(s,a)$.*
- ***Transition oracle:*** *Given a state s and action a, returns the next state $T(s,a)$.*
- ***Feature oracle:*** *Given a state s (or a state-action pair (s,a)), returns the corresponding d-dimensional feature vector $\phi(s)$ or $\phi(s,a)$.*

Each oracle call has constant runtime, and all input/output sizes are polynomial in the feature dimension d.

We assume that the optimal value functions Q^* and V^* are linear in these features (Definition 3.1). Our goal is to prove the following computational lower bound for finding an approximate optimal policy:

Theorem 4.3. *Unless $\mathsf{NP} = \mathsf{RP}$, there is no polynomial-time algorithm which, given access to a deterministic MDP M with at least two actions and horizon $H = O(d)$, where the optimal value functions Q^* and V^* are linear in d-dimensional features ϕ (Definition 3.1), outputs a policy π satisfying $V^\pi > V^* - 1/4$ with constant probability.*

We construct such MDPs from 3-SAT formulas in a way that an efficient algorithm for solving the MDP would yield an efficient algorithm for solving 3-SAT.

4.2 Linear Infinite-Horizon MDP

To build intuition, we begin with an infinite-horizon deterministic MDP derived from a CNF formula. Consider the example:

$$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$$

We define an infinite-horizon MDP (whose unrolling is illustrated in Figure 4.1) as follows:

- Each state corresponds to a partial assignment $w \in \{0,1\}^v$ of the SAT variables. The root corresponds to the all-zero assignment.
- The process terminates when the agent reaches a pre-decided satisfying assignment w^*.
- From any state w, an unsatisfied clause is selected, and the agent chooses among three actions: flipping one of the three variables in that clause.
- Each action incurs a reward of -1.

Because each action has a reward of -1, the optimal policy minimizes the path length to the satisfying assignment w^*. This leads to the following:

Lemma 4.4. *The optimal value functions Q^* and V^* are linear in w and w^*. Specifically, for a state s with assignment w:*

$$V^*(s) = -D(w, w^*),$$

where $D(w, w^*)$ is the Hamming distance between w and w^*. Since $D(w, w^*) = \frac{1}{2}(v - \langle w, w^* \rangle)$, it is linear in both w and w^*. Moreover, $Q^*(s,a) = V^*(T(s,a)) - 1$.

While conceptually clean, this construction is infinite-horizon. Truncating the tree to make it finite-horizon would require inserting leaf rewards that depend on w^*, which cannot be simulated efficiently.

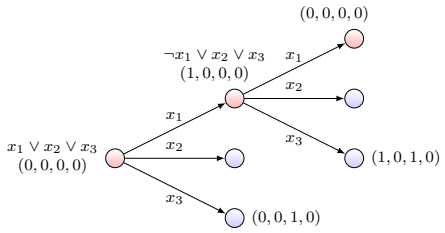

Figure 4.1: Unrolling of an infinite-horizon MDP based on SAT clauses. Each node corresponds to an assignment; actions flip variables in an unsatisfied clause.

4.3 Linear Finite-Horizon MDP

To address this, we define a finite-horizon MDP with carefully constructed leaf rewards. Rewards are zero everywhere except at the leaves. At a leaf with assignment w and depth l, the expected reward is:

$$\mathbb{E}[R(w,l)] = \left(1 - \frac{l + D(w,w^*)}{H+v}\right)^r.$$

This form ensures that the optimal value function has the same structure:

Lemma 4.5. *The optimal value function at a state s with assignment w and depth l is*

$$V^*(s) = \left(1 - \frac{l + D(w,w^*)}{H+v}\right)^r.$$

This function is a degree-r polynomial in the inner product $\langle w, w^* \rangle$, and thus linear in $d = v^r$-dimensional features.

Sketch. See [11] for details. The greedy policy that flips variables to reduce $D(w,w^*)$ by 1 per step (and thus increases l by 1) maintains $D(w,w^*) + l$ as a constant. Since the reward is decreasing in this quantity, the greedy policy is optimal, and the value function inherits the same functional form as the reward. □

Suppose we truncate the tree at depth $H = d = v^r$. Then the maximum reward at the final level is:

$$\left(1 - \frac{H}{H+v}\right)^r = \left(\frac{v}{H+v}\right)^r = v^{-O(r^2)}.$$

Since any algorithm restricted to polynomial time in d and H can reach only $\text{poly}(v^r)$ states, it will observe negligible reward at the leaves and gain no useful information.

In conclusion, while statistically efficient algorithms exist under the assumption that V^* and Q^* are linear in known features, computationally efficient algorithms do not—unless $\mathsf{NP} = \mathsf{RP}$.

Bibliography

[1] Farama Foundation. Cartpole – gymnasium classic control environment, 2023. `https://gymnasium.farama.org/environments/classic_control/cart_pole/`.

[2] Farama Foundation. Pong – ale atari environment, 2023. `https://ale.farama.org/environments/pong/`.

[3] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897. PMLR, 2015.

[4] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. In *arXiv preprint arXiv:1707.06347*, 2017.

[5] Rémi Munos. Error bounds for approximate value iteration. In *Proceedings of the National Conference on Artificial Intelligence*, volume 20, page 1006. AAAI Press, 2005.

[6] Chi Jin, Zhuoran Yang, Zhaoran Wang, and Michael I Jordan. Provably efficient reinforcement learning with linear function approximation. In *Conference on Learning Theory*, pages 2137–2143. PMLR, 2020.

[7] Zheng Wen and Benjamin Van Roy. Efficient exploration and value function generalization in deterministic systems. *Advances in Neural Information Processing Systems*, 26, 2013.

[8] Nan Jiang, Akshay Krishnamurthy, Alekh Agarwal, John Langford, and Robert E Schapire. Contextual decision processes with low bellman rank are pac-learnable. In *International Conference on Machine Learning*, pages 1704–1713. PMLR, 2017.

[9] Simon Du, Sham Kakade, Jason Lee, Shachar Lovett, Gaurav Mahajan, Wen Sun, and Ruosong Wang. Bilinear classes: A structural framework for provable generalization in rl. In *International Conference on Machine Learning*, pages 2826–2836. PMLR, 2021.

[10] Daniel Russo and Benjamin Van Roy. Eluder dimension and the sample complexity of optimistic exploration. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 2256–2264, Red Hook, NY, USA, 2013. Curran Associates Inc.

[11] Daniel Kane, Sihan Liu, Shachar Lovett, and Gaurav Mahajan. Computational-statistical gap in reinforcement learning. In *Conference on Learning Theory*, pages 1282–1302. PMLR, 2022.