Roadmap

Today’s class will cover some basic approaches in reinforcement learning (RL) [SB18]. The agenda includes:

- Introduction: What is RL?
- Mathematical Foundations: Markov Decision Processes
- Value Iteration and the Q-Learning Algorithm
- Policy Iteration and Policy Gradient Methods

1 Introduction: What is RL?

**RL vs. Supervised/Unsupervised Learning.** Putting RL into context, we can compare it to the two learning paradigms we have seen so far. In supervised learning, we are given paired data \((x, y)\) and our goal is to learn a function \(F : x \rightarrow y\). Some common examples of supervised learning tasks are classification and regression problems. In unsupervised learning, our data have no labels. Given the data \(x\), we want to learn its underlying structure. So far in the class, we have seen unsupervised approaches, like k-means and nearest neighbor search. The RL framework is summarized in Figure 1.

![Figure 1: RL Framework](image-url)
There is an agent and an environment. The agent interacts with the environment by choosing actions that modify its state, while the environment provides a numeric reward for each state. The goal in RL is to learn how to select actions that maximize the rewards collected over time. Often a complicating factor in RL is that the state and action spaces are very large, so approximations are needed. We will cover approaches to address this issue in subsequent sections.

Examples. We next present a few examples of problems where RL is relevant.

- The cart-pole problem in Figure 2 can be phrased in the RL paradigm. The agent must choose how much force to put onto the cart to drive it forwards while balancing the pole upright at each time step. The environment applies gravity and other forces to the cart and the pole. Rewards are generated in each time step so that the pole remains upright while the cart moves forward.

![Cart-Pole Problem](image)

**Objective**: Balance a pole on top of a movable cart

**State**: angle, angular speed, position, horizontal velocity

**Action**: horizontal force applied on the cart

**Reward**: 1 at each time step if the pole is upright

- Similarly, robot locomotion (Figure 3) can be phrased as a RL problem, where the action space is the placement of the robot’s joints and the rewards are set as function of being upright and the distance to a target location.

![Robot Locomotion](image)

**Objective**: Make the robot move forward

**State**: Angle and position of the joints

**Action**: Torques applied on joints

**Reward**: 1 at each time step upright + forward movement

Figure 2: Cart-Pole Problem

Figure 3: Robot Locomotion
• Finally, another example where RL has successfully been applied is for playing games, like 
Atari video games (Figure 4). The agent is the player of the game and the objective is 
to maximize the total score in the game. Another example of a game where RL achieved 
remarkable results is Go.

![Figure 4: Atari](image)

**Objective:** Complete the game with the highest score

**State:** Raw pixel inputs of the game state
**Action:** Game controls e.g. Left, Right, Up, Down
**Reward:** Score increase/decrease at each time step

Figure 4: Atari

# Mathematical Foundations: Markov Decision Processes

**Markov Decision Processes.** Now we turn to mathematically formalizing the concepts of RL. 
Our problem of interest can be framed as a Markov Decision Process (MDP). An MDP is defined 
by a 5-tuple \((S, \mathcal{A}, p, \mathcal{E}, \gamma)\), where

- \(S\) is the set of possible states,
- \(\mathcal{A}\) is the set of actions an agent can choose to take,
- \(p_0\) is the initial state distribution,
- \(\mathcal{E}\) is the environment distribution that, given a (state, action) pair, specifies the distribution 
of rewards and next state,
- \(\gamma\) is the discount rate (must be \(\leq 1\)) that determines the weighting of rewards over time.

This is the typical formulation of RL, but it also possible to have some variations. For example, 
sometimes the action set can depend upon the current state or there can be terminal states.

Putting it all together, it is important to point out that a MDP happens over a series of time 
steps. When \(t = 0\), the initial state \(s_0\) is sampled from \(p(s_0)\). For each subsequent timestep, the 
agent selects an action \(a_t\). The environment samples the next state \(s_{t+1}\) and the reward \(r_t\) from 
the environment distribution \(\mathcal{E}(\cdot|s_t, a_t)\).

An important distinction is whether the MDP is known or unknown. In the setting that we know 
all the parameters of the MDP, we face a Planning Problem. When the state space and reward 
function are unknown, we call this an RL Problem.
**Policy.** Assuming we have a known MDP, we define a policy $\pi : S \rightarrow \Delta(A)$ which is a function that maps a state to a distribution of actions to take. The goal in RL can then be formalized as trying to find the optimal policy $\pi^*$ that maximizes the expected rewards according to:

$$\pi^* = \arg \max \pi \mathbb{E}\left[ \sum_{t \geq 0} \gamma^t r_t | \pi \right].$$

Following a policy $\pi$ produces sample trajectories in the form: $s_0, \alpha_0, r_0, s_1, \alpha_1, r_1, \ldots$

**Example: Grid World.** We can visualize the notion of a policy by looking at a simple grid problem. In this grid world, the objective is to reach one of the two goal states (denoted by $\star$). A policy in grid world specifies the direction in which to move at each state (location) in the grid. A completely random policy could look something like the left of Figure 5, while an optimal policy would look like the right side of Figure 5.

![Random Policy vs Optimal Policy](image)

**Figure 5: Grid World**

**Value function, Q-value function, Optimal Policy.** In order to find the optimal policy $\pi^*$ we define the value and Q-value functions. The value function for a policy $\pi$ is defined at each state as:

$$V^\pi(s) = \mathbb{E}\left[ \sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi \right],$$

and expresses how good the current state $s$ is in terms of the expected cumulative reward from following policy $\pi$, assuming we start at $s$.

In a similar way, the Q-value function for a policy $\pi$ is defined at each (state, action) pair $(s, a)$ as:

$$Q^\pi(s, a) = \mathbb{E}\left[ \sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right],$$

and expresses how good this (state, action) pair is in terms of the expected cumulative reward from taking action $\alpha$ in state $s$ and then following the policy $\pi$.

Using these functions, we can formulate some additional constraints on the optimal policy $\pi^*$. The optimal Q-value function

$$Q^*(s, a) = \max \pi \mathbb{E}\left[ \gamma^t r_t | s_0 = s, a_0 = a, \pi \right]$$

4
expresses the maximum expected cumulative reward achievable from a given (state, action) pair. The function must satisfy

\[ Q^*(s, a) = E_{(r, s') \sim \mathcal{E}(\cdot|s, a)} \left[ r + \gamma \max_{a'} Q^*(s', a') \right], \]

which is one of the most important mathematical constructs of RL, known as the Bellman equation. The Bellman equation captures the optimal path through the (state, action) space of the MDP. It requires that \( Q^* \) be consistent with itself, leading to a convenient fixed point structure for finding the optimal policy in our MDP. Note that the optimal policy \( \pi^* \) can be extracted from \( Q^* \), as

\[ \pi^*(s) = \arg \max_a Q^*(s, a). \]

3 Value Iteration

The Value Iteration Approach enables us to identify the optimal policy by manipulating either the Value function or the Q-value function. A common method for finding the optimal policy via manipulation of the Value function is Temporal Difference (TD) learning; here, we focus on utilizing the Q-value function. We first present the approach for the Planning Problem where the MDP is known, and then proceed with the RL Problem where the MDP is unknown. Finally, we combine the Value Iteration Approach with function approximators, so that it can scale to significantly larger problems.

3.1 Value Iteration for the Planning Problem

Under the circumstances that the MDP is known, we can find the optimal policy by finding \( Q^* \) through an iterative process. The idea is to use the Bellman equation as an iterative update. We start by initializing \( Q_0(s, a) = 0 \) for all (state, action) pairs. Then, for each \( t \geq 0 \), we do the following update for every (state, action) pair:

\[ Q_t(s, a) = E[r + \gamma \max_{a'} Q_{t-1}(s', a')|s, a] \]

Note that when \( Q_t = Q^* \), no further updates will occur, thus it is a fixed point.

**Theorem 1.** \( Q_t \to Q^* \) as \( t \to \infty \).

**Proof Sketch of Theorem 1.** Define an operator \( \mathcal{H} \) over functions \( q : S \times A \to \mathbb{R} \), taking a candidate Q-function as input. Let

\[ (\mathcal{H}q)(s, a) = E[r + \gamma \max_{a'} q(s', a')|s, a]. \]

At each step of value iteration, we apply our operator \( \mathcal{H} \) to \( Q_{t-1} \) to compute our new estimate \( Q_t \).

**Claim:** \( \mathcal{H} \) is a contraction map under the sup norm, thus we have the following property:

\[ ||\mathcal{H}q_1 - \mathcal{H}q_2||_{\infty} \leq \gamma \cdot ||q_1 - q_2||_{\infty} \]

By Banach’s fixed point theorem, \( \mathcal{H} \) has a unique fixed point (which we already identified as \( Q^* \)). Thus our iteration will bring us arbitrarily close to the fixed point where \( \mathcal{H}Q^* = Q^* \). We have now shown that our iterative process always converges. \(\square\)
3.2 Value Iteration for RL

In the context of RL, the underlying MDP is not assumed to be known. Again, our goal is to learn the optimal Q-value function. We do this by iteratively interacting with the environment. There exist two types of approaches for the problem:

- **Model-Based Approaches**: The intuition here is that the agent tries to understand the environment; it interacts with it and learns the state transition and reward models from the interactions. As a result, the RL problem is reduced to a Planning Problem, and any planning approach (e.g., Value Iteration as presented in Section 3.1) can then be used to find an optimal policy.

- **Model-Free Approaches**: Here the agent does not try to explicitly model the environment; instead it only cares about learning how to play. In particular, it seeks to directly identify an optimal policy through its interaction with the environment.

Next, we present the Q-Learning Algorithm (Algorithm 1) [WD92], which is an example of a model-free approach. As its name suggests, our goal is to learn the optimal Q-value function $Q^*$ via interaction with the environment. The main idea is to still use the Bellman equation, but make updates based on sampled trajectories.

**Algorithm 1 Q-Learning**

1. Initialize $Q_0(s, \alpha) \leftarrow 0$ (or any other value)
2. Select arbitrary policy $\pi : \mathcal{S} \rightarrow \Delta(\mathcal{A})$ such that $\pi(s)(\alpha) > 0$, $\forall (s, \alpha) \in \mathcal{S} \times \mathcal{A}$
3. Use policy $\pi$ to generate sample trajectory $s_0, \alpha_0, r_0, s_1, \alpha_1, r_1, ...$; if, for some $i \geq 0$, $s_i$ is terminal, restart
4. $t \leftarrow 0$
5. while convergence criterion not satisfied do
6. $Q_{t+1}(s_t, \alpha_t) \leftarrow Q_t(s_t, \alpha_t) + \eta_t(s_t, \alpha_t) \left[ r_t + \gamma \max_{\alpha'} Q_t(s_{t+1}, \alpha') - Q_t(s_t, \alpha_t) \right]$
7. $Q_{t+1}(s, \alpha) \leftarrow Q_t(s, \alpha)$, $\forall (s, \alpha) \in \mathcal{S} \times \mathcal{A} \setminus \{(s_t, \alpha_t)\}$
8. $t \leftarrow t + 1$
9. end while
10. Return $Q_t$

The update step in line 6 of Algorithm 1 can be thought of as a gradient step, where the function $\eta_t(s_t, \alpha_t) < 1$ is the learning rate. Also, notice that Algorithm 1 imposes no constraints on the way the samples are generated, i.e., the samples are generated using a different policy than the one associated with the Q-value function that we actually update. This is called an *off-policy* method. An alternative approach would be to generate samples *on-policy*, in which case we would also have to balance the exploration-exploitation trade-off; one common such approach is the SARSA Algorithm [RN94]. The number of updates we make at point $(s, \alpha)$ is determined by the number of samples in our dataset that include this particular point. If there exist terminal states, or the game has a finite horizon, then we perform multiple episodes of the game, until we collect enough data.

Theorem 2 guarantees the convergence of Algorithm 1. The first assumption allows the possibility to update infinitely many times so, if our graph is strongly connected, we will eventually explore
the entire space; we can deal with the opposite case (where the graph is not strongly connected) using multiple random restarts.

**Theorem 2.** If, \( \forall (s, \alpha) \in S \times A \), the step sizes satisfy

\[
\sum_t \eta_t(s, \alpha) = +\infty, \quad \sum_t \eta_t^2(s, \alpha) < +\infty,
\]

then, with probability 1, \( Q_t \rightarrow Q^* \) as \( t \rightarrow +\infty \).

### 3.3 Value Iteration via Function Approximation

The main limitation of the Q-Learning Algorithm is that it needs to maintain \( Q(s, \alpha), \forall (s, \alpha) \in S \times A \), and hence it is not scalable when the state-action space is large. This motivates the use of function approximators for \( Q \), that is, we postulate that \( Q^*(s, \alpha) \approx Q(s, \alpha; \theta^*) \), where \( Q(\cdot, \cdot; \theta) \) is some function approximator.

**Linear Approximator.** The simplest approach would be to use a linear approximation for \( Q \), i.e., \( Q^*(s, \alpha) \approx \theta^T \phi(s, \alpha) \), \( \theta \in \mathbb{R}^k \), for some set of \( k \) basis functions \( \phi \). In doing so, updating the parameters \( \theta \) of the architecture during each iteration of the Q-learning Algorithm is computationally easy.

**Deep Q-Learning.** In what follows, we examine the use of a neural network to approximate \( Q \), so \( Q(\cdot, \cdot; \theta) \) can be thought of as a neural network parameterized by \( \theta \). The resulting approach is called deep Q-learning. Recall that our goal is to enforce the Bellman equation. We start by initializing the function approximator \( Q(\cdot, \cdot; \theta) \) at some arbitrary \( \theta_0 \). We then run many episodes of interaction with the environment, indexed by \( i = 1, 2, ... \). Let \( \rho \in \Delta(S \times A) \) be the empirical distribution over all (state, action) pairs. We iteratively update the parameters \( \theta_{i-1} \) to \( \theta_i \) as follows:

1. First, we do a **forward pass**, enforcing the Bellman equation on average w.r.t. \( \rho \) over all (state, action) pairs encountered in episode \( i \):

\[
L_i(\theta_i) = \mathbb{E}_{s, \alpha \sim \rho(\cdot)} \left[ (y_i(s, \alpha) - Q(s, \alpha; \theta_i))^2 \right],
\]

where \( y_i(s, \alpha) = \mathbb{E}_{r, s' \sim \hat{E}(\cdot|s, a)} [r + \gamma \max_{\alpha'} Q(s', \alpha'; \theta_{i-1})] \) and \( \hat{E} \) denotes the empirical environment distribution. That is, \( \forall (s, a) \) pair encountered in our sampling in episode \( i \), we compare the value of \( Q \) that we were expecting to get according to our current parameters to what we actually got in each episode. In order to find the action \( \alpha' \) that maximizes \( Q(s', \alpha'; \theta_{i-1}) \), we can either try all possible actions (in case the action space is discrete), or do a gradient descent step over the action parameter.

2. Second, we do a **backward pass**, where we perform a gradient update of the loss with respect to the Q-value function parameters \( \theta \):

\[
\nabla_\theta L_i(\theta_i) = \mathbb{E}_{s, \alpha \sim \rho(\cdot), r, s' \sim \hat{E}(\cdot|s, a)} \left[ 2 \left( r + \gamma \max_{\alpha'} Q(s', \alpha'; \theta_{i-1}) - Q(s, \alpha; \theta_i) \right) \nabla_\theta Q(s, \alpha; \theta_i) \right].
\]
Example: Atari. We return to the Atari Example (Figure 4). [MKS13] train a neural network parameterized by weight-vector $\theta$, that inputs a state $s$ (stack of last 4 raw-pixel frames after some preprocessing) and outputs a $d$-dimensional vector $(Q(s, \alpha_1), \ldots, Q(s, \alpha_d))$, where $d$ is the number of possible actions (between 4 and 18 depending on the specific Atari game). The network architecture is shown in Figure 6. The network can be trained efficiently by performing a single feedforward pass to compute the Q-values for all actions from the current state.

![Atari Q-network Architecture](image)

**Figure 6: Atari Q-network Architecture**

Training the Network: Experience Replay. Learning from batches of consecutive samples is problematic for the following two reasons. First, in most realistic settings, the samples are correlated, which can lead to inefficient learning. Second, the current Q-network parameters determine the next training samples, which can lead to bad feedback loops (e.g., if maximizing action is to move left, training samples will be dominated by samples from left-hand size). A method to address these problems is experience replay, where we continually update a replay memory table of transitions $(s_t, \alpha_t, r_t, s_{t+1})$ as more episodes are played. The Q-network is trained on random mini-batches of transitions from the replay memory, instead of consecutive samples. Thus, each transition contributes to multiple weight updates, which also implies greater data efficiency.

Putting it all together, the deep Q-learning algorithm with experience replay for Atari games is presented in Figure 7. The first step is to initialize the replay memory $D$ and Q-network. We then play $M$ episodes (full games). Within each episode, we initialize the state as the starting game-screen pixels and, for each timestep $t$ of the game we select an action $\alpha_t$, either, with small probability, randomly (explore) $=$, or greedily from the current policy (exploit). After taking action $\alpha_t$ and encountering the associated reward $r_t$ and next state $s_{t+1}$, we store this transition in the replay memory. Finally, we sample a random mini-batch of transitions from replay memory and perform a gradient descent step (experience replay).
4 Policy Iteration

In many applications, the Q-value function can be excessively complicated. For example, consider a robot grasping an object; the state is very high-dimensional so it can be hard to learn the exact value of every (state, action) pair, even using function approximators. Nevertheless, the policy can be much simpler: just close the robot’s hand. Motivated by this, one may consider directly learning a policy, e.g., finding the best policy from a collection of policies. This approach is known as Policy Iteration.

4.1 Policy Iteration for the Planning Problem

We again start with the Planning Problem, where the underlying MDP is known. As in the Value Iteration Approach, we use the Bellman equation as an iterative update. We initially pick an arbitrary starting policy \( \pi_0 \). Then, for each \( t \geq 0 \), we first evaluate our current policy by solving a linear system, namely

\[
V(s) = \mathbb{E}_{\alpha \sim \pi_t, (r,s') \sim \mathcal{E}(.|s,\alpha)} \left[ r + \gamma V(s') \right], \quad \forall s,
\]

and proceed by updating our policy as

\[
\pi_{t+1}(s) \leftarrow \arg\max_{\alpha} \mathbb{E}_{(r,s') \sim \mathcal{E}(.|s,\alpha)} \left[ r + \gamma V(s') \right], \quad \forall s.
\]

**Theorem 3.** \( \pi_t \to \pi^* \) as \( t \to \infty \).

Theorem 3 can also be proved by fixed point arguments, along the lines of the proof of Theorem 1.
4.2 Policy Iteration for RL

We now focus on the RL Problem and, specifically, on a family of methods called Policy Gradient Methods. In contrast to the Q-Learning Algorithm we presented for value iteration, the Policy Gradient Methods are on-policy methods. The basic idea is to define a class of parameterized policies $\Pi = \{\pi, \theta \in \mathbb{R}^k\}$ and, for each policy, define its value $J(\theta) = \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t | \pi_\theta \right]$. Then, the goal is to find the optimal policy within our candidate set: $\theta^* = \arg\max_{\theta} J(\theta)$. This is done using gradient ascent on the policy parameters.

**REINFORCE Algorithm.** Next we introduce one important policy gradient method, namely, the REINFORCE algorithm [W92]. Let $\tau = (s_0, \alpha_0, r_0, s_1, \ldots)$ be a trajectory whose distribution $p(\cdot; \theta)$ depends on both the environment (sampling rewards and next states given previous states and actions) and the policy (which guides us in sampling actions given states). Let $r(\tau) = \sum_{t \geq 0} \gamma^t r_t$ be the reward of trajectory $\tau$. Then, the expected reward is

$$J(\theta) = \mathbb{E}_{\tau \sim p(\tau; \theta)} \left[ r(\tau) \right] = \int_{\tau} r(\tau) p(\tau; \theta) d\tau.$$

We typically cannot simply differentiate this reward function, as computing the gradient of the probability density function of trajectories w.r.t. the policy parameters $\theta$ is challenging:

$$\nabla_\theta J(\theta) = \int_{\tau} r(\tau) \nabla_\theta p(\tau; \theta) d\tau.$$

However, we can use the following trick

$$\nabla_\theta p(\tau; \theta) = p(\tau; \theta) \frac{\nabla_\theta p(\tau; \theta)}{p(\tau; \theta)} = p(\tau; \theta) \nabla_\theta \log p(\tau; \theta),$$

and inject the resulting expression back, to get

$$\nabla_\theta J(\theta) = \int_{\tau} r(\tau) p(\tau; \theta) \nabla_\theta \log p(\tau; \theta) d\tau = \mathbb{E}_{\tau \sim p(\tau; \theta)} \left[ r(\tau) \nabla_\theta \log p(\tau; \theta) \right].$$

It turns out that we can estimate this expectation without knowing the environment; we have

$$p(\tau; \theta) = p(s_0) \prod_{t \geq 0} p(s_{t+1} | s_t, \alpha_t) \pi_\theta(\alpha_t | s_t),$$

and thus

$$\log p(\tau; \theta) = \log p(s_0) \sum_{t \geq 0} \log p(s_{t+1} | s_t, \alpha_t) + \log \pi_\theta(\alpha_t | s_t),$$

and, finally, differentiating

$$\nabla_\theta \log p(\tau; \theta) = \sum_{t \geq 0} \nabla_\theta \log \pi_\theta(\alpha_t | s_t),$$

which does not depend on the transition probabilities. Therefore, an unbiased estimator of the $\nabla_\theta J(\theta)$ can be computed by sampling trajectory $\tau$ from the measure defined by the environment.
and policy $\pi_\theta$, then outputting the empirical mean of $\sum_{t \geq 0} r(\tau) \nabla_\theta \log \pi_\theta(\alpha_t | s_t)$. The intuition behind the gradient estimator

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau \left[ \sum_{t \geq 0} r(\tau) \nabla_\theta \log \pi_\theta(\alpha_t | s_t) \right]$$

is that, if $r(\tau) > 0$, we push up the conditional probabilities of the chosen actions, whereas if $r(\tau) < 0$, we push them down. The main limitation of this estimator is its high variance because attributing credit to individual actions taken is hard.

**Variance Reduction.** The estimator we derived can be written in more detail as

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau \left[ \left( \sum_{t \geq 0} \gamma^t r_t \right) \cdot \left( \sum_{t \geq 0} \nabla_\theta \log \pi_\theta(\alpha_t | s_t) \right) \right].$$

By similar manipulations we can get a slightly different expression

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau \left[ \sum_{t \geq 0} \nabla_\theta \log \pi_\theta(\alpha_t | s_t) \cdot \left( \sum_{\ell \geq t} \gamma^\ell r_\ell \right) \right],$$

and reduce the variance of the estimator, using this expression together with an arbitrary function $b : \mathcal{S} \rightarrow \mathbb{R}$, as follows:

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau \left[ \sum_{t \geq 0} \nabla_\theta \log \pi_\theta(\alpha_t | s_t) \cdot \gamma^t \cdot \left( \sum_{\ell \geq t} \gamma^{\ell-t} r_\ell - b(s_t) \right) \right].$$

Intuitively, $\sum_{\ell \geq t} \gamma^{\ell-t} r_\ell$ represents the reward function starting from state $s_t$. The function $b(s)$ can be thought of as a baseline reward, that is, the anticipated reward of state $s$. Subtracting $b(s_t)$ expresses the perspective that the log-probability of taking action $\alpha_t$ should be increased proportionally to how much returns are better than anticipated.

The next challenge is how to choose the baseline rewards. A simple choice would be to maintain a constant moving average of rewards experienced so far from all trajectories. A better choice would be to push up the probability of an action from a state, if this action was better than the anticipated value of what we should get from that state. Hence, we would like to maintain a value function $V(s)$ that measures our anticipated value for state $s$.

**Actor-Critic Algorithm.** The improved choice for a baseline function gives rise to Actor-Critic methods (Figure 8, left), where an actor ($\theta$) maintains a policy and takes actions, and a critic ($\phi$) keeps statistics about values of states or (state, action) pairs. Both may use function approximations $\pi_\theta, V_\phi$ (or $Q_\phi$) (Figure 8, middle). Algorithm 2 illustrates the form of Actor-Critic methods.
**Algorithm 2** Actor-Critic

1: Initialize actor parameters $\theta$, critic parameters $\phi$
2: for iteration = 1, 2, ... do
3: Sample $m$ trajectories under the current policy $\pi_\theta$
4: $\Delta \theta \leftarrow 0$
5: for episode $i = 1, ..., m$ do
6: for $t = 1, ..., T$ do
7: $A_t \leftarrow \sum_{t' \geq t} \gamma^{t' - t} r_t - V_\phi(s_t')$
8: $\Delta \theta \leftarrow \Delta \theta + A_t \nabla_\theta \log \pi_\theta (a_t | s_t)$
9: end for
10: $\Delta \phi \leftarrow \sum_i \sum_t \nabla_\phi ||A_i||^2$
11: $\theta \leftarrow \alpha \Delta \theta$
12: $\phi \leftarrow \beta \Delta \phi$
13: end for
14: end for

Return $\pi_\theta$

A well-cited example of such an approach is the Least-Squares Policy Iteration (LSPI) Algorithm [LP03] (Figure 8, right), a model-free, off-policy method, that combines linear approximation of the Q-value function with approximate policy iteration. The main advantages of LSPI are its data efficiency, as it can reuse in each iteration samples collected in any manner, as well as its simplicity (at each iteration, it solves a linear system), which allows it to scale to large problems.

---

**5 Conclusion**

In this lecture, we introduced RL, set its mathematical foundations, and examined a representative family of methods for each of two common approaches in RL, namely:

- **Q-learning for Value Iteration**, which does not always work, but when it works, it is usually more sample-efficient. The challenge is exploration of the state/action space. When function approximators are used, we lose our theoretical guarantees since we are approximating the Bellman equation with a complicated function approximator.

- **Policy gradient methods for Policy Iteration**, which is very general but suffers from high variance, thus a large number of samples is required. However, we can guarantee convergence to a local minimum of $J(\cdot)$, which is often good enough!
References


