Reinforcement Learning

Yunlong Song (宋运龙)

If you want to master something, teach it. — Richard Feynman

Introduction

Reinforcement Learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. Actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics—*trial-and-error* search and *delayed reward*—are the two most important distinguishing features of reinforcement learning.

Elements of Reinforcement Learning

- **Policy**: A policy is a mapping from perceived states of the environment to actions to be taken when in those states.
- **Reward Signal**: A reward signal defines the goal in a reinforcement learning problem. On each time step, the environment sends to the reinforcement learning agent a single number called the reward.
- **Value Function**: The value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state.
- **Model**: A model predicts what the environment will do next.

Markov Decision Process

A Markov Decision Process (MDP) is a 5-tuple (S, A, P, R, γ), where

- S is a finite set of states,
- A is a finite set of actions,
- P is a state transition probability matrix,
- R is a reward function,
- γ is a discount factor.

The value function $V^\pi(s)$ is the expected return starting from state s and then following policy π.

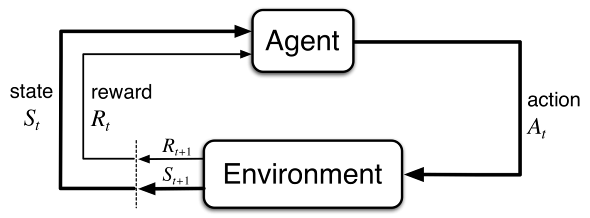

Figure 1: Agent-Environment interaction is MDP.

Returns are the sum of rewards,

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}.$$

(state) Value Function can be written recursively as

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma V^\pi(S_{t+1}) | S_t = s]. \end{aligned}$$

(state-action) Value Function is the expected return starting from state s, taking action a, and then following policy π,

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma Q^\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]. \end{aligned}$$

Bellman Equation for $V^\pi(s)$ is

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r | s, a)[r + \gamma V^\pi(s')].$$

The key idea behind the Bellman equation is that the value of a state is the sum of the immediate reward and the value of the next state.

Optimal Value Function is the maximum value function over all policies,

$$V^*(s) = \max_\pi V^\pi(s).$$

A Unified View of RL Methods

How to solve an MDP?

- *Dynamic Programming*
- *Monte-Carlo Methods*
- *Temporal-Difference Learning*
- *Function Approximation*
- *Policy Gradient Methods*
- *Actor-Critic Methods*

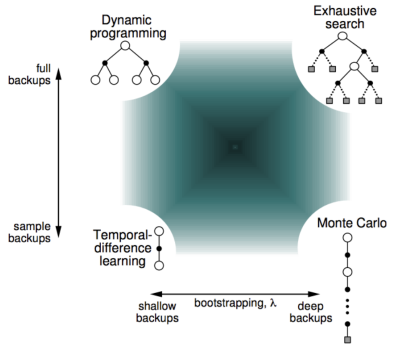

Figure 2: Unified view of RL.

Dynamic Programming

Dynamic programming methods assume that the agent has full knowledge of the MDP. The agent uses this knowledge to plan its actions ahead of time.

Bellman's Principle of Optimality

An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

Bellman Optimality Equation for $V^*(s)$ is

$$V^*(s) = \max_a \sum_{s',r} p(s', r | s, a)[r + \gamma V^*(s')].$$

Bellman Optimality Equation for $Q^*(s,a)$ is

$$Q^*(s,a) = \sum_{s',r} p(s', r | s, a)[r + \gamma \max_{a'} Q^*(s', a')].$$

DP backup:

$$V(S_t) \leftarrow \max_a \sum_{s',r} p(s', r | s, a)[r + \gamma V(s')].$$

Monte-Carlo Methods

Monte-Carlo methods learn directly from episodes of experience. They do not require a model of the environment.

- MC methods learn directly from episodes of experience.
- MC is model-free: no knowledge of MDP transitions / rewards.
- MC learns from complete episodes: no bootstrapping.
- MC uses the simplest possible idea: value = mean return.

Monte-Carlo backup:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)].$$

Temporal-Difference Learning

Temporal-difference (TD) learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap).

- TD methods update estimates based in part on other learned estimates.
- TD methods update estimates based on current estimate.
- TD methods are model-free: no knowledge of MDP transitions / rewards.
- TD methods learn from incomplete episodes: online, continual learning.

TD backup:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)].$$

Sarsa: On-Policy TD Control

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)].$$

Q-learning: Off-Policy TD Control

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)].$$

Function Approximation

In many reinforcement learning problems, the state space is too large to store the value of each state. In such cases, we can use function approximation to approximate the value function.

Linear Function Approximation

$$\hat{v}(s, \mathbf{w}) = \mathbf{x}(s)^T \mathbf{w} = \sum_{j=1}^{d} x_j(s) w_j.$$

Gradient Descent

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[G_t - \hat{v}(S_t, \mathbf{w})]\nabla \hat{v}(S_t, \mathbf{w}).$$

Deep Q-Network (DQN)

- Experience replay: store experience and sample mini-batches.
- Fixed Q-targets: use a separate network to estimate the target.

Optimize MSE between Q-network and target.

$$L(\theta) = \mathbb{E}_{s,a,r,s' \sim U(D)} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \right)^2 \right].$$

where θ^- are the parameters of the target network, which are fixed for a certain number of steps. Here, $U(D)$ is the uniform distribution over the replay buffer D.

Policy Gradient Methods

Policy gradient methods directly learn the policy, rather than learning the value function and then deriving the policy. Policy gradient methods can learn stochastic policies.

Policy Gradient Theorem

The policy gradient is given by

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[\nabla_\theta \log \pi(a|s) \Phi^\pi(s, a) \right].$$

REINFORCE The policy gradienet can be derived using the log-derivative trick, Mathematically, the optimization problem can be formulated as:

$$\max_\theta \quad J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)}[R(\tau)] = \int \pi_\theta(\tau) R(\tau) d\tau \qquad (1)$$

where $R(\tau)$ is the task objective, and $\pi_\theta(\tau)$ is the distribution over trajectories τ. The gradient of the objective function can be computed as:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \int \nabla_\theta \pi_\theta(\tau) R(\tau) d\tau \\
&= \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) R(\tau) d\tau \quad \text{log-trick} \\
&= \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\nabla_\theta \log \pi_\theta(\tau) R(\tau) \right] \\
&= \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right) \right] \\
&\approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t^i|s_t^i) r(s_t^i, a_t^i) \\
&= \mathbb{E}_\pi \left[\nabla_\theta \log \pi_\theta(a|s) \Phi^\pi(s, a) \right].
\end{aligned}$$

Natural Policy Gradient The natural policy gradient is the gradient of the policy with respect to the policy parameters, scaled by the Fisher information matrix. The natural gradient is given by:

$$\nabla_\theta J(\theta) = F^{-1}(\theta) \nabla_\theta J(\theta).$$

where $F(\theta)$ is the Fisher information matrix.

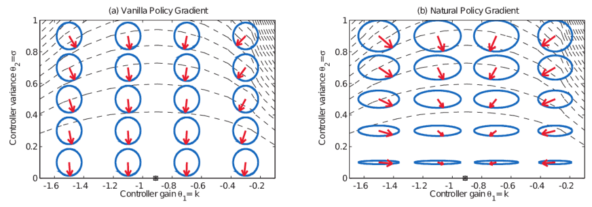

Figure 3: "Vanilla" Policy Gradient vs. Natural Policy Gradient.

Actor-Critic Methods

Actor-critic methods combine the benefits of both policy-based and value-based methods. The actor is the policy, and the critic is the value function. The policy gradient has high variance, and the value function has high bias. Actor-critic methods combine the two to reduce the variance of the policy gradient. As a result, the policy gradient is:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[\nabla_\theta \log \pi(a|s) \Phi^\pi(s, a) \right].$$

where $\Phi^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$ is the advantage function.

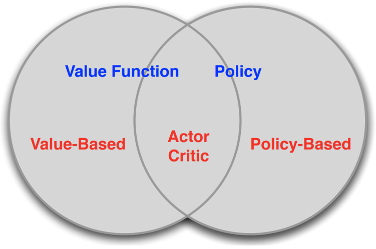

Figure 4: Actor-Critic architecture.

Deterministic Policy Gradient

The deterministic policy gradient is given by:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[\nabla_\theta \mu(s) \nabla_a Q(s, a)|_{s=s_t, a=\mu(s_t)} \right].$$

where $\mu(s)$ is the deterministic policy, and $Q(s, a)$ is the action-value function.

Deep Deterministic Policy Gradient

DDPG is an algorithm which concurrently learns a Q-function and a deterministic policy $\mu(s)$. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy. Suppose the approximator is a deep neural network, the Q-function is updated by minimizing the loss:

$$L(\theta) = \mathbb{E}_{s,a,r,s' \sim U(D)} \left[\left(r + \gamma Q(s', \mu(s'; \theta^\mu); \theta^Q) - Q(s, a; \theta^Q) \right)^2 \right].$$

where θ^μ are the parameters of the policy network, and θ^Q are the parameters of the Q-network.

Relative Entropy Policy Search

Relative entropy policy search (REPS) is a policy search method that uses the KL-divergence to constrain the policy updates. The policy search problem can be formulated as a constrained optimization problem:

$$\max_{\theta} \quad J(\theta) = \mathbb{E}_{\pi_\theta(\tau)}[R(\tau)]$$
$$\text{s.t.} \quad D_{KL}(\pi_\theta(\tau)||\pi_{\theta_{old}}(\tau)) \leq \delta.$$

The optimization problem can be solved using the Lagrange multiplier method:

$$\max_{\theta} \quad J(\theta) - \beta D_{KL}(\pi_\theta(\tau)||\pi_{\theta_{old}}(\tau)).$$

The problem can be solved by first optimizing the Lagrangian with respect to the policy, and then updating the Lagrange multiplier β. TRPO can be viewed as a variant of REPS that uses the natural policy gradient and the Fisher information matrix to constrain the policy updates.
The KL-divergence is given by:

$$D_{KL}(\pi_\theta(\tau)||\pi_{\theta_{old}}(\tau)) = \mathbb{E}_{\pi_\theta(\tau)}\left[\log \frac{\pi_\theta(\tau)}{\pi_{\theta_{old}}(\tau)}\right].$$

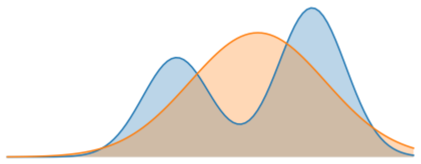

Figure 5: KL-divergence measures the difference between two distributions.

Proximal Policy Optimization

Proximal policy optimization (PPO) is a policy optimization method that uses a clipped surrogate objective to prevent large policy updates. The clipped surrogate objective is given by:

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

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, and $\hat{A}_t$ is the advantage function.

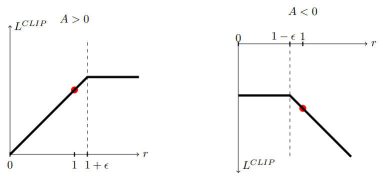

Figure 6: PPO Clip.

Soft Actor-Critic

Soft actor-critic (SAC) is an off-policy actor-critic method that uses the maximum entropy reinforcement learning framework. The objective function is given by:

$$J(\theta) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty}\gamma^t(r(s_t,a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t)))\right].$$

where $\mathcal{H}(\pi(\cdot|s_t))$ is the entropy of the policy.

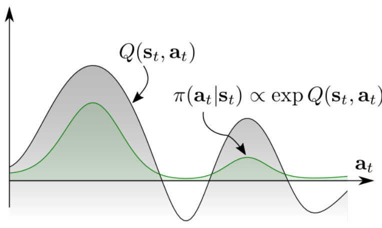

Figure 7: A multimodal Q-function.

Model-based Reinforcement Learning

Model-based reinforcement learning methods learn a model of the environment and use the model to plan ahead.

Model-based RL

- Learn a model of the environment.
- Use the model to plan ahead.
- Model-based methods can be more sample-efficient than model-free methods.

How to Represent the Model

- *Tabular models*
- *Linear models*, e.g. linear regression.
- *Non-linear models*, e.g. neural networks.
- *Probabilistic models* e.g. Gaussian processes.

Dyna-Q Algorithm Dyna integrates learning, planning, model learning, and reactive execution. The planning method is the random-sample one-step tabular Q-planning. The direct RL method is tabular Q-learning. The model learning method is also tabular.

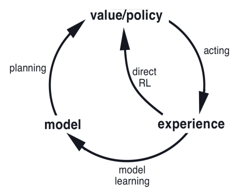

Figure 8: Dyna-Q architecture.

Monte-Carlo Tree Search in AlphaGo

- AlphaGo uses Monte-Carlo tree search (MCTS) to plan ahead.
- MCTS is a model-based reinforcement learning method.
- AlphaGo uses a deep neural network to approximate the value function.

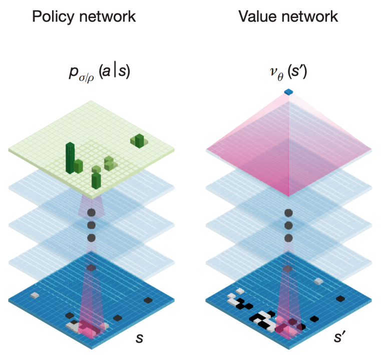

Figure 9: AlphaGo architecture.

PILCO: Probabilistic Inference for Learning Control

PILCO is a model-based reinforcement learning method that uses Gaussian processes to model the dynamics of the environment. PILCO uses a probabilistic model to propagate uncertainty through the planning process. Policy evaluation is performed in closed form using approximate inference. Policy gradients are computed *analytically* for policy improvement.

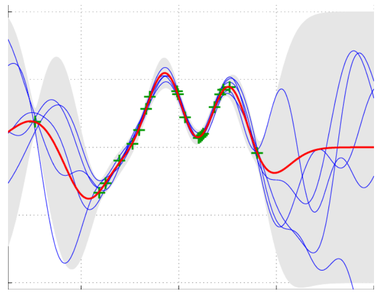

Figure 10: Gaussain process.