Deep Reinforcement Learning Notes (DS)

Dongda Li
dli160@syr.edu

Contents

1 Background

I started learning Reinforcement Learning in 2018, and I first learned it from the book *Deep Reinforcement Learning Hands-On* by Maxim Lapan. That book taught me some high level concepts of Reinforcement Learning and how to implement it using PyTorch step by step. However, when I dug deeper into Reinforcement Learning, I found that the high level intuition was not enough. So I read *Reinforcement Learning: An Introduction* by S. G. (available at `http://incompleteideas.net/book/bookdraft2017nov5.pdf`), and by following the course *Reinforcement Learning* by David Silver (see `https://www.youtube.com/watch?v=2pWv7GOvuf0`), I gained a deeper understanding of RL. For the code implementations from the book and course, refer to the GitHub repository at `https://github.com/dennybritz/reinforcement-learning`.

Here are some of my notes taken while attending the course. For some concepts and ideas that are hard to understand, I add some of my own explanations and intuitions. I omit some simpler concepts in these notes; hopefully, this note will also help you start your RL tour.

2 1. Introduction

RL Features

- Reward signal
- Feedback delay
- Sequence is not i.i.d.
- Actions affect subsequent data

Why Using Discounted Reward?

- Mathematically convenient.
- Avoids **infinite** returns in cyclic Markov processes.
- We are not very confident about our **prediction of reward**; perhaps we are only confident about the near future steps.
- Humans show a preference for immediate reward.
- It is sometimes possible to use an undiscounted reward.

3 2. MDP

In an MDP, the reward is an **action reward**, not a state reward!

$$R_s^a = E\left[R_{t+1} \mid S_t = s, A_t = a\right]$$

The Bellman Optimality Equation is **non-linear**, so we solve it using iterative methods.

4 3. Planning by Dynamic Programming

Planning (when you clearly know the MDP model and try to find an optimal policy)

Prediction: Given an MDP and a policy, you output the value function (policy evaluation).

Control: Given an MDP, you output the optimal value function and optimal policy (solving the MDP).

- Policy Evaluation.
- Policy Iteration:
  - Policy Evaluation (run for k steps until convergence).
  - Policy Improvement:

    * If we iterate policy evaluation and improvement repeatedly, knowing the MDP, we will eventually obtain the optimal policy (as proved). Thus, policy iteration solves the MDP.

- Value Iteration:
  1. Value update (one step of policy evaluation).
  2. Policy improvement (one step greedy based on the updated value).

  Iterating this also solves the MDP.

Asynchronous Dynamic Programming

- In-place dynamic programming (update the old value immediately with the new value, not waiting for all states to update).
- Prioritized sweeping (based on the error in value iteration).
- Real-time dynamic programming (run the game in real-time).

5 4. Model-free Prediction

Model-free prediction is accomplished by sampling.

Monte-Carlo Learning

Every update in Monte-Carlo learning must span a **full episode**.

- **First-Visit Monte-Carlo Policy Evaluation:**
  Run the agent following the policy; the **first** time that state s is visited in an episode, perform the following calculations:

$$N(s) \leftarrow N(s) + 1, \quad S(s) \leftarrow S(s) + G_t, \quad V(s) = \frac{S(s)}{N(s)},$$

  and $V(s) \to v_\pi$ as $N(s) \to \infty$.

- **Every-Visit Monte-Carlo Policy Evaluation:**
  Run the agent following the policy, and each time state s is visited in an episode (even if in a loop), update.
  Incremental Mean:

$$\begin{aligned}\mu_k &= \frac{1}{k}\sum_{j=1}^{k} x_j \\ &= \frac{1}{k}\Big(x_k + \sum_{j=1}^{k-1} x_j\Big) \\ &= \frac{1}{k}\Big(x_k + (k-1)\mu_{k-1}\Big) \\ &= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})\end{aligned}$$

  Thus, by the incremental mean:

$$N(S_t) \leftarrow N(S_t) + 1, \quad V(S_t) \leftarrow V(S_t) + \frac{1}{N_t}(G_t - V(S_t)).$$

  In non-stationary problems, it may be useful to track a running mean, i.e.,

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

Temporal-Difference (TD) Learning

TD learning uses **incomplete** episodes and bootstraps the reward:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$$

and

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

The TD target is

$$G_t = R_{t+1} + \gamma V(S_{t+1}) \quad \text{(TD(0))}.$$

The TD error is

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t).$$

TD(λ) — Balancing between MC and TD

Let the TD target look n steps into the future. If n is very large and the episode is terminal, then it is equivalent to Monte-Carlo.

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}),$$
$$V(S_t) \leftarrow V(S_t) + \alpha\big(G_t^{(n)} - V(S_t)\big).$$

Averaging n-step returns produces **forward TD**(λ):

$$G_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)},$$
$$V(S_t) \leftarrow V(S_t) + \alpha\big(G_t^{\lambda} - V(S_t)\big).$$

Eligibility Traces combine frequency and recency heuristics:

$$E_0(s) = 0,$$
$$E_t(s) = \gamma\lambda E_{t-1}(s) + 1(S_t = s).$$

Backward TD(λ) (using eligibility traces):

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t),$$
$$V(s) \leftarrow V(s) + \alpha\, \delta_t\, E_t(s).$$

If updates are done offline (i.e., in an episode using the old value), then the sum of forward TD(λ) equals the sum of backward TD(λ):

$$\sum_{t=1}^{T} \alpha\, \delta_t\, E_t(s) = \sum_{t=1}^{T} \alpha\big(G_t^{\lambda} - V(S_t)\big) 1(S_t = s).$$

6 5. Model-free Control

An ϵ-greedy policy is used to add exploration to ensure that the policy both improves and explores the environment.

On-policy Monte-Carlo Control

For every episode:

1. **Policy Evaluation:** Perform Monte-Carlo policy evaluation to estimate $Q \approx q_\pi$.
2. **Policy Improvement:** Use an ϵ-greedy policy improvement based on $Q(s,a)$.

Greedy in the limit with infinite exploration (GLIE) will eventually find the optimal solution.

6.0.1 GLIE Monte-Carlo Control

For the kth episode, set $\epsilon \leftarrow 1/k$. As k increases, ϵ_k reduces to zero, and the optimal policy is obtained.

On-policy TD Learning

Sarsa:

$$Q(S,A) \leftarrow Q(S,A) + \alpha\Big(R + \gamma Q(S',A') - Q(S,A)\Big)$$

On-Policy Sarsa:

For every time-step:

- **Policy Evaluation:** Use Sarsa to estimate $Q \approx q_\pi$.
- **Policy Improvement:** Apply ϵ-greedy policy improvement based on $Q(s,a)$.

Forward n-step Sarsa leads to Sarsa(λ), analogous to TD(λ).

Eligibility Traces:

$$E_0(s,a) = 0,$$
$$E_t(s,a) = \gamma\lambda E_{t-1}(s,a) + 1(S_t = s,\ A_t = a).$$

Backward Sarsa(λ) updates, for all (s,a) at each time-step:

$$\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t),$$
$$Q(s,a) \leftarrow Q(s,a) + \alpha\, \delta_t\, E_t(s,a).$$

Intuition: The current state-action pair's reward and value influence all other state-action pairs, with more influence on those that are more recent and frequent. Using only one-step Sarsa would update only one state-action pair per reward, making learning slower.

Off-policy Learning

6.0.2 Importance Sampling

$$\begin{aligned} E_{X\sim P}[f(X)] &= \sum_X P(X) f(X) \\ &= \sum_X Q(X) \frac{P(X)}{Q(X)} f(X) \\ &= E_{X\sim Q}\left[\frac{P(X)}{Q(X)} f(X)\right] \end{aligned}$$

For off-policy TD, the update is:

$$V(s_t) \leftarrow V(s_t) + \alpha\left(\frac{\pi(A_t \mid S_t)}{\mu(A_t \mid S_t)}\Big(R_{t+1} + \gamma V(S_{t+1}) - V(s_t)\Big)\right)$$

6.0.3 Q-learning

In Q-learning, the next action is chosen using the behavior policy $A_{t+1} \sim \mu(\cdot \mid S_t)$, but we update using a target policy $A' \sim \pi(\cdot \mid S_t)$:

$$Q(S,A) \leftarrow Q(S,A) + \alpha\Big(R_{t+1} + \gamma Q(S_{t+1}, A') - Q(S,A)\Big)$$

No matter what action is actually taken next, we update Q according to our target policy. Thus, the Q-values converge to those of the target policy π.

Off-policy Control with Q-learning: The target policy is greedy with respect to $Q(s,a)$:

$$\pi(S_{t+1}) = \arg\max_{a'} Q(S_{t+1}, a')$$

The behavior policy μ can be, for example, ϵ-greedy with respect to $Q(s,a)$ or even a completely random policy; it does not matter because the update is off-policy.

The Q-learning update becomes:

$$Q(S,A) \leftarrow Q(S,A) + \alpha\Big(R_{t+1} + \gamma \max_{a'} Q(S', a') - Q(S,A)\Big)$$

and Q-learning converges to the optimal action-value function $Q(s,a) \rightarrow q_*(s,a)$.

Note: Q-learning can be used both off-policy and on-policy. For on-policy, if you use an ϵ-greedy policy update, Sarsa is a good on-policy method; using Q-learning is also acceptable since ϵ-greedy is similar to the max-Q policy.

7 6. Value Function Approximation

Before this lecture, we discussed **tabular learning** (maintaining a Q-table or value table).

7.1 Introduction

7.1.1 Why?

- The state space is large.
- The state space can be continuous.

7.1.2 Value Function Approximation

We approximate the value function and action-value function as:

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s),$$
$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a).$$

7.1.3 Approximator Considerations

- Non-stationarity: State values change as the policy changes.
- Non-i.i.d.: Samples are generated according to the policy.

7.2 Incremental Methods

7.2.1 Basic SGD for Value Function Approximation

Using stochastic gradient descent (SGD) with feature vectors:

$$\mathbf{x}(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$$

Linear value function approximation:

$$\hat{v}(s, \mathbf{w}) = \mathbf{x}(s)^T \mathbf{w} = \sum_{j=1}^{n} x_j(s)\, w_j,$$
$$J(\mathbf{w}) = E_\pi\Big[\big(v_\pi(s) - \hat{v}(s, \mathbf{w})\big)^2\Big].$$

The gradient update is:

$$\Delta \mathbf{w} = \alpha\big(v_\pi(s) - \hat{v}(s, \mathbf{w})\big)\nabla_{\mathbf{w}}\hat{v}(s, \mathbf{w}) = \alpha\big(v_\pi(s) - \hat{v}(s, \mathbf{w})\big)\mathbf{x}(s).$$

7.2.2 Table Lookup as a Special Case

A table lookup is a special case of linear approximation where the feature vector is:

$$\mathbf{x}(s) = \begin{pmatrix} 1(s = s_1) \\ \vdots \\ 1(s = s_n) \end{pmatrix},$$

and then

$$\hat{v}(s, \mathbf{w}) = \mathbf{x}(s)^T \mathbf{w} = \sum_{i=1}^{n} 1(s = s_i) w_i.$$

7.2.3 Incremental Prediction Algorithms

Supervision:

- For Monte-Carlo (MC), the target is the return G_t:
$$\Delta\mathbf{w} = \alpha\big(G_t - \hat{v}(S_t, \mathbf{w})\big)\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w}).$$
- For TD(0), the target is the TD target $R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w})$:
$$\Delta\mathbf{w} = \alpha\Big(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})\Big)\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w}).$$
- Note: The TD target contains $\hat{v}(S_{t+1}, \mathbf{w})$, which depends on $\mathbf{w}$, but we do not differentiate through it (we treat it as a constant at each time step).
- For TD(λ), the target is the λ-return G_t^λ:
$$\Delta\mathbf{w} = \alpha\big(G_t^\lambda - \hat{v}(S_t, \mathbf{w})\big)\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w}).$$
In the backward view of linear TD(λ):
$$\begin{aligned}\delta_t &= R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}),\\ E_t &= \gamma\lambda E_{t-1} + \mathbf{x}(S_t),\\ \Delta\mathbf{w} &= \alpha\,\delta_t\,E_t.\end{aligned}$$

8 7. Policy Gradient Methods

8.1 Introduction

8.1.1 Policy-based Reinforcement Learning

We directly parameterize the policy:
$$\pi_\theta(s, a) = \mathcal{P}[a \mid s, \theta].$$

Advantages:

- Better convergence properties.
- Effective in high-dimensional or **continuous action spaces**.
- Can learn stochastic policies.

Disadvantages:

- Convergence to a local rather than global optimum.
- Evaluating a policy is typically inefficient and high variance.

8.1.2 Policy Gradient

Let $J(\theta)$ be the policy objective function. To find a local **maximum** of the policy objective function, we perform:
$$\Delta\theta = \alpha\,\nabla_\theta J(\theta),$$
where
$$\nabla_\theta J(\theta) = \begin{pmatrix}\frac{\partial J(\theta)}{\partial\theta_1}\\ \vdots\\ \frac{\partial J(\theta)}{\partial\theta_n}\end{pmatrix}.$$

Score Function Trick:
$$\nabla_\theta\pi(s, a) = \pi_\theta(s, a)\,\nabla_\theta\log\pi_\theta(s, a).$$
The *score function* is $\nabla_\theta\log\pi_\theta(s, a)$.

Policy Examples:

- Softmax policy for discrete actions.
- Gaussian policy for continuous action spaces.

For one-step MDPs, applying the score function trick:

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\pi_\theta}[r] = \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \pi_\theta(s,a)\,\mathcal{R}_{s,a}, \\ \nabla J(\theta) &= \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \pi_\theta(s,a)\,\nabla_\theta \log \pi_\theta(s,a)\,\mathcal{R}_{s,a} \\ &= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\, r]. \end{aligned}$$

8.1.3 Policy Gradient Theorem

The policy gradient is given by:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\big[\nabla_\theta \log \pi_\theta(s,a)\, Q^{\pi_\theta}(s,a)\big].$$

8.2 Monte-Carlo Policy Gradient (REINFORCE)

Using the return v_t as an unbiased sample of $Q^{\pi_\theta}(s_t,a_t)$:

$$\Delta\theta_t = \alpha\,\nabla_\theta \log \pi_\theta(s_t,a_t)\, v_t, \quad \text{with } v_t = G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots.$$

Pseudo-code for REINFORCE:

function REINFORCE
  Initialize θ arbitrarily
  for each episode $\{s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, R_T\} \sim \pi_\theta$ **do**
    for $t = 1$ to $T-1$ **do**
      $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t,a_t) v_t$
    end for
  end for
  return θ
end function

REINFORCE suffers from a high variance problem since v_t is estimated by sampling.

8.3 Actor-Critic Policy Gradient

8.3.1 Idea

Use a critic to estimate the action-value function:

$$Q_w(s,a) \approx Q^{\pi_\theta}(s,a).$$

The actor-critic algorithm approximates the policy gradient as:

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}\big[\nabla_\theta \log \pi_\theta(s,a)\, Q_w(s,a)\big],$$

and the update becomes:

$$\Delta\theta = \alpha\,\nabla_\theta \log \pi_\theta(s,a)\, Q_w(s,a).$$

8.3.2 Action-Value Actor-Critic

Using a linear function approximator $Q_w(s,a) = \phi(s,a)^T w$:

- The critic updates w using TD(0).
- The actor updates θ using the policy gradient.

Pseudo-code for QAC:

Algorithm 1 QAC

procedure QAC
  Initialize state s and policy parameters θ
  Sample action $a \sim \pi_\theta(s)$
  for each step **do**
    Sample reward $r = \mathbb{R}(s, a)$
    Sample transition $s' \sim P(s' \mid s, a)$
    Sample action $a' \sim \pi_\theta(s')$
    $\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$
    $\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$
    $w = w + \beta \delta \psi(s, a)$
    $s \leftarrow s'; a \leftarrow a'$
  end for
end procedure

Observation: Value-based learning is a special case of actor-critic, since the greedy policy derived from Q (when the policy gradient step size is very large) will assign probability nearly 1 to the action with maximum Q.

8.3.3 Reducing Variance using a Baseline

Subtracting a baseline function $B(s)$ from the policy gradient can reduce variance without changing its expectation:

$$\begin{aligned}
\mathbb{E}_{\pi_\theta}\big[\nabla_\theta \log \pi_\theta(s, a) B(s)\big] &= \sum_{s \in \mathcal{S}} d^{\pi_\theta}(s) \sum_{a} \nabla_\theta \pi_\theta(s, a) B(s) \\
&= \sum_{s \in \mathcal{S}} d^{\pi_\theta}(s) B(s) \nabla_\theta \sum_{a \in \mathcal{A}} \pi_\theta(s, a) \\
&= \sum_{s \in \mathcal{S}} d^{\pi_\theta}(s) B(s) \nabla_\theta (1) \\
&= 0.
\end{aligned}$$

A good baseline is the state value function: $B(s) = V^{\pi_\theta}(s)$. Then, we can define the advantage function:

$$A^{\pi_\theta}(s, a) = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s)$$

and the policy gradient becomes:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\big[\nabla_\theta \log \pi_\theta(s, a)\, A^{\pi_\theta}(s, a)\big].$$

Estimating the Advantage Function:

- Use two networks to estimate Q and V separately (more complex).
- More commonly, use bootstrapping via the TD error:

  $$\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s),$$

  which is an unbiased estimate of the advantage:

  $$E_{\pi_\theta}\big[\delta^{\pi_\theta} \mid s, a\big] = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s) = A^{\pi_\theta}(s, a).$$

Thus,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\big[\nabla_\theta \log \pi_\theta(s, a)\, \delta^{\pi_\theta}\big].$$

In practice, an approximate TD error for one step is:

$$\delta_v = r + \gamma V_v(s') - V_v(s).$$

For the critic, we can use methods such as MC, TD(0), TD(λ), or TD(λ) with eligibility traces.

Examples:

- MC Policy Gradient:
$$\Delta\theta = \alpha\left(v_t - V_v(s_t)\right)\nabla_\theta \log \pi_\theta(s_t, a_t)$$
- TD(0):
$$\Delta\theta = \alpha\left(r + \gamma V_v(s_{t+1}) - V_v(s_t)\right)\nabla_\theta \log \pi_\theta(s_t, a_t)$$
- TD(λ):
$$\Delta\theta = \alpha\left(v_t^\lambda + \gamma V_v(s_{t+1}) - V_v(s_t)\right)\nabla_\theta \log \pi_\theta(s_t, a_t)$$
- TD(λ) with Eligibility Traces (backward view):
$$\begin{aligned}\delta_t &= r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t),\\ e_{t+1} &= \lambda\, e_t + \nabla_\theta \log \pi_\theta(s, a),\\ \Delta\theta &= \alpha\, e_t.\end{aligned}$$

For continuous action spaces, Gaussian policies are often used, but due to the noise inherent in Gaussian distributions, it is sometimes preferable to use a *deterministic policy* (by selecting the mean) to reduce noise and facilitate convergence. This leads to the **Deterministic Policy Gradient (DPG)** algorithm.

8.3.4 Deterministic Policy Gradient (Off-policy)

For a deterministic policy:
$$a_t = \mu(s_t \mid \theta^\mu),$$
with a Q-network parameterized by θ^Q and the state distribution under the behavior policy ρ^β, the critic loss is:
$$\begin{aligned}L(\theta^Q) &= \mathbb{E}_{s_t\sim\rho^\beta,\, a_t\sim\beta,\, r_t\sim E}\Big[\left(Q(s_t, a_t \mid \theta^Q) - y_t\right)^2\Big],\\ y_t &= r(s_t, a_t) + \gamma\, Q\big(s_{t+1}, \mu(s_{t+1}) \mid \theta^Q\big).\end{aligned}$$
The actor's objective is:
$$\begin{aligned}J(\theta^\mu) &= \mathbb{E}_{s\sim\rho^\beta}\Big[Q\big(s, \mu(s \mid \theta^\mu) \mid \theta^Q\big)\Big],\\ \nabla_{\theta^\mu} J &\approx \mathbb{E}_{s\sim\rho^\beta}\Big[\nabla_a Q(s, a \mid \theta^Q)\big|_{a=\mu(s)}\, \nabla_{\theta^\mu}\mu(s \mid \theta^\mu)\Big].\end{aligned}$$

To improve training stability, target networks are used for both the critic and actor, updated by a *soft update*:
$$\begin{aligned}\theta^{Q'} &\leftarrow \tau\,\theta^Q + (1-\tau)\theta^{Q'},\\ \theta^{\mu'} &\leftarrow \tau\,\theta^\mu + (1-\tau)\theta^{\mu'},\end{aligned}$$
with τ set very small (e.g., $\tau = 0.001$).

Additionally, noise is added to the deterministic action during exploration:
$$\mu'(s_t) = \mu(s_t \mid \theta_t^\mu) + \mathcal{N}_t,$$
where $\mathcal{N}_t$ is noise (e.g., Ornstein-Uhlenbeck noise).

9 8. Integrating Learning and Planning

9.1 Introduction

Model-free RL:

- No model.
- Learn the value function (and/or policy) directly from experience.

Model-based RL:

- Learn a model from experience.
- Plan the value function (and/or policy) using the model.

We define a model as $\mathcal{M} = \langle \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$, where

$$S_{t+1} \sim \mathcal{P}_\eta(s_{t+1} \mid s_t, A_t), \quad R_{t+1} = \mathcal{R}_\eta(R_{t+1} \mid s_t, A_t).$$

Model learning from experience $\{S_1, A_1, R_2, \ldots, S_T\}$ is performed via supervised learning:

$$\begin{gathered} S_1, A_1 \to R_2, S_2, \\ S_2, A_2 \to R_3, S_3, \\ \vdots \\ S_{T-1}, A_{T-1} \to R_T, S_T. \end{gathered}$$

Here, learning $s, a \to r$ is a regression problem, and learning $s, a \to s'$ is a density estimation problem.

9.2 Planning with a Model

9.2.1 Sample-based Planning

1. Sample experience from the model.
2. Apply model-free RL methods to the samples, such as Monte-Carlo control, Sarsa, or Q-learning.

The performance of model-based RL is limited to the optimal policy for the approximate MDP.

9.3 Integrated Architectures

Integrating learning and planning is exemplified by the **Dyna** framework:

- Learn a model from real experience.
- Learn and plan the value function (and/or policy) using both real and simulated experience.

9.4 Simulation-Based Search

- **Forward Search:** Select the best action by lookahead.
- Build a search tree with the current state s_t at the root.
- Solve the sub-MDP starting from the current state.

9.4.1 Simulation-Based Search Process

1. Simulate episodes of experience from the current state using the model.
2. Apply model-free RL to the simulated episodes (e.g., Monte-Carlo search, TD search).

9.4.2 Sample Monte-Carlo Search

- Given a model $\mathcal{M}_v$ and a simulation policy π:
    1. For each action $a \in \mathcal{A}$, simulate K episodes from the current (real) state s_t:
    $$\{s_t, a, R^k_{t+1}, S^k_{t+1}, A^k_{t+1}, \ldots, s^k_T\}^K_{k=1} \sim \mathcal{M}_v, \pi.$$
    2. Evaluate the action by computing the mean return:
    $$Q(s_t, a) = \frac{1}{K} \sum_{k=1}^{K} G_t \quad \xrightarrow{P} \quad q_\pi(s_t, a).$$
- Select the action with maximum estimated value:
$$a_t = \arg\max_{a \in \mathcal{A}} Q(s_t, a).$$

9.4.3 Monte-Carlo Tree Search (MCTS)

- Given a model $\mathcal{M}_v$, simulate K episodes from the current state s_t using the simulation policy π:
$$\{s_t, A_t^k, R_{t+1}^k, S_{t+1}^k, A_{t+1}^k, \ldots, s_T^k\}_{k=1}^K \sim \mathcal{M}_v, \pi.$$
- Build a search tree of visited states and actions.
- Evaluate states $Q(s,a)$ by the mean return of episodes passing through s,a:
$$Q(s_t, a) = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{u=t}^{T} \mathbf{1}(s_u, A_u = (s,a))\, G_u \quad \xrightarrow{P} \quad q_\pi(s_t, a).$$
- After search is finished, select the real action with maximum value:
$$a_t = \arg\max_{a \in \mathcal{A}} Q(s_t, a).$$

Each simulation consists of two phases:

- **Tree Policy (improves):** Pick actions to maximize $Q(s,a)$.
- **Default Policy (fixed):** Pick actions randomly.

Note: Q-values are updated on the entire subtree, not only at the current state. After each search episode, the policy is improved based on the updated Q-values and a new search begins. With progress, the search exploits promising directions while still exploring others (e.g., via MCTS with Upper Confidence Bounds as in AlphaZero).

Temporal-Difference Search: For example, update using Sarsa:
$$\Delta Q(S,A) = \alpha\Big(R + \gamma Q(S', A') - Q(S,A)\Big).$$
One may also use function approximation for simulated Q-values.

Dyna-2:

- **Long-term memory (real experience):** Use TD learning.
- **Short-term memory (working memory):** Use simulated experience with TD search & TD learning.

10 9. Exploration and Exploitation

10.1 Ways to Explore

- **Random Exploration:**
  - Use Gaussian noise in continuous action spaces.
  - ϵ-greedy: choose a random action with probability ϵ.
  - Softmax: select an action based on the softmax of the policy distribution.
- **Optimism in the Face of Uncertainty:** Prefer to explore state/actions with highest uncertainty.
  - Optimistic Initialization.
  - UCB (Upper Confidence Bounds).
  - Thompson Sampling.
- **Information State Space:**
  - Gittins indices.
  - Bayes-adaptive MDPs.

State-action exploration versus parameter exploration.

10.2 Multi-arm Bandit

Total Regret:

$$
\begin{aligned}
L_t &= \mathbb{E}\left[\sum_{\tau=1}^{t}\big(V^* - Q(a_\tau)\big)\right] \\
&= \sum_{a\in\mathcal{A}} \mathbb{E}[N_t(a)]\big(V^* - Q(a)\big) \\
&= \sum_{a\in\mathcal{A}} \mathbb{E}[N_t(a)]\,\Delta a.
\end{aligned}
$$

Optimistic Initialization:

- Initialize $Q(a)$ to a high value.
- Then act greedily.
- This leads to linear regret.

ϵ-greedy:

- Also leads to linear regret.
- Decaying ϵ-greedy (with properly tuned decay) can yield sub-linear regret (often logarithmic in t).

The regret lower bound (logarithmic bound):

$$
\lim_{t\to\infty} L_t \geq \log t \sum_{a:\Delta a>0} \frac{\Delta a}{KL\big(\mathcal{R}^a \,\|\, \mathcal{R}^{a_*}\big)}.
$$

10.2.1 Optimism in the Face of Uncertainty: Upper Confidence Bounds (UCB)

- Estimate an upper confidence $U_t(a)$ for each action value such that with high probability,
$$
Q(a) \leq \hat{Q}_t(a) + U_t(a).
$$
- The upper confidence depends on the number of times $N(s)$ has been sampled.
- Select the action maximizing the upper confidence bound:
$$
A_t = \arg\max_{a\in\mathcal{A}}\Big[Q(s_t, a) + U_t(a)\Big].
$$

Theorem (Hoeffding's Inequality):

Let $x_1, \ldots, x_t$ be i.i.d. random variables in $[0,1]$, and let $\overline{X}_t = \frac{1}{t}\sum_{\tau=1}^{t} x_\tau$. Then,

$$
\mathbb{P}\Big[\mathbb{E}[X] > \overline{X}_t + u\Big] \leq e^{-2tu^2}.
$$

Applying Hoeffding's inequality to the rewards of the bandit for a given action a:

$$
\mathbb{P}\Big[Q(a) > \hat{Q}(a) + U_t(a)\Big] \leq e^{-2N_t(a)U_t(a)^2}.
$$

If we set a probability p such that this holds:

$$
e^{-2N_t(a)U_t(a)^2} = p,
$$

then solving for $U_t(a)$ gives:

$$
U_t(a) = \sqrt{\frac{-\log p}{2N_t(a)}}.
$$

If we let $p = t^{-4}$, then:

$$
U_t(a) = \sqrt{\frac{2\log t}{N_t(a)}}.
$$

This ensures we select the optimal action as $t \to \infty$.

UCB1 Algorithm:

$$A_t = \arg\max_{a \in \mathcal{A}} \left[Q(s_t, a) + \sqrt{\frac{2 \log t}{N_t(a)}} \right].$$

The UCB algorithm achieves logarithmic asymptotic total regret:

$$\lim_{t \to \infty} L_t \leq 8 \log t \sum_{a : \Delta > 0} \Delta a.$$

Bayesian Bandits: Probability matching (Thompson Sampling) is optimal for the one-armed bandit, though it may not be as effective in MDPs.

10.3 Solving Information State Space Bandits — MDP

Define an MDP on the information state space.

10.4 MDP Exploration with UCB

In an MDP, UCB can be generalized as:

$$A_t = \arg\max_{a \in \mathcal{A}} \Big[Q(s_t, a) + U_t(s_t, a) \Big].$$

Another algorithm is the R-Max algorithm.