CS-566 Deep Reinforcement Learning

Model-Free Learning - I
Reward Sampling

Nazar Khan
Department of Computer Science
University of the Punjab

Model-Free Learning: Motivation

- So far: Value Iteration (known model-based) computes the policy using the transition model.
- Problem: In many environments, the transition probabilities are **unknown**.
- Solution: Use **model-free algorithms** that learn directly from experience.
- Key milestone: Enabled reinforcement learning to work in real-world problems.

Model-Free Learning: Overview

- Focus: **value-based** model-free algorithms.
- Instead of knowing the transition model:
  - The agent interacts with the environment.
  - Learns from sampled rewards and state transitions.
- Goal: Learn an optimal policy π^* without knowing transition dynamics.

Three Principles of Model-Free Learning

- There are three principles of model-free learning.
  1. *Reward Sampling*: How should rewards be sampled from the environment?
  2. *Action Selection*: How does the agent decide which action to take?
  3. *Learning from Rewards*: How to use reward to make the agent better?
- Different answers to these questions lead to different methods of reinforcement learning.
- In this lecture: Reward Sampling.

Monte Carlo Sampling: Intuition

- Idea: Learn from **complete episodes**.
- Generate a random episode by interacting with the environment.
- Use its return to update the value function at the visited states.
- Named *Monte Carlo* after the famous casino, due to random sampling.

Two Loops in Monte Carlo Learning

1. Loop over time steps of the episode.
2. Loop over many episodes until values converge.

Monte Carlo Sampling

Pseudocode

1. **Initialization:** Start with arbitrary $Q(s, a)$ values.
2. **Episode loop:** Generate many episodes.
3. **Within each episode:**
   - Collect (s, a, r) tuples until terminal state.
4. **Return calculation:** Work backwards to compute G_t for each time step $t \in \{T, T-1, \ldots, 1, 0\}$.

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

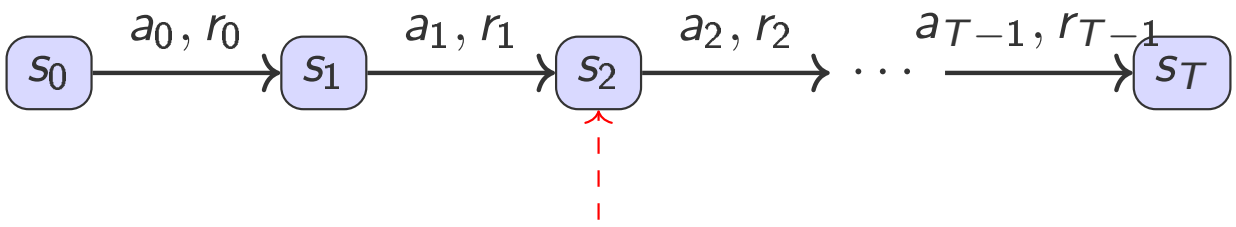
5. **Update rule:**

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \underbrace{\big(G_t - Q(s_t, a_t)\big)}_{\text{Monte Carlo Error}}$$

Note

Uses **incremental implementation**, suitable for non-stationary environments.

Monte Carlo Sampling: Illustration



$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

Key Idea

At the end of the episode, compute G_t for each state s_t visited and use it to update the value function.

Monte Carlo Sampling: Code Structure

```
def monte_carlo(n_samples, ep_length, alpha, gamma):
    # 0: initialize
    t = 0; total_t = 0
    Qsa = []

    # sample n_times
    while total_t < n_samples:

        # 1: generate a full episode
        s = env.reset()
        s_ep = []
        a_ep = []
        r_ep = []
        for t in range(ep_length):
            a = select_action(s, Qsa)
            s_next, r, done = env.step(a)
            s_ep.append(s)
            a_ep.append(a)
```

Monte Carlo Sampling: Code Structure

```
            r_ep.append(r)

            total_t += 1
            if done or total_t >= n_times:
                break;
            s = s_next

        # 2: update Q function with a full episode (incremental
        # implementation)
        g = 0.0
        for t in reversed(range(len(a_ep))):
            s = s_ep[t]; a = a_ep[t]
            g = r_ep[t] + gamma * g
            Qsa[s,a] = Qsa[s,a] + alpha * (g - Qsa[s,a])

    return Qsa

def select_action(s, Qsa):
```

Monte Carlo Sampling: Code Structure

```
    # policy is egreedy
    epsilon = 0.1
    if np.random.rand() < epsilon:
        a = np.random.randint(low=0,high=env.n_actions)
    else:
        a = argmax(Qsa[s])
    return a

env = gym.make('Taxi-v3')
monte_carlo(n_samples=10000, ep_length=100, alpha=0.1, gamma
    =0.99)
```

High Variance of Monte Carlo Estimates

- Consider estimating G_t not once but a 100 times.
- Since action selection is random, you will get a random trace for each of the 100 estimates.
- Each trace can become really really different from the others.
- So the 100 estimates of G_t can also be really really different from each other.
- Therefore, estimates computed from MC Sampling have high variance.

Monte Carlo Sampling: Pros and Cons

Advantages

- Conceptually simple.
- Works without knowing transitions.
- Easy to implement.

Disadvantages

- Must wait until end of episode to update values.
- Inefficient in long episodes.
- High variance in estimates.

Motivation for Next Step

Leads to **Temporal Difference (TD) learning**, which updates values after each step by bootstrapping.

Temporal Difference (TD) Learning

Bootstrapping and Model-Free Updates

- Recall: In **Value Iteration**, Bellman's equation computes values recursively using successor states.
- In model-free RL, we don't have the transition model $T(s, a, s')$.
- But we can still refine estimates step by step from sampled experience.
- This is called **bootstrapping**: refine old estimates with new updates.

Idea

Use the difference between successive time steps to update the current value estimate.

Bootstrapping Explained

- "Pull yourself up by your bootstraps" $\rightarrow$ refine estimates iteratively.
- Bellman recursion is itself a form of bootstrapping.
- The value of a state depends on the values of its successor states.

$$V(s) = \max_a \mathbb{E}[r + \gamma V(s')]$$

Model-based RL
Compute expectation $\mathbb{E}$ using transition probabilities.

Model-free RL
Use sample transitions (s, r, s') instead of knowing transition probabilities.

Key Question

How can we compute the value of a state using only sampled transitions?

TD Learning Update Rule[1]

$$V(s_t) \leftarrow V(s_t) + \alpha \Big[\underbrace{r_{t+1} + \gamma V(s_{t+1}) - V(s_t)}_{\text{temporal difference error } \delta} \Big]$$

Interpretation

- $V(s_t)$ predicts future reward from *now*.
- $V(s_{t+1})$ predicts future reward from the *next time step*.
- $r_{t+1} + \gamma V(s_{t+1})$ represents a *one-step lookahead* estimate of the total return from s_t.
- $\delta = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ is the difference between the estimate after looking one-step ahead and the estimate now.
- *TD Learning* updates current estimate by adding the (scaled) *temporal difference error*.

[1] Richard S Sutton. 'Learning to predict by the methods of temporal differences'. In: *Machine Learning* 3.1 (1988), pp. 9–44.

Alternative Formulation of TD Learning

$$V(s_t) \leftarrow \alpha\big[r_{t+1} + \gamma V(s_{t+1})\big] + (1-\alpha)V(s_t)$$

- Weighted average between:
  - $V(s_t)$: current estimate of future reward
  - $r_{t+1} + \gamma V(s_{t+1})$: new estimate of future reward *after looking one-step into the future*.

Implementation

Learning takes place on Q, not V

While the TD update equation is theoretically introduced in terms of V, in learning implementations, it is applied to Q.

- Leads to two different TD Learning implementations.
  1. **SARSA (On-policy learning)**

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big[\underbrace{R(s_t, a_t) + \gamma Q(s_{t+1}, a_{t+1})}_{\text{policy } \pi \text{ for behaviour and learning}} - Q(s_t, a_t)\Big]$$

  2. **Q-Learning (Off-policy learning)**

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big[\underbrace{R(s_t, a_t)}_{\pi \text{ for behaviour}} + \gamma \underbrace{\max_{a'} Q(s_{t+1}, a')}_{\pi^* \text{ for learning}} - Q(s_t, a_t)\Big]$$

SARSA

```
# Temporal Difference SARSA
Q = np.zeros((n_states, n_actions))

for episode in range(n_episodes):
    s = env.reset()
    a = epsilon_greedy(Q, s, epsilon)
    done = False

    while not done:
        s_next, r, done, _ = env.step(a)
        a_next = epsilon_greedy(Q, s_next, epsilon)

        # TD update (SARSA)
        Q[s,a] = Q[s,a] + alpha * (
            r + gamma * Q[s_next, a_next] - Q[s,a]
        )

        s, a = s_next, a_next
```

Q-learning

```
# Temporal Difference Q-learning
Q = np.zeros((n_states, n_actions))

for episode in range(n_episodes):
    s = env.reset()
    done = False

    while not done:
        a = epsilon_greedy(Q, s, epsilon)
        s_next, r, done, _ = env.step(a)

        # TD update (Q-learning)
        Q[s,a] = Q[s,a] + alpha * (
            r + gamma * np.max(Q[s_next,:]) - Q[s,a]
        )

        s = s_next
```

Advantages and Impact

- TD combines ideas of **Monte Carlo** (sample-based) and **Dynamic Programming** (bootstrapping).
- More efficient than full-episode Monte Carlo: updates can occur at each time step.
- Enabled model-free learning in complex domains.

Famous Application

TD-Gammon[a] developed by Gerald Tesauro[b] beat world champions in Backgammon using TD learning.

[a] Gerald Tesauro. 'Temporal difference learning and TD-Gammon'. In: *Communications of the ACM* 38.3 (1995), pp. 58–68.

[b] Earned his PhD in Physics in 1986.

Bias–Variance Trade-off

Monte Carlo vs. Temporal Difference

- Key difference between Monte Carlo (MC) and Temporal Difference (TD):
  - MC: no bootstrapping
  - TD: uses bootstrapping
- Bootstrapping introduces a trade-off between **bias** and **variance**.

Monte Carlo Characteristics

- MC waits until the **end of an episode** to update values.
- Uses many random action choices $\rightarrow$ updates are **unbiased**.
- Randomness across full episodes $\rightarrow$ **high variance** in returns.

Monte Carlo

Low Bias / High Variance

Temporal Difference Characteristics

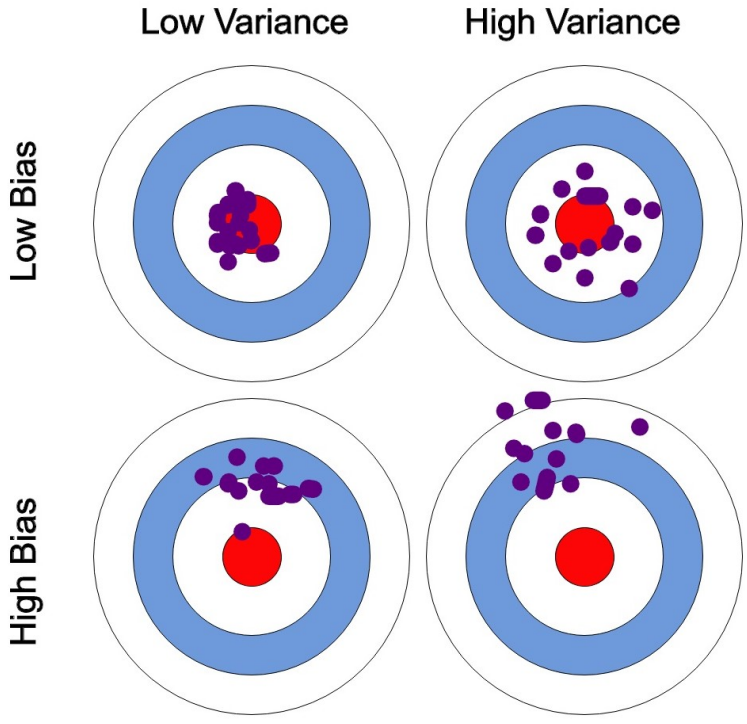
- TD updates the value function **after every step**.
- Old values are reused in updates $\rightarrow$ **bias is introduced**.
- But because updates are incremental, variance is **lower**.

Temporal Difference

High Bias / Low Variance

Bias–Variance Illustrated

The Dartboard Analogy

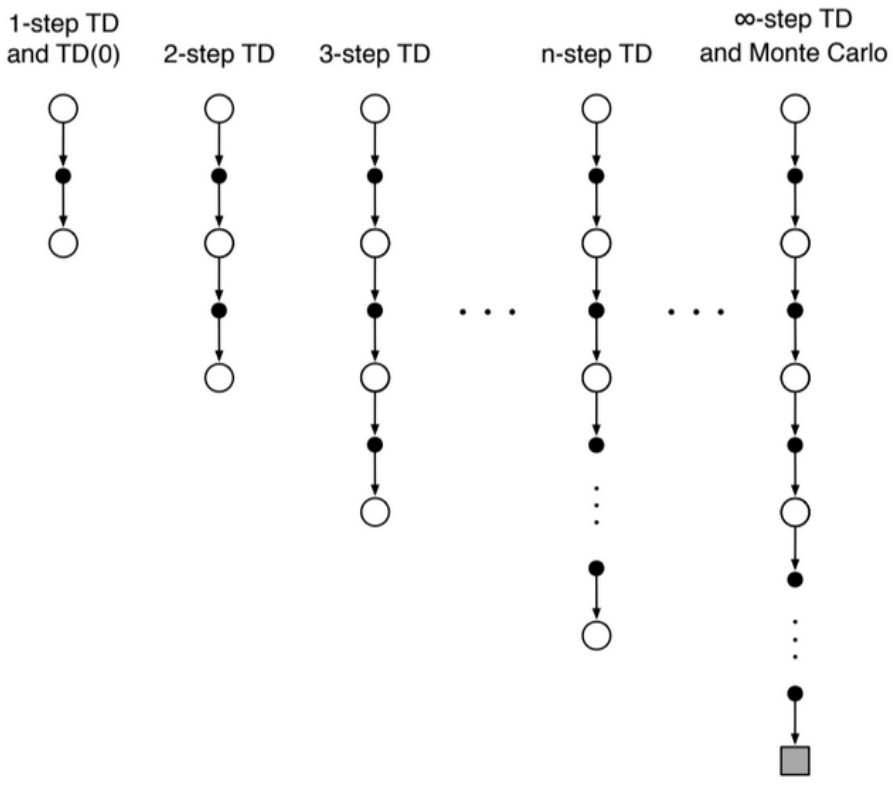


- High bias $\rightarrow$ shots far from target center.
- High variance $\rightarrow$ shots spread out.
- Goal: balance accuracy (low bias) and consistency (low variance).

Finding Middle Ground: N-step Methods

- Can we combine the best of MC and TD?
- Idea: use *n*-**step returns**.
  - Not a full episode (like MC).
  - Not just one step (like TD).
  - Instead: update after n steps.
- Results in **medium bias** and **medium variance**.

MC, TD, and N-step Compared



- Monte Carlo $\rightarrow$ full-episode updates.
- TD $\rightarrow$ single-step bootstrapping.
- n-step $\rightarrow$ compromise: updates after n steps.

Finding a Policy from Value Functions
Value-based Learning

- Goal of reinforcement learning: construct a policy π with the highest cumulative reward.
- In the **value-based approach**, we use $V(s)$ or $Q(s, a)$ to guide action selection.
- In discrete action spaces:
  - At least one action has the highest value.
  - That action becomes the best choice in the policy.

Optimal Policy

$$\pi^\star = \max_\pi V^\pi(s) = \max_{a,\pi} Q^\pi(s, a)$$

$$a^\star = \arg\max_{a \in A} Q^\star(s, a)$$

Value-based Policy Extraction

- The optimal policy sequence $\pi^\star(s)$ is recovered by:
  1. Learning $Q^\star(s, a)$ or $V^\star(s)$.
  2. Selecting $a^\star = \arg\max_a Q^\star(s, a)$ at each state.
- This is why methods are called **value-based**: the policy comes from values.

Key Idea

Value function $\longrightarrow$ Best actions $\longrightarrow$ Policy

Summary

- In this lecture, we learned different approaches to computing reward of sampled actions.
  - Full-episode
  - One-step
  - n-step
- Different ways of computing reward lead to different methods of RL.
  - Monte Carlo
  - TD Learning
- *Next lecture:* How should actions be sampled?