

The **sample average method** estimates the value of an action  $Q_t(a)$  by averaging all rewards received from taking that action prior to time  $t$ . Here's a structured breakdown:

### Formula:

$$Q_t(a) = \frac{\text{Sum of rewards when action } a \text{ was taken prior to } t}{\text{Number of times action } a \text{ was taken prior to } t}$$

### Key Components:

1. **Sum of Rewards:** Total rewards accumulated from choosing action  $a$  before time  $t$ .
2. **Count of Action:** Number of times action  $a$  has been selected before time  $t$ .

### Incremental Update:

Instead of storing all past rewards, update  $Q_t(a)$  incrementally when action  $a$  is taken at time  $t$ :

$$Q_{t+1}(a) = Q_t(a) + \frac{1}{N_{t+1}(a)} (R_t - Q_t(a))$$

where:

- $N_t(a)$  = Number of times  $a$  was taken before  $t$ .
- $R_t$  = Reward received at time  $t$ .



## Why Use Incremental Updates in the Sample Average Method?

The **sample average method** computes the action-value estimate  $Q_t(a)$  as the average of all rewards received when action  $a$  was taken. While the straightforward approach is to store all past rewards and recompute the average each time, this is inefficient. Instead, we use an **incremental update rule** for several key reasons:

---

### 1. Efficiency in Computation & Memory

- **Problem with Naive Averaging:**
  - If we store all rewards for an action, the memory requirement grows linearly with time.
  - Recomputing the average from scratch every time is computationally expensive (requires summing all past rewards each update).
- **Solution with Incremental Update:**
  - Instead of storing all rewards, we only keep track of:
    - The current estimate  $Q_t(a)$ .
    - The count  $N_t(a)$  (number of times  $a$  was taken).
  - The new estimate is computed in constant time  $O(1)$  per update:

$$Q_{t+1}(a) = Q_t(a) + \frac{1}{N_{t+1}(a)} (R_t - Q_t(a))$$



- No need to store past rewards, saving memory.
- 

## 2. Numerical Stability

- Problem with Summing Rewards:
    - If rewards are large or numerous, summing them can lead to **overflow** (if numbers get too big for storage).
    - Floating-point precision errors can accumulate over time.
  - Solution with Incremental Update:
    - Each update only involves the latest reward and a small correction term, avoiding large sums.
- 

## 3. Adaptability to Non-Stationary Problems

- Problem with Standard Averaging:
  - The sample average method assumes rewards come from a **stationary distribution** (i.e., the true mean reward doesn't change over time).
  - If the environment is **non-stationary** (e.g., the best action changes over time), old rewards should have less influence.



- **Solution with Incremental Update:**

- We can modify the update rule to **discount old rewards** by using a constant step size  $\alpha$  instead of  $\frac{1}{N_t(a)}$ :

$$Q_{t+1}(a) = Q_t(a) + \alpha (R_t - Q_t(a))$$

- This gives more weight to recent rewards, making the estimate adapt to changes.
- 

## 4. Online Learning Compatibility

- **Problem with Batch Updates:**

- Some learning scenarios require **online updates** (i.e., learning from one sample at a time without revisiting past data).
- Storing all rewards is impractical in real-time systems (e.g., robotics, trading algorithms).

- **Solution with Incremental Update:**

- The agent can update  $Q_t(a)$  immediately after each new reward, making it suitable for streaming/online learning.

The Upper Confidence Bound (UCB) formula is a key component of the **UCB algorithm**, which is used in the **multi-armed bandit problem** to balance exploration and exploitation. The UCB formula helps an agent decide which action (or "arm") to take next by considering both the estimated value of the action and the uncertainty in that estimate.

The UCB formula for selecting an action  $a$  at time  $t$  is:

$$\text{UCB}(a) = \hat{Q}(a) + c \sqrt{\frac{\ln t}{N(a)}}$$

### Explanation of Terms:

1.  $\hat{Q}(a)$ : The estimated mean reward for action  $a$  based on previous observations.
2.  $c$ : A constant (often  $c = \sqrt{2}$ ) that controls the degree of exploration. Higher values of  $c$  encourage more exploration.
3.  $t$ : The total number of time steps (or rounds) so far.
4.  $N(a)$ : The number of times action  $a$  has been selected up to time  $t$ .
5.  $\sqrt{\frac{\ln t}{N(a)}}$ : The exploration term, which accounts for the uncertainty in the estimate of  $\hat{Q}(a)$ . Actions that have been tried fewer times will have a larger exploration term, encouraging the agent to explore them.



## Intuition:

- $\hat{Q}(a)$ : Represents the exploitation component, favoring actions with higher estimated rewards.
- $\sqrt{\frac{\ln t}{N(a)}}$ : Represents the exploration component, favoring actions that have been tried fewer times or have higher uncertainty.
- The UCB formula balances these two components to ensure that the agent explores sufficiently while also exploiting the best-known actions.

## Steps of the UCB Algorithm:

1. Initialize:
  - Set  $N(a) = 0$  and  $\hat{Q}(a) = 0$  for all actions  $a$ .
2. For each time step  $t = 1, 2, \dots$ :
  - Select the action  $a$  that maximizes  $UCB(a)$ .
  - Observe the reward  $r$  for the selected action.
  - Update the estimates:
    - $N(a) \leftarrow N(a) + 1$
    - $\hat{Q}(a) \leftarrow \hat{Q}(a) + \frac{r - \hat{Q}(a)}{N(a)}$
3. Repeat until the end of the time horizon.

## Key Properties:

- **Regret Bound:** The UCB algorithm achieves logarithmic regret, meaning the total regret (difference between the optimal reward and the reward obtained by the algorithm) grows logarithmically with time.
- **Adaptive Exploration:** The exploration term decreases as an action is tried more often, naturally shifting the focus to exploitation over time.

## Example:

Suppose there are 3 actions, and at time  $t = 10$ :

- Action 1:  $N(1) = 5, \hat{Q}(1) = 0.8$
- Action 2:  $N(2) = 3, \hat{Q}(2) = 0.9$
- Action 3:  $N(3) = 2, \hat{Q}(3) = 0.7$

Using  $c = \sqrt{2}$ , the UCB values are:

- $UCB(1) = 0.8 + \sqrt{2} \sqrt{\frac{\ln 10}{5}}$
- $UCB(2) = 0.9 + \sqrt{2} \sqrt{\frac{\ln 10}{3}}$
- $UCB(3) = 0.7 + \sqrt{2} \sqrt{\frac{\ln 10}{2}}$

The agent selects the action with the highest UCB value. This ensures a balance between exploring less tried actions and exploiting actions with high estimated rewards. 

The **total discounted reward** is a key concept in reinforcement learning that represents the cumulative reward an agent receives over time, with future rewards discounted by a factor  $\gamma$ . It is used to evaluate the long-term performance of a policy or to guide learning in algorithms like **Q-learning** and **Policy Gradient Methods**.

The formula for the total discounted reward  $G_t$  starting from time step  $t$  is:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

### Explanation of Terms:

1.  $G_t$ : The total discounted reward starting from time step  $t$ .
2.  $\gamma$ : The discount factor ( $0 \leq \gamma < 1$ ), which determines the importance of future rewards. A smaller  $\gamma$  means the agent prioritizes immediate rewards, while a larger  $\gamma$  means the agent considers future rewards more heavily.
3.  $r_{t+k+1}$ : The reward received at time step  $t + k + 1$ .

### Intuition:

- The total discounted reward  $G_t$  is the sum of all future rewards, but each reward is weighted by  $\gamma^k$ , where  $k$  is the number of time steps into the future.

- This ensures that rewards further in the future contribute less to the total reward, reflecting the idea that immediate rewards are more certain and valuable than distant ones.

### Finite Horizon Case:

In episodic tasks (where the agent's interaction ends after a finite number of steps  $T$ ), the total discounted reward is calculated as:

$$G_t = \sum_{k=0}^{T-t} \gamma^k r_{t+k+1}$$

Here,  $T$  is the final time step of the episode.

### Recursive Relationship:

The total discounted reward  $G_t$  can also be expressed recursively:

$$G_t = r_{t+1} + \gamma G_{t+1}$$

This relationship is fundamental to many reinforcement learning algorithms, such as **TD learning** and **Dynamic Programming**.

## 1. $t$ :

- $t$  represents the **current time step** from which the total discounted reward is being calculated.
  - It is the starting point for summing up future rewards.
  - For example, if  $t = 0$ , the formula calculates the total discounted reward starting from the first time step.
- 

## 2. $k$ :

- $k$  is the **index of summation** and represents the number of time steps into the future relative to the current time step  $t$ .
  - It starts at  $k = 0$  (the immediate next time step) and goes to  $k = \infty$  (all future time steps).
  - For each value of  $k$ , the reward  $r_{t+k+1}$  is the reward received  $k + 1$  time steps after the current time step  $t$ .
-

## $\epsilon$ -Greedy Action Selection in Reinforcement Learning

The  $\epsilon$ -greedy (epsilon-greedy) strategy is a simple and widely-used method for balancing **exploration** (trying new actions to discover their rewards) and **exploitation** (choosing the best-known action to maximize rewards). It is commonly used in algorithms like multi-armed bandits, Q-learning, and deep reinforcement learning.

---

### How $\epsilon$ -Greedy Works

1. With probability  $1 - \epsilon$ :

- Choose the **best-known (greedy) action** (the one with the highest estimated value  $Q_t(a)$ ).
- This is **exploitation** (maximizing immediate reward).

2. With probability  $\epsilon$ :

- Choose a **random action** (uniformly or weighted by some distribution).
- This is **exploration** (discovering potentially better actions).

Mathematically:

$$a_t = \begin{cases} \arg \max_a Q_t(a) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases}$$



# Why Use $\epsilon$ -Greedy?

## 1. Balances Exploration & Exploitation

- Pure greedy ( $\epsilon = 0$ ) may get stuck in suboptimal actions if initial estimates are wrong.
- Pure random ( $\epsilon = 1$ ) wastes too much time on bad actions.
- $\epsilon$ -greedy ensures we mostly exploit but occasionally explore.

## 2. Simple to Implement

- No complex calculations (unlike methods like UCB or Thompson sampling).
- Works well in practice for many problems.

## 3. Controlled Exploration

- The value of  $\epsilon$  (e.g., 0.1, 0.01) determines how much exploration happens.
- Can be adjusted over time (e.g., decay  $\epsilon$  slowly to exploit more later).

## 1. State Value Function ( $V^\pi(s)$ )

The **state value function**  $V^\pi(s)$  represents the expected return (cumulative discounted reward) when starting in state  $s$  and following policy  $\pi$  thereafter.

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

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

- $\pi(a|s)$ : Probability of taking action  $a$  in state  $s$  under policy  $\pi$ .
- $p(s',r|s,a)$ : Transition probability of reaching state  $s'$  with reward  $r$  from  $(s,a)$ .
- $\gamma$ : Discount factor ( $0 \leq \gamma \leq 1$ ).
- $V^\pi(s')$ : Value of the next state  $s'$ .

Interpretation:

- The value of state  $s$  is the average (over all actions) of the immediate reward plus the discounted value of the next state.

## 2. Action Value Function ( $Q^\pi(s, a)$ )

The **action value function**  $Q^\pi(s, a)$  represents the expected return when taking action  $a$  in state  $s$  and thereafter following policy  $\pi$ .

Bellman Equation for  $Q^\pi(s, a)$ :

$$Q^\pi(s, a) = \sum_{s', r} p(s', r | s, a) \left[ r + \gamma \sum_{a'} \pi(a' | s') Q^\pi(s', a') \right]$$

- $p(s', r | s, a)$ : Transition dynamics.
- $\pi(a' | s')$ : Policy's probability of choosing  $a'$  in  $s'$ .
- $Q^\pi(s', a')$ : Value of the next state-action pair.

Interpretation:

- The value of  $(s, a)$  is the immediate reward plus the discounted average value of the next state-action pair under  $\pi$ .

## Bellman Optimality Equations

For the optimal value functions ( $V^*(s)$  and  $Q^*(s, a)$ ), the Bellman equations become:

Optimal State Value Function:

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

Optimal Action Value Function:

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

Key Insight:

- The optimal policy  $\pi^*$  chooses actions greedily w.r.t.  $Q^*$ :

$$\pi^*(a|s) = \begin{cases} 1 & \text{if } a = \arg \max_a Q^*(s, a), \\ 0 & \text{otherwise.} \end{cases}$$

## State Value Function ( $V^\pi(s)$ )

- **Definition:**  
 $V^\pi(s)$  is the expected cumulative discounted return when starting in state  $s$  and following policy  $\pi$  thereafter (i.e., for all future steps).
- **Key Point:**
  - At every step (including the current state  $s$ ), actions are chosen according to the policy  $\pi$ .
  - It averages over all possible actions in state  $s$ , weighted by their probabilities under  $\pi$ .

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

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

Interpretation:

- "What is the expected long-term return if I start in  $s$  and follow  $\pi$  forever?"
- The policy  $\pi$  decides all actions, including the first one in  $s$ .

## Action Value Function ( $Q^\pi(s, a)$ )

- Definition:  
 $Q^\pi(s, a)$  is the expected cumulative discounted return when starting in state  $s$ , taking action  $a$  immediately, and then following policy  $\pi$  thereafter.
- Key Point:
  - The first action is fixed (given as  $a$ ), but all future actions are chosen by  $\pi$ .

Bellman Equation for  $Q^\pi(s, a)$ :

$$Q^\pi(s, a) = \sum_{s', r} p(s', r | s, a) \left[ r + \gamma \sum_{a'} \pi(a' | s') Q^\pi(s', a') \right]$$

Interpretation:

- "What is the expected long-term return if I take action  $a$  now in  $s$ , then follow  $\pi$  afterwards?"
- The policy  $\pi$  only decides actions from the next state onward.

## Policy Iteration in Detail

Policy Iteration is a **dynamic programming** algorithm used to find the optimal policy in a **Markov Decision Process (MDP)**. It consists of two main steps:

1. **Policy Evaluation** – Compute the value function for a given policy.
2. **Policy Improvement** – Update the policy greedily based on the computed value function.

The algorithm iterates between these two steps until the policy converges to the optimal policy.

### 1. Components of an MDP

Before diving into Policy Iteration, recall that an MDP consists of:

- **States (S)**: Possible situations the agent can be in.
- **Actions (A)**: Choices available to the agent.
- **Transition Model ( $P(s'|s, a)$ )**: Probability of moving to state  $s'$  from state  $s$  after taking action  $a$ .
- **Reward Function ( $R(s, a, s')$ )**: Immediate reward received after transitioning from  $s$  to  $s'$  via action  $a$ .
- **Discount Factor ( $\gamma$ )**: Determines the importance of future rewards ( $0 \leq \gamma \leq 1$ ).



## 2. Policy Iteration Algorithm

### Step 1: Initialization

- Start with an arbitrary policy  $\pi_0$ .
- Initialize value function  $V(s)$  for all states (often set to 0).

### Step 2: Policy Evaluation

Given a policy  $\pi$ , compute the state-value function  $V^\pi(s)$  by solving the **Bellman Expectation Equation** iteratively:

$$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s, a) [R(s, a, s') + \gamma V_k(s')]$$

- Repeat until  $V$  converges (i.e., changes are below a small threshold  $\theta$ ).

### Step 3: Policy Improvement

Update the policy greedily by selecting actions that maximize the expected return:

$$\pi'(s) = \arg \max_a \sum_{s'} P(s'|s, a) [R(s, a, s') + \gamma V^\pi(s')]$$

### Step 4: Check for Convergence

- If  $\pi' = \pi$ , stop and return  $\pi$  as the optimal policy.
- Else, set  $\pi = \pi'$  and go back to **Policy Evaluation**.

### 3. Example Walkthrough

Consider a simple Gridworld with 4 states:

- States:  $S = \{1, 2, 3, 4\}$
- Actions:  $A = \{\text{Left}, \text{Right}\}$
- Rewards:  $R = 0$  everywhere except a terminal state with  $+1$ .
- Discount Factor:  $\gamma = 0.9$

#### Iteration 1: Initial Policy $\pi_0$

Assume  $\pi_0$  is "always go Right."

#### Policy Evaluation

- Compute  $V^\pi(s)$  using Bellman updates until convergence.

#### Policy Improvement

- For each state, select the action that maximizes  $V^\pi$ .
- If going Left gives a higher value than Right, update  $\pi$ .

#### Iteration 2: Improved Policy $\pi_1$

- Repeat evaluation and improvement.
- Continue until policy stops changing.

## 4. Why Policy Iteration Works

- **Policy Evaluation** ensures accurate value estimates for the current policy.
  - **Policy Improvement** guarantees the new policy is at least as good as the old one (or better).
  - The algorithm converges because there are finitely many policies, and each iteration strictly improves the policy unless it is already optimal.
- 

## 5. Policy Iteration vs. Value Iteration

Feature	Policy Iteration	Value Iteration
Approach	Alternates between policy evaluation & improvement	Directly updates value function
Convergence	Faster (fewer iterations)	Slower (more iterations)
Computation	More expensive per iteration (full policy evaluation)	Cheaper per iteration (one sweep)
Use Case	Better when policies converge quickly	Better when value function converges quickly

## 6. Pseudocode

text

 Copy  Download

1. Initialize  $V(s)$  and  $\pi(s)$  arbitrarily.
2. Repeat until policy converges:
  - a. Policy Evaluation:
    - Repeat until  $V$  converges:
      - For each  $s$  in  $S$ :
$$V(s) = \sum \pi(a|s) \sum P(s'|s,a) [R(s,a,s') + \gamma V(s')]$$
  - b. Policy Improvement:
    - For each  $s$  in  $S$ :
$$\pi(s) = \operatorname{argmax}_a \sum P(s'|s,a) [R(s,a,s') + \gamma V(s')]$$
3. Return optimal policy  $\pi$ .

## 7. Conclusion

Policy Iteration is a robust method for finding the optimal policy in an MDP. It leverages **policy evaluation** to compute state values and **policy improvement** to refine the policy iteratively. While it can be computationally intensive, it often converges faster than Value Iteration in practice.

Let's consider a 3-state Gridworld to illustrate Policy Iteration step-by-step.

---

## 1. Problem Setup

### MDP Components:

- States (S):  $\{1, 2, 3\}$  (State 3 is terminal).
- Actions (A): {Left (L), Right (R)}.
- Rewards (R):
  - Moving into State 3 gives +1.
  - All other transitions give 0.
- Transition Probabilities (P):
  - Actions succeed with probability 1 (deterministic).
- Discount Factor ( $\gamma$ ): 0.9.

### Initial Policy (Random Policy):

- $\pi_0(s) = \text{Right}$  for all states.

## 2. Policy Iteration Steps

### Iteration 1: Policy Evaluation (Compute $V^{\pi_0}$ )

We solve the Bellman equation for  $\pi_0$ :

$$V^{\pi_0}(s) = \sum_{s'} P(s'|s, \pi_0(s)) [R(s, \pi_0(s), s') + \gamma V^{\pi_0}(s')]$$

Initialize  $V_0(s) = 0$  for all states.

We update  $V$  iteratively until convergence (let's do 2 steps for simplicity).

First Update ( $k=1$ ):

- State 1 (Right → State 2):

$$V_1(1) = R(1, R, 2) + \gamma V_0(2) = 0 + 0.9 \times 0 = 0$$

- State 2 (Right → State 3):

$$V_1(2) = R(2, R, 3) + \gamma V_0(3) = 1 + 0.9 \times 0 = 1$$

- State 3 (Terminal):

$$V_1(3) = 0 \quad (\text{no future rewards})$$



## Second Update (k=2):

- State 1 (Right → State 2):

$$V_2(1) = R(1, R, 2) + \gamma V_1(2) = 0 + 0.9 \times 1 = 0.9$$

- State 2 (Right → State 3):

$$V_2(2) = R(2, R, 3) + \gamma V_1(3) = 1 + 0.9 \times 0 = 1$$

- State 3 (Terminal):

$$V_2(3) = 0$$

After more iterations,  $V$  converges to:

$$V^{\pi_0} = [0.9, 1, 0]$$

---

## Iteration 1: Policy Improvement

Now, we improve the policy greedily using  $V^{\pi_0}$ :

$$\pi_1(s) = \arg \max_a \sum_{s'} P(s'|s, a) [R(s, a, s') + \gamma V^{\pi_0}(s')]$$

### State 1: Choose Left or Right?

- Action Left ( $\rightarrow$  State 1):

$$Q(1, L) = R(1, L, 1) + \gamma V^{\pi_0}(1) = 0 + 0.9 \times 0.9 = 0.81$$

- Action Right ( $\rightarrow$  State 2):

$$Q(1, R) = R(1, R, 2) + \gamma V^{\pi_0}(2) = 0 + 0.9 \times 1 = 0.9$$

- Best action: Right ( $0.9 > 0.81$ )  $\rightarrow \pi_1(1) = \text{Right}$ .

### State 2: Choose Left or Right?

- Action Left ( $\rightarrow$  State 1):

$$Q(2, L) = R(2, L, 1) + \gamma V^{\pi_0}(1) = 0 + 0.9 \times 0.9 = 0.81$$

- Action Right ( $\rightarrow$  State 3):

$$Q(2, R) = R(2, R, 3) + \gamma V^{\pi_0}(3) = 1 + 0.9 \times 0 = 1$$

- Best action: Right ( $1 > 0.81$ )  $\rightarrow \pi_1(2) = \text{Right}$ .

### State 3: Terminal (No Action)

$$\pi_1(3) = \text{None}$$



New Policy:

$$\pi_1 = [\text{Right}, \text{Right}, \text{None}]$$

Since  $\pi_1 = \pi_0$ , the policy has not improved yet. We need another iteration.

---

## Iteration 2: Policy Evaluation (Compute $V^{\pi_1}$ )

We already computed  $V^{\pi_0} = [0.9, 1, 0]$ . Since  $\pi_1 = \pi_0$ ,  $V^{\pi_1}$  remains the same.

---

## Iteration 2: Policy Improvement

Again, we check if a better action exists.

State 1:

- $Q(1, L) = 0.81, Q(1, R) = 0.9 \rightarrow$  Still Right.

State 2:

- $Q(2, L) = 0.81, Q(2, R) = 1 \rightarrow$  Still Right.



Policy remains unchanged.

$$\pi_2 = [\text{Right}, \text{Right}, \text{None}]$$

Since  $\pi_2 = \pi_1$ , the algorithm converges.

---

### 3. Is This the Optimal Policy?

The final policy  $\pi_* = [\text{Right}, \text{Right}, \text{None}]$  is indeed optimal:

- From **State 1**, moving **Right** → **State 2** gives the best path to the reward.
  - From **State 2**, moving **Right** → **State 3** gives the immediate reward.
- 

### 4. Key Observations

1. **Policy Evaluation** computes how good the current policy is.
2. **Policy Improvement** makes the policy greedier based on  $V$ .
3. If the policy doesn't change, it means we've found the **optimal policy**.



## 5. Alternative Scenario (If Initial Policy was Different)

Suppose the initial policy was  $\pi_0 = [\text{Left}, \text{Left}, \text{None}]$ .

### First Policy Evaluation:

- $V^{\pi_0}(1) = 0 + 0.9V(1) \rightarrow V(1) = 0$ .
- $V^{\pi_0}(2) = 0 + 0.9V(1) = 0$ .
- $V^{\pi_0}(3) = 0$ .

### First Policy Improvement:

- State 1:  $Q(1, R) = 0 + 0.9V(2) = 0$  vs  $Q(1, L) = 0$ .  
→ Both actions are equal, so policy could stay or change randomly.
- State 2:  $Q(2, R) = 1 + 0 = 1$  vs  $Q(2, L) = 0$ .  
→ **Right is better!**  
→  $\pi_1(2) = \text{Right}$ .

Now, the policy improves to  $[\text{Left}, \text{Right}, \text{None}]$ , and further iterations refine it to the optimal policy.

## 6. Conclusion

This example shows how Policy Iteration:

1. Starts with a random policy.
2. Evaluates its value function.
3. Improves the policy greedily.
4. Repeats until convergence.

**Final Optimal Policy:**

$$\pi_* = [\text{Right}, \text{Right}, \text{None}]$$

with value function:

$$V_* = [0.9, 1, 0]$$

## Value Iteration: Detailed Explanation with Example

Value Iteration is another **dynamic programming** algorithm used to find the **optimal policy** in a Markov Decision Process (MDP). Unlike Policy Iteration (which alternates between policy evaluation and improvement), Value Iteration **directly computes the optimal value function**  $V^*$  and then extracts the optimal policy  $\pi^*$ .

---

### 1. Key Idea

- **Goal:** Find  $V^*$  (optimal value function) such that:

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

- **Approach:** Update  $V(s)$  iteratively using the **Bellman Optimality Equation** until convergence.
- **Once  $V^*$  is found,** derive the optimal policy:

$$\pi^*(s) = \arg \max_a \sum_{s'} P(s'|s, a) [R(s, a, s') + \gamma V^*(s')]$$

## 2. Value Iteration Algorithm

Steps:

1. Initialize  $V(s) = 0$  for all states.
2. Repeat until convergence (small threshold  $\theta$ ):
  - For each state  $s$ :

$$V_{k+1}(s) = \max_a \sum_{s'} P(s'|s, a) [R(s, a, s') + \gamma V_k(s')]$$

3. Extract the optimal policy:

$$\pi^*(s) = \arg \max_a \sum_{s'} P(s'|s, a) [R(s, a, s') + \gamma V^*(s')]$$

## MDP Setup:

- **States:**  $\{1, 2, 3\}$  (State 3 is terminal).
- **Actions:** {Left (L), Right (R)}.
- **Rewards:**
  - $R(2, R, 3) = +1$  (reaching terminal state).
  - All other rewards = 0.
- **Discount Factor**  $\gamma = 0.9$ .
- **Deterministic transitions.**

## Initialization:

$$V_0 = [0, 0, 0]$$

## Iteration 1 ( $k=0 \rightarrow k=1$ ):

- **State 1:**

$$Q(1, L) = 0 + 0.9 \times V_0(1) = 0$$

$$Q(1, R) = 0 + 0.9 \times V_0(2) = 0$$

$$V_1(1) = \max(0, 0) = 0$$



- State 2:

$$Q(2, L) = 0 + 0.9 \times V_0(1) = 0$$

$$Q(2, R) = 1 + 0.9 \times V_0(3) = 1$$

$$V_1(2) = \max(0, 1) = 1$$

- State 3 (Terminal):

$$V_1(3) = 0$$

Updated Value Function:

$$V_1 = [0, 1, 0]$$

## Iteration 2 ( $k=1 \rightarrow k=2$ ):

- State 1:

$$Q(1, L) = 0 + 0.9 \times V_1(1) = 0$$

$$Q(1, R) = 0 + 0.9 \times V_1(2) = 0.9$$

$$V_2(1) = \max(0, 0.9) = 0.9$$

- State 2:

$$Q(2, L) = 0 + 0.9 \times V_1(1) = 0$$

$$Q(2, R) = 1 + 0.9 \times V_1(3) = 1$$

$$V_2(2) = \max(0, 1) = 1$$

- State 3:

$$V_2(3) = 0$$

Updated Value Function:

$$V_2 = [0.9, 1, 0]$$

### Iteration 3 ( $k=2 \rightarrow k=3$ ):

- State 1:

$$Q(1, L) = 0 + 0.9 \times V_2(1) = 0.81$$

$$Q(1, R) = 0 + 0.9 \times V_2(2) = 0.9$$

$$V_3(1) = \max(0.81, 0.9) = 0.9$$

- State 2:

$$Q(2, L) = 0 + 0.9 \times V_2(1) = 0.81$$

$$Q(2, R) = 1 + 0.9 \times V_2(3) = 1$$

$$V_3(2) = \max(0.81, 1) = 1$$

- State 3:

$$V_3(3) = 0$$

Updated Value Function:

$$V_3 = [0.9, 1, 0]$$

Since  $V_3 = V_2$ , the algorithm **converges**.



## 4. Extracting the Optimal Policy

Now, compute  $\pi^*$  using  $V^* = [0.9, 1, 0]$ :

- State 1:

$$Q(1, L) = 0 + 0.9 \times 0.9 = 0.81$$

$$Q(1, R) = 0 + 0.9 \times 1 = 0.9$$

$$\pi^*(1) = \text{Right}$$

- State 2:

$$Q(2, L) = 0 + 0.9 \times 0.9 = 0.81$$

$$Q(2, R) = 1 + 0.9 \times 0 = 1$$

$$\pi^*(2) = \text{Right}$$

- State 3:

$$\pi^*(3) = \text{None}$$

Final Optimal Policy:

$$\pi^* = [\text{Right}, \text{Right}, \text{None}]$$



## 5. Key Differences from Policy Iteration

Feature	Value Iteration	Policy Iteration
Approach	Directly updates $V$ using Bellman optimality	Alternates between policy evaluation & improve
Convergence	Slower (requires more iterations)	Faster (fewer iterations)
Computation	Cheaper per iteration (one sweep)	More expensive (full policy evaluation)
Use Case	Better when $V$ converges quickly	Better when policies converge quickly



## 6. When to Use Value Iteration?

- When the **transition model is known** (MDP is fully specified).
- When **memory efficiency** is important (no need to store intermediate policies).
- When **only the optimal value function is needed**.

## 7. Pseudocode

text

 Copy  Download

```
1. Initialize  $V(s) = 0$  for all  $s$ .  
2. Repeat until convergence ( $\Delta < \theta$ ):  
     $\Delta = 0$   
    For each  $s$  in  $S$ :  
         $v = V(s)$   
         $V(s) = \max_a \sum P(s'|s,a) [R(s,a,s') + \gamma V(s')]$   
         $\Delta = \max(\Delta, |v - V(s)|)$   
3. Extract optimal policy:  
 $\pi^*(s) = \operatorname{argmax}_a \sum P(s'|s,a) [R(s,a,s') + \gamma V(s')]$ 
```

## 8. Conclusion

Value Iteration is a **powerful and efficient** method for finding the optimal policy in MDPs. It **bypasses explicit policy evaluation** and directly computes  $V^*$ , making it suitable for problems where the value function converges smoothly.

# 1. Policy Improvement Logic

The algorithm checks **all states** to see if the policy has changed:

python

 Copy  Download

```
policy_stable = True # Assume policy is stable initially

for each s in S:
    old_action = π_old(s)
    π_new(s) = argmax_a Q(s,a) # Greedy improvement
    if old_action != π_new(s):
        policy_stable = False # Policy changed → need another evaluation
```

## Key Points:

1. **policy\_stable** starts as **True**.
  - It only flips to **False** if **any state** has a changed action.
2. Even if 3/4 states are stable, but 1 changes:
  - **policy\_stable** becomes **False**, and the algorithm repeats Policy Evaluation.
3. Why check all states if the first change flips **policy\_stable**?
  - Because we still need to compute the **new policy** for all states (even if we already know the policy is unstable).
  - The loop ensures the **entire policy** is updated before the next evaluation.



## 2. Your Example: 4 States

State	Old Action ( $\pi_{\text{old}}$ )	New Action ( $\pi_{\text{new}}$ )	policy_stable	Status
1	Left	Left	Stays True	
2	Right	Right	Stays True	
3	Left	Left	Stays True	
4	Right	Left (changed)	Flips to False	

### What Happens?

- After State 4, `policy_stable = False`, but the loop **still completes** for all states.
- The algorithm then **re-runs Policy Evaluation** with the updated policy (even if only 1 action changed).

## 3. Why Not Exit Early?

You might ask: "Why not break the loop as soon as `policy_stable = False`?"

- Reason:** The new policy must be **fully defined** for all states before the next evaluation.
- Partial updates could lead to incorrect value estimates in the next iteration.



In reinforcement learning (RL), **target policy** and **behavioral policy** are two distinct policies that play different roles in the learning process, particularly in **off-policy** learning algorithms.

## 1. Behavioral Policy (Behavior Policy)

- **Definition:** The policy used to **select actions** and **interact with the environment** to generate experience (trajectories, transitions, etc.).
- **Purpose:** It governs how the agent explores the environment by taking actions (e.g.,  $\epsilon$ -greedy in Q-learning).
- **Key Points:**
  - It is the policy that **actually generates the data**.
  - It can be **stochastic** (e.g., random exploration) or **deterministic**, but it typically encourages exploration.
  - In **off-policy** learning, the behavioral policy is different from the target policy.

**Example:** In Q-learning, the agent might use an  $\epsilon$ -greedy policy (behavior policy) to explore while learning a greedy policy (target policy).

## 2. Target Policy

- **Definition:** The policy that is being **learned and improved** based on the experience generated by the behavioral policy.
- **Purpose:** It represents the **optimal or desired policy** the agent aims to learn.
- **Key Points:**
  - It is **not necessarily used for exploration** (unless the algorithm is **on-policy**).
  - In **off-policy** methods, the target policy may be **greedy** (e.g., Q-learning) or a **learned policy** (e.g., policy gradient methods).
  - The agent updates its estimates (e.g., Q-values) **towards the target policy's expected returns**.

**Example:** In Q-learning, the target policy is the **greedy policy** (always choosing the action with the highest Q-value), while the behavioral policy is  $\epsilon$ -greedy.

### Difference Between On-Policy and Off-Policy Learning

Aspect	On-Policy Learning	Off-Policy Learning
Behavior Policy	Same as target policy (e.g., SARSA)	Different from target policy (e.g., Q-learning)
Target Policy	Learned from its own actions	Learned from another policy's actions
Example	SARSA, REINFORCE	Q-learning, DDPG



## Example: Q-Learning (Off-Policy)

- Behavior Policy ( $\epsilon$ -greedy): Explores by sometimes taking random actions.
- Target Policy (greedy): Learns the best action based on Q-values.

## Example: SARSA (On-Policy)

- Behavior Policy = Target Policy ( $\epsilon$ -greedy): Both exploration and learning use the same policy.

## Summary

- Behavioral Policy → How the agent acts in the environment (exploration).
- Target Policy → What the agent learns (exploitation).

Understanding these concepts is crucial for designing and implementing RL algorithms, especially in **off-policy learning** where the behavior and target policies differ.

## Key Difference

- Behavioral Policy → How he explores (tries things, makes mistakes).
  - Target Policy → What he learns (the best way after learning).
- 

## Video Game Example

- Behavioral Policy (Playing Randomly Sometimes):
    - You press buttons randomly to see what happens.
  - Target Policy (Optimal Strategy):
    - You learn: "*Pressing 'A' makes me jump over enemies—that's the best move!*"
- 

## One-Liner Summary

- Behavioral Policy = "What I do to learn" (trial & error).
- Target Policy = "What I want to become" (the best strategy).

Here's a complete algorithm for Monte Carlo Generalized Policy Iteration (MC-GPI) with exploring starts:

## Algorithm: Monte Carlo GPI with Exploring Starts

text

Copy Download

Initialize:

```
 $\pi(s) \leftarrow$  arbitrary action for all  $s \in S$ 
 $Q(s,a) \leftarrow$  arbitrary value for all  $s \in S, a \in A(s)$ 
>Returns(s,a)  $\leftarrow$  empty list for all  $s \in S, a \in A(s)$ 
```

Repeat forever:

# Policy Evaluation (Monte Carlo estimation of Q)

Generate an episode using exploring starts:

1. Start episode with random  $(s_0, a_0)$  where  $s_0 \in S, a_0 \in A(s_0)$
2. Follow policy  $\pi$  until termination:  $s_0, a_0, r_1, s_1, a_1, r_2, \dots, s_{T-1}, a_{T-1}, r_T, s_T$

$G \leftarrow 0$

For each step  $t = T-1, T-2, \dots, 0$ :

$G \leftarrow \gamma G + r_{t+1}$

Unless  $(s_t, a_t)$  appears in  $s_0, a_0, s_1, a_1, \dots, s_{t-1}, a_{t-1}$ :

Append  $G$  to  $Returns(s_t, a_t)$

$Q(s_t, a_t) \leftarrow \text{average}(Returns(s_t, a_t))$

# Policy Improvement (greedy policy update)

For each  $s$  in the episode:

$\pi(s) \leftarrow \text{argmax}_a Q(s,a)$

(with random selection among maximizing actions if there are ties)

1. **Exploring Starts:** Every episode begins with a random state-action pair  $(s_0, a_0)$ , ensuring all pairs are visited infinitely often in the limit.
2. **First-Visit MC:** For each state-action pair in the episode, we only consider the first visit when calculating returns (indicated by the "unless" condition).
3. **Generalized Policy Iteration:** Alternates between:
  - Policy Evaluation: Estimating  $Q^\pi$  using Monte Carlo returns
  - Policy Improvement: Updating  $\pi$  to be greedy with respect to  $Q$
4. **On-Policy:** The same policy is used for both generating episodes and being improved.

## Practical Implementation Notes:

- In practice, you would run this for many iterations until convergence
- The "average(Returns(s,a))" can be implemented incrementally:

text

 Copy  Download

```
Q(s, a) ← Q(s, a) + (G - Q(s, a)) / (number of times (s, a) has been visited)
```

- $\gamma$  is the discount factor ( $0 \leq \gamma \leq 1$ )
- The algorithm assumes episodic tasks with finite returns

This algorithm is guaranteed to converge to the optimal policy under the conditions of the GPI theorem given that all state-action pairs are visited infinitely often (ensured by exploring starts). 

## Policy $\pi(s)$ Definition:

- A **policy** is a mapping from states ( $s$ ) to actions ( $a$ ), determining the agent's behavior.
- It can be:
  - **Deterministic:** Directly outputs a single action ( $\pi(s) = a$ ).
  - **Stochastic:** Outputs a probability distribution over actions ( $\pi(a|s)$  = *probability of taking action a in state s*).

## In Monte Carlo GPI with Exploring Starts:

- Initially,  $\pi(s)$  is set to an **arbitrary action** for all states (often random).
- After each episode, the policy is **improved greedily** based on the current action-value estimates  $Q(s,a)$ :

$$\pi(s) \leftarrow \arg \max_a Q(s, a)$$

- If multiple actions have the same maximal Q-value, one is chosen randomly.

## Key Points:

1.  $\pi(s)$  is the **decision rule** telling the agent what to do in state  $s$ .
2. It is **updated iteratively** in Generalized Policy Iteration (GPI):
  - **Policy Evaluation:** Estimates  $Q(s,a)$  (how good actions are).
  - **Policy Improvement:** Updates  $\pi(s)$  to pick the best action ( $\text{argmax } Q$ ).



## Example:

- If in state  $s$ , the Q-values are:
  - $Q(s, \text{"left"}) = 10$
  - $Q(s, \text{"right"}) = 15$
- Then the improved policy becomes:

$$\pi(s) = \text{"right"} \quad (\text{since } \arg \max_a Q(s, a) = \text{"right"})$$

Thus,  $\pi(s)$  is the action selection rule, refined over time to maximize returns.

# 1. $\epsilon$ -Greedy Monte Carlo Control (On-Policy)

This approach maintains a soft policy while learning, balancing exploration and exploitation.

## Algorithm: $\epsilon$ -Greedy MC Control

text

 Copy  Download

Initialize:

```
 $\pi \leftarrow$  an arbitrary  $\epsilon$ -soft policy (e.g.,  $\epsilon$ -greedy with respect to random Q)  
 $Q(s, a) \leftarrow$  arbitrary values for all  $s \in S, a \in A(s)$   
 $Returns(s, a) \leftarrow$  empty list for all  $s \in S, a \in A(s)$ 
```

Repeat forever:

    Generate an episode following  $\pi$ :

```
 $s_0, a_0, r_1, s_1, a_1, r_2, \dots, s_{T-1}, a_{T-1}, r_T, s_T$ 
```

$G \leftarrow 0$

    For each step  $t = T-1, T-2, \dots, 0$ :

```
 $G \leftarrow \gamma G + r_{t+1}$ 
```

    Unless  $(s_t, a_t)$  appears in  $s_0, a_0, \dots, s_{t-1}, a_{t-1}$ :

```
        Append  $G$  to  $Returns(s_t, a_t)$ 
```

```
         $Q(s_t, a_t) \leftarrow \text{average}(Returns(s_t, a_t))$ 
```

    For each  $s$  in the episode:

```
 $a^* \leftarrow \text{argmax}_a Q(s, a)$  (with random tie-breaking)
```

    For all  $a \in A(s)$ :

```
 $\pi(a|s) \leftarrow \begin{cases} 1 - \epsilon + \epsilon/|A(s)| & \text{if } a = a^* \\ \epsilon/|A(s)| & \text{otherwise} \end{cases}$ 
```



## Key Features:

- Maintains an  $\epsilon$ -soft policy throughout learning
- Gradually shifts probability toward greedy actions while maintaining exploration
- Converges to the optimal  $\epsilon$ -soft policy

## 2. Softmax Monte Carlo Control

An alternative approach using action probabilities weighted by Q-values.

### Algorithm: Softmax MC Control

text

 Copy  Download

Initialize:

```
Q(s,a) ← arbitrary values  
 $\pi(a|s)$  ← softmax( $Q(s,a)/\tau$ ) with high initial temperature  $\tau$   
Returns(s,a) ← empty lists
```

Repeat forever:

    Generate episode following  $\pi$

$G \leftarrow 0$

    For  $t = T-1, \dots, 0$ :

$G \leftarrow \gamma G + r_{t+1}$

        Unless  $(s_t, a_t)$  appears earlier in episode:

            Update Returns( $s_t, a_t$ ) with  $G$

$Q(s_t, a_t) \leftarrow \text{average}(\text>Returns(s_t, a_t))$

    For each  $s$  in episode:

        Update  $\pi(a|s) \leftarrow \exp(Q(s,a)/\tau) / \sum_{a'} \exp(Q(s,a')/\tau)$

    Optionally: Anneal  $\tau$  (reduce over time)

### 3. Off-Policy Monte Carlo Control with Importance Sampling

For learning the optimal policy while following a soft behavior policy.

#### Algorithm: Off-Policy MC Control

text

 Copy  Download

Initialize:

```
π ← deterministic greedy policy (w.r.t. random Q)
b ← soft policy (e.g., ε-greedy)
Q(s,a) ← arbitrary
C(s,a) ← 0 (cumulative weights)
```

Repeat:

```
Generate episode using b: s0, a0, r1, ..., rT, sT
G ← 0
W ← 1
For t = T-1, ..., 0:
    G ← γG + r_{t+1}
    C(st, at) ← C(st, at) + W
    Q(st, at) ← Q(st, at) + W/C(st, at) * (G - Q(st, at))
    π(st) ← argmaxa Q(st, a)
    If at ≠ π(st):
        Exit inner loop (skip remaining steps)
    W ← W / b(at|st)
```



## Key Differences from Exploring Starts:

1. **No need for exploring starts** - the soft policy ensures continual exploration
2. **On-policy methods** learn the optimal  $\epsilon$ -soft policy rather than strictly optimal
3. **Off-policy methods** can learn optimal policy while following soft policy
4. **Temperature parameter** in softmax controls exploration-exploitation tradeoff

All of these approaches guarantee convergence to optimal policies under suitable conditions (GPI theorem, infinite visits to all state-action pairs, and proper exploration).

## 1. Monte Carlo GPI with Exploring Starts (First Algorithm)

- **Policy Type:** Learns a **deterministic** (greedy) optimal policy.
- **Exploration Method:** Uses **exploring starts** (random initial state-action pairs).
- **Update Rule:** Directly sets  $\pi(s) = \operatorname{argmax}_a Q(s, a)$  after each episode.
- **Convergence:** Guaranteed to converge to the **strictly optimal policy** (if all state-action pairs are visited infinitely often).

## 2. $\epsilon$ -Greedy Monte Carlo Control (Second Algorithm)

- **Policy Type:** Maintains a **soft ( $\epsilon$ -greedy)** policy throughout learning.
- **Exploration Method:** Uses  **$\epsilon$  probability** to take random actions (no need for exploring starts).
- **Update Rule:** Adjusts probabilities:
  - Best action gets  $1 - \epsilon + \epsilon / |A(s)|$  probability.
  - Other actions get  $\epsilon / |A(s)|$  probability.
- **Convergence:** Converges to the **best  $\epsilon$ -soft policy** (near-optimal, but always explores).

## Punchline Differences

Feature	Exploring Starts (First Algo)	$\epsilon$ -Greedy (Second Algo)
Policy Type	Deterministic (greedy)	Soft ( $\epsilon$ -greedy)
Exploration	Forced via random starts	Built-in via $\epsilon$
Optimality	Strictly optimal	Near-optimal ( $\epsilon$ -soft)
Use Case	When random starts are feasible	When always-on exploration is needed

### Which to use?

- Exploring Starts if you can control episode beginnings (e.g., simulated environments).
- $\epsilon$ -Greedy if you need persistent exploration (e.g., real-world RL, no reset control).

Both are Monte Carlo control methods but differ in how they balance exploration vs. exploitation! 

In Off-Policy Monte Carlo Control with Importance Sampling, the variable  $W$  represents the **importance sampling ratio**, which is used to adjust the updates of the action-value function  $Q(s, a)$  to account for the difference between:

- **Target policy ( $\pi$ )** – The policy being learned (usually deterministic and greedy).
- **Behavior policy ( $b$ )** – The policy used to generate episodes (usually soft/exploratory, like  $\epsilon$ -greedy).

## What is $W$ ?

$$W = \prod_{k=0}^t \frac{\pi(a_k | s_k)}{b(a_k | s_k)}$$

- It is the **product of likelihood ratios** for each action taken in the episode up to time  $t$ .
- Measures **how much more (or less) likely** the actions were under  $\pi$  compared to  $b$ .

## Role of $W$ in the Algorithm

### 1. Initialization:

- $W = 1$  at the start of backward episode processing.

### 2. Update Rule:

- After each step, update:



## 2. Update Rule:

- After each step, update:

$$W \leftarrow W \cdot \frac{\pi(a_t|s_t)}{b(a_t|s_t)}$$

- If  $\pi$  is deterministic and greedy, then:

- $\pi(a_t|s_t) = 1$  if  $a_t$  is the greedy action, else 0.
- Thus, if the episode takes a non-greedy action,  $W$  becomes 0, and later steps are ignored (since they couldn't have happened under  $\pi$ ).

## 3. Weighting the Returns:

- When updating  $Q(s, a)$ , the return  $G$  is scaled by  $W$ :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{W}{C(s_t, a_t)}(G - Q(s_t, a_t))$$

- This ensures that updates are corrected for the policy mismatch between  $b$  and  $\pi$ .

## Key Implications

- If  $b$  is exploratory (e.g.,  $\epsilon$ -greedy) and  $\pi$  is greedy,  $W$  drops to 0 when the behavior policy picks a non-greedy action.
- This early termination avoids updates from trajectories that  $\pi$  would never take.
- $W$  ensures unbiased learning of  $Q^\pi$  even when following  $b$ .



## Example

Suppose:

- $\pi$  always chooses action "Up" (greedy).
- $b$  chooses "Up" with 80% and "Down" with 20%.

For an episode:  $(s_1, \text{Up}), (s_2, \text{Down})$ :

- At  $t = 0$ :  $W = \frac{1}{0.8} = 1.25$
- At  $t = 1$ :  $W = 1.25 \times \frac{0}{0.2} = 0$  (since  $\pi$  would never pick "Down") → Stop updating further steps.

Thus,  $W$  controls how much credit the target policy  $\pi$  gets for the episode's actions.

## Summary

- $W$  = Importance weight correcting for policy differences.
- Critical for off-policy learning to converge to  $Q^\pi$  while following  $b$ .
- Causes early termination if  $b$  deviates from  $\pi$ .

This makes off-policy MC data-efficient (can reuse exploratory episodes) but high-variance (due to product terms in  $W$ ).

Formula:

$$W = \prod_{k=0}^t \frac{\pi(a_k|s_k)}{b(a_k|s_k)}$$

Pronunciation (Step-by-Step):

1. "W equals" → The variable  $W$  is defined as...
2. "The product from  $k$  equals zero to  $t$ " →  $\prod_{k=0}^t$
3. "of  $\pi$  of  $a_k$  given  $s_k$ " →  $\pi(a_k|s_k)$
4. "divided by  $b$  of  $a_k$  given  $s_k$ " →  $\overline{b(a_k|s_k)}$

Full Pronunciation:

" $W$  equals the product from  $k$  equals zero to  $t$  of  $\pi$  of  $a$ -sub- $k$  given  $s$ -sub- $k$ , divided by  $b$  of  $a$ -sub- $k$  given  $s$ -sub- $k$ ."

Simplified (Natural Language):

" $W$  is the product over all time steps  $k$  from zero to  $t$  of the ratio between the target policy's probability of action  $a_k$  in state  $s_k$  and the behavior policy's probability of that same action."

## Key Symbols & Terms:

Symbol	Pronunciation	Meaning
$\prod$	"Product"	Multiplication of terms
$\pi(a_k \parallel s_k)$	"Pi of $a$ -sub- $k$ given $s$ -sub- $k$ "	Target policy's action probability
$b(a_k \parallel s_k)$	" $b$ of $a$ -sub- $k$ given $s$ -sub- $k$ "	Behavior policy's action probability
$W$	"Importance weight $W$ "	Cumulative correction factor

## Example with Numbers (for Clarity):

Suppose:

- At  $k = 0$ :  $\pi(\text{Up}|s_0) = 1.0$ ,  $b(\text{Up}|s_0) = 0.8 \rightarrow \text{Ratio} = \frac{1.0}{0.8} = 1.25$
- At  $k = 1$ :  $\pi(\text{Right}|s_1) = 0$ ,  $b(\text{Right}|s_1) = 0.1 \rightarrow \text{Ratio} = \frac{0}{0.1} = 0$

Pronounced as:

" $W$  is one-point-two-five times zero, which equals zero."

## Why It Matters:

- $W$  adjusts updates to account for **policy mismatch** (off-policy learning).
- If  $W = 0$ , the episode is **ignored** for updates (actions didn't match  $\pi$ ).

# One-Step Tabular Q-Planning Algorithm (Complete)

This algorithm is a **model-based** reinforcement learning method that updates the Q-table by simulating experiences from a learned model of the environment.

---

## Algorithm Steps

### 1. Initialization

- Initialize Q-table:  $Q(s, a) = 0$  (or small random values) for all  $s \in \mathcal{S}, a \in \mathcal{A}$ .
- Initialize an empty **environment model**  $\hat{\mathcal{M}}(s, a)$  (stores  $r, s'$  for each  $(s, a)$  pair).
- Set learning rate  $\alpha \in (0, 1]$  (default:  $\alpha = 1$ ).
- Set discount factor  $\gamma \in [0, 1]$ .

### 2. Loop for Each Planning Step

Repeat for  $N$  planning steps (or until convergence):

1. Sample a state-action pair  $(s, a)$  from the model:

- Either uniformly random, or prioritize states with high uncertainty.
- Alternatively, sample from past real experiences (replay buffer).

2. Query the model to get predicted reward and next state:

- $r, s' \leftarrow \hat{\mathcal{M}}(s, a)$ .

3. Compute TD target (one-step lookahead):

$$\text{target} = r + \gamma \cdot \max_{a'} Q(s', a')$$

4. Update Q-value using semi-gradient Q-learning:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot (\text{target} - Q(s, a))$$

5. (Optional) Improve policy (e.g.,  $\epsilon$ -greedy):

$$\pi(s) = \arg \max_a Q(s, a)$$

3. Interleave with Real Experience (Optional)

- If interacting with the real environment, update the model  $\hat{\mathcal{M}}$  with observed transitions  $(s, a, r, s')$ .
- Perform planning updates in the background while collecting real data.

## Key Properties

- **Model-based:** Requires (or learns) a dynamics model  $\hat{\mathcal{M}}$ .
  - **Off-policy:** Updates Q-values using simulated experiences.
  - **Sample-efficient:** Can improve policies without new environment interactions.
  - **Flexible:** Works with **random sampling** or **prioritized sweeping** (focusing on important states).
- 

## Example Use Cases

1. **Dyna-Q:** Combines real experience + simulated Q-planning.
2. **Batch RL:** Learns entirely from a fixed dataset by replaying transitions.
3. **Planning in MDPs:** When the full environment model is known.

Certainly! The line "planning only uses imagined experience/simulated experience" means that Q-planning updates the Q-table **without interacting with the real environment**. Instead, it relies on:

## 1. Imagined/Simulated Experience

- The algorithm **samples transitions**  $(s, a, r, s')$  from a learned model of the environment (rather than executing actions in the real world).
- These samples are "**imagined**" because they come from predictions (like a dream or simulation).

## 2. Contrast with Real Experience (e.g., Q-Learning)

Q-Learning (Real Experience)	Q-Planning (Simulated Experience)
Learns by <b>actually taking actions</b> in the environment.	Learns by <b>simulating actions</b> using a model.
Needs <b>interaction</b> with the real world.	Can improve policy <b>offline</b> (no real interactions).
Slow (limited by environment steps).	Fast (can generate unlimited simulated data).

## 3. Why Use Simulated Experience?

- **Sample Efficiency:** Reuse past data or simulate new transitions without costly real interactions.
- **Safe Exploration:** Test risky actions in simulation first.
- **Parallel Learning:** Many "imagined" updates can happen in the background.



## Example

Suppose an AI learns a model where:

- In state  $s_1$ , action  $a_1$  leads to  $s_2$  with reward  $+5$ .

Instead of **actually trying  $a_1$  in the real world**, Q-planning:

1. **Imagines**  $(s_1, a_1, +5, s_2)$ .
2. Updates  $Q(s_1, a_1)$  using this **simulated transition**.

This is why it's called "**model-based**" RL—it relies on a model to generate fake (but useful) experiences.

---

**Key Takeaway:**

Q-planning **does not require real environment steps**—it learns purely from **simulated data generated by a model**, making it computationally efficient but dependent on model accuracy.

# Dyna Architecture: Combining Real and Simulated Experience

The Dyna architecture is a hybrid reinforcement learning framework that **integrates real-world experience with model-based planning** to improve sample efficiency. It was introduced by Richard Sutton as a way to blend:

- Direct RL (Q-learning/SARSA) → Learns from **real experience**.
  - Model-based Planning (Q-planning) → Learns from **simulated experience**.
- 

## Core Idea

Dyna alternates between:

1. Taking **real actions** in the environment (like Q-learning).
2. **Simulating additional learning steps** using a learned model (like Q-planning).

This allows the agent to:

- **Learn faster** by replaying past experiences.
- **Plan ahead** without additional costly interactions.



# Key Components of Dyna

Component	Role
Real Experience (Q-learning)	Updates Q-table using actual environment interactions.
Learned Model (Model-based)	Stores transitions $(s, a) \rightarrow (r, s')$ to simulate future steps.
Simulated Experience (Q-planning)	Replays past/simulated transitions to refine Q-values.
Replay Memory	Stores past experiences for sampling in planning.

## Dyna Variants

1. **Dyna-Q** (Standard)
  - Uses random sampling from memory for planning.
2. **Dyna-Q+** (Exploration Bonus)
  - Encourages exploration by adding rewards for rarely visited states.
3. **Prioritized Sweeping Dyna**
  - Focuses planning on **high-error transitions** for faster convergence.



## Why Dyna Works?

- **Faster Learning:** Each real step is augmented with multiple simulated updates.
  - **Better Sample Efficiency:** Reuses past experiences for additional learning.
  - **Balances Real & Simulated Data:** Avoids over-reliance on an imperfect model.
- 

## Example: Maze Navigation

- **Real Experience:** Agent takes steps, hits walls, finds rewards.
  - **Simulated Experience:** Replays past moves in memory to "imagine" better paths.
  - **Result:** Learns optimal policy with **fewer real steps** than pure Q-learning.
- 

## Limitations

- **Model Bias:** If the learned model is wrong, planning misleads the agent.
- **Memory Usage:** Storing many experiences can be costly.

## Search Control in Dyna (Short & Clear Explanation)

Search control = Deciding which simulated experiences to use for planning.

It answers:

?" "Which (state, action) pairs should we sample from the model to improve learning?"

Methods of Search Control:

1. Random Sampling (Dyna-Q) → Pick past experiences randomly.
2. Prioritized Sweeping → Focus on transitions where Q-values changed the most.
3. Heuristic Selection → Prefer states near goals or high uncertainty.

Why It Matters?

- Smart search control = Faster learning (avoids wasting time on irrelevant simulations).
- Poor search control = Slow or inefficient planning.

Example:

- In a maze, prioritizing simulations near the goal speeds up learning vs. replaying random moves.

---

In One Line:

"Search control picks the best 'imagined' experiences to train on, making planning efficient."



## Why Q-Learning is Off-Policy? (Short & Clear Answer)

Because it learns the optimal Q-values (max policy) while following a different behavior policy (e.g.,  $\epsilon$ -greedy).

- Target Policy ( $\pi^*$ ): Uses  $\max$  over Q-values ( $a' = \operatorname{argmax} Q(s', a')$ ).
- Behavior Policy ( $\pi$ ): Uses exploration (e.g., random actions via  $\epsilon$ -greedy).

**Key Point:** It updates Q-values based on the *best future action* (even if not taken), making it **off-policy**.

Contrast with On-Policy (e.g., SARSA):

- SARSA follows and updates the *same* policy (including exploration moves).

---

In One Line:

"Q-learning learns the optimal policy while acting sub-optimally (off-policy)."

Q-learning is off-policy because it learns about the best possible policy (greedy) while actually behaving differently (e.g., exploring randomly).

Key points:

1. It *learns* using the best future action ( max Q-value )
2. But it *acts* using exploration (like  $\epsilon$ -greedy)
3. The policy it learns (greedy) is different from how it behaves (exploratory) → hence "off-policy"

Example:

- While exploring, you might take random actions
- But Q-learning updates assuming you'll take the best action next time
- This mismatch makes it off-policy

Parameterizing the value function is a common technique in reinforcement learning (RL) and dynamic programming when dealing with large or continuous state spaces. Instead of storing the value function  $V(s)$  or action-value function  $Q(s, a)$  explicitly for every state (or state-action pair), we approximate it using a parameterized function, such as a linear function, neural network, or other function approximator.

### Key Concepts:

1. **Value Function ( $V(s)$ ):** Estimates the expected return (cumulative reward) from state  $s$  following a policy  $\pi$ .
2. **Action-Value Function ( $Q(s, a)$ ):** Estimates the expected return from taking action  $a$  in state  $s$  and following policy  $\pi$  thereafter.
3. **Parameterization:** Represent  $V$  or  $Q$  using a function with parameters  $\theta$ , e.g.,  $V_\theta(s)$  or  $Q_\theta(s, a)$ .

## Common Approaches:

### 1. Linear Function Approximation:

Represent the value function as a linear combination of features:

$$V_{\theta}(s) = \theta^T \phi(s)$$

where:

- $\phi(s)$  is a feature vector representing state  $s$ .
- $\theta$  is a weight vector to be learned.

### 2. Neural Networks (Deep RL):

Use a deep neural network to approximate  $V$  or  $Q$ :

$$V_{\theta}(s) = \text{NN}_{\theta}(s)$$

where  $\text{NN}_{\theta}$  is a neural network with parameters  $\theta$ . This is the basis of Deep Reinforcement Learning (e.g., DQN, Policy Gradients).

### 3. Kernel Methods:

Use kernel functions to represent the value function nonlinearly, though this is less common in practice due to scalability issues.

## Algorithms for Learning Parameterized Value Functions:

### 1. TD Learning with Function Approximation:

- Update rule for  $V_\theta(s)$ :

$$\theta \leftarrow \theta + \alpha[r + \gamma V_\theta(s') - V_\theta(s)]\nabla_\theta V_\theta(s)$$

- Where  $\alpha$  is the learning rate,  $\gamma$  is the discount factor, and  $s'$  is the next state.

### 2. Q-Learning with Function Approximation:

- Update rule for  $Q_\theta(s, a)$ :

$$\theta \leftarrow \theta + \alpha[r + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s, a)]\nabla_\theta Q_\theta(s, a)$$

### 3. Policy Gradient Methods:

- Directly optimize the policy  $\pi_\theta$  using gradients of the expected return with respect to  $\theta$ , often using a value function as a baseline (e.g., Actor-Critic methods).

## Challenges:

1. **Approximation Error:** The function approximator may not perfectly represent the true value function.
2. **Stability and Convergence:** Unlike tabular methods, function approximation can diverge or become unstable if not carefully tuned (e.g., due to correlated updates or non-stationarity).
3. **Feature Engineering:** For linear methods, the choice of features  $\phi(s)$  is critical to performance.
4. **Overfitting:** In deep RL, overfitting can occur if the network is too complex relative to the data.



## Example: Linear Value Function Approximation

Suppose we have a feature extractor  $\phi(s)$  for states. The value function is:

$$V_\theta(s) = \theta_1\phi_1(s) + \theta_2\phi_2(s) + \cdots + \theta_n\phi_n(s)$$

The update rule for TD(0) is:

$$\theta \leftarrow \theta + \alpha[r + \gamma\theta^T\phi(s') - \theta^T\phi(s)]\phi(s)$$

## Deep RL Example (DQN):

In Deep Q-Networks (DQN),  $Q_\theta(s, a)$  is represented by a neural network. The loss function is:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s, a, r, s')} \left[ \left( r + \gamma \max_{a'} Q_{\theta^-}(s', a') - Q_\theta(s, a) \right)^2 \right]$$

where  $\theta^-$  are the target network parameters (fixed for stability).

## Summary:

Parameterizing the value function allows RL algorithms to scale to large or continuous state spaces by generalizing across states. The choice of parameterization (linear, neural network, etc.) and learning algorithm (TD, Q-learning, etc.) depends on the problem and computational resources.

## Why We Need Parameterized Value Functions

In reinforcement learning (RL), the **value function**  $V(s)$  estimates how good it is to be in a state  $s$ , and the **action-value function**  $Q(s, a)$  estimates how good it is to take action  $a$  in state  $s$ .

In small, discrete environments (like a grid world with few states), we can store exact values for every state in a table:

- $V(s)$  is a table mapping each state  $s$  to its value.
- $Q(s, a)$  is a table mapping each state-action pair  $(s, a)$  to its value.

But in **large or continuous spaces** (like robotic control, video games, or financial markets), storing all possible  $V(s)$  or  $Q(s, a)$  values becomes impossible because:

1. **Too many states:** E.g., a robot's state might be its position, velocity, etc., leading to infinite possibilities.
2. **Memory constraints:** Storing a table for all states is impractical.
3. **Generalization:** Even if we visit some states, we need to estimate values for unseen states.

## Solution: Parameterized Approximation

Instead of storing exact values, we approximate  $V(s)$  or  $Q(s, a)$  using a **parameterized function** (a model with adjustable parameters  $\theta$ ):

$$V_\theta(s) \approx V(s) \quad \text{or} \quad Q_\theta(s, a) \approx Q(s, a)$$



Key Idea:

- The function  $V_\theta(s)$  or  $Q_\theta(s, a)$  generalizes across similar states.
- We learn the parameters  $\theta$  to make the approximation as accurate as possible.

## Types of Parameterized Functions

### 1. Linear Approximation

- Represent  $V(s)$  as a weighted sum of features:

$$V_\theta(s) = \theta_1\phi_1(s) + \theta_2\phi_2(s) + \cdots + \theta_n\phi_n(s)$$

- Here,  $\phi(s) = [\phi_1(s), \phi_2(s), \dots, \phi_n(s)]$  are hand-designed features (e.g., distance to goal, velocity, etc.).
- Learning: Adjust  $\theta$  to minimize the error between  $V_\theta(s)$  and the true  $V(s)$ .

### 2. Neural Networks (Deep RL)

- Use a deep neural network (DNN) to approximate  $V(s)$  or  $Q(s, a)$ :

$$V_\theta(s) = \text{NeuralNetwork}_\theta(s)$$

- Advantage: Automatically learns useful features (no manual engineering).
- Example: Deep Q-Networks (DQN) approximate  $Q(s, a)$  with a DNN.

### 3. Kernel Methods

- Use similarity-based representations (less common in modern RL).



## How Learning Works

Instead of updating a lookup table, we update the **parameters**  $\theta$  to reduce prediction error.

### Example: TD Learning with Linear Approximation

1. Compute the **TD error** (difference between predicted and observed value):

$$\delta = r + \gamma V_\theta(s') - V_\theta(s)$$

2. Update  $\theta$  to reduce this error:

$$\theta \leftarrow \theta + \alpha \cdot \delta \cdot \nabla_\theta V_\theta(s)$$

(where  $\nabla_\theta V_\theta(s) = \phi(s)$  for linear approximation).

### Deep RL Example (DQN)

1. Use a neural network  $Q_\theta(s, a)$  to predict action values.
2. Minimize the loss:

$$\mathcal{L}(\theta) = \left( r + \gamma \max_{a'} Q_{\theta^-}(s', a') - Q_\theta(s, a) \right)^2$$

(where  $\theta^-$  is a **target network** for stability).



## Why This Works

- **Generalization:** The function approximator predicts values for **unseen states** based on learned patterns.
- **Efficiency:** Much less memory than storing a giant table.
- **Scalability:** Works in high-dimensional spaces (e.g., images, sensor data).

## Challenges

1. **Approximation Error:** The function may not perfectly fit the true  $V(s)$ .
2. **Stability Issues:** Unlike tabular methods, convergence isn't guaranteed (needs careful tuning).
3. **Feature Dependency:** Linear methods require good features; deep methods need proper architecture.

## Summary

Instead of storing a table of values, we:

1. Use a **parameterized function** (linear model, neural net, etc.) to approximate  $V(s)$  or  $Q(s, a)$ .
2. **Learn the parameters  $\theta$**  using TD learning, gradient descent, etc.
3. **Generalize** to unseen states, making RL feasible in complex environments.

## Mean Squared Value Error (MSVE) in Reinforcement Learning

The Mean Squared Value Error (MSVE) is a key metric used to evaluate how well a parameterized value function  $V_\theta(s)$  approximates the true value function  $V^\pi(s)$  under a policy  $\pi$ . It measures the average squared difference between the estimated values and the true values across all states.

---

### 1. Definition of MSVE

The MSVE is defined as:

$$\text{MSVE}(\theta) = \mathbb{E}_{s \sim d^\pi(s)} \left[ (V^\pi(s) - V_\theta(s))^2 \right]$$

where:

- $V^\pi(s)$  = True value function (expected return under policy  $\pi$ ).
- $V_\theta(s)$  = Estimated value function (parameterized by  $\theta$ ).
- $d^\pi(s)$  = Stationary state distribution under  $\pi$  (how often each state is visited).



## 2. Interpretation

- **Goal:** Minimize MSVE to make  $V_\theta(s)$  as close as possible to  $V^\pi(s)$ .
  - **If MSVE = 0:** Perfect approximation.
  - **If MSVE is high:** Poor approximation (large errors in value predictions).
- 

## 3. Why Use MSVE?

- Provides a **quantitative measure** of how good the approximation is.
  - Used as an **optimization objective** in algorithms like TD learning and **gradient-based value function approximation**.
  - Helps in **comparing different function approximators** (e.g., linear vs. neural networks).
- 

## 4. Connection to TD Learning

In Temporal Difference (TD) learning, we update  $\theta$  to reduce the **TD error**:

$$\delta = r + \gamma V_\theta(s') - V_\theta(s)$$

The **MSVE** is indirectly minimized by reducing the expected squared TD error:



$$\mathbb{E}[\delta^2] \propto \text{MSVE}$$

Thus, TD methods like  $\text{TD}(\lambda)$  and Gradient TD aim to minimize MSVE.

---

## 5. Gradient Descent for Minimizing MSVE

To optimize  $\theta$ , we can use **stochastic gradient descent (SGD)** on the MSVE:

$$\nabla_{\theta} \text{MSVE}(\theta) = -2\mathbb{E}_{s \sim d^{\pi}(s)} [(V^{\pi}(s) - V_{\theta}(s)) \nabla_{\theta} V_{\theta}(s)]$$

In practice, we use **sampled updates** (e.g., Monte Carlo or TD updates):

$$\theta \leftarrow \theta + \alpha (V^{\pi}(s) - V_{\theta}(s)) \nabla_{\theta} V_{\theta}(s)$$

(Since we don't know  $V^{\pi}(s)$ , we replace it with **bootstrapped targets** like  $r + \gamma V_{\theta}(s')$ .)

---

## 6. Challenges with MSVE

### 1. Bias-Variance Tradeoff:

- High-capacity models (e.g., deep neural networks) can overfit.
- Low-capacity models (e.g., linear approximation) may underfit.



## 2. Non-Stationary Targets:

- In TD learning, the target  $r + \gamma V_\theta(s')$  depends on  $\theta$ , making optimization unstable.

## 3. Distribution Shift:

- The state distribution  $d^\pi(s)$  changes as the policy improves, affecting MSVE.
- 

## 7. Alternatives to Pure MSVE

Since MSVE can be hard to optimize directly, some methods use:

- **Projected Bellman Error (PBE):** Projects TD errors onto the function space.
  - **Bellman Residual Minimization:** Minimizes  $\|V_\theta - \mathcal{T}^\pi V_\theta\|$ , where  $\mathcal{T}^\pi$  is the Bellman operator.
- 

## 8. Example: Linear Approximation MSVE

Suppose:

- $V_\theta(s) = \theta^T \phi(s)$  (linear function approximation).
- The MSVE becomes:



- The MSVE becomes:

$$\text{MSVE}(\theta) = \mathbb{E}_{s \sim d^\pi(s)} \left[ (V^\pi(s) - \theta^T \phi(s))^2 \right]$$

The gradient is:

$$\nabla_\theta \text{MSVE} = -2\mathbb{E} [\phi(s) (V^\pi(s) - \theta^T \phi(s))]$$

We can estimate this using sampled states and TD targets.

---

## 9. Practical Implications

- Deep RL (DQN, Actor-Critic): Uses variants of MSVE (e.g., Huber loss, MSE on Q-values).
  - Policy Evaluation: MSVE measures how well we estimate values before improving the policy.
  - Convergence: For linear approximators, MSVE converges under certain conditions (see Tsitsiklis & Van Roy, 1997).
- 

## 10. Summary

- MSVE measures the average squared error between true and estimated value functions.
- Minimizing MSVE improves value prediction accuracy.
- Used in TD learning, gradient-based methods, and deep RL.
- Challenges include non-stationarity and approximation errors.

The Gradient Monte Carlo (GMC) algorithm is a method for estimating the value function in reinforcement learning (RL) using gradient descent and Monte Carlo (MC) sampling. It is particularly useful for **policy evaluation** in problems with large or continuous state spaces where tabular methods are impractical.

---

## Key Concepts

### 1. Monte Carlo (MC):

- Learns from complete episodes of experience (e.g., until termination).
- Uses empirical returns (observed cumulative rewards) for value estimation.

### 2. Gradient Descent:

- Adjusts parameters (weights) of a function approximator (e.g., linear function, neural network) to minimize the error between predicted and actual returns.

### 3. Function Approximation:

- Instead of storing a value for each state, the algorithm represents the value function as:

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$

where  $\mathbf{w}$  are the weights (parameters) of the approximator.

## Gradient Monte Carlo Algorithm

Objective:

Minimize the **mean squared error (MSE)** between the true value  $v_\pi(s)$  and its approximation  $\hat{v}(s, \mathbf{w})$ :

$$J(\mathbf{w}) = \mathbb{E}_\pi [(v_\pi(S) - \hat{v}(S, \mathbf{w}))^2]$$

Update Rule (Stochastic Gradient Descent):

For each state  $S_t$  in an episode, update the weights  $\mathbf{w}$  using:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha [G_t - \hat{v}(S_t, \mathbf{w}_t)] \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}_t)$$

where:

- $G_t$  = actual return observed after  $S_t$  (Monte Carlo target).
- $\alpha$  = step size (learning rate).
- $\nabla_{\mathbf{w}} \hat{v}$  = gradient of the value function w.r.t.  $\mathbf{w}$ .

## Steps of the Algorithm

### 1. Initialize:

- Weights  $\mathbf{w}$  (e.g., randomly or zero).
- Learning rate  $\alpha$ .

### 2. Generate Episodes:

- Follow policy  $\pi$  to produce trajectories  $(S_0, A_0, R_1, \dots, S_T)$ .

### 3. For Each Episode:

- Compute  $G_t$  (return from time  $t$  onward) for each state  $S_t$ .
- Update  $\mathbf{w}$  using the gradient descent rule above.

### 4. Repeat:

- Until convergence (weights stabilize).

---

## Important Notes

- **Bias-Variance Tradeoff:** MC returns  $G_t$  are unbiased but high-variance.
- **Function Approximation Choice:**
  - Linear:  $\hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s)$ , where  $\mathbf{x}(s)$  is a feature vector.
  - Nonlinear: Neural networks can also be used.
- **On-Policy:** Evaluates the policy  $\pi$  used to generate episodes.



## Comparison with TD Methods

Feature	Gradient MC	Temporal Difference (TD)
Target	$G_t$ (full return)	$R_{t+1} + \gamma \hat{v}(S_{t+1})$
Variance	High	Lower
Bias	Unbiased	Biased (bootstrapping)
Convergence	Converges to local optimum	May diverge with approximation

## Example (Linear Function Approximation)

If  $\hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s)$ , the update becomes:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha [G_t - \mathbf{w}_t^T \mathbf{x}(S_t)] \mathbf{x}(S_t)$$

This is similar to the Least Mean Square (LMS) update.

## Applications

- Large/continuous state spaces (e.g., robotics, game AI).
- Policy evaluation in policy gradient methods.



Yes! In the **Gradient Monte Carlo (GMC)** algorithm,  $G_t$  represents the **total discounted return** from time  $t$  onward, following the policy  $\pi$ . It is defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$$

where:

- $R_{t+k}$  = reward received at time  $t + k$ ,
  - $\gamma$  = discount factor ( $0 \leq \gamma \leq 1$ ),
  - $T$  = terminal time step of the episode.
- 

## Key Points About $G_t$ in GMC

### 1. Monte Carlo Nature:

- $G_t$  is the **actual observed return** from state  $S_t$  until the end of the episode (unlike TD methods, which bootstrap).
- Because it relies on full trajectories, it is **unbiased** but has **high variance**.

### 2. Discounting:

- If  $\gamma = 1$ ,  $G_t$  is the **undiscounted return** (sum of all future rewards).
- If  $\gamma < 1$ , later rewards are weighted less (avoids infinite returns in infinite-horizon problems). 

### 3. Use in Gradient Update:

- The term  $G_t - \hat{v}(S_t, \mathbf{w})$  acts as the **error signal** (similar to supervised learning).
  - The goal is to adjust  $\mathbf{w}$  so that  $\hat{v}(S_t, \mathbf{w})$  better predicts  $G_t$ .
- 

### Comparison of $G_t$ in MC vs. TD

Feature	Monte Carlo (GMC)	Temporal Difference (TD)
Target	$G_t$ (full return)	$R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$
Bias	Unbiased	Biased (due to bootstrapping)
Variance	High (depends on entire trajectory)	Lower (uses single-step rewards)

---

### Example

Consider an episode with rewards  $[R_1, R_2, R_3] = [1, -2, 3]$  and  $\gamma = 0.9$ :

- $G_0 = 1 + 0.9 \times (-2) + 0.9^2 \times 3 = 1 - 1.8 + 2.43 = 1.63$ ,
- $G_1 = -2 + 0.9 \times 3 = -2 + 2.7 = 0.7$ ,



- $G_2 = 3$ .

The GMC update for  $S_1$  would use  $G_1 = 0.7$  as the target.

---

## When to Use Discounting

- Set  $\gamma < 1$  for:
    - Infinite-horizon problems (to ensure finite returns),
    - Preferring earlier rewards (common in RL).
  - Set  $\gamma = 1$  for:
    - Episodic tasks where the exact sum of rewards matters (e.g., board games).
- 

## Pseudocode Recap

For each episode:

1. Compute  $G_t$  for all  $t$  using the full trajectory.
2. For each  $S_t$ , update:

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



The **Semi-Gradient TD(0)** algorithm is a temporal difference (TD) learning method that combines **bootstrapping** (like TD) with **function approximation** (e.g., linear or neural networks). Unlike Monte Carlo methods (which wait until the end of an episode to update), TD(0) updates estimates after each step, making it more efficient for online learning.

However, it is called "semi-gradient" because it ignores the dependency of the target (which includes the current weight vector  $\mathbf{w}$ ) on the weights, leading to a biased but stable update.

---

## Semi-Gradient TD(0) Algorithm

### Objective

Minimize the error between the current value estimate  $\hat{v}(S_t, \mathbf{w})$  and the **TD target**:

$$\text{TD Target} = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$$

Unlike Monte Carlo, this target **bootstraps** from the current estimate  $\hat{v}(S_{t+1}, \mathbf{w})$ .

## Update Rule

For each transition  $(S_t, R_{t+1}, S_{t+1})$ :

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha [R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)] \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}_t)$$

where:

- $\alpha$  = learning rate,
  - $\gamma$  = discount factor,
  - $\nabla_{\mathbf{w}} \hat{v}$  = gradient of the value function w.r.t.  $\mathbf{w}$ .
- 

## Why "Semi-Gradient"?

- The **true gradient** should consider that the TD target  $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$  also depends on  $\mathbf{w}$ .
- However, in **semi-gradient TD**, we **ignore this dependency** and treat the target as a fixed value (even though it changes with  $\mathbf{w}$ ).
- This makes the update **not a true gradient descent**, but it often works well in practice and avoids divergence.

## Comparison: Semi-Gradient TD(0) vs. Gradient Monte Carlo

Feature	Semi-Gradient TD(0)	Gradient Monte Carlo (GMC)
Update Target	$R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$ (bootstrapped)	$G_t$ (full return)
Bias	Biased (due to bootstrapping)	Unbiased
Variance	Lower (single-step updates)	Higher (depends on full trajectory)
Convergence	Stable under function approximation	Converges to a local optimum
Online Learning	Yes (updates after each step)	No (requires full episodes)

## Pseudocode for Semi-Gradient TD(0)

### 1. Initialize:

- Weight vector  $\mathbf{w}$  (e.g., randomly or zeros).
- Learning rate  $\alpha$ , discount factor  $\gamma$ .

### 2. Loop for each episode:

- Initialize state  $S$ .
- Repeat until episode ends:
  - Take action  $A$  (using policy  $\pi$ ), observe  $R, S'$ .
  - Compute TD error:

$$\delta = R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$$

- Update weights:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \cdot \delta \cdot \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$$

- $S \leftarrow S'$ .

## Example: Linear Function Approximation

If  $\hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s)$ , then:

$$\nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$$

The update simplifies to:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R + \gamma \mathbf{w}^T \mathbf{x}(S') - \mathbf{w}^T \mathbf{x}(S)] \mathbf{x}(S)$$

---

## Key Takeaways

1. **Semi-Gradient TD(0) is more efficient** than Monte Carlo for online learning (updates after each step).
2. **It is biased** because it ignores the effect of  $\mathbf{w}$  on the target.
3. **It is stable** under linear function approximation (unlike true gradient TD, which can diverge).
4. Used in algorithms like SARSA and Q-learning with function approximation.

# Simple Explanation: Why is Semi-Gradient TD(0) "Semi-Gradient" and Biased?

## 1. What "Semi-Gradient" Means

- In **true gradient descent**, we compute the gradient of the **entloss function**, including **how the target affects the weights**.
- But in **TD(0)**, the target is:

$$\text{Target} = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$$

- This target **depends on the current weights  $\mathbf{w}$**  (because  $\hat{v}(S_{t+1}, \mathbf{w})$  uses  $\mathbf{w}$ ).
- **Semi-gradient ignores this dependency** and treats the target as a fixed value (even though it changes as  $\mathbf{w}$  changes).
- Thus, we only take the gradient of  $\hat{v}(S_t, \mathbf{w})$ , **not the full loss**, making it "semi" gradient.

## 2. Why This Causes Bias

- Since we **pretend the target is fixed**, the update does **not follow the true gradient** of the loss function.
- This means:
  - The algorithm **does not truly minimize the error** in the best possible way.
  - The solution can be **biased** (systematically wrong) compared to Monte Carlo (which uses the true return  $G_t$ ).



## Example: Why the Target Depends on $\mathbf{w}$

Let's say:

- Current state value estimate:  $\hat{v}(S_t, \mathbf{w}) = 1.0$
- Next state value estimate:  $\hat{v}(S_{t+1}, \mathbf{w}) = 2.0$
- Observed reward:  $R_{t+1} = 1.0$
- Discount factor:  $\gamma = 0.9$

The TD target is:

$$\text{Target} = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) = 1.0 + 0.9 \times 2.0 = 2.8$$

But if we update  $\mathbf{w}$ ,  $\hat{v}(S_{t+1}, \mathbf{w})$  also changes!

Semi-gradient ignores this, treating the target as fixed at 2.8, even though it should adjust as  $\mathbf{w}$  changes.

Great question! Let's break it down **exactly** where TD(0) "ignores" the dependency on  $\mathbf{w}$  in the gradient calculation.

---

## Full Gradient vs. Semi-Gradient TD(0)

We want to minimize the **squared TD error**:

$$J(\mathbf{w}) = \frac{1}{2} \left[ \underbrace{R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})}_{\text{TD target}} - \hat{v}(S_t, \mathbf{w}) \right]^2$$

### 1. True Gradient (If We Account for Everything)

The **full gradient** of  $J(\mathbf{w})$  w.r.t.  $\mathbf{w}$  requires differentiating **both**:

- $\hat{v}(S_t, \mathbf{w})$  (current estimate),
- **and**  $\hat{v}(S_{t+1}, \mathbf{w})$  (inside the TD target).

Using the chain rule:

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = [\hat{v}(S_t, \mathbf{w}) - (R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}))] \cdot [\nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) - \gamma \nabla_{\mathbf{w}} \hat{v}(S_{t+1}, \mathbf{w})]$$



## 2. Semi-Gradient TD(0) (What We Actually Do)

Instead, TD(0) ignores the  $\nabla_{\mathbf{w}} \hat{v}(S_{t+1}, \mathbf{w})$  term, pretending the TD target is a fixed label (like in supervised learning).

This simplifies the update to:

$$\nabla_{\mathbf{w}} J(\mathbf{w}) \approx [\hat{v}(S_t, \mathbf{w}) - (R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}))] \cdot \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w})$$

---

### Where Exactly Does TD(0) Ignore $\mathbf{w}$ ?

In the TD target:

$$\text{Target} = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$$

- True gradient would differentiate  $\hat{v}(S_{t+1}, \mathbf{w}) \rightarrow$  adds a  $-\gamma \nabla_{\mathbf{w}} \hat{v}(S_{t+1}, \mathbf{w})$  term.
- Semi-gradient ignores this and treats the target as a constant, so:

$$\nabla_{\mathbf{w}} \text{Target} \approx 0 \quad (\text{in reality, it's } \gamma \nabla_{\mathbf{w}} \hat{v}(S_{t+1}, \mathbf{w}))$$

---

## Why This Matters

- Ignoring  $\nabla_{\mathbf{w}} \hat{v}(S_{t+1}, \mathbf{w})$  introduces bias:  
The update doesn't follow the **true direction** of steepest descent.
  - But it's more stable:  
Including the full gradient can cause divergence with function approximation (see *Tsitsiklis & Van Roy, 1997*).
- 

## Visualization

Term	Full Gradient	Semi-Gradient (TD(0))
TD Target Gradient	$\gamma \nabla_{\mathbf{w}} \hat{v}(S_{t+1}, \mathbf{w})$	0 (ignored)
Update Direction	More accurate, but risky	Stable, but biased

## Example (Linear Case)

If  $\hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s)$ :

- Full gradient:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R + \gamma \mathbf{w}^T \mathbf{x}(S') - \mathbf{w}^T \mathbf{x}(S)] [\mathbf{x}(S) - \gamma \mathbf{x}(S')]$$

- Semi-gradient (TD(0)):

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R + \gamma \mathbf{w}^T \mathbf{x}(S') - \mathbf{w}^T \mathbf{x}(S)] \mathbf{x}(S)$$

(The  $\gamma \mathbf{x}(S')$  term is dropped!)

---

## Key Takeaways

1. TD(0) ignores  $\nabla_{\mathbf{w}} \hat{v}(S_{t+1}, \mathbf{w})$  in the target, treating it as constant.
2. This makes it "semi"-gradient (only part of the true gradient is used).
3. Result: Biased but stable updates.

In Episodic SARSA with Function Approximation, the input consists of several key components that define the learning process. Here's a breakdown of the required inputs:

## Inputs for Episodic SARSA with Function Approximation:

### 1. Environment (MDP):

- A Markov Decision Process defined by states ( $\mathcal{S}$ ), actions ( $\mathcal{A}$ ), rewards ( $\mathcal{R}$ ), transition dynamics, and discount factor ( $\gamma$ ).

### 2. Function Approximator (e.g., a linear function or neural network):

- A parameterized weight vector  $\mathbf{w}$  (or  $\theta$ ) that approximates the action-value function:

$$\hat{Q}(s, a, \mathbf{w}) \approx Q(s, a)$$

- A feature representation  $\mathbf{x}(s, a)$  for state-action pairs (e.g., tile coding, neural network embeddings).

### 3. Learning Rate ( $\alpha$ ):

- A step-size parameter controlling how much the weights are updated per step.

### 4. Exploration Strategy (e.g., $\epsilon$ -greedy):

- A policy derived from  $\hat{Q}$  that balances exploration and exploitation.

### 5. Discount Factor ( $\gamma$ ):

- Determines the importance of future rewards (typically  $0 \leq \gamma < 1$ ).

## 6. Number of Episodes ( $N$ ):

- The total number of training episodes to run.

## 7. Initial Parameters ( $w_0$ ):

- Initial weights for the function approximator (often initialized randomly or to zero).

## Pseudocode Summary:

text

 Copy  Download

### Input:

- Environment ( $S, A, R, P, \gamma$ )
- Function approximator  $Q^*(s, a, w)$  with feature extractor  $x(s, a)$
- Learning rate  $\alpha$
- Exploration rate  $\epsilon$
- Discount factor  $\gamma$
- Number of episodes  $N$

Initialize  $w$  arbitrarily (e.g.,  $w = \theta$ )

For each episode:

    Initialize state  $s$

    Choose action  $a$  using  $\epsilon$ -greedy policy from  $Q^*(s, \cdot, w)$

    Repeat until episode terminates:

        Take action  $a$ , observe reward  $r$  and next state  $s'$

        Choose next action  $a'$  using  $\epsilon$ -greedy policy from  $Q^*(s', \cdot, w)$

        Compute TD error:  $\delta = r + \gamma Q^*(s', a', w) - Q^*(s, a, w)$

        Update weights:  $w \leftarrow w + \alpha \cdot \delta \cdot \nabla_w Q^*(s, a, w)$

$s \leftarrow s'$ ,  $a \leftarrow a'$

## Key Notes:

- Instead of a Q-table, the algorithm uses **function approximation** (linear, neural networks, etc.) to generalize across states.
- The **gradient**  $\nabla_w \hat{Q}(s, a, w)$  is required for weight updates (e.g., for linear approximation,  $\nabla_w \hat{Q} = \mathbf{x}(s, a)$ ).
- The algorithm is **on-policy** (updates are based on the current  $\epsilon$ -greedy policy).

In Differential SARSA (an extension of SARSA for average reward settings), the **average reward** ( $\bar{R}$ ) is used to adapt the algorithm to **non-discounted, continuing tasks** where the goal is to maximize the **long-term average reward per step** rather than the cumulative discounted reward.

---

## Key Idea of Differential SARSA

Unlike discounted SARSA (which uses  $\gamma$  to weigh future rewards), Differential SARSA:

1. Estimates the **average reward per time step** ( $\bar{R}$ ).
2. Uses it to compute TD errors without discounting ( $\gamma = 1$ ).
3. Adjusts Q-values based on how much an action performs *relative* to the average.

This is useful in **continuing tasks** (e.g., robotics, real-time control) where episodes don't terminate naturally.

## How Average Reward ( $\bar{R}$ ) is Used

### 1. TD Error in Differential SARSA:

$$\delta_t = R_{t+1} - \bar{R}_t + Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$$

- $R_{t+1} - \bar{R}_t$ : Immediate reward *relative* to the current average.
- No discounting ( $\gamma = 1$ ) because we care about **long-term average performance**.

### 2. Updating the Average Reward:

$$\bar{R}_{t+1} = \bar{R}_t + \beta \delta_t$$

- $\beta$ : Step-size parameter for the average reward (separate from  $\alpha$  for Q-values).

### 3. Updating Q-values (SARSA-style):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \delta_t$$

## Inputs for Differential SARSA

### 1. Step-sizes:

- $\alpha$  (for Q-values).
- $\beta$  (for  $\bar{R}$ , typically smaller than  $\alpha$ ).

### 2. No discounting ( $\gamma = 1$ ).

### 3. Exploration strategy (e.g., $\varepsilon$ -greedy).

---

## Why Use Differential SARSA?

- Discounted formulations can fail in continuing tasks (e.g.,  $\gamma$ -based returns grow infinitely).
- Optimizes the *rate of reward* (e.g., "maximize average throughput in a network").

Initialize:

$Q(s, a)$  arbitrarily  
 $R^- = 0$   
 $\alpha, \beta$  = small step sizes

Loop for each step:

Take action  $A$ , observe  $R, S'$   
Choose  $A'$  from  $S'$  using  $\epsilon$ -greedy( $Q$ )  
 $\delta = R - R^- + Q(S', A') - Q(S, A)$   
 $R^- \leftarrow R^- + \beta\delta$   
 $Q(S, A) \leftarrow Q(S, A) + \alpha\delta$   
 $S \leftarrow S', A \leftarrow A'$

## Comparison with Discounted SARSA

Feature	Discounted SARSA	Differential SARSA
Objective	Maximize $\sum \gamma^t R_t$	Maximize $\lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=1}^T R_t$
TD Error	$R + \gamma Q(S', A') - Q(S, A)$	$R - \bar{R} + Q(S', A') - Q(S, A)$
Use Case	Episodic tasks	Continuing tasks



**Policy Parameterization** refers to directly modeling the policy (the agent's strategy) using a parameterized function, rather than deriving it indirectly (e.g., via value functions like in Q-learning). This is a core idea in **Policy Gradient Methods** in reinforcement learning (RL).

## Key Concepts:

### 1. Policy ( $\pi$ ):

- A policy defines the agent's behavior:  $\pi(a|s)$  is the probability of taking action  $a$  in state  $s$ .
- In **policy parameterization**, we represent  $\pi$  as a function (e.g., a neural network) with tunable parameters  $\theta$ :  $\pi_\theta(a|s)$ .

### 2. Direct Parameterization:

- Instead of learning a value function (like  $V(s)$  or  $Q(s, a)$ ) and then deriving a policy from it (e.g.,  $\epsilon$ -greedy), we directly learn  $\pi_\theta(a|s)$ .
- The parameters  $\theta$  are updated to maximize expected reward.

### 3. Advantages:

- Can handle **stochastic policies** naturally (useful in partially observable or adversarial environments).
- Better for **high-dimensional or continuous action spaces** (unlike value-based methods that require argmax over actions).
- More stable learning in some cases (avoids the "greedification" problem of value-based methods).

#### 4. Disadvantages:

- Typically higher variance in gradient estimates (though techniques like advantage estimation help).
- May converge to local optima.

### Common Policy Parameterizations:

#### 1. Softmax Policy (Discrete Actions):

- For discrete actions, the policy is often a softmax over action scores:

$$\pi_{\theta}(a|s) = \frac{e^{f_{\theta}(s,a)}}{\sum_{a'} e^{f_{\theta}(s,a')}}$$

- Here,  $f_{\theta}(s, a)$  is a learned function (e.g., a neural network).

#### 2. Gaussian Policy (Continuous Actions):

- For continuous actions, the policy can output a mean ( $\mu_{\theta}(s)$ ) and variance ( $\sigma_{\theta}(s)$ ) for a Gaussian distribution:

$$\pi_{\theta}(a|s) = \mathcal{N}(a|\mu_{\theta}(s), \sigma_{\theta}(s))$$

- The agent samples actions from this distribution during training.

#### 3. Deterministic Policy (e.g., DDPG):

- In some cases, the policy is deterministic:  $a = \mu_{\theta}(s)$ .
- Used in Deep Deterministic Policy Gradient (DDPG).



## Policy Gradient Theorem:

The fundamental result for policy gradients is:

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

where  $J(\theta)$  is the expected return, and  $Q^{\pi_{\theta}}(s, a)$  is the action-value function.

## Algorithms Using Policy Parameterization:

- REINFORCE: Vanilla policy gradient.
- PPO (Proximal Policy Optimization): Clips gradients to avoid large updates.
- A3C (Advantage Actor-Critic): Combines policy gradients with value function approximation.
- DDPG: For continuous action spaces (uses a deterministic policy).

how it is different from function approximation there we also use neural network to store state value/action value???

Great question! The key difference lies in **what the neural network represents** and **how it is used** in reinforcement learning (RL). Let's break it down:

---

## 1. Function Approximation (Value-Based Methods)

- **What it does:**

The neural network approximates a **value function** (either  $V(s)$  or  $Q(s, a)$ ).

- Example: In DQN, the network outputs  $Q(s, a)$  for each possible action.

- **Goal:**

Learn to predict the expected return (cumulative reward) for a state or state-action pair.

- **Policy Derivation:**

The policy is **indirectly derived** from the value function (e.g.,  $\epsilon$ -greedy policy over  $Q(s, a)$ ).

- **Update Rule:**

Minimize the Bellman error (e.g., MSE between predicted  $Q(s, a)$  and target  $Q(s, a)$ ).



## 2. Policy Parameterization (Policy-Based Methods)

- **What it does:**

The neural network directly represents the **policy**  $\pi_\theta(a|s)$ , outputting action probabilities (or parameters of a distribution, like mean/variance for continuous actions).

- **Goal:**

Optimize the policy parameters  $\theta$  to **maximize expected reward** (via gradient ascent).

- **Policy Derivation:**

The policy is **explicitly defined** by the network (no need for argmax over Q-values).

- **Update Rule:**

Follow the policy gradient (e.g., REINFORCE, PPO).

## Key Differences

Feature	Function Approximation (Value-Based)	Policy Parameterization (Policy-Based)
Network Output	$V(s)$ or $Q(s, a)$	$\pi_\theta(a)$
Policy Derivation	Indirect (e.g., $\epsilon$ -greedy)	Direct (sampling from $\pi_\theta$ )
Optimization	Minimize Bellman error (MSE)	Maximize expected reward (gradient)
Handles Stochastic Policies	No (unless combined with policy gradients)	Yes (natively supports stochastic policies)
Action Space	Discrete (hard for continuous)	Discrete or continuous (flexible)
Examples	DQN, SARSA	REINFORCE, PPO, SAC

## Why Policy Parameterization?

### 1. Continuous Action Spaces:

- Value-based methods struggle because they require  $\arg \max_a Q(s, a)$ , which is intractable in continuous spaces.
- Policy networks can output parameters of a distribution (e.g., Gaussian) for sampling.

### 2. Stochastic Policies:

- Useful in partially observable environments (e.g., adversarial games).
- Value-based methods typically learn deterministic policies (unless explicitly modified).

### 3. Avoids Greedy Exploitation:

- Q-learning's  $\arg \max$  can lead to brittle policies (overfitting to noisy Q-values).
- Policy gradients optimize for long-term reward directly.

---

## Hybrid Approaches (Actor-Critic)

Many modern algorithms (e.g., A3C, PPO, SAC) combine both ideas:

- **Actor (Policy Network):** Directly parameterizes  $\pi_\theta(a|s)$ .
- **Critic (Value Network):** Approximates  $V(s)$  or  $Q(s, a)$  to reduce variance in policy updates.



## Summary

- **Function Approximation** learns "how good" states/actions are (value functions).
- **Policy Parameterization** learns "what to do" directly (policy).
- **Trade-offs:** Value-based methods are more sample-efficient but less flexible; policy-based methods handle complex action spaces but have higher variance.

Yes! Unlike  $\epsilon$ -greedy (which explores randomly with fixed probability  $\epsilon$ ), softmax policy parameterization adjusts action probabilities based on learned scores, providing **more nuanced exploration**. It balances exploration/exploitation by favoring promising actions while still trying others, making it more adaptive than  $\epsilon$ -greedy's brute-force randomness.

#### Key Difference:

- $\epsilon$ -greedy: Explores *uniformly* (all non-optimal actions get equal chance).
- Softmax: Explores *adaptively* (better actions get higher probabilities, but suboptimal ones are still sampled occasionally).

Example: In a maze,  $\epsilon$ -greedy might randomly bump into walls, while softmax would prioritize directions that seem promising but still explore others.

The **Policy Gradient Theorem** is a foundational result in reinforcement learning that provides a way to directly optimize a parameterized policy (e.g., a neural network) by calculating the gradient of expected reward with respect to the policy parameters. Here's the core idea:

### Policy Gradient Theorem Formula:

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

Where:

- $J(\theta)$  = Expected return (total reward) under policy  $\pi_{\theta}$ .
- $\pi_{\theta}(a|s)$  = Probability of taking action  $a$  in state  $s$  under policy  $\theta$ .
- $Q^{\pi_{\theta}}(s, a)$  = Expected return after taking action  $a$  in state  $s$ .
- $\nabla_{\theta} \log \pi_{\theta}(a|s)$  = "Score function" (how to adjust  $\theta$  to increase  $\pi_{\theta}(a|s)$ ).

---

### Key Intuitions:

1. **Goal:** Maximize  $J(\theta)$  by ascending the gradient  $\nabla_{\theta} J(\theta)$ .
2. **No Need for Value Function?**
  - The theorem uses  $Q^{\pi_{\theta}}(s, a)$ , but in practice, we often approximate it (e.g., with Monte Carlo returns or a critic network).



3. **Local Improvement:** The gradient points in the direction that increases the probability of **high-reward actions**.
- 

### Why It Matters:

- Enables **policy optimization** without explicitly modeling environment dynamics.
  - Works for **both discrete and continuous action spaces**.
  - Forms the basis for algorithms like **REINFORCE**, **A3C**, and **PPO**.
- 

### Example (REINFORCE Algorithm):

1. Sample a trajectory under  $\pi_\theta$ .
2. For each step  $(s, a, r)$ , compute:

$$\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a|s) \cdot G_t$$

(where  $G_t$  is the empirical return from that step).

3. Update  $\theta$  via gradient ascent.



## How to Say the Policy Gradient Formula:

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

### Verbal Breakdown:

1. "Gradient of  $J$  with respect to  $\theta$ " ( $\nabla_{\theta} J(\theta)$ )  
→ "How to tweak policy parameters ( $\theta$ ) to increase total reward ( $J$ )"
2. "Equals the expectation under policy  $\pi_{\theta}$ " ( $\mathbb{E}_{\pi_{\theta}}[\dots]$ )  
→ "Average over all actions/states encountered by the current policy"
3. "Gradient of  $\log \pi_{\theta}$  of a given  $s$ " ( $\nabla_{\theta} \log \pi_{\theta}(a|s)$ )  
→ "Direction to increase the probability of action  $a$  in state  $s$ "
4. "Times  $Q$ - $\pi_{\theta}$  of  $s$  comma  $a$ " ( $\cdot Q^{\pi_{\theta}}(s, a)$ )  
→ "Weighted by how good action  $a$  is in state  $s$ "

### Shortcut Phrase:

"The policy gradient equals the average of [log-policy-gradient times Q-value] over all experiences."

# Actor-Critic Algorithm in Reinforcement Learning

The Actor-Critic algorithm is a hybrid approach that combines value-based and policy-based methods in reinforcement learning. Here's a complete explanation of the algorithm:

## Core Components

1. **Actor:** The policy function  $\pi(a|s;\theta)$  that selects actions (policy-based)
2. **Critic:** The value function  $V(s;w)$  that evaluates states (value-based)

## Algorithm Steps

text

 Copy  Download

### 1. Initialize:

- Actor parameters  $\theta$  (policy)
- Critic parameters  $w$  (value function)
- Learning rates  $\alpha_\theta$ ,  $\alpha_w$
- Discount factor  $\gamma$
- Empty replay buffer (if using experience replay)

2. For each episode:
  3. Initialize state  $s$
  4. While episode not terminated:
    5. Select action  $a \sim \pi(a|s;\theta)$  (with exploration noise if needed)
    6. Execute  $a$ , observe reward  $r$  and next state  $s'$
    7. Compute TD error:
$$\delta = r + \gamma V(s';w) - V(s;w)$$
    8. Update Critic (value function):
$$w \leftarrow w + \alpha_w * \delta * \nabla_w V(s;w)$$
    9. Update Actor (policy):
$$\theta \leftarrow \theta + \alpha_\theta * \delta * \nabla_\theta \log \pi(a|s;\theta)$$
  10.  $s \leftarrow s'$

## Variants and Enhancements

1. Advantage Actor-Critic (A2C):
  - Uses advantage function  $A(s,a) = Q(s,a) - V(s)$  instead of TD error
  - More stable as it reduces variance
2. Asynchronous Advantage Actor-Critic (A3C):
  - Multiple parallel actors that update a global network
  - Improves exploration and training speed
3. Generalized Advantage Estimation (GAE):
  - Balances bias and variance in advantage estimation
  - Uses an exponential average of k-step advantage estimates
4. Soft Actor-Critic (SAC):
  - Incorporates entropy regularization
  - Off-policy variant with temperature parameter



# Practical Implementation Considerations

## 1. Network Architectures:

- Can use separate networks for actor and critic
- Or shared lower layers with separate output heads

## 2. Exploration:

- Typically uses the stochastic policy's inherent exploration
- Can add entropy regularization to encourage exploration

## 3. Hyperparameter Tuning:

- Learning rates for actor and critic need careful balancing
- Discount factor  $\gamma$  affects credit assignment

## 4. Stability Techniques:

- Gradient clipping
- Target networks (like in DQN)
- Experience replay (though more common in pure value methods)

The Actor-Critic framework is widely used in modern RL because it combines the best of both policy gradient methods (direct policy optimization) and value function methods (lower variance updates).

## Actor-Critic in Simple Words

The Actor-Critic algorithm is a Reinforcement Learning (RL) method that combines two key ideas:

1. **Actor (The Decision-Maker)** → Learns the **best policy** (what action to take).
2. **Critic (The Evaluator)** → Judges how good the action was and gives **feedback**.

## How It Works?

- The **Actor** (policy) picks an action based on the current state.
- The **Critic** (value function) checks if the action was good or bad by estimating future rewards.
- The **Critic's feedback (TD Error)** helps the **Actor** improve its decisions.

## Simple Analogy

Think of a **student (Actor)** and a **teacher (Critic)**:

- The **student** tries solving problems (takes actions).
- The **teacher** checks the answers and gives corrections (feedback).
- The **student** improves based on the teacher's feedback.

## Why Use Actor-Critic?

- Faster learning** than pure policy methods (like REINFORCE).
- More stable** than pure value-based methods (like Q-Learning).