1 Intro

2 Imitation learning

$\mathbf{s}_t$	state
$\mathbf{o}_t$	observation
$\mathbf{a}_t$	action
$\pi_\theta(\mathbf{a}_t \mid \mathbf{o}_t)$	policy
$\pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t)$	policy (fully observed)

In regular supervised learning, we have the i.i.d assumption, i.e. training points does not affect each other. However, we don't have such assumption in RL, taking actions affects next observations.

3 Policy Gradient Methods

3.1 Motivation

Value-based methods (e.g., Q-learning) estimate action values and then derive the policy indirectly by taking the action with the highest value. In contrast, **policy gradient methods** directly optimize the parameters of a policy $\pi_\theta(a|s)$ with respect to expected return. They are especially suitable for:

- Continuous or high-dimensional action spaces
- Stochastic policies (exploration via inherent randomness)

—

3.2 Problem Setup

Let the agent follow a parameterized policy $\pi_\theta(a|s)$. The goal is to maximize the expected discounted return:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)]$$

where a trajectory $\tau = (s_0, a_0, s_1, a_1, \ldots)$ is generated according to:

$$p(\tau|\theta) = \rho_0(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t|s_t) P(s_{t+1}|s_t, a_t),$$

and the cumulative discounted reward is

$$R(\tau) = \sum_{t=0}^{T-1} \gamma^t r(s_t, a_t).$$

—

3.3 The Policy Gradient Theorem

We aim to compute the gradient of $J(\theta)$ with respect to θ:

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)].$$

Expanding the expectation:

$$\nabla_\theta J(\theta) = \int R(\tau)\nabla_\theta p(\tau|\theta)d\tau.$$

Applying the **log-derivative trick**:

$$\nabla_\theta p(\tau|\theta) = p(\tau|\theta)\nabla_\theta \log p(\tau|\theta),$$

we obtain:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[R(\tau)\,\nabla_\theta \log p(\tau|\theta)\right].$$

Now note that:

$$\log p(\tau|\theta) = \sum_{t=0}^{T-1} \log \pi_\theta(a_t|s_t),$$

so:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} R(\tau)\,\nabla_\theta \log \pi_\theta(a_t|s_t)\right].$$

This is the foundation of the **REINFORCE algorithm**.

—

3.4 Reducing Variance with Return-to-Go

Instead of using the total return $R(\tau)$ for each timestep, we can use the *return-to-go* G_t:

$$G_t = \sum_{k=t}^{T-1} \gamma^{k-t} r(s_k, a_k)$$

Then, the gradient estimator becomes:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} G_t\,\nabla_\theta \log \pi_\theta(a_t|s_t)\right].$$

This reduces variance since it only attributes future rewards to current actions.

—

3.5 Baseline and Advantage Function

To further reduce variance, we subtract a baseline $b(s_t)$ that does not depend on the action:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} (G_t - b(s_t))\,\nabla_\theta \log \pi_\theta(a_t|s_t)\right].$$

A common and effective choice is:

$$b(s_t) = V^\pi(s_t)$$

Then, the term $G_t - V^\pi(s_t)$ is the **advantage function**:

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$$

Thus, the general policy gradient becomes:

$$\boxed{\nabla_\theta J(\theta) = \mathbb{E}_{s_t, a_t \sim \pi_\theta} \left[A^\pi(s_t, a_t) \, \nabla_\theta \log \pi_\theta(a_t|s_t) \right].}$$

—

3.6 ActorCritic Methods

In practice, we approximate:

- $A^\pi(s, a)$ or $V^\pi(s)$ using a critic network (value function estimator)
- $\pi_\theta(a|s)$ using an actor network (policy)

The critic is trained by minimizing:

$$L_V(\phi) = \frac{1}{2} \mathbb{E}_{s_t \sim \pi_\theta} \left[(V_\phi(s_t) - G_t)^2 \right].$$

The actor is updated via:

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{s_t, a_t \sim \pi_\theta} \left[(G_t - V_\phi(s_t)) \, \nabla_\theta \log \pi_\theta(a_t|s_t) \right].$$

—

3.7 Intuitive Summary

- The policy gradient optimizes expected return directly.
- The gradient direction is determined by:

  $$\nabla_\theta \log \pi_\theta(a_t|s_t)$$

  scaled by how good the action was:

  $$A^\pi(s_t, a_t)$$

- The baseline reduces variance without introducing bias.
- Actor-Critic = policy gradient + value function approximation.

—

3.8 Algorithm: REINFORCE

Algorithm 1: REINFORCE Algorithm

Initialize policy parameters θ ;
repeat
  Collect trajectory $\tau = (s_0, a_0, r_0, \ldots, s_T)$ by running π_θ ;
  for *each time step* $t = 0, \ldots, T-1$ **do**
    Compute return-to-go $G_t = \sum_{k=t}^{T-1} \gamma^{k-t} r_k$;
    Update policy: $\theta \leftarrow \theta + \alpha G_t \nabla_\theta \log \pi_\theta(a_t|s_t)$;
until;

4 Fitted Q-Iteration, Function Approximation, and Bellman Operator

4.1 Function Approximators

In reinforcement learning, we aim to estimate the optimal action-value function $Q^*(s, a)$. When the state-action space is large or continuous, we cannot store one value per pair, so we approximate it with a parameterized function $Q_\phi(s, a)$:

$$Q_\phi(s, a) \approx Q^*(s, a)$$

Common types of function approximators include:

- **Tabular:** Each (s, a) pair has a separate entry.
- **Linear:** $Q_\phi(s, a) = \phi^\top x(s, a)$
- **Nonlinear:** Neural networks $Q_\phi(s, a) = f_\phi(s, a)$
- **Kernel / Gaussian process:** $Q_\phi(s, a) = \sum_i \alpha_i k((s, a), (s_i, a_i))$

—

4.2 The Bellman Optimality Operator

The Bellman optimality equation is defined as:

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

Define the **Bellman optimality operator** $\mathcal{T}^*$ acting on any function Q:

$$(\mathcal{T}^* Q)(s, a) = r(s, a) + \gamma \mathbb{E}_{s'} \left[\max_{a'} Q(s', a') \right]$$

Then the optimal Q^* satisfies the fixed-point condition:

$$Q^* = \mathcal{T}^* Q^*$$

That is, Q^* is the fixed point of the Bellman operator.

—

4.3 Contraction Property of the Bellman Operator

The Bellman operator $\mathcal{T}^*$ is a γ-**contraction mapping** under the sup norm $\| \cdot \|_\infty$:

$$\|\mathcal{T}^* Q_1 - \mathcal{T}^* Q_2\|_\infty \leq \gamma \|Q_1 - Q_2\|_\infty, \quad \forall Q_1, Q_2$$

This implies that repeated application of $\mathcal{T}^*$ moves any Q closer to Q^*.

—

4.4 Banach Fixed-Point Theorem

Theorem 4.1 (Banach Fixed-Point Theorem)**.** *Let (X, d) be a complete metric space and $T : X \to X$ be a contraction mapping, i.e.*

$$d(Tx, Ty) \leq \gamma d(x, y), \quad 0 \leq \gamma < 1.$$

Then:

1. *T has a unique fixed point $x^* \in X$.*
2. *Starting from any $x_0 \in X$, the sequence $x_{k+1} = T(x_k)$ converges to x^*.*

Applying this theorem with $X = \mathbb{R}^{|S||A|}$, $d = \|\cdot\|_\infty$, and $T = \mathcal{T}^*$, we conclude that Q^* is the unique fixed point and

$$Q_{k+1} = \mathcal{T}^* Q_k \quad \Longrightarrow \quad Q_k \to Q^*.$$

—

4.5 Function Approximation and the Projected Bellman Operator

When using function approximators such as neural networks, we can only represent a subset of all possible functions. Hence, we define a **projected Bellman operator**:

$$\mathcal{T}_\Pi Q = \Pi \mathcal{T}^* Q,$$

where Π denotes projection onto the function space representable by Q_ϕ.

However, $\mathcal{T}_\Pi$ is generally *not* a contraction mapping:

$$\|\mathcal{T}_\Pi Q_1 - \mathcal{T}_\Pi Q_2\|_\infty \not\leq \gamma \|Q_1 - Q_2\|_\infty,$$

so convergence is not guaranteed. This explains why nonlinear function approximators such as deep neural networks may cause instability or divergence.

—

4.6 Summary Table

Setting	Q Representation	Bellman Operator Type	Convergence Guara
Tabular	Lookup table	Exact $\mathcal{T}^*$	Guaranteed (Banach the
Linear Approximation	$Q_\phi(s,a) = \phi^\top x(s,a)$	Approximate	Sometimes guarante
Neural Networks	$Q_\phi(s,a) = f_\phi(s,a)$	Projected $\mathcal{T}_\Pi$	No theoretical guaran

—

4.7 Intuitive Summary

> The Bellman operator $\mathcal{T}^*$ "pulls" any Q-function toward the optimal fixed point Q^*. Fitted Q-Iteration iteratively applies this operator via function approximation. In the tabular or linear case, this process is a contraction and converges to Q^*. In the nonlinear case (e.g., deep neural networks), the projection breaks contraction, and convergence is no longer guaranteed.

Table 1: Comparison among Q-Iteration, Q-Learning, and Online Q-Iteration

Aspect	Q-Iteration	Q-Learning	Online Q-Iteration (Function Approximation)
Mathematical form	$Q_{k+1}(s,a) = r(s,a) + \gamma \sum_{s'} P(s'\|s,a) \max_{a'} Q_k(s',a')$	$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)]$	$\phi \leftarrow \phi - \alpha \nabla_\phi Q_\phi(s_i,a_i)\,[Q_\phi(s_i,a_i) - (r_i + \gamma \max_{a'} Q_\phi(s_i',a'))]$
Core idea	Exact Bellman iteration using known model	Sample-based approximation of Bellman operator	Gradient descent on Bellman error with function approximation
Environment model $P(s'\|s,a)$	Required (model-based)	Not required (model-free)	Not required (model-free)
Update type	Batch / synchronous (all states)	Incremental / online (per step)	Online stochastic gradient (per sample)
Function representation	Tabular (explicit $Q(s,a)$)	Tabular (explicit $Q(s,a)$)	Parameteric $Q_\varphi(s,a)$ (e.g., neural network)
Learning signal	Exact expectation $\mathbb{E}_{s'}[\cdot]$	Single transition sample	Mini-batch / single-sample gradient step
Policy type	Greedy: $\pi(s) = \arg\max_a Q(s,a)$	Off-policy (e.g., ϵ-greedy for exploration)	Off-policy, with many choices of behavior policy
Objective interpretation	Find fixed point of Bellman optimality operator $\mathcal{T}^*$	Stochastic approximation to fixed point	Minimize Bellman error: $\frac{1}{2}\mathbb{E}[(Q_\phi - \mathcal{T}^* Q_\phi)^2]$
Data usage	Uses full transition probabilities	Uses sampled transitions (s,a,r,s')	Uses replay buffer / online samples (s,a,r,s')
Convergence guarantee	Guaranteed (Bellman contraction)	Guaranteed under conditions (RobbinsMonro)	Not guaranteed (depends on function class / optimization)
Computation cost	High (full state sweep)	Low (one-step updates)	High per update (backpropagation, nonlinear fit)
Example algorithms	Value Iteration, Dynamic Programming	Tabular Q-Learning	DQN, Double DQN, Fitted Q-Iteration

Table 2: Comparison between Online and Offline Reinforcement Learning

Aspect	Online RL	Offline RL
Data source	Collected in real time during interaction with the environment	Pre-collected static dataset (no further environment interaction)
Update timing	Update parameters after every step or episode	Train model using fixed data, all updates done offline
Environment access	Required (agent interacts continuously)	Not required (training purely from data)
Data distribution	Continuously changes as policy improves	Fixed; determined by behavior policy that collected data
Advantages	Adaptive, can learn from continuous feedback	Safe, data-efficient, usable in domains where interaction is costly
Challenges	Instability, high variance, explorationexploitation tradeoff	Distributional shift, extrapolation error, overestimation
Typical algorithms	Q-Learning, SARSA, ActorCritic, DDPG, PPO	Fitted Q-Iteration, DQN (offline), CQL, IQL, Behavior Cloning

Table 3: Comparison between On-policy and Off-policy Learning

Aspect	On-policy	Off-policy
Definition	Learns the value of the *same* policy used to generate data	Learns the value of a *different* (target) policy than data-generating policy
Behavior vs. target policy	Identical ($\pi_b = \pi_t$)	Different ($\pi_b \neq \pi_t$)
Exploration method	Exploration built into policy (e.g., ϵ-greedy within current policy)	Can reuse off-policy data and experience replay
Advantages	Stable, lower bias, directly improves current policy	Data-efficient, can leverage past experience and other agents' data
Challenges	Requires fresh data, less efficient	Risk of distribution mismatch, instability in importance sampling
Typical algorithms	SARSA, A2C, PPO, Policy Gradient (on-policy)	Q-Learning, DQN, DDPG, SAC, Fitted Q-Iteration
Relation to data	Must collect new data from current policy	Can learn from arbitrary datasets (even offline)

5 Deep RL with Q-Fucntions

5.1 Why Q-Learning is not Gradient Descent

5.1.1 The Apparent Gradient Descent Form

In function-approximation Q-learning, the update rule is:

$$\phi \leftarrow \phi - \alpha \nabla_\phi Q_\phi(s_i,a_i)\Big(Q_\phi(s_i,a_i) - y_i\Big), \quad y_i = r(s_i,a_i) + \gamma \max_{a'} Q_\phi(s_i',a')$$

This looks like minimizing the following squared error:

$$L(\phi) = \frac{1}{2}(Q_\phi(s_i,a_i) - y_i)^2.$$

At first glance, it appears to be a standard gradient descent step.

—

5.1.2 The Hidden Problem: *Target Depends on* ϕ

However, the target y_i itself depends on the same parameters ϕ through $Q_\phi(s_i',a')$. If we compute the true gradient of $L(\phi)$:

$$\nabla_\phi L(\phi) = (Q_\phi(s_i, a_i) - y_i)\Big[\nabla_\phi Q_\phi(s_i, a_i) - \nabla_\phi\Big(r(s_i, a_i) + \gamma \max_{a'} Q_\phi(s_i', a')\Big)\Big],$$

we can see that the second term propagates gradient through the target value y_i. In practice, **Q-learning ignores this term**:

$$\text{no gradient through target:} \quad \nabla_\phi y_i = 0.$$

Hence, Q-learning performs a *semi-gradient* update, not a true gradient descent step.

—

5.1.3 Why Ignoring the Gradient is Necessary

If gradients were allowed to flow through the target y_i, the optimization would become unstable because:

- The target y_i changes as Q_ϕ changes — a moving target.
- The learning objective is not fixed; both sides of the Bellman equation depend on the same parameters.

Therefore, Q-learning treats y_i as a constant:

$$y_i = \text{stop\_gradient}\Big(r_i + \gamma \max_{a'} Q_\phi(s_i', a')\Big),$$

turning the update into a **fixed-point iteration** rather than a true gradient-based optimization.

—

5.1.4 Comparison: Gradient Descent vs Q-Learning Update

Aspect	**True Gradient Descent**	**Q-Learning Update**
Objective	Minimize fixed loss function $L(\phi)$	Satisfy Bellman fixed-point equation
Target y	Constant (does not depend on ϕ)	Depends on ϕ, but gradient stopped
Gradient flow	Through both prediction and target	Through prediction only
Type of optimization	Standard supervised learning (convex/nonconvex)	Bootstrapped fixed-point iteration
Stability	Usually stable (well-defined loss landscape)	Can diverge (moving target problem)
Interpretation	Minimizing a static objective	Approximating the Bellman operator $\mathcal{T}^*$
Example	Linear regression, cross-entropy loss	TD-learning, Q-learning, DQN

—

5.1.5 Conceptual Interpretation

- In gradient descent, we descend a *fixed hill*; the loss surface does not move as we update parameters.
- In Q-learning, the "hill" itself moves, because the target depends on the same parameters — we are chasing a moving target.

—

5.1.6 Connection to DQN

To mitigate the instability caused by a moving target, DQN introduces a **target network** Q_{ϕ^-} for computing targets:

$$y_i = r_i + \gamma \max_{a'} Q_{\phi^-}(s_i', a'), \quad \text{with } \phi^- \text{ updated slowly (periodically frozen).}$$

This makes the target temporarily constant, bringing the update closer to a stable gradient descent process.

—

5.1.7 Summary

> **Q-learning is not true gradient descent**, because it ignores gradients through its bootstrapped target. It is a fixed-point iteration aiming to satisfy the Bellman equation, not to minimize a static differentiable loss.

5.2 A More General View of Q-Learning and DQN

5.2.1 The Three-Process View

Q-learning with replay buffer and target network can be decomposed into three main processes:

1. **Process 1: Data Collection** Interact with the environment using a policy $\pi(a|s)$ (e.g., ϵ-greedy), collect transitions (s, a, s', r), and store them in the replay buffer $\mathcal{B}$.

2. **Process 2: Target Update** Update target network parameters periodically or smoothly:
$$\phi' \leftarrow \tau\phi' + (1-\tau)\phi \quad \text{or} \quad \phi' \leftarrow \phi$$
This provides a stable target for Q-learning.

3. **Process 3: Q-Function Regression** Sample batches from the replay buffer and perform gradient descent:
$$\phi \leftarrow \phi - \alpha \sum_i \nabla_\phi Q_\phi(s_i, a_i) \Big(Q_\phi(s_i, a_i) - [r_i + \gamma \max_{a'} Q_{\phi'}(s_i', a')] \Big)$$

The overall structure is visualized as follows:

Environment $\leftrightarrow \pi(a|s)$Replay Buffer $\longrightarrow$ Current Parameters $(\phi) \longrightarrow$ Target Parameters $(\phi') \longrightarrow$ Q-fu

Old transitions are evicted periodically to maintain buffer freshness.

—

5.2.2 Process Interaction Overview

- **Process 1 — Data Collection:** Generates new transitions (s, a, s', r) and inserts them into the replay buffer. It determines how the agent explores the environment.
- **Process 2 — Target Update:** Updates target parameters ϕ', either hard or soft, defining how frequently the target values change.
- **Process 3 — Q-Function Regression:** Uses sampled batches from $\mathcal{B}$ to update ϕ by minimizing the TD error.

These processes may run at different speeds or be nested within each other, leading to different algorithmic variants.

—

5.2.3 Algorithm Comparisons in the Three-Process Framework

Algorithm	Process 1 (Data Collection)	Process 2 (Target Update)	Process 3 (Q-Function Regression)
Online Q-Learning	Collects new transition, immediately discards old ones.	No separate target network (same ϕ used).	All processes run simultaneously at same speed.
DQN	Process 1 and 3 run together (real-time data collection and regression).	Process 2 (target update) runs slowly (periodically or softly).	Regression uses replay buffer to decorrelate samples.
Fitted Q-Iteration	Data collection (process 1) is outer loop.	Target update (process 2) inside it.	Regression (process 3) runs as an inner optimization loop until convergence.

—

5.2.4 Intuitive Summary

- Reinforcement learning can be understood as the interaction of three asynchronous processes:
  1. Data generation (environment sampling)
  2. Target computation (stabilization)
  3. Function approximation (learning)
- Different RL algorithms can be viewed as different synchronization choices between these processes.
- DQN improves upon online Q-learning by decoupling these processes and allowing stable off-policy updates.

—

5.2.5 Key Insights

- The replay buffer provides a bridge between **data collection** and **Q-function learning**.
- The target network acts as a temporal buffer between **learning** and **bootstrapping**.
- By adjusting their relative update speeds, we can control the trade-off between *stability* and *learning speed*.

—

5.2.6 Summary of Process Speed Hierarchies

Online Q-learning:	Process 1, 2, 3 all run at the same rate.
DQN:	Process 1 & 3 run together; Process 2 (target) is slower.
Fitted Q-Iteration:	Process 3 nested inside Process 2, which is nested inside Process 1.

5.3 Q-learning with N-step Returns

In standard Q-learning, the target value is computed using a single-step reward:

$$y_t = r_t + \gamma \max_a Q_{\phi'}(s_{t+1}, a)$$

This approach relies heavily on bootstrapping from the next state, which can introduce high bias and slow propagation of reward information.

To address this, we can use **N-step returns** to incorporate multiple future rewards:

$$y_{j,t} = \sum_{t'=t}^{t+N-1} \gamma^{t'-t} r_{j,t'} + \gamma^N \max_{a_{j,t+N}} Q_{\phi'}(s_{j,t+N}, a_{j,t+N})$$

This estimates the expected return for policy π as:

$$Q^\pi(s_{j,t}, a_{j,t}) \approx y_{j,t}$$

where the policy π is assumed to be:

$$\pi(a_t|s_t) = \begin{cases} 1, & \text{if } a_t = \arg\max_a Q_\phi(s_t, a) \\ 0, & \text{otherwise.} \end{cases}$$

—

Advantages

- **Less biased target values:** Incorporates multiple real rewards, reducing dependence on inaccurate Q-values early in training.
- **Faster learning:** Reward information propagates backward more quickly, improving sample efficiency.

—

Limitation: Off-policy Bias

The N-step formulation assumes that all intermediate transitions $(s_{t'}, a_{t'})$ for $t' - t < N - 1$ are generated by the same policy π. This assumption holds for **on-policy** learning, but breaks for **off-policy** settings (e.g., when using a replay buffer).

Therefore, the N-step target is:

- **Only theoretically correct when learning on-policy.**
- Off-policy data can introduce bias because transitions may come from older behavior policies.

For $N = 1$, this issue disappears since only one transition (s_t, a_t, r_t, s_{t+1}) is required, which is valid for any policy.

—

Practical Fixes for Off-policy Settings

1. **Ignore the problem** Simply apply N-step targets without correction. In practice, this often works well for small N.
2. **Cut the trace** Dynamically choose N to include only on-policy segments. Effective when most samples are near on-policy and the action space is small.
3. **Importance sampling** Weight each step by $\prod_t \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}$ to correct off-policy bias. This is theoretically correct but may increase variance.

—

Summary

Aspect	Advantage	Limitation / Concern
Bias	Reduces bias when Q-values are inaccurate	Only unbiased for on-policy data
Learning Speed	Propagates reward information faster	May introduce instability off-policy
Implementation	Easy to extend from standard Q-learning	Needs multiple consecutive transitions
Off-policy Use	Can be corrected via importance sampling	High variance in correction weights

—

Reference

For a formal treatment, see:

Munos et al., *"Safe and Efficient Off-Policy Reinforcement Learning"*, 2016.

5.4 Double Deep Q-learning (Double DQN)

Motivation

Although the classic Deep Q-learning (DQN) algorithm greatly stabilizes Q-learning with replay buffers and target networks, it still suffers from **overestimation bias**.

In standard DQN, the target is computed as:

$$y_t = r_t + \gamma \max_{a'} Q_{\phi'}(s_{t+1}, a')$$

Since both the selection and evaluation of the maximizing action are done using the same Q-network $Q_{\phi'}$, any noise or over-optimistic estimate in $Q_{\phi'}$ tends to be **amplified by the** max **operator**.

This phenomenon can systematically overestimate Q-values, leading to suboptimal policies.

—

Key Idea

Double DQN (van Hasselt et al., 2016) addresses this problem by *decoupling action selection and evaluation*.

Instead of using the same network for both roles, Double DQN uses:

- The **online network** Q_ϕ to **select** the greedy action.
- The **target network** $Q_{\phi'}$ to **evaluate** that action.

The new target value is defined as:

$$y_t^{\text{DoubleDQN}} = r_t + \gamma\, Q_{\phi'}\Big(s_{t+1}, \arg\max_{a'} Q_\phi(s_{t+1}, a')\Big)$$

That is, the arg max is computed by the current (online) parameters ϕ, but the value is obtained from the slower-moving target parameters ϕ'.

—

Algorithm

1. Collect transitions (s_t, a_t, r_t, s_{t+1}) into replay buffer $\mathcal{B}$.
2. Sample a mini-batch $\{(s_j, a_j, r_j, s'_j)\}$ from $\mathcal{B}$.
3. Compute targets using:

$$y_j = r_j + \gamma Q_{\phi'}(s'_j, \arg\max_{a'} Q_\phi(s'_j, a'))$$

4. Update the online network parameters ϕ by minimizing:

$$L(\phi) = \frac{1}{2}\sum_j \Big(Q_\phi(s_j, a_j) - y_j\Big)^2$$

5. Periodically update the target network:

$$\phi' \leftarrow \tau\phi' + (1-\tau)\phi \quad \text{(soft update, or hard copy every } N \text{ steps)}$$

—

Intuition

In standard DQN:

same network for both selection and evaluation ⇒ biased overestimation

In Double DQN:

$$\text{two networks: } \begin{cases} \text{Selection: } a^* = \arg\max_a Q_\phi(s', a) \\ \text{Evaluation: } Q_{\phi'}(s', a^*) \end{cases} \Rightarrow \text{more accurate targets, reduced bias.}$$

This results in more stable learning, especially in stochastic environments where rewards and value estimates are noisy.

—

Advantages and Disadvantages

Aspect	**Advantage**	**Limitation / Concern**
Estimation Bias	Reduces overestimation in Q-values	May introduce slight underestimation
Stability	Improves training stability compared to vanilla DQN	Still sensitive to hyperparameters
Computation	Requires only minor changes to DQN (same architecture)	Needs careful synchronization between ϕ and ϕ'
Empirical Results	Performs significantly better on Atari benchmarks	Benefits diminish for very large networks or continuous actions

—

Summary

- Double DQN separates action selection and evaluation, effectively correcting the overestimation bias of standard DQN.
- It retains the computational simplicity of DQN while yielding more stable and reliable performance.
- It is now considered a standard component in most modern deep reinforcement learning algorithms (e.g., Rainbow DQN, Dueling DQN).

—

Reference

van Hasselt, H., Guez, A., and Silver, D. (2016). *Deep Reinforcement Learning with Double Q-learning.* In Proceedings of the 30th AAAI Conference on Artificial Intelligence.