Fundamentals

What is Reinforcement Learning?

- **Big picture:** RL is about **learning through interaction**.
- An **agent** interacts with an **environment**:
    - At each step: the agent is in a **state** (s),
    - Chooses an **action** (a),
    - Gets a **reward** (r) and a new **state** (s').

The goal: **maximise total rewards over time.**

Rewards as Feedback

- Rewards tell the agent if what it did was **good or bad**.
- Example:
    - In a maze, **+1** for reaching the goal, **0** otherwise.
    - In a game, **+100** for winning, **–100** for losing.

The agent doesn't know what's best initially — it must **explore, try actions, and learn from feedback**.

Value Functions

The agent needs a way to estimate "how good" things are.

That's where **value functions** come in:

- **State Value $V(s)$:** how good is it to be in state s?
- **Action Value $Q(s,a)$:** how good is it to take action aa in state s?

These are like the agent's **internal map of expectations**.

Bellman Equations

The **Bellman equations** update the agent's knowledge.

They say:

> The value of now = reward now + value of the future.

- For state values:
  $V(s) = \max_a \mathbb{E}[\,r + \gamma V(s') \mid s, a\,]$
- For action values:
  $Q(s,a) = \mathbb{E}[\,r + \gamma \max_{a'} Q(s',a') \mid s, a\,]$

Bellman updates = the **mathematical glue** that connects present and future.

Learning Approaches

Now, **how do we actually use experience to learn values/policies?**

1. **Dynamic Programming (DP)**
   - Works if we know the environment probabilities (rare in practice).
2. **Monte Carlo (MC)**
   - Learn from complete episodes → average total returns.
3. **Temporal Difference (TD)**
   - Update values **step by step** without waiting for episode to finish.
   - Variants:
     - **SARSA (on-policy)**: learns from the actual action taken.
     - **Q-learning (off-policy)**: learns from the best action possible.

Function Approximation

For small problems, we can keep a **table of** $Q(s, a)$.

But in big problems (chess, Atari, robotics) → too many states, So :

- Use a **function (like a neural network)** to approximate Q or V.
  - **Deep Q-Networks (DQN):** use a neural net to approximate Q-values.
  - **Policy Gradients:** learn the policy directly instead of values.

Where UVFAs Come In

UVFAs extend the idea of value functions:

- Instead of just $V(s)$ or $Q(s, a)$, we also include the **goal**:
  $V(s, g), \quad Q(s, a, g)$
- This lets the agent **generalize across different goals**.

Example: instead of learning separately for “get to the red square” and “get to the blue square,” the UVFA learns one function that works for both.

The world model: an MDP

RL classically assumes the environment is a **Markov Decision Process (MDP)**

$\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$

- $\mathcal{S}$: set of states
- $\mathcal{A}$: set of actions (may depend on ss; often assume a fixed set)
- $P(s' \mid s, a)$: transition kernel (stationary dynamics)

- $R(r \mid s, a, s')$ or $\bar{r}(s, a) = \mathbb{E}[R \mid s, a]$: reward distribution or its mean
- $\gamma \in [0, 1)$: discount factor (preference for sooner rewards and to ensure convergence)

Markov property: the next state & reward depend **only** on the current state-action, not the entire past:

$$\Pr(S_{t+1} = s', R_{t+1} = r \mid S_0, A_0, \dots, S_t = s, A_t = a) = \Pr(S_{t+1} = s', R_{t+1} = r \mid s, a).$$

> If observations aren't fully Markov (partial observability), you have a POMDP; you can restore Markov-ness by augmenting state with a belief or memory (recurrent nets, filters, reward machines, etc.).

Trajectories & probability space

A **trajectory** (episode) of length T is

$$\tau = (s_0, a_0, r_1, s_1, a_1, r_2, \dots, s_T).$$

Given a **policy** $\pi(a \mid s)$ and initial-state distribution ρ_0,

$$p_\pi(\tau) = \rho_0(s_0) \prod_{t=0}^{T-1} \left[\pi(a_t \mid s_t)\; P(s_{t+1} \mid s_t, a_t)\right].$$

Expectations "under π" mean integrating quantities over $p_\pi(\tau)$.

Returns (what we try to maximize)

The **return** is the cumulative reward from time tt.

(a) discounted infinite-horizon (most common)

$$G_t \;=\; \sum_{k=0}^{\infty} \gamma^k\, R_{t+1+k}, \quad \gamma \in [0, 1).$$

If rewards are bounded $|R_t| \le R_{\max}$, then $|G_t| \le \frac{R_{\max}}{1-\gamma}$ (converges).

(b) episodic finite-horizon (length TT)

$$G_t \;=\; \sum_{k=0}^{T-t-1} R_{t+1+k} \quad (\text{often } \gamma = 1).$$

(c) average-reward (continuing tasks)

$$\rho(\pi) \;=\; \lim_{T\to\infty} \frac{1}{T}\, \mathbb{E}_\pi\Big[\sum_{t=0}^{T-1} R_{t+1}\Big].$$

(Useful for tasks without natural episode ends; analysis differs from discounted case.)

Objectives (prediction vs control)

- **policy** π: mapping $s \mapsto \text{distribution over } \mathcal{A}$. Can be stochastic or deterministic.
- **evaluation (prediction): for a fixed π, estimate how good it is (value functions below).**
- **control: find π^* that maximizes the objective (return).**

Discounted objective (control):

$$J(\pi) \;=\; \mathbb{E}_{\tau\sim p_\pi}\big[\,G_0\,\big] \quad \text{and seek} \quad \pi^* \in \arg\max_\pi J(\pi).$$

Key fact: Under standard conditions in discounted MDPs, there exists an **optimal stationary deterministic** policy π^*.

Value functions (define "how good")

These are expectations of returns under a policy π\pi:

- **state-value**:

$V^\pi(s) = \mathbb{E}_\pi[\, G_t \mid S_t = s]$

- **action-value**:

$Q^\pi(s,a) = \mathbb{E}_\pi[\, G_t \mid S_t = s, A_t = a]$

- **advantage**:

$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s).$

Optimal counterparts:

$V^*(s) = \sup_\pi V^\pi(s), \qquad Q^*(s,a) = \sup_\pi Q^\pi(s,a).$

Bellman relations (the recursive glue)

policy evaluation (for a fixed π\pi)

$V^\pi(s) = \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s,a)\Big(\bar{r}(s,a,s') + \gamma V^\pi(s')\Big),$

$Q^\pi(s,a) = \sum_{s'} P(s' \mid s,a)\Big(\bar{r}(s,a,s') + \gamma \sum_{a'} \pi(a' \mid s') Q^\pi(s',a')\Big).$

optimality (control)

$V^*(s) = \max_a \sum_{s'} P(s' \mid s,a)\Big(\bar{r}(s,a,s') + \gamma V^*(s')\Big),$

$Q^*(s,a) = \sum_{s'} P(s' \mid s,a)\Big(\bar{r}(s,a,s') + \gamma \max_{a'} Q^*(s',a')\Big).$

These arise by unrolling the definition of G_t one step and using the Markov property.

Occupancy measures (useful later for gradients)

For discounted problems, define the **discounted state(-action) visitation**:

$d_\pi(s) = (1-\gamma)\sum_{t=0}^\infty \gamma^t \Pr(S_t = s \mid \pi), \quad d_\pi(s,a) = d_\pi(s)\,\pi(a \mid s).$

Then the objective can be written as

$J(\pi) = \frac{1}{1-\gamma}\,\mathbb{E}_{(s,a)\sim d_\pi}\big[\bar{r}(s,a)\big].$

(Handy for policy-gradient derivations and intuition about "where the agent spends time.")

Task types & terminal handling

- **episodic**: trajectories end in an absorbing terminal s_{term} with zero future reward.
- **continuing**: no terminal; use $\gamma < 1$ or average-reward.

- **finite-horizon**: decisions over $t = 0, \ldots, T-1$ with horizon-aware value functions $V_t(s)$.

Reward design & shaping (critical in practice)

- **scaling rewards** by a positive constant α scales all values by α and **doesn't** change the optimal policy (discounted case).
- **adding a constant** cc per step:
  - discounted continuing: adds $\frac{c}{1-\gamma}$ to all state/action values → **policy ranking unchanged**.
  - episodic with variable lengths: adds $\sum_{k=0}^{T-1} \gamma^k c$ which **can** change rankings if episode lengths differ.
- **potential-based shaping** (safe): add $F(s, a, s') = \gamma\Phi(s') - \Phi(s)$.

  It preserves all optimal policies because it exactly telescopes in the Bellman equations (shifts values but not argmax).

Sketch (why safe): shaped Q'target becomes

$$r + F + \gamma \max_{a'} Q'(s', a') = \underbrace{r + \gamma \max_{a'} Q(s', a')}_{\text{original}} + \gamma\Phi(s') - \Phi(s),$$

so $Q'(s, a) = Q(s, a) - \Phi(s) + \text{const}$ and the maximizing actions are identical.

Exploration vs exploitation (the tension)

- **exploit** the current best estimate to get reward now.
- **explore** uncertain actions/states to improve future estimates.

  Mechanisms: ε-greedy, softmax/boltzmann, optimism (UCB), entropy bonuses (in policy gradients), intrinsic motivation, etc. (We'll detail later when we hit algorithms.)

Notation cheat-sheet (we'll reuse)

- S_t, A_t, R_{t+1}: state, action, reward at time tt.
- $\pi(a \mid s)$: policy.
- V^π, Q^π, A^π: value, action-value, advantage.
- G_t: return.
- $P(s'|s, a)$: dynamics. $\bar{r}(s, a, s') = \mathbb{E}[R|s, a, s']$.
- γ: discount.
- α: stepsize/learning rate.
- $d_\pi(s), d_\pi(s, a)$: discounted visitation distributions.

Tiny sanity checks + micro-derivations

(1) convergence of G_t (discounted):

If $|R_{t+1}| \le R_{\max}$, then
$|G_t| \le \sum_{k\ge 0} \gamma^k R_{\max} = \frac{R_{\max}}{1-\gamma}$.

(2) adding constant cc (discounted continuing):
$\tilde{G}_t = \sum_{k\ge 0} \gamma^k (R_{t+1+k} + c) = G_t + \frac{c}{1-\gamma}$.
So $\tilde{Q}^\pi(s,a) = Q^\pi(s,a) + \frac{c}{1-\gamma} \to$ same argmax over actions.

(3) optimal deterministic stationary policy exists (discounted MDP):

Follows from contraction mapping of the Bellman optimality operator and measurability/compactness assumptions; fixed point Q^* induces a greedy deterministic $\pi^*(s) \in \arg\max_a Q^*(s,a)$.

Minimal code: computing returns (for intuition)

```
def discounted_returns(rewards, gamma):
    # rewards = [r1, r2, ..., rT]
    G = 0.0
    out = [0.0]*len(rewards)
    for t in reversed(range(len(rewards))):
        G = rewards[t] + gamma * G
        out[t] = G
    return out
```

- Try a few reward sequences and γ\gamma values to see how “the future” weighs in.

Rewards as Feedback

Rewards are the *signals* that guide the agent. They define **what the agent should care about**.

Without rewards, the agent has no reason to prefer one action over another.

What is a reward?

- **reward (R_{t+1})**: a scalar signal given **after** an action at time tt, before the next state.
- it's *local* feedback: “good” (+), “bad” (–), or “neutral” (0).

Formally, in an MDP:

$R_{t+1} \sim R(\cdot \mid S_t, A_t, S_{t+1})$,

a random variable possibly depending on the transition.

= think of it as **the teacher’s hint** — not the full lesson.

Reward vs return

The agent doesn't just care about the immediate reward, but about the **long-term return**:
$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+1+k}$.

- R_{t+1}: one step's feedback.
- G_t: total "future goodness" from now on.

reward = local feedback; return = what the agent maximizes.

Examples

- **maze**:
  - +1+1 when reaching goal, 00 otherwise.
  - problem: reward is **sparse** → agent learns slowly.
- **Atari (Breakout)**:
  - +1 per brick destroyed, –1 life lost.
  - reward is **dense** and more informative.
- **self-driving car**:
  - +100 for safe arrival, –100 for crash, small negative per second to encourage speed.
  - here, design shapes behaviour.

Sparse vs dense rewards

- **sparse**: agent rarely sees nonzero rewards → exploration hard.
- **dense**: frequent signals guide learning quickly, but risk **reward hacking** (agent exploits loopholes).

Example: if you reward "distance traveled" → agent might spin in circles to farm reward.

Discounted reward intuition

Why multiply future rewards by γ^k?

1. **Uncertainty about the future** (further future is less reliable).
2. **Mathematical convergence** (ensures infinite sum converges).
3. **Preference for immediacy** (like financial discounting).

- $\gamma \approx 0$: agent cares only about immediate rewards (short-sighted).
- $\gamma \approx 1$: agent cares about long-term outcome (far-sighted).

Shaping rewards

Sometimes the natural reward is too sparse (e.g., +1 only at goal).

We can add **shaping terms** to speed learning:

- **potential-based shaping** (safe):

$F(s, a, s') = \gamma\Phi(s') - \Phi(s)$.

This preserves optimal policies (only shifts values).

- **dangerous shaping**: giving extra rewards that may change what "optimal" means.

Deterministic vs stochastic rewards

- **deterministic**: same reward every time for a given transition.
- **stochastic**: reward distribution with variance.
  - Example: slot machines (bandits) give random payouts.

Agent must learn **expected rewards**:

$\bar{r}(s, a) = \mathbb{E}[R \mid s, a]$.

Reward scaling & clipping

In practice (especially with deep nets):

- **clipping** rewards (e.g., $[-1, 1]$) prevents exploding updates.
- **normalising** or scaling helps stabilise learning.
- but scaling changes relative magnitudes → may alter behaviour if not done carefully.

Reward hacking

Agents may exploit poorly designed rewards:

- Example: a cleaning robot rewarded for "collected dirt" → dumps dirt to pick it up again.
- This is why **reward design** is critical in real-world tasks.

Minimal derivation: why shifting rewards by a constant can be safe

Suppose discounted return with constant c:

$\tilde{G}_t = \sum_{k=0}^{\infty} \gamma^k (R_{t+1+k} + c)$.

$= G_t + \frac{c}{1-\gamma}$.

So every state's value shifts by the same constant → **policy ranking is unchanged**.

But in **finite-horizon or undiscounted episodic tasks**, this is not guaranteed safe.

Code intuition

```
def discounted_return(rewards, gamma=0.9):
    G, out = 0, []
    for r in reversed(rewards):
        G = r + gamma * G
        out.insert(0, G)
    return out
```

```
print(discounted_return([0,0,0,1], gamma=0.9))
# -> [0.729, 0.81, 0.9, 1.0]
```

Only the last step had a +1 reward, but discounting spreads “credit” back in time.

Summary

- reward = local scalar feedback.
- return = cumulative discounted reward.
- reward design crucial (sparse vs dense, shaping).
- shifting rewards may or may not preserve optimal policies.
- discounting balances short vs long term.

Value Functions (Deep Dive)

rewards are the raw signal.

value functions are how the agent predicts long-term goodness from states and actions.

they are the foundation for almost everything in rl.

state-value function (under policy π)

$V^\pi(s) = \mathbb{E}_\pi\left[G_t \mid S_t = s\right]$

- “expected return if I start in s and follow policy π.”

action-value function

$Q^\pi(s,a) = \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right]$

- “expected return if I start in s, take action a, then follow π.”

advantage function

$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$

- measures “how much better is action aa compared to average behavior at ss.”

relation between v and q

since $V^\pi(s)$ is the expectation over the policy’s action choices:

$V^\pi(s) = \sum_a \pi(a \mid s)\, Q^\pi(s,a).$

Bellman expectation equations (policy evaluation)

these come directly from unrolling the return one step.

for **v**:

$V^\pi(s) = \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s,a)\Big(\bar{r}(s,a,s') + \gamma V^\pi(s')\Big).$

for **q**:

$$Q^\pi(s,a) = \sum_{s'} P(s' \mid s,a)\Big(\bar{r}(s,a,s') + \gamma \sum_{a'} \pi(a' \mid s') Q^\pi(s',a')\Big).$$

derivation idea:

start from $V^\pi(s) = \mathbb{E}[G_t]$, write $G_t = R_{t+1} + \gamma G_{t+1}$, expand expectation, use Markov property.

Bellman optimality equations

define optimal value functions:

$$V^*(s) = \max_\pi V^\pi(s), \qquad Q^*(s,a) = \max_\pi Q^\pi(s,a).$$

then

$$V^*(s) = \max_a \sum_{s'} P(s' \mid s,a)\Big(\bar{r}(s,a,s') + \gamma V^*(s')\Big),$$

$$Q^*(s,a) = \sum_{s'} P(s' \mid s,a)\Big(\bar{r}(s,a,s') + \gamma \max_{a'} Q^*(s',a')\Big).$$

these are **fixed-point equations** solved by dynamic programming, or approximated via learning.

The bellman operators

define an operator TπT^\pi on value functions:

$$(T^\pi V)(s) = \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s,a)\big(\bar{r}(s,a,s') + \gamma V(s')\big).$$

- T^π is a γ **contraction** (Banach fixed-point theorem).
- unique fixed point = V^π.
- iterative application converges: $V_{k+1} = T^\pi V_k \to V^\pi$.

for optimality:

$$(T^* V)(s) = \max_a \sum_{s'} P(s' \mid s,a)\big(\bar{r}(s,a,s') + \gamma V(s')\big).$$

- contraction → unique fixed point V^*.

this is why rl learning converges.

Prediction vs control (context for uvfa)

- **prediction:** given π, estimate V^π, Q^π.
- **control:** improve policy toward optimal using estimates.

most algorithms alternate between these two.

Connection to uvfa

uvfa generalizes these functions by **adding a goal input**:

$$V^\pi(s,g), \quad Q^\pi(s,a,g).$$

same definitions, just conditioned on **goal gg**.

this allows **generalization across tasks/goals**.

small numerical example

imagine a 2-state mdp:

- states: s_1, s_2
- actions: a_1, a_2
- transitions: from $s_1, a_1 \to s_2$ with reward +1; from $s_1, a_2 \to s_1$ with reward 0.
- $\gamma = 0.9$.

compute $Q\pi(s1, a1)Q^\pi(s_1, a_1)$:

take a_1 at s_1, go to s_2: reward +1.

suppose from s_2, policy ends (no reward).

so

$Q^\pi(s_1, a_1) = 1 + 0.9 \cdot 0 = 1.$

while

$Q^\pi(s_1, a_2) = 0 + 0.9V^\pi(s_1).$

so action a_1 is better.

code snippet

```
import numpy as np

# tiny mdp: 2 states, 2 actions
# transitions: dict[(s,a)] = (next_state, reward)
transitions = {
    (0,0): (1,1),   # s0,a0 -> s1, r=1
    (0,1): (0,0),   # s0,a1 -> s0, r=0
    (1,0): (1,0),   # terminal self-loop
    (1,1): (1,0)
}
gamma = 0.9

def Q_value(state, action, V):
    ns, r = transitions[(state,action)]
    return r + gamma*V[ns]

V = np.zeros(2)
Q = np.zeros((2,2))

for s in [0,1]:
    for a in [0,1]:
        Q[s,a] = Q_value(s,a,V)
```

```
print("Q-values:", Q)
```

summary

- **value functions = expectations of return**.
- **bellman equations** tie today's value to reward + future value.
- **optimality equations** define V∗,Q∗V^*, Q^*.
- convergence guaranteed via contraction.
- uvfa = same thing but conditioned on goals.

Bellman Equations (Derivations + Intuition)

recall the return

from step 1:

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

recursive property (just peel off the first term):

$$G_t = R_{t+1} + \gamma G_{t+1}.$$

this recursion is the foundation of the bellman equations.

State-value function under policy π

definition:

$$V^\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s].$$

substitute recursion for G_t:

$$V^\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s].$$

now split expectation:

$$V^\pi(s) = \mathbb{E}_\pi[R_{t+1} \mid S_t = s] + \gamma \mathbb{E}_\pi[G_{t+1} \mid S_t = s].$$

but notice:

$$\mathbb{E}_\pi[G_{t+1} \mid S_t = s] = \mathbb{E}_\pi[V^\pi(S_{t+1}) \mid S_t = s].$$

so:

$$V^\pi(s) = \mathbb{E}_{a\sim\pi,\, s'\sim P}[R(s,a,s') + \gamma V^\pi(s') \mid s].$$

Action-value function under policy π

definition:

$$Q^\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a].$$

apply recursion:

$Q^\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid s,a]$.

but G_{t+1} depends on next state s' and the future policy.

so:

$Q^\pi(s,a) = \sum_{s'} P(s' \mid s,a)\Big(\bar{r}(s,a,s') + \gamma \sum_{a'} \pi(a' \mid s') Q^\pi(s',a')\Big)$.

Relationship between v and q

since $V^\pi(s) =$ expected value of chosen action:

$V^\pi(s) = \sum_a \pi(a \mid s) Q^\pi(s,a)$.

Optimality equations

define optimal value functions:

$V^*(s) = \max_\pi V^\pi(s), \quad Q^*(s,a) = \max_\pi Q^\pi(s,a)$.

then:

optimal state-value:

$V^*(s) = \max_a \sum_{s'} P(s' \mid s,a)\big(\bar{r}(s,a,s') + \gamma V^*(s')\big)$.

optimal action-value:

$Q^*(s,a) = \sum_{s'} P(s' \mid s,a)\big(\bar{r}(s,a,s') + \gamma \max_{a'} Q^*(s',a')\big)$.

The operators (mathematical machinery)

policy evaluation operator

$(T^\pi V)(s) = \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s,a)[\bar{r}(s,a,s') + \gamma V(s')]$.

- contraction mapping with modulus γ\gamma.
- unique fixed point: V^π.
- iterative application: $V_{k+1} = T^\pi V_k \to V^\pi$.

optimality operator

$(T^* V)(s) = \max_a \sum_{s'} P(s' \mid s,a)[\bar{r}(s,a,s') + \gamma V(s')]$.

- contraction too.
- fixed point = V^*.
- iterative application: **value iteration**.

Intuition (plain words)

- bellman equations are basically saying:

  “the value of now = immediate reward + discounted value of tomorrow.”

- for optimality:

  "the best choice now = immediate reward + best possible tomorrow."

Tiny numerical example

mdp:

- states: s_1, s_2.
- actions: left, right.
- transition:
  - from s_1, action right $\rightarrow s_2$, reward +1.
  - from s_1, action left $\rightarrow s_1$, reward 0.
  - s_2 is terminal (value 0).
- $\gamma = 0.9$.

compute:

$Q^*(s_1, \text{right}) = 1 + 0.9 \cdot 0 = 1,$

$Q^*(s_1, \text{left}) = 0 + 0.9V^*(s_1).$

so:

$V^*(s_1) = \max\{1,\, 0.9V^*(s_1)\}.$

solve: if $V^*(s_1) = 1$, consistent.

so optimal policy: go right.

minimal code illustration

```
# value iteration (bellman optimality updates)
import numpy as np

states = ["s1","s2"]
actions = ["left","right"]

# transition: (next_state, reward)
P = {
    ("s1","right"): [("s2",1.0,1.0)],   # to s2, r=1
    ("s1","left"): [("s1",1.0,0.0)],    # to s1, r=0
    ("s2","right"): [("s2",1.0,0.0)],
    ("s2","left"): [("s2",1.0,0.0)],
}
gamma=0.9
V = {s:0 for s in states}

for it in range(10):
```

```
    newV={}
    for s in states:
        values=[]
        for a in actions:
            val=0
            for (s2,prob,r) in P[(s,a)]:
                val += prob*(r+gamma*V[s2])
            values.append(val)
        newV[s]=max(values)
    V=newV
    print(it,V)
```

summary

- bellman equations come from return recursion.
- **expectation form:** for fixed policy.
- **optimality form:** for best possible.
- contraction property guarantees unique solution.
- basis for dynamic programming, temporal-difference learning, q-learning, uvfa.

Learning Approaches

The bellman equations tell us what values should satisfy.

but how do we actually **compute or learn** them? (3 main families):

- **dynamic programming,**
- **monte carlo,**
- **temporal difference.**

Dynamic programming (DP)

- Assumes **we know the model (**: transition probabilities $P(s'|s,a)$ and expected rewards $\bar{r}(s,a)$.
- then we can compute value functions exactly (iteratively).

Policy evaluation (iterative)

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

- start with arbitrary V_0.
- repeatedly apply Bellman expectation operator T^π.
- converges to V^π.

Policy improvement

given V^π, construct new greedy policy:

$\pi'(s) \in \arg\max_a \sum_{s'} P(s'|s,a)\Big[\bar{r}(s,a,s') + \gamma V^\pi(s')\Big].$

Policy iteration

- alternate **policy evaluation** and **policy improvement** until stable.

Value iteration

- shortcut: directly update using Bellman optimality operator:

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

- faster, often used in textbooks.

limitation: requires full model, which is rarely available in practice.

Monte carlo (MC) methods

- no model needed.
- instead: learn values from **sampled episodes**.
- key: use **empirical returns**.

First-visit mc

for each state s, when it's first visited in an episode:

- compute actual return G_t from that timestep to episode end.
- average all such returns:

$V(s) \approx \frac{1}{N(s)} \sum_{i=1}^{N(s)} G^{(i)}.$

Every-visit mc

same but include *all* visits (not just first).

Action-values

- we can also estimate $Q(s,a)$ by averaging returns following first visit to pair (s,a).
- then choose actions $\pi(s) = \arg\max_a Q(s,a)$.

pros & cons

-
  - unbiased estimates.
- – need **complete episodes** (delayed updates).
- – high variance.

Temporal difference (TD) methods

- blend dp + mc ideas.
- update after **one step**, bootstrapping from existing estimates.
- no need to wait until episode ends.

td(0) for state values

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

- target $R_{t+1} + \gamma V(s_{t+1})$.
- error = **TD error**:

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

sarsa (on-policy td for q)

update using the action actually taken next:

$Q(s_t, a_t) \quad \leftarrow \quad Q(s_t, a_t) + \alpha\big(R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big).$

- learns the value of the **current policy** (including exploration).

q-learning (off-policy td)

update using the *best action* in next state:

$Q(s_t, a_t) \quad \leftarrow \quad Q(s_t, a_t) + \alpha\big(R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)\big).$

- learns the **optimal policy** regardless of behaviour policy.

pros & cons

-
  - incremental (online).
-
  - doesn't need model.
- – can be biased (bootstrapping).
- – stability issues with function approximation.

comparing approaches

Approach	Needs model?	Updates when?	Variance	Bias	Examples
DP	Yes	Iterative sweeps	0 (exact)	none	Value Iteration
MC	No	After episode	High	none	MC control
TD	No	After each step	Lower	Biased	SARSA, Q-learning

- **dp** = solve with equations (requires knowing everything).
- **mc** = “learn by playing full games and averaging results.”

- **td** = “update immediately using guesses of the future.”

modern rl (dqn, actor–critic, uvfa) is built on **td learning** with **function approximation**.

code - monte carlo (first-visit)

```
import collections

def mc_prediction(env, policy, episodes=1000, gamma=0.9):
    V = collections.defaultdict(float)
    returns = collections.defaultdict(list)
    for _ in range(episodes):
        # generate episode
        s = env.reset()
        traj = []
        done = False
        while not done:
            a = policy(s)
            s2, r, done, _ = env.step(a)
            traj.append((s,a,r))
            s = s2
        # compute returns backward
        G = 0
        visited = set()
        for t in reversed(range(len(traj))):
            s,a,r = traj[t]
            G = r + gamma*G
            if s not in visited:
                returns[s].append(G)
                V[s] = sum(returns[s])/len(returns[s])
                visited.add(s)
    return V
```

td(0) update

```
def td_update(V, s, r, s2, alpha=0.1, gamma=0.9):
    V[s] = V[s] + alpha * (r + gamma*V[s2] - V[s])
```

summary

- 3 families: dp (needs model), mc (episodes), td (step-by-step).
- td generalizes mc and dp.
- td error = “surprise” drives learning.
- q-learning = off-policy td that converges to optimal.

Tabular Algorithms (small/finite MDPs)

We assume:

- finite state set $\mathcal{S}$, finite action set $\mathcal{A}$
- episodic or continuing, discounted with $\gamma \in [0, 1)$
- we can store a table $Q(s, a)$ for all (s, a)

Core idea: learn **action values** $Q(s, a)$ from interaction. A **behaviour policy** selects actions; updates push Q toward Bellman targets.

TD error & targets (shared intuition)

At time t, after $(s_t, a_t, r_{t+1}, s_{t+1})$ we form a **target** y_t and update:

$$\delta_t \;=\; y_t - Q(s_t, a_t), \qquad Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t\, \delta_t.$$

- $\alpha_t \in (0, 1]$ is stepsize (learning rate).
- The **only** difference between algorithms is how y_t is defined.

SARSA — on-policy TD control

Name: State–**A**ction–**R**eward–**S**tate–**A**ction (all five appear in the update).

- You learn the value of **the policy you actually follow**, including its exploration.
- After you get to s_{t+1}, you also choose the **next action** $a_{t+1} \sim \pi(\cdot|s_{t+1})$ (the same behavior policy).

  Target:

  $$y_t \;=\; r_{t+1} + \gamma\, Q(s_{t+1}, a_{t+1}).$$

- Update:

  $$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \Big(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \Big).$$

Policy improvement

Use an ε-greedy policy w.r.t. current Q:

$$\pi(a|s) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}|} & \text{if } a \in \arg\max_{a'} Q(s, a'), \\ \frac{\varepsilon}{|\mathcal{A}|} & \text{otherwise.} \end{cases}$$

As Q improves, this gradually makes the greedy actions more likely.

Convergence (sketch)

If the policy is **GLIE** (Greedy in the Limit with Infinite Exploration):

- **Infinite exploration:** every (s, a) is visited infinitely often (e.g., $\sum_t \varepsilon_t = \infty$ with $\varepsilon_t \to 0$).
- **Greedy in the limit:** $\varepsilon_t \to 0$ so the behaviour becomes greedy.

With suitable step sizes α_t satisfying Robbins–Monro conditions $(\sum_t \alpha_t = \infty, \sum_t \alpha_t^2 < \infty)$, SARSA converges **with probability 1** to the **optimal** action-values Q^* in finite MDPs.

> Intuition: SARSA tracks the Bellman operator of the current ε-greedy policy and, as $\varepsilon \to 0$, that policy tends to greedy $\Rightarrow$ fixed point tends to Q^*.

Q-learning — off-policy TD control

- You behave with some exploratory policy (e.g.,ε greedy), **but** learn values for the **greedy target policy**.
- Target uses the **greedy action** at the next state, not the sampled action:

  $y_t \;=\; r_{t+1} + \gamma \; \max_{a'} Q(s_{t+1}, a').$
- Update:

  $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \Big(r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \Big).$

Off-policy meaning

- **Behaviour policy:** what you use to act (e.g., ε greedy).
- **Target policy:** the greedy policy $\arg\max_a Q(s, a)$ (what you learn **about**).

  Q-learning's backup ignores the behaviour's next action and bootstraps to the **max** over actions.

Convergence (sketch)

For finite MDPs, if

- all (s, a) are visited infinitely often (persistent exploration), and
- stepsizes satisfy Robbins–Monro,

  then Q-learning converges **with probability 1** to Q^*.

> Intuition: the update is a stochastic approximation to the Bellman optimality operator T^*Q.That operator is a γ-contraction, so iterates converge to its unique fixed point Q^*.

On-policy vs Off-policy: when to use what?

- **SARSA (on-policy):**
  - Safer with risky exploration (target includes the exploratory action you will actually take).
  - Often preferred in environments where exploratory actions can lead to catastrophic states you want to value lower.
- **Q-learning (off-policy):**
  - More aggressive/impatient; learns the **best-case** value for the next step.
  - Can overestimate due to the $\max$ over noisy estimates (see Practical pitfalls & fixes).

Tiny numeric walk-through

Consider a tiny continuing MDP:

- Two states s_A, s_B, two actions L, R.
- From s_A :
  - R: go to s_B, reward $+1$.
  - L: stay in s_A, reward .
- From s_B : self-loop with reward 0.
- $\gamma = 0.9$. Initialize $Q = 0$.

Suppose first transition is $s_A, a = R, r = +1, s_B$.

- **Q-learning** target:

  $y = 1 + 0.9 \cdot \max\{Q(s_B, L), Q(s_B, R)\} = 1 + 0.9 \cdot 0 = 1.$

  Update $Q(s_A, R) \leftarrow 0 + \alpha(1 - 0) = \alpha$.
- **SARSA** target needs next action a' at s_B. If ε greedy picks $a' = L$ (say):

  $y = 1 + 0.9 \cdot Q(s_B, L) = 1.$

  First update identical. Differences appear when the **next** state has nonzero Q and SARSA uses $Q(s', a')$ vs Q-learning uses $\max_{a'} Q(s', a')$.

Pseudocode you can reuse

SARSA (episodic)

```
initialize Q[s,a] arbitrarily
for each episode:
    s = env.reset()
    a = epsilon_greedy(Q[s, :], eps)
    done = False
    while not done:
        s2, r, done, info = env.step(a)
        a2 = epsilon_greedy(Q[s2, :], eps)
        td_target = r + gamma * Q[s2, a2] * (0 if done else 1)
        Q[s, a] += alpha * (td_target - Q[s, a])
        s, a = s2, a2
    # (optional) decay eps
```

Q-learning (episodic)

```
initialize Q[s,a] arbitrarily
for each episode:
    s = env.reset()
```

```
    done = False
    while not done:
        a = epsilon_greedy(Q[s, :], eps)
        s2, r, done, info = env.step(a)
        td_target = r + gamma * (0 if done else np.max(Q[s2, :]))
        Q[s, a] += alpha * (td_target - Q[s, a])
        s = s2
    # (optional) decay eps
```

Good defaults (tabular):

- $\alpha \in [0.1, 0.5]$ (can decay slowly).
- $\varepsilon_t = \max(\varepsilon_{\min}, \varepsilon_0 \cdot \mathrm{decay}^k)$ or $\varepsilon_t = c/(c+t)$.
- Ensure sufficient exploration early; reduce later.

Practical pitfalls & fixes (important!)

- **Maximisation bias (Q-learning):**

  $\max_{a'} Q(s', a')$ over **noisy** estimates inflates targets.

  Fix: Double Q-learning — maintain Q^A, Q^B and decouple argmax and evaluation:

  $y = r + \gamma\, Q^B\big(s', \arg\max_a Q^A(s', a)\big)$

  (and swap roles stochastically).

- **Non-stationarity from ε-greedy decay:**

  If you decay ε too fast, you may stop exploring before learning stabilizes. Keep a floor (e.g., $\varepsilon_{\min} = 0.05$).

- **Learning rate too high:**

  Can cause divergence/noise. Use a small constant or a diminishing schedule.

- **Terminal handling:**

  When s_{t+1} is terminal, set the bootstrap term to 0:

  $y_t = r_{t+1}$.

- **Stochastic transitions:**

  Variance in targets is normal; more episodes smooth it out.

Convergence conditions (clean statements)

Let the MDP be finite, $\gamma < 1$. Suppose:

- Every state–action pair is visited infinitely often (**persistent exploration**).
- Step sizes $\alpha_t(s, a)$ satisfy Robbins–Monro:

  $\sum_t \alpha_t(s, a) = \infty$ and $\sum_t \alpha_t^2(s, a) < \infty$.

Then:

- **Q-learning** updates converge w.p.1 to Q^*.
- **SARSA** with a **GLIE** policy (e.g., $\varepsilon_t \to 0$ but $\sum_t \varepsilon_t = \infty$) converges w.p.1 to Q^*.

(Proof skeletons rely on stochastic approximation to a contraction mapping's fixed point.)

Eligibility traces (bonus: SARSA(λ))

Bridges MC and TD by backing up **multi-step returns**:

- Maintain trace $e(s, a)$ that decays:
  $e_t(s, a) = \gamma\lambda e_{t-1}(s, a) + \mathbf{1}\{s_t = s, a_t = a\}.$
- Update **all** visited pairs proportionally to their trace:
  $Q \leftarrow Q + \alpha\, \delta_t\, e.$
- $\lambda = 0 \to$ SARSA(0) (pure TD); $\lambda = 1 \to$ MC-like.
  Often speeds learning substantially.

What you should be able to do now

- Write SARSA and Q-learning updates **from memory**.
- Explain **on-policy vs off-policy** in one sentence.
- State the **convergence conditions** and why max\max causes bias.
- Implement either algorithm on a gridworld or bandit-like toy and see learning curves.

Function Approximation (from first principles to DQN family)

Why approximate?

- tabular $Q(s, a)$ is $|\mathcal{S}| \times |\mathcal{A}|$ numbers → impossible when $|\mathcal{S}|$ is huge (images, sensors).
- idea: represent V, Q, π with a **parametric function** $f_\theta(\cdot)$ with shared parameters θ that generalize across states.

Common choices:

- **Linear:** $Q_\theta(s, a) = \phi(s, a)^\top \theta.$
- **Neural nets (MLP/CNN/RNN):** $Q_\theta(s, a) = \mathrm{NN}_\theta([s, a])$ or $Q_\theta(s, \cdot)$ outputs all actions at once.

Supervised view of TD learning

TD learning creates **training targets** from interaction.

For Q-learning style backups:

$y_t \;=\; r_{t+1} + \gamma \max_{a'} Q_{\bar{\theta}}(s_{t+1}, a') \cdot \mathbf{1}\{\text{not terminal}\}$

Then minimize squared error:

$$\mathcal{L}_t(\theta) \ = \ \big(Q_\theta(s_t, a_t) - y_t\big)^2.$$

Stochastic gradient step:

$$\theta \leftarrow \theta - \alpha \, \nabla_\theta \mathcal{L}_t(\theta) = \theta - 2\alpha\big(Q_\theta - y_t\big) \, \nabla_\theta Q_\theta(s_t, a_t).$$

Key: target y_t depends on a **stale/slow** parameter set $\bar{\theta}$ to avoid chasing a moving target (target networks; SEE target networks 6.4).

The deadly triad (why naive deep TD can diverge)

Combining these three is risky:

1. **Function approximation** (NNs)
2. **Bootstrapping** (target uses own predictions V/Q)
3. **Off-policy** learning (behavior $\neq$ target policy)

Together → instability/divergence (can blow up even on simple problems). DQN adds engineering **stabilisers**.

Experience replay (stabiliser #1)

Maintain a buffer $\mathcal{D}$ of past transitions $(s, a, r, s', \text{done})$.

- Train on **uniform random mini-batches** from $\mathcal{D}$ ⇒ breaks temporal correlations, improves sample efficiency.
- Implements off-policy reuse of data.
- Typical sizes: 10510^5–10610^6. Warm-up before training.

Target computation per sampled transition:

$$y = r + \gamma(1 - \text{done}) \ \max_{a'} Q_{\bar{\theta}}(s', a').$$

Target networks (stabiliser #2)

Keep a **separate, slowly-updated** copy of the network $Q_{\bar{\theta}}$ (the "target network").

- **Hard update:** every C steps: $\bar{\theta} \leftarrow \theta$.
- **Soft update (Polyak):** $\bar{\theta} \leftarrow \tau\theta + (1 - \tau)\bar{\theta}$ with $\tau \ll 1$.

Purpose: make the target yy **semi-stationary** so optimisation doesn't chase itself.

The DQN algorithm (canonical)

Model: a NN that takes state s and outputs a vector $Q_\theta(s, \cdot) \in \mathbb{R}^{|\mathcal{A}|}$.

Loss (per batch):

$$\mathcal{L}(\theta) = \tfrac{1}{B} \textstyle\sum_{i=1}^{B} \big(Q_\theta(s_i, a_i) - y_i\big)^2, \quad y_i = r_i + \gamma(1 - d_i) \max_{a'} Q_{\bar{\theta}}(s_i', a').$$

Training loop (sketch):

1. Interact with env using ε greedy from Q_θ.

2. Store transitions in replay $\mathcal{D}$.
3. Sample mini-batch from $\mathcal{D}$.
4. Compute targets y with **target network** $Q_{\bar{\theta}}$.
5. Backprop MSE; update θ.
6. Periodically update $\bar{\theta}$.

Stability extras (common in practice):

- reward **clipping** to $[-1, 1]$ for Atari.
- gradient **clipping** (e.g., global norm).
- state **normalisation** (e.g., pixel preprocessing: grayscale, downscale, stack frames).

Double DQN (fix overestimation bias)

Q-learning target uses $\max_{a'}$ over **noisy** estimates ⇒ optimistic bias.

Double DQN decouples **argmax** and **evaluation**:

$$a^\star \;=\; \arg\max_{a'} Q_\theta(s', a') \quad \text{(online net for selection)}$$
$$y \;=\; r + \gamma(1-d)\, Q_{\bar{\theta}}(s', a^\star) \quad \text{(target net for evaluation)}$$

This significantly reduces bias and improves stability.

Dueling networks (better value/advantage structure)

Parameterize:

$$Q_\theta(s,a) = V_\theta(s) + \Big(A_\theta(s,a) - \tfrac{1}{|\mathcal{A}|}\sum_{a'} A_\theta(s,a')\Big).$$

- Separate streams for **state value** $V(s)$ and **advantages** $A(s, a)$.
- Helps learning useful state values even when action effects are subtle (e.g., many similar actions).

Prioritized replay (learn more from “surprising” samples)

Sample transitions with probability proportional to **priority** $p_i = |\delta_i|^\alpha$ where TD error $\delta_i = y_i - Q_\theta(s_i, a_i)$.

Use **importance-sampling (IS) weights** to correct bias:

$$w_i = \Big(\tfrac{1}{N}\,\tfrac{1}{P(i)}\Big)^\beta, \quad \text{normalize } w_i \leftarrow \tfrac{w_i}{\max_j w_j}.$$

Anneal β∈[0,1]\beta \in [0,1] from small to large over training.

n-step returns (multi-step targets)

Trade bias/variance by bootstrapping after nn steps:

$$y^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+1+k} \;+\; \gamma^n(1-d_{t+n}) \max_{a'} Q_{\bar{\theta}}(s_{t+n}, a').$$

Faster propagation of rewards; common in Rainbow / Ape-X DQN.

Distributional RL (optional but powerful)

Instead of predicting the **mean** Q, predict distribution of returns $Z(s, a)$.

- **C51:** categorical distribution with fixed atoms, cross-entropy loss with projection.
- **QR-DQN:** quantile regression for return quantiles.

  Often improves sample efficiency and exploration.

Stability & implementation checklist

Architecture

- For low-dim states: 2–3 layer MLP (e.g., 256–512 units, ReLU/SiLU).
- For images: CNN backbone (e.g., DQN conv stack) + MLP head.

Initialization

- Fan-in (Kaiming/He) for ReLU; orthogonal for linear layers is robust.
- Output layer small init helps prevent huge Q-values initially.

Normalization

- If observations vary in scale, standardize/normalize inputs.
- For pixels: divide by 255, optionally mean-subtract.

Targets

- detach target y from graph; use **target network**.
- handle terminals: zero bootstrap when done.

Optimization

- Adam with lr in $[1e{-}4, 3e{-}4]$ typical.
- Batch size 32–256.
- Gradient norm clip (e.g., 10.0).

Exploration

- ε greedy schedule: e.g., from $1.0 \to 0.1$ over first $1e6$ frames (Atari-style), or faster for small tasks.
- For continuous actions, use actor–critic (DDPG/SAC/TD3) rather than DQN.

Replay

- Capacity: $1e5$–$1e6$.
- Start learning after warm-up (e.g., $1e4$ steps).
- Sample uniformly or prioritized (with IS correction).

Loss scaling

- Huber loss $\ell_\kappa(x)$ instead of MSE is common: less sensitive to outliers.

Precise DQN/Double DQN training pseudocode

```
# Q: s -> R^{|A|}; target Qbar (same arch)
Q = QNetwork()
Qbar = deepcopy(Q)

replay = ReplayBuffer(capacity=1_000_000)
eps = eps_start  # e.g., 1.0
optimizer = Adam(Q.parameters(), lr=3e-4)
gamma = 0.99
tau = 0.005  # if soft updates; else use hard update every C steps
step = 0

s = env.reset()
while step < max_steps:
    # 1) behave epsilon-greedy
    if random.random() < eps:
        a = random_action()
    else:
        with torch.no_grad():
            a = Q(s).argmax().item()

    s2, r, done, info = env.step(a)
    replay.add(s, a, r, s2, done)
    s = s2 if not done else env.reset()
    step += 1

    # decay exploration
    eps = max(eps_end, eps - (eps_start-eps_end)/eps_decay_steps)

    # 2) learn after warm-up
    if len(replay) < batch_size or step < warmup_steps:
        continue

    batch = replay.sample(batch_size)
    s_b, a_b, r_b, s2_b, d_b = batch  # tensors

    with torch.no_grad():
        # Double DQN target:
        a_star = Q(s2_b).argmax(dim=1, keepdim=True)
        y = r_b + gamma * (1 - d_b) * Qbar(s2_b).gather(1, a_star).sque
eze(1)

    q_pred = Q(s_b).gather(1, a_b).squeeze(1)
    loss = F.smooth_l1_loss(q_pred, y)  # Huber
```

```
    optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_norm_(Q.parameters(), 10.0)
    optimizer.step()

    # 3) update target network
    # soft:
    for p, p_bar in zip(Q.parameters(), Qbar.parameters()):
        p_bar.data.mul_(1 - tau).add_(tau * p.data)
    # (or hard: if step % C == 0: Qbar.load_state_dict(Q.state_dict()))
```

When DQN isn't the right tool

- **Continuous action spaces** (steering angle, torque): argmax over actions is undefined. Prefer:
  - **DDPG/TD3** (deterministic actor–critic with target policy smoothing in TD3)
  - **SAC** (stochastic actor–critic with entropy regularization; very robust)
- **High-dimensional partially observable tasks**: add recurrence (DRQN) or frame stacking; consider **transformers** for longer memory.

Common failure modes & diagnostics

- **Divergence / NaNs**: check learning rate, reward scale, gradient clip, target update cadence.
- **Q-value explosion**: clip rewards, reduce lr, use Huber loss, ensure target network is slow.
- **No learning**: exploration too low; replay too small; not enough warm-up; incorrect terminal handling.
- **Overestimation**: use **Double DQN**.
- **Action-insensitive states**: consider **Dueling**.

Tying back to UVFA

Everything above extends by **conditioning on the goal** g:

- Inputs become$[s, g]$ (concat or more sophisticated fusion).
- Targets keep the **same goal** in the bootstrap:
  $y = r + \gamma(1-d)\ \max_{a'} Q_{\bar{\theta}}(s', a', g).$
- Double DQN, dueling, prioritized replay, n-step targets → **all apply** unchanged, just feed gg.

you should now be able to

- write the DQN loss and its gradient
- explain **why** target networks and replay stabilize learning
- implement Double DQN (selection/evaluation split)
- choose between DQN / DDQN / Dueling / Prioritized / n-step variants

- diagnose instability and fix it

Policy Gradients (from scratch)

setting

- Stochastic policy $\pi_\theta(a \mid s)$ (differentiable in θ).
- Objective (discounted, start-state distribution ρ_0):

$J(\theta) = \mathbb{E}_{\tau \sim p_\theta}\Big[G_0\Big], \quad p_\theta(\tau) = \rho_0(s_0) \prod_{t \geq 0} \pi_\theta(a_t \mid s_t) P(s_{t+1} \mid s_t, a_t).$

We want $\nabla_\theta J(\theta)$

Likelihood-ratio (score function) trick

Use: $\nabla_\theta \log p_\theta(\tau) = \sum_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)$(dynamics cancel since P doesn't depend on θ).

$\nabla_\theta J = \nabla_\theta \int p_\theta(\tau) G_0 \, d\tau = \int p_\theta(\tau) \, \nabla_\theta \log p_\theta(\tau) \, G_0 \, d\tau = \mathbb{E}_{\tau \sim p_\theta}\Big[\Big(\sum_{t \geq 0} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\Big) \, G_0\Big].$

Causality lets us replace G_0 by **future return from** t:

$\nabla_\theta J = \mathbb{E}\Big[\sum_{t \geq 0} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \, G_t\Big].$

This is the **REINFORCE** gradient.

State-action value form (policy gradient theorem)

Define discounted (unnormalized) visitation $d_\theta(s) = (1-\gamma) \sum_{t \geq 0} \gamma^t \Pr(S_t = s \mid \theta)$. Then:

$\nabla_\theta J(\theta) = \frac{1}{1-\gamma} \, \mathbb{E}_{s \sim d_\theta, \, a \sim \pi_\theta}\big[\nabla_\theta \log \pi_\theta(a \mid s) \, Q^{\pi_\theta}(s,a)\big].$

Sketch: Write gradient with G_t, condition on (s_t, a_t), take expectation over futures $\rightarrow Q^\pi(s_t, a_t)$, and convert time-sum to d_θ.

Baselines (variance reduction)

We can subtract any **state-dependent** baseline $b(s)$ without bias:

$\mathbb{E}\big[\nabla_\theta \log \pi_\theta(a \mid s) \, b(s)\big] = 0.$

Proof (one-line):

$\sum_a \pi_\theta(a \mid s) \nabla_\theta \log \pi_\theta(a \mid s) b(s) = b(s) \nabla_\theta \sum_a \pi_\theta(a \mid s) = b(s) \nabla_\theta 1 = 0.$

Thus replace QπQ^\pi with **advantage** $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$ using $b(s) = V^\pi(s)$:

$\nabla_\theta J(\theta) = \frac{1}{1-\gamma} \, \mathbb{E}\big[\nabla_\theta \log \pi_\theta(a \mid s) \, A^\pi(s,a)\big].$

Advantages shrink variance dramatically.

REINFORCE (Monte Carlo policy gradient)

- Use trajectory returns as unbiased estimates:

$g(\theta) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \left(G_t - b(s_t)\right).$

- Common baseline: a learned value function$V_w(s)$ (the "critic").

Pros: unbiased, simple.

Cons: high variance; updates only after episode (unless using partial returns).

Actor–critic (TD critic; lower variance)

Learn a critic $V_w(s) \approx V^\pi(s)$ (or QwQ_w) with TD:

$\delta_t = r_{t+1} + \gamma V_w(s_{t+1}) - V_w(s_t)$

Actor update uses **advantage estimate**:

$\Delta\theta \propto \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, \hat{A}_t, \quad \hat{A}_t \in \{\delta_t,\ n\text{-step},\ \text{GAE}\}.$

Critic update minimises $(\delta_t)^2$ (or MSE to VV-targets).

This is the foundation of **A2C/A3C**.

Generalised advantage estimation (GAE-λ)

Balance bias–variance by exponentially weighting multi-step TD residuals:

$\hat{A}_t^{(\text{GAE-}\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l\, \delta_{t+l}, \quad \delta_t = r_{t+1} + \gamma V_w(s_{t+1}) - V_w(s_t).$

- $\lambda = 0$: one-step TD (low var, higher bias).
- $\lambda = 1$: MC-style long returns (low bias, high var).
- Typical: $\lambda \in [0.9, 0.97]$.

Entropy regularisation (encourage exploration)

Augment objective with policy entropy:

$J_{\text{ent}}(\theta) = J(\theta) + \beta\, \mathbb{E}_s[\mathcal{H}(\pi_\theta(\cdot \mid s))], \quad \mathcal{H} = -\sum_a \pi \log \pi.$

Actor gradient gets an extra $\beta\, \nabla_\theta \mathcal{H}$ term → prevents premature collapse.

Natural policy gradient (Fisher preconditioning)

Steepest ascent under KL geometry:

$\theta \leftarrow \theta + \alpha\, F(\theta)^{-1}\, g, \quad g = \nabla_\theta J,\ F = \mathbb{E}[\nabla \log \pi\, \nabla \log \pi^\top].$

- In practice: **TRPO** approximates this with a constrained step $\max_\theta \hat{J}(\theta)$ s.t. $\mathbb{E}[\text{KL}(\pi_{\theta_{\text{old}}} \| \pi_\theta)] \le \delta.$
- **PPO** simplifies with a clipped surrogate (below).

PPO (practical, robust actor–critic)

Define probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}.$

Clipped surrogate objective:

$$L^{\mathrm{CLIP}}(\theta) = \mathbb{E}\Big[\min\big(r_t(\theta)\,\hat{A}_t,\ \mathrm{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\,\hat{A}_t\big)\Big] \ -\ c_v\,\mathrm{MSE}(V_w(s_t), \hat{V}_t)\ +\ c_H\,\mathcal{H}(\pi_\theta(\cdot \mid s_t)).$$

- $\epsilon \in [0.1, 0.3]$ (trust region proxy).
- Optimized with minibatch SGD over a fixed set of trajectories ("epochs").

Deterministic policy gradients (for continuous actions)

Deterministic policy $\mu_\theta(s)$.

Objective $J = \mathbb{E}_{s\sim d^\mu}[Q^\mu(s, \mu_\theta(s))]$.

Gradient (DPG theorem):

$$\nabla_\theta J = \mathbb{E}_{s\sim d^\mu}\left[\nabla_\theta \mu_\theta(s)\,\nabla_a Q^\mu(s,a)\big|_{a=\mu_\theta(s)}\right].$$

Leads to **DDPG/TD3**; add target nets, noise for exploration, policy smoothing (TD3).

Common parameterizations & gradients

Discrete actions (softmax head):

$$\pi_\theta(a \mid s) = \frac{\exp(z_\theta(s)_a)}{\sum_b \exp(z_\theta(s)_b)}, \quad \nabla_\theta \log \pi_\theta(a \mid s) = \nabla_\theta z_\theta(s)_a - \sum_b \pi_\theta(b \mid s)\nabla_\theta z_\theta(s)_b.$$

Gaussian policy (continuous):

$$\pi_\theta(a \mid s) = \mathcal{N}\big(\mu_\theta(s), \sigma_\theta^2(s) I\big), \quad \nabla_\theta \log \pi = \frac{(a-\mu_\theta)}{\sigma_\theta^2}\nabla_\theta \mu_\theta\ +\ \left(\frac{(a-\mu_\theta)^2}{\sigma_\theta^2} - 1\right)\nabla_\theta \log \sigma_\theta.$$

(Implement with tanh squashing + log-prob correction as needed.)

Variance, bias, and credit assignment

- **MC (REINFORCE):** unbiased but high variance (all future rewards attributed equally to a_t).
- **TD critics:** introduce bias via bootstrap but cut variance.
- **GAE:** tunable bias–variance.
- **Normalization:** standardize $\hat{A}_t$ per batch (zero mean, unit std) to stabilize.

putting it together — minimal actor–critic (A2C-style)

```
# one step of on-policy actor-critic with GAE and PPO-like batching (co
nceptual)
obs_buf, act_buf, rew_buf, val_buf, logp_buf, done_buf = collect_rollou
t(env, pi, V, horizon)
# compute targets
vals = V(obs_buf)
with torch.no_grad():
    adv = compute_gae(rew_buf, vals, done_buf, gamma=0.99, lam=0.95)  #
GAE-λ
```

```
    ret = adv + vals
# normalize advantages
adv = (adv - adv.mean()) / (adv.std() + 1e-8)

for epoch in range(update_epochs):
    for batch in minibatches(obs_buf, act_buf, ret, adv, logp_buf):
        logp_new, entropy = pi.logp_and_ent(batch.obs, batch.act)   # s
tochastic policy
        ratio = torch.exp(logp_new - batch.logp_old)
        # PPO clipped surrogate
        unclipped = ratio * batch.adv
        clipped = torch.clamp(ratio, 1-eps, 1+eps) * batch.adv
        actor_loss = -torch.mean(torch.min(unclipped, clipped)) - cH*to
rch.mean(entropy)
        value_loss = F.mse_loss(V(batch.obs), batch.ret)
        loss = actor_loss + cV*value_loss
        optimizer.zero_grad(); loss.backward(); clip_grad_norm_(params,
max_norm); optimizer.step()
```

sanity checks you should pass

- derive $\nabla_\theta J$ via likelihood ratio without looking.
- prove the **baseline trick** in one line.
- explain **why** $\nabla \log \pi$ makes dynamics P vanish from the gradient.
- write the **actor** update with advantage, and the **critic** TD loss.
- explain GAE and the γ, λ roles.
- state what PPO's clip does (prevents destructive large policy updates).

how this connects to UVFA

- Replace s with (s, g).
- Actor: $\pi_\theta(a \mid s, g)$.
- Critic: $V_w(s, g)$ or $Q_w(s, a, g)$.
- Advantage, TD error, GAE computed **conditioned on the same goal gg**.
- Off-policy goal relabeling (e.g., HER) can further improve sample efficiency.

takeaway

- Policy gradients optimize behaviour **directly** via $\nabla_\theta \log \pi \times \text{advantage}.$
- Baselines/critics reduce variance; GAE balances bias–variance.
- PPO/TRPO stabilize updates; DPG/TD3/SAC handle continuous actions.
- The same machinery applies to **goal-conditioned** settings for UVFA/GC-actor–critic.

Actor–Critic (the practical workhorse)

Big picture

- **Actor**: a differentiable policy $\pi_\theta(a \mid s)$ you want to improve.
- **Critic**: a differentiable value estimator (either $V_w(s)$ or $Q_w(s,a)$) that tells the actor **how good** its choices are.
- **Update logic** (one line):

$\underbrace{\Delta\theta}_{\text{actor}} \propto \underbrace{\nabla_\theta \log \pi_\theta(a_t \mid s_t)}_{\text{sensitivity}} \cdot \underbrace{\hat{A}_t}_{\text{“how much better than usual”}}, \quad \underbrace{\Delta w}_{\text{critic}} \propto \nabla_w \tfrac{1}{2}(\text{TD target} - \text{prediction})^2.$

Which critic? V critic vs Q critic

V critic (most common, on-policy)

- Critic predicts Vw(s)≈Vπ(s)V_w(s)\approx V^\pi(s).
- Advantage estimates:
  - **1-step TD**: $\hat{A}_t = r_{t+1} + \gamma V_w(s_{t+1}) - V_w(s_t) = \delta_t$.
  - $n-$**step**: $\hat{A}_t = \sum_{k=0}^{n-1} \gamma^k r_{t+1+k} + \gamma^n V_w(s_{t+n}) - V_w(s_t)$.
  - **GAE(λ)**: $\hat{A}_t = \sum_{l \geq 0} (\gamma\lambda)^l \delta_{t+l}$ (best bias–variance tradeoff).

Q critic (common in off-policy / continuous action)

- Critic predicts $Q_w(s,a) \approx Q^\pi(s,a)$.
- Actor update uses either:
  - **Stochastic policy**: $\nabla_\theta J = \mathbb{E}\big[\nabla_\theta \log \pi_\theta(a \mid s)\, Q_w(s,a)\big]$.
  - **Deterministic policy** $\mu_\theta(s)$ (DPG/TD3/DDPG):
    $\nabla_\theta J = \mathbb{E}_{s\sim d^\mu}\big[\nabla_\theta \mu_\theta(s)\, \nabla_a Q_w(s,a)\big|_{a=\mu_\theta(s)}\big]$.

Critic learning (TD losses)

Value critic (V)

- **TD error**: $\delta_t = r_{t+1} + \gamma(1-d_{t+1}) V_w(s_{t+1}) - V_w(s_t)$.
- **Loss**: $\mathcal{L}_V(w) = \frac{1}{2}\,\mathbb{E}[\delta_t^2]$ (or MSE to nnstep/GAE targets).

Action-value critic (QQ)

- **Target** (on-policy, SARSA-style): $y_t = r_{t+1} + \gamma(1-d_{t+1})\,\mathbb{E}_{a'\sim\pi}[Q_w(s_{t+1}, a')]$.

- **Target** (off-policy, DDPG/TD3): $y_t = r_{t+1} + \gamma(1 - d_{t+1})\, Q_{\bar{w}}(s_{t+1}, \mu_{\bar{\theta}}(s_{t+1}))$ with slow/target nets.
- **Loss**: $L_Q(w) = \frac{1}{2}\,\mathbb{E}[(Q_w(s_t, a_t) - y_t)^2]$.

> Always stop gradient through targets; use target networks for stability in off-policy.

Actor learning (advantages + regularisation)

- **Core actor loss (maximize surrogate)**:

$$\mathcal{J}(\theta) = \mathbb{E}\big[\log \pi_\theta(a_t \mid s_t)\, \hat{A}_t\big].$$

- **Entropy bonus** for exploration: $\beta\,\mathbb{E}[\mathcal{H}(\pi_\theta(\cdot \mid s))]$.
- **Final loss to minimize** (sign flip):

$$\mathcal{L}_{\text{actor}}(\theta) = -\mathbb{E}\big[\log \pi_\theta(a_t \mid s_t)\, \hat{A}_t\big] - \beta\,\mathbb{E}[\mathcal{H}(\pi_\theta(\cdot \mid s))].$$

Advantage normalization (per batch) stabilizes:

$$\hat{A} \leftarrow (\hat{A} - \text{mean})/(\text{std} + 10^{-8}).$$

A2C / A3C: canonical on-policy actor–critic

A2C (Advantage Actor–Critic, synchronous)

- Collect TT steps from NN parallel envs.
- Build **nnstep / GAE** targets.
- Optimize actor & critic on that batch.

A3C (asynchronous)

- Many workers run envs and update a shared set of params asynchronously (Hogwild-style).

Typical combined loss:

$$\mathcal{L}(\theta, w) = \underbrace{-\mathbb{E}[\log \pi_\theta(a \mid s)\, \hat{A}]}_{\text{actor}} + \; c_v\, \underbrace{\mathbb{E}[(V_w(s) - \hat{V})^2]}_{\text{critic}} - \; c_H\, \underbrace{\mathbb{E}[\mathcal{H}(\pi_\theta(\cdot \mid s))]}_{\text{entropy}}$$

with $c_v \in [0.25, 1]$, $c_H \in [10^{-3}, 10^{-2}]$ as starting points.

nn-step return for $\hat{V}$:

$$\hat{V}_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+1+k} + \gamma^n (1 - d_{t+n}) V_w(s_{t+n}).$$

PPO/TRPO as stabilised actor–critic (quick tie-in)

- **TRPO**: maximize surrogate $\mathbb{E}[\frac{\pi_\theta}{\pi_{\text{old}}}\hat{A}]$ under a **KL trust region** constraint → natural-gradient-like step.
- **PPO**: replace constraint with **clipped ratio**; same actor–critic ingredients (critic loss + entropy).

  You already saw the PPO objective in section “Policy Gradients”; here it’s “just” a safer actor **wrapper** around A2C-style updates.

Compatible function approximation & natural actor–critic (theory gem)

If you choose a Q-critic of the **compatible** form

$Q_w(s,a) \;=\; w^\top \nabla_\theta \log \pi_\theta(a \mid s),$

and fit ww by least squares to the true advantage, then the actor update

$\Delta\theta \propto \mathbb{E}[\nabla_\theta \log \pi \, Q_w]$

equals a **natural gradient** step:

$\Delta\theta \;=\; F(\theta)^{-1}\, \nabla_\theta J(\theta), \quad F = \mathbb{E}\big[\nabla \log \pi \, \nabla \log \pi^\top\big]$

("Fisher" matrix).

This explains why well-trained critics can give **geometry-aware** policy updates.

Continuous actions: DDPG / TD3 (deterministic actor–critic)

- **Actor** outputs $\mu_\theta(s) \in \mathbb{R}^d$.
- **Critic** learns $Q_w(s,a)$.
- **Targets** with slow nets $\bar\theta, \bar w$.
- **Exploration** via action noise (OU or clipped Gaussian).
- **TD3** fixes DDPG instabilities:
  - **Target policy smoothing** (noise on target action),
  - **Clipped double** Q (min of two critics),
  - **Delayed policy updates** (update actor slower than critic).

Practical actor–critic recipe (discrete actions, on-policy A2C/PPO-style)

Data collection

1. Run π_θ for T steps (optionally NN envs in parallel).
2. Store $(s_t, a_t, r_{t+1}, s_{t+1}, d_{t+1}, \log \pi_\theta(a_t|s_t), V_w(s_t))$.

Targets

- Compute GAE: $\delta_t = r_{t+1} + \gamma(1-d_{t+1})V_w(s_{t+1}) - V_w(s_t)$;
  $\hat A_t = \sum_{l\ge 0} (\gamma\lambda)^l \delta_{t+l}$;
  $\hat V_t = \hat A_t + V_w(s_t)$.
- Normalize $\hat A$ in batch.

Optimization

- Actor loss: $-\mathbb{E}[\log \pi_\theta(a_t|s_t)\, \hat A_t] - c_H\, \mathbb{E}[\mathcal{H}]$.
- Critic loss: $c_v\, \mathbb{E}[(V_w(s_t) - \hat V_t)^2]$.

- Sum, backprop, gradient clip (e.g., 0.5–1.0), Adam (e.g., 3×10−43\times 10^{-4}).

Hyperparameters (good starting points)

- $\gamma = 0.99, \lambda = 0.95.$
- Entropy coef $c_H \in [0.001, 0.02]$ (anneal down).
- Critic coef $c_v \in [0.25, 1]$.
- Batch horizon $T = 128 \dots 2048$.
- PPO clip $\epsilon = 0.1 \dots 0.2$, 4–10 epochs, minibatch size 64–256.

Failure modes & fixes (checklist)

- **Critic lag / collapse** → Actor follows bad advantages.
  - Increase critic capacity; raise c_v; more TD steps (n-step/GAE); lower lr.
- **Entropy too low (early collapse)** → Poor exploration.
  - Increase c_H, add entropy schedule, larger batches.
- **High variance advantages** → Unstable actor loss.
  - Normalize $\hat{A}$, use GAE, gradient clip.
- **Over-fitting critic** → Low train loss, poor returns.
  - Stronger regularisation, early stopping on critic epochs per batch.
- **Partial observability** → Non-Markovian inputs.
  - Add recurrence (LSTM/GRU), stack frames, or explicit memory.

Goal-conditioned Actor–Critic (bridge to UVFA)

Make **goal** g an explicit input everywhere:

Policy: $\pi_\theta(a \mid s, g)$

Critic: $V_w(s, g)$ or $Q_w(s, a, g)$

Advantages/TD: computed with the **same gg**

- **Actor loss**:

$$\mathcal{L}_{\text{actor}}(\theta) = -\mathbb{E}[\log \pi_\theta(a_t \mid s_t, g)\, \hat{A}_t(s_t, a_t, g)] - c_H\, \mathbb{E}[\mathcal{H}(\pi_\theta(\cdot \mid s_t, g))].$$

- **Critic TD error** (value-critic):

$$\delta_t(g) = r_{t+1} + \gamma(1 - d_{t+1})V_w(s_{t+1}, g) - V_w(s_t, g).$$

- **Relabeling (HER-style, optional)**: for sparse rewards, **relabel gg** using future states in the trajectory to create extra successful samples.

Architecturally, feed gg by concatenation $[s, g]$, FiLM/conditioning layers, or separate encoders with fusion.

Everything from A2C/PPO/DDPG/TD3 carries over unchanged after replacing s with (s, g).

Minimal pseudocode (on-policy, goal-conditioned A2C/PPO core)

```
# assume: pi_theta(a|s,g)  ;  V_w(s,g)
rollout = collect(env, pi_theta, horizon=T)   # tuples (s,g,a,r,s2,don
e,logp_t, V_t)

# compute GAE advantages per (s,g)
adv, ret = compute_GAE(rollout.rew, rollout.V, rollout.done, gamma=0.9
9, lam=0.95)
adv = (adv - adv.mean()) / (adv.std() + 1e-8)

for epoch in range(update_epochs):
    for batch in minibatches(rollout.s, rollout.g, rollout.a, rollout.l
ogp, adv, ret):
        logp_new, ent = pi.logp_and_entropy(batch.s, batch.g, batch.a)
        ratio = torch.exp(logp_new - batch.logp)  # PPO; for A2C just u
se logp_new
        actor_obj = torch.min(ratio * batch.adv,
                              torch.clamp(ratio, 1-eps, 1+eps) * batch.
adv).mean()
        value_loss = F.mse_loss(V(batch.s, batch.g), batch.ret)
        loss = -actor_obj + c_v*value_loss - c_H*ent.mean()
        optimizer.zero_grad(); loss.backward(); clip_grad_norm_(params,
max_norm); optimizer.step()
```

What you should be able to do now

- Write the **actor** and **critic** losses from memory (with GAE).
- Explain VVcritic vs QQcritic choices and when to use each.
- Implement A2C/A3C or PPO and make it stable (entropy, clip, GAE, value loss).
- Explain the **compatible critic** result and how it connects to **natural gradients**.
- Extend everything to **goal-conditioned** inputs (s, g) for UVFA/GC-actor–critic.

Goal Conditioning (theory → algorithms → tricks)

Intuition

In many tasks the **dynamics** don't change, only the **goal** (reach a location, pick up an object, match a pattern). Instead of learning a separate agent per goal, learn **one conditional agent**:

- policy $\pi(a \mid s,\ g)$
- values $V(s,\ g),\ Q(s, a,\ g)$

This lets you generalize across goals and reuse data.

goal-conditioned MDP (GMDP) formalization

Let an environment MDP be $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$. Introduce a **goal space** $\mathcal{G}$ and **goal distribution** p(g)p(g). For each goal $g \in \mathcal{G}$ we have a **goal-specific reward** $R_g(r \mid s, a, s')$ (dynamics usually **do not** depend on gg: P shared).

Training objective (discounted, episodic or continuing):

$$J(\pi) \;=\; \mathbb{E}_{g\sim p(g)}\,\mathbb{E}_{\tau\sim p(\tau\mid\pi,g)}\big[\,G_0\,\big].$$

You can also condition the **initial state** on g (e.g., curriculum), but the standard case samples g at episode start and keeps it fixed.

goal-conditioned value functions

For a fixed policy $\pi(\cdot \mid s, g)$:

- **State value**

$$V^\pi(s,g) = \mathbb{E}_\pi[\,G_t \mid S_t = s,\ g]\,.$$

- **Action value**

$$Q^\pi(s,a,g) = \mathbb{E}_\pi[\,G_t \mid S_t = s,\ A_t = a,\ g]\,.$$

- **Advantage**

$$A^\pi(s,a,g) = Q^\pi(s,a,g) - V^\pi(s,g).$$

Bellman expectation (fixed g):

$$V^\pi(s,g) = \textstyle\sum_a \pi(a \mid s,g) \sum_{s'} P(s' \mid s,a)\big(\bar r_g(s,a,s') + \gamma V^\pi(s',g)\big),$$

$$Q^\pi(s,a,g) = \textstyle\sum_{s'} P(s' \mid s,a)\Big(\bar r_g(s,a,s') + \gamma \sum_{a'} \pi(a' \mid s',g) Q^\pi(s',a',g)\Big).$$

Bellman optimality (fixed gg):

$$V^*(s,g) = \max_a \textstyle\sum_{s'} P(s' \mid s,a)\big(\bar r_g(s,a,s') + \gamma V^*(s',g)\big),$$

$$Q^*(s,a,g) = \textstyle\sum_{s'} P(s' \mid s,a)\big(\bar r_g(s,a,s') + \gamma \max_{a'} Q^*(s',a',g)\big).$$

> For each gg, the corresponding Bellman operator is a γ\gamma-contraction ⇒ unique fixed point $V^\pi(\cdot,g)$ and $V^*(\cdot,g)$.

goal-conditioned policy gradients

Define the discounted state visitation **conditioned on** g:

$$d_{\pi,g}(s) = (1-\gamma) \textstyle\sum_{t\ge 0} \gamma^t\ \Pr(S_t = s \mid \pi, g).$$

Then the **policy gradient theorem** extends directly:

$$\nabla_\theta J(\theta) = \tfrac{1}{1-\gamma}\,\mathbb{E}_{g\sim p(g),\, s\sim d_{\pi_\theta,g},\, a\sim\pi_\theta(\cdot\mid s,g)}\big[\,\nabla_\theta \log \pi_\theta(a \mid s,g)\ Q^{\pi_\theta}(s,a,g)\,\big].$$

Variance reduction with a **goal-conditioned baseline** b(s,g)b(s,g) keeps the gradient unbiased; choose $b = V^\pi(s,g)$ to get advantages:

$$\nabla_\theta J = \tfrac{1}{1-\gamma}\,\mathbb{E}\big[\nabla_\theta \log \pi_\theta(a \mid s,g)\ A^\pi(s,a,g)\big].$$

defining goal rewards (design patterns)

(A) state-matching goals

Goal is a target state g (or a feature of it $f(g)$). Common choices:

- **Sparse**: $r_g(s,a,s') = \mathbf{1}\{d(f(s'), f(g)) \le \epsilon\}$.
- **Shaped**: $r_g(s,a,s') = -d(f(s'), f(g))$ (or change in distance).

Caution: pure distance shaping can change optimal policies. Use **potential-based shaping** to be safe:

$$F_g(s,a,s') = \gamma\Phi_g(s') - \Phi_g(s), \quad \text{e.g., } \Phi_g(s) = -\lambda\, d(f(s), f(g)).$$

Adding F_g to any base reward preserves optimal policies (for each fixed g).

(B) object/achievement goals

Binary predicates (door open, key collected). Rewards often sparse; use HER and curricula.

(C) instruction / language goals

gg is text (e.g., "put the red block on blue block"). Encode with a text encoder, align with state embedding; reward from a success classifier or programmatic checker.

terminals

Treat success as **absorbing** with 0 future reward; bootstrap is zero when done.

Architectures: conditioning on gg

- **Concatenation**: input $[s, g]$ to the network; simple and effective.
- **Difference features**: include $[s, g, s-g]$ (for positional goals).
- **FiLM / conditional BN**: generate per-channel scales/shifts from gg to modulate state features.
- **Cross-attention**: queries from state, keys/values from goal (useful for set- or language-goals).
- **Separate encoders** $z_s = E_s(s), z_g = E_g(g) \to$ fuse via concat, Hadamard product, or attention.
- **Normalization**: standardize goal coordinates to match state scale.

Any value/policy head can be conditioned; e.g., dueling head for $Q(s,a,g)$:

$$Q(s,a,g) = V(s,g) + \Big(A(s,a,g) - \text{mean}_{a'} A(s,a',g)\Big).$$

Data efficiency: Hindsight Experience Replay (HER)

For sparse goal rewards, **relabel goals** after the fact.

Mechanism (episodic):

1. Roll out with commanded goal gg, collect transitions (s_t, a_t, r_t, s_{t+1}).

2. For each transition, sample a **hindsight goal** g′g' from future states in the episode (e.g., $g' = s_k$ for some $k \geq t$).
3. Recompute reward using $g' : r' = \mathbf{1}\{d(f(s_{t+1}), f(g')) \leq \epsilon\}$.
4. Store both original and relabeled samples; train your **universal** critic/policy on the union.

Goal sampling strategies: `future` (t+), `final`, `episode`, `random`.

k-relabels: 2–8 per real transition are common.

Notes:

- HER changes the effective goal distribution to a mixture; this is fine because we are learning **universal** $Q(s, a, g)$ or $\pi(a|s, g)$.
- Works best when success can be checked programmatically from state (deterministic success predicate).
- Combine with **Double DQN / SAC / PPO**; the relabeling only affects the reward and the goal input.

goal-conditioned algorithms (drop-in recipes)

(A) GC-Double DQN (discrete actions)

Target for a batch item (s, a, r, s', d, g) :

$a^\star = \arg\max_{a'} Q_\theta(s', a', g), \qquad y = r + \gamma(1 - d)\, Q_{\bar{\theta}}(s', a^\star, g).$

Loss: $\big(Q_\theta(s, a, g) - y\big)^2$.

Use **HER** by replacing g and recomputing r for extra samples.

(B) GC-SAC (continuous actions, robust off-policy)

Two critics $Q_{\theta_1}, Q_{\theta_2}$, stochastic policy $\pi_\phi(a \mid s, g)$, temperature α.

Critic target:

$y = r + \gamma(1 - d) \left[\min_{i=1,2} Q_{\bar{\theta}_i}(s', a', g) - \alpha \log \pi_\phi(a' \mid s', g)\right], \quad a' \sim \pi_\phi(\cdot \mid s', g).$

Actor objective (reparameterized):

$\min_\phi \; \mathbb{E}_{s,g}\left[\alpha \log \pi_\phi(a \mid s, g) - \min_i Q_{\theta_i}(s, a, g)\right], \quad a \sim \pi_\phi(\cdot \mid s, g).$

HER: relabel gg and recompute sparse reward; keep the entropy term unchanged.

(C) GC-PPO (on-policy)

Same as PPO but inputs include g; compute **GAE** with $V(s, g)$; ratios/clipping unchanged.

sampling goals & curricula

- **Uniform over** $\mathcal{G}$ can be too hard (most goals far).
- **Distance-based**: sample goals within a radius from current states.

- **Success-rate targeting**: pick goals to keep success around ~30–70% (Goldilocks zone).
- **Frontier/ALP-GMM**: model learning progress over goals and sample where progress is highest.
- **HER-driven**: bias toward goals you actually reached (works well early on).

evaluation metrics

- **Success rate@ε**: fraction of episodes reaching $d(f(s_T), f(g)) \leq \epsilon$.
- **Time-to-goal / steps-to-success**.
- **Return** (if rewards shaped).
- **SPL** (Success weighted by Path Length) for navigation.
- **Generalization**: train goals vs. **held-out** goals (OOD tests).

pitfalls & fixes

- **Wrong distance/scale** → poor shaping & generalization.

  Fix: normalize coordinates; learn a goal/state embedding with metric learning.
- **Reward shaping breaks optimality**.

  Fix: use **potential-based** shaping $F_g = \gamma\Phi_g(s') - \Phi_g(s)$.
- **Ambiguous goals** (many states satisfy): non-stationary credit.

  Fix: define canonical success or use dense auxiliary signals.
- **Negative transfer across goals**.

  Fix: separate encoders per object/slot; multi-head critics; curriculum.
- **HER with stochastic success**: relabeled rewards become noisy.

  Fix: increase relabels; prefer deterministic success checks; combine with shaped auxiliary losses.

minimal pseudocode

GC-Double DQN + HER (sketch)

```
for each env step:
    g = current_goal()                    # sampled at episode start
    a = eps_greedy(Q(s, ·, g))
    s2, r, done = env.step(a)
    buffer.add(s, a, r, s2, done, g)

    # HER relabels (k times)
    for g_her in sample_future_goals(episode_buffer, from_state=s2, k=
4):
        r_her = success(s2, g_her)  # 0/1 based on epsilon
        buffer.add(s, a, r_her, s2, done, g_her)
```

```
    if ready_to_train():
        batch = buffer.sample(B)
        y = []
        with torch.no_grad():
            a_star = Q_online(batch.s2, :, batch.g).argmax(dim=1, keepd
im=True)
            q_targ = Q_target(batch.s2, :, batch.g).gather(1, a_star).s
queeze(1)
            y = batch.r + gamma * (1 - batch.done) * q_targ

        q_pred = Q_online(batch.s, :, batch.g).gather(1, batch.a).squee
ze(1)
        loss = huber(q_pred - y)
        optimize(loss); update_target()
```

GC-PPO (core)

```
# collect on-policy trajectories with (s,g)
adv, ret = compute_GAE(rew, V(s,g), done, gamma, lam)
adv = (adv - adv.mean())/(adv.std()+1e-8)
ratio = exp(logp_new(a|s,g) - logp_old)
actor = -mean(min(ratio*adv, clip(ratio,1-eps,1+eps)*adv)) - cH*entropy
critic = cV * mse(V(s,g), ret)
loss = actor + critic
optimize(loss)
```

tie-in to UVFA (what changes next)

UVFA is exactly the **goal-conditioned value** idea with a **single function approximator** over (s, g) :

$Q_\theta(s, a, g) \approx Q^*(s, a; g)$

trained with Q-learning-style targets **at fixed** g (Double/Distributional/Dueling variants carry over). In "UVFA" section we'll formalize UVFA's generalization story, cover **successor features** for fast transfer across **reward functions**, and show how to wire all of this into actor–critic.

should now be able to

- write Bellman equations for V,QV,Q **conditioned on gg**,
- derive the **goal-conditioned policy gradient**,
- implement **GC-DQN / GC-SAC / GC-PPO**,
- use **HER** and design safe shaping for goals.

Universal Value Function Approximators

what is a UVFA?

A **UVFA** is a single function that approximates value **across states *and* goals**:

- **state-value:** $V_\theta(s, g) \approx V^*(s; g)$
- **action-value:** $Q_\theta(s, a, g) \approx Q^*(s, a; g)$

Here, the **dynamics** $P(s'|s, a)$ are shared; **rewards** depend on the goal gg via R_g. Instead of learning one Q per goal, we learn **one universal** Q that conditions on g and can **interpolate/extrapolate** to unseen goals.

goal-conditioned Bellman equations (fixed gg)

All Bellman relations from steps 3/9 apply **for each fixed goal** g:

Expectation (policy evaluation):

$$Q^\pi(s, a, g) = \sum_{s'} P(s'|s, a)\Big(\bar{r}_g(s, a, s') + \gamma \sum_{a'} \pi(a'|s', g)\, Q^\pi(s', a', g)\Big).$$

Optimality (control):

$$Q^*(s, a, g) = \sum_{s'} P(s'|s, a)\Big(\bar{r}_g(s, a, s') + \gamma \max_{a'} Q^*(s', a', g)\Big).$$

UVFA simply says: **approximate $Q^*(\cdot, \cdot, g)$ for many g with one network $Q_\theta(\cdot, \cdot, g)$**.

learning objective (universal TD)

Sample goals from a goal distribution p(g)p(g). Minimize TD error **averaged over goals**:

$$\min_\theta \ \mathbb{E}_{g\sim p(g)}\ \mathbb{E}_{(s,a,r_g,s',d)\sim\mathcal{D}}\Big[\big(Q_\theta(s, a, g) - y(s, a, s', r_g, g)\big)^2\Big],$$

with (Double-DQN target, discrete actions)

$$a^\star = \arg\max_{a'} Q_\theta(s', a', g), \quad y = r_g + \gamma(1-d)\, Q_{\bar{\theta}}(s', a^\star, g).$$

For value-based **on-policy** evaluation, replace max by expectation under $\pi(\cdot|s', g)$.

> For actor–critic (below), this same "universal conditioning" applies to $V(s, g)$, $Q(s, a, g)$, and $\pi(a|s, g)$.

why can a single function generalise across goals?

Three ingredients typically make UVFA work:

1. **Shared dynamics:** only the reward changes with g, so many transitions (s, a, s') are informative for many goals.
2. **Smoothness in goal space:** if reward/optimal value varies smoothly with g, a continuous $Q_\theta(s, a, g)$ can interpolate.
3. **Representation learning:** shared encoders for s and g learn **features** useful across goals (e.g., distances, success predicates).

Formally, for each g the Bellman operator is a γ-contraction; UVFA training is solving many coupled fixed-point problems by **jointly fitting** a function class with shared parameters.

architectures (how to inject gg)

- **Concatenation:** $x = [\mathrm{enc}_s(s) \,\|\, \mathrm{enc}_g(g)] \rightarrow$ MLP/CNN head to $Q(s, a, g)$ (or $Q(s, \cdot, g)$).
- **Difference features:** for positional goals, include $[s, g, s - g]$.
- **FiLM/cond-BN:** modulate state features with scales/shifts produced from g.
- **Cross-attention:** queries from state, keys/values from goal (useful for set/language goals).
- **Dual encoders:** separate encoders E_s, E_g with fusion (concat, elementwise product, attention).

Heads:

- Discrete: network outputs $Q(s, \cdot, g)$ (one value per action).
- Dueling head for UVFA: $Q = V(s, g) + \big(A(s, a, g) - \mathrm{mean}_{a'} A(s, a', g)\big)$.
- Continuous: two critics $Q_{\theta_1}, Q_{\theta_2}$ (SAC/TD3 style) that take (s, a, g).

rewards for goals (designs you'll use)

- **Sparse success:** $r_g(s, a, s') = \mathbf{1}\{d(f(s'), f(g)) \le \epsilon\}$.
- **Potential-based shaping (safe):** add $F_g = \gamma \Phi_g(s') - \Phi_g(s)$ (e.g., $\Phi_g = -\lambda\, d(f(s), f(g))$). Preserves optimality.
- **Task predicates:** success if "object X at Y", "door open", etc.
- **Language goals:** encode text gg (e.g., transformer) and use a programmatic or learned success signal.

Terminal handling: when success/termination, bootstrap term 0.

data: HER + curricula (crucial in practice)

HER (Hindsight Experience Replay): relabel goals with states you actually reached to convert failures into successes.

- For a transition from episode E at time t, sample a future state s_k with $k \ge t$; set $g' = f(s_k)$, recompute $r_{g'}$; store $(s_t, a_t, r_{g'}, s_{t+1}, d, g')$.
- Do $k = 2..8$ relabels per real transition.
- Works with UVFA naturally because Q_θ is universal over goals.

Curricula / goal sampling:

- **Distance-based sampling** (nearby goals early).
- **Success-rate targeting** (sample difficulty to keep ~30–70% success).
- **Frontier/ALP-GMM** (sample where learning progress is highest).
- **Mixture** of commanded goals + HER goals.

UVFA + actor–critic (on-policy & off-policy)

On-policy (PPO/A2C) with goals:

- Policy $\pi_\theta(a|s,g)$, critic $V_w(s,g)$.
- GAE advantages with the **same goal** gg:
  $\delta_t(g) = r_{t+1} + \gamma(1-d_{t+1})V_w(s_{t+1},g) - V_w(s_t,g).$
- PPO objective unchanged; just feed gg.

Off-policy (SAC/TD3) with goals:

- Critics $Q_{\theta_i}(s,a,g)$, stochastic policy $\pi_\phi(a|s,g)$.
- SAC target (with entropy):
  $y = r + \gamma(1-d)\left[\min_i Q_{\bar{\theta}_i}(s',a',g) - \alpha \log \pi_\phi(a'|s',g)\right].$

Hybrid: learn both a universal critic and a universal policy; still HER-compatible.

successor features (SF) & GPI — fast transfer across **reward functions**

If rewards decompose as

$r_g(s,a,s') = \phi(s,a,s')^\top w_g \quad (\text{shared features } \phi,\ \text{goal weights } w_g),$

then the **(optimal) Q** decomposes as

$Q_g^*(s,a) = \psi^*(s,a)^\top w_g,$

where $\psi^*(s,a) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t \phi(S_t, A_t, S_{t+1})\right]$ are **successor features** (goal-independent).
Learn ψ\psi once; adapt to new gg by changing wgw_g — **no environment interaction** required.

GPI (Generalized Policy Improvement): given a set of policies {πi}\{\pi_i\} (e.g., optimal for different wiw_i), define

$Q^{\text{GPI}}(s,a) = \max_i Q_g^{\pi_i}(s,a).$

GPI **never hurts** and often improves over any single πi\pi_i for a new w_g.

UVFA + SF gives **instant transfer** across linear reward families.

evaluation & generalization

- **In-distribution goals:** success rate, return, steps-to-goal.
- **OOD goals:** held-out locations/texts; measure interpolation vs extrapolation.
- **Ablations:** with/without HER, with/without shaping, goal encoders, curriculum strategies.
- **Diagnostics:** value error vs distance to goal; success vs goal radius; learning curves per goal bin.

failure modes & fixes

- **Bad goal representation / scale mismatch:** normalize goal coordinates; learn embeddings; use FiLM.

- **Negative transfer across goals:** separate encoders or multi-head critics; curriculum.
- **Reward shaping changes optimality:** use **potential-based shaping** only.
- **HER with stochastic success:** increase relabels; prefer deterministic success checks; combine with dense auxiliary losses.
- **Overestimation (max over noisy Q): Double DQN**, **Clipped double Q** (TD3/SAC).
- **Exploration collapse:** entropy bonus (policy methods), ε\varepsilonschedules, count-based/RND bonuses.

UVFA training pseudocode (discrete actions, Double-DQN + HER)

```
# Q_online(s, ·, g), Q_target(s, ·, g)
replay = Replay(capacity)
for each episode:
    g = sample_goal()          # from curriculum / goal sampler
    s = env.reset(g)           # or env.reset(); store g separately
    episode_traj = []
    done = False
    while not done:
        a = epsilon_greedy(Q_online(s, :, g), eps)
        s2, r, done, info = env.step(a)
        episode_traj.append((s,a,r,s2,done))
        replay.add(s, a, r, s2, done, g)
        s = s2

    # HER relabels (k times per transition)
    for t,(s,a,r,s2,done) in enumerate(episode_traj):
        for _ in range(k_relabels):
            g_her = goal_from_future(episode_traj, t)    # e.g., final
state position
            r_her = success_reward(s2, g_her)            # 0/1 or shape
d
            replay.add(s, a, r_her, s2, done, g_her)

    # learn
    for _ in range(train_steps_per_episode):
        (S,A,R,S2,D,G) = replay.sample(B)
        with torch.no_grad():
            A_star = Q_online(S2, :, G).argmax(dim=1, keepdim=True)
            Y = R + gamma*(1 - D) * Q_target(S2, :, G).gather(1, A_sta
r).squeeze(1)
        Q_pred = Q_online(S, :, G).gather(1, A).squeeze(1)
        loss = huber(Q_pred - Y)
        optimize(loss)
```

```
        soft_update(Q_target, Q_online, tau)  # or hard copy every C st
eps
```

quick recipes (choose your stack)

- **Discrete actions, sparse goals: GC-Double DQN + HER + (n-step, prioritized)**.
- **Continuous actions: GC-SAC** (two critics, entropy), add HER & curricula.
- **On-policy / when stability matters: GC-PPO** with GAE; HER via off-policy buffer is trickier—prefer off-policy stacks for HER.

"do i get it?" checks

- Write the **UVFA Bellman target** $y = r + \gamma(1-d)\max_{a'} Q_{\bar{\theta}}(s', a', g)$ without peeking.
- Explain **why** Double DQN reduces overestimation with goals unchanged.
- State **how HER changes** the training distribution and why UVFA makes that valid.
- For linear reward families, derive $Q_g^*(s,a) = \psi^*(s,a)^\top w_g$.
- Describe one **goal curriculum** and why it helps.

tying back to your project

1. Start with **GC-Double DQN + HER** on a gridworld (coordinate goals).
2. Add **potential-based shaping** and compare success curves.
3. Swap to **GC-SAC** if you move to continuous control.
4. Add **successor features** if your goals differ by reward weights.
5. Evaluate **ID vs OOD goals**, ablate encoders/curricula.