1

RL

core

policy
"behaviour rule"
"if A do B"

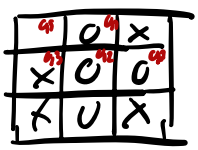
reward signal
single num per time-step
"how well it did"
goal: maximize it

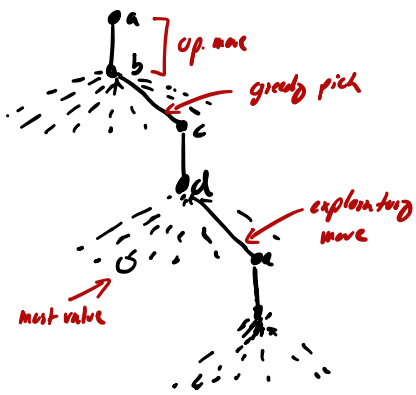
value function
predicted total future reward started from a given state
"estimated from experience"
"key is to figure out estimates"

(model)
predicts outcomes
"literal what-if"
simulates multiple future steps
- planning
- model/free
- trial and error

these

Tic Tac Toe





TD style

$$V(S_t) \leftarrow V(S_t) + \alpha[V(S_{t+1}) - V(S_t)]$$

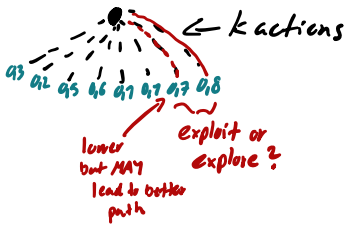
predictor error
+ next state better
− worse
learning rate

Part 1

(2)

K-bandit problem

- simplest RL problem



$q_*(a) \doteq \mathbb{E}[R_t \mid A_t = a]$

- $q_*(a)$: true action value function, "true expected reward"
- $\mathbb{E}$: Expectation (average of all possible outcomes)
- for action 'a'
- A_t: actual action at t
- R_t: random reward at step t
- "given that" at t, we picked a

$$Q_t(a) = \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t}$$

$A_t \doteq \operatorname{argmax}_a Q_t(a)$ ← exploitation greedy

ε-greedy: ex. $\varepsilon = 0.2$ → 80% we exploit, 20% we pick random arm to explore

Incremental Implementation

instead of: $Q_n = \frac{R_1 + R_2 + \dots + R_n}{n}$ ← storing R's each time

we can: $R_1 + \dots + R_n = n \cdot Q_n$

so: $Q_{n+1} = \frac{n \cdot Q_n + R_{n+1}}{n+1} = Q_n + \frac{1}{n+1}\left(R_{n+1} - Q_n\right)$

- Q_n: we 'nudge' the old average
- $R_{n+1} - Q_n$: error

NewEstimate = OldEstimate + α[Target − OldEstimate]

Non stationary problem

Stationary: $\alpha = \frac{1}{n}$

Non-stationary: $\alpha \in (0,1]$

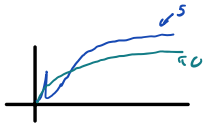
ex. $\alpha = 0.1$ ↗ trust 90% of old info and 10% of new → 0.9^k rewards from many steps ago count to almost nothing

$$(1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i$$

- $(1-\alpha)^n Q_1$: exponentially shrinking remainder of initial guess
- R_i: R_i decays the further back in time i is.

Optimistic initial values

if instead of $Q_1(a) = 0$ we start with $Q_1(a) = 5$ → more exploration, the final values will always be less than the old.



Upper-Confidence-Bound Action Selection

$$A_t \doteq \operatorname{argmax}_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]$$

- $Q_t(a)$: curr. avg. rew. estim.
- $\sqrt{\frac{\ln t}{N_t(a)}}$: explor. bonus
- how large bonus
- grows slowly over time

big confidence at start, then gets smaller

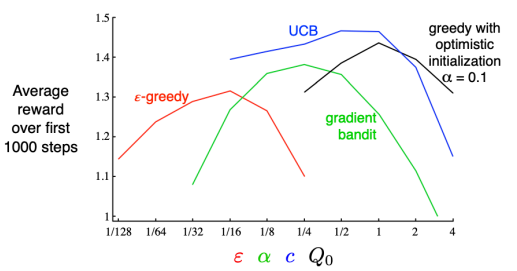
We satisfy both goals:
- exploit action with highest estm. average so far
- explore action with uncertain estimates, because bonus $\sqrt{\ln t / N_t(a)}$ keeps at play

Associative Search

2 slot machines: A: (0.1, 0.2) → normally we should know with one we are pulling → 0.5

B: (0.9, 0.8)

→ but! if we do know which we are pulling: learn 2 mini-policies → if A, pull 2nd; if B, pull 1st → 0.55



Gradient bandit algorithms

- rather than estimating each $q_*(a)$, we learn a set of preferences and turn them to probabilities (via soft-max)
- each arm, preference $H_t(a)$
- we apply Bolt2mann or Gibbs softmax:

$$\Pr\{A_t = a\} = \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}}$$

- prob. of picking a on step t
- if H_t large, $\Pr_t$ large
- if the similar, similar prop.

Update Rule:

- after choose A_t we: compute $\bar{R}_t = \frac{1}{t}\sum_{i=1}^{t} R_i$ ← best guess of typical reward

then:

$$H_{t+1}(A_t) \doteq H_t(A_t) + \alpha\left(R_t - \bar{R}_t\right)\left(1 - \pi_t(A_t)\right), \quad \text{and}$$
$$H_{t+1}(a) \doteq H_t(a) - \alpha\left(R_t - \bar{R}_t\right)\pi_t(a), \quad \text{for all } a \neq A_t,$$

- add
- reduce
- normalizing factor
- ensures we spread the penalty proportionally

repeat $t+1$

(3) Finite Markov Decision Processes

MDP = states + actions + rewards + dynamics

Markov property: Future is independent from the past, given the present state
Ex. in maze, we need past states, to know where we came from → not MARKov.

transition prop. function $p(s', r \mid s, a)$
"prob. of going to s' and seeing reward r after taking a in s"

Central property
the next state (S_{t+1}, R_{t+1}) depends only on (S_t, R_t) not earlier

Agent vs Environment (outside of agent's control)
- he may fully know env, but still struggle (Rubik's)

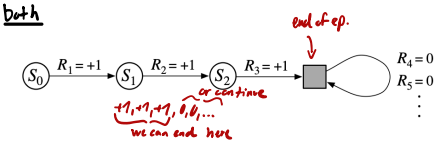
Decision Making
- consider how action affects not only the next immediate reward, but also the future state you end up in

Rewards
- reward signal is way of communicating to agent what you want it to achieve not how you want it achieved.

Returns and episodes

episodic
$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T,$$
(to converge for infinite tasks)
continuing
$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$
where γ is a parameter, $0 \le \gamma \le 1$, called the *discount rate*.

both



$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k,$$

Policies and value Functions / Bellman

π tells agent how to choose actions given a state

$\pi(a|s) = Pr\{A_t = a \mid S_t = s\}$
if in s, it picks a with probability $\pi(a|s)$

deterministic (picks which a has $p = 1$) × stochastic (random) spreads p over multiple actions

State-value Function: how good it is to start in s and follow π

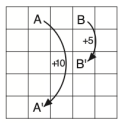
$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \text{ for all } s \in \mathcal{S},$$
Bellman $v_\pi(s)$
- we can compare which states are better under π

Action-value Function: -ll- start in s, take a, follow π

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]$$
Bellman $q_\pi(s,a)$
- which a is best in s, given follow π

Ex. GridWorld

$\pi(a|s) = 0.25$



Actions

policy $\gamma = 0.9$

3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

$R = 0$

$$\text{RHS} = \sum_{a \in \{N,S,E,W\}} 0.25 \times [0 + \gamma v_\pi(\text{neighbor})]$$
RHS = $0.25(0.9 \cdot 0.9 + 0.9 \cdot 2.3 + 0.9 \cdot 0.4 - 0.9 \cdot 0.4) = 0.635 \approx 0.7$

Bellman Function simply

Value of s = immediate reward + discount × value of next state

$v(s) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \mid S_t = s]$
it is the expected sum of discounted rewards from that point onward

Easy ex.
A →(+2) B →(+3) C →(+4) END (0) $\gamma = 0.5$
we want: $v(A)$ $v(B)$ $v(C)$
$v(C) = 1 + 0.5 \cdot v(\text{END}) = 1$ (R + 0)
$v(B) = 3 + 0.5 \cdot v(C) = 3.5$
$v(A) = 2 + 0.5 \cdot v(B) = 3.75$

Multiple choice
$$v_\pi(s) = \sum \pi(a|s) \sum p(s', r \mid s, a)[r + \gamma v(s')]$$
avg over which action π chooses; avg over next state and reward

$v_\pi(s) = 0.7[3 + 0.9 \cdot 0] + 0.3[1 + 0.9 \cdot 0] = 2.4$

Adding constants to immediate rewards in Continuing

$R'_t = R_t + c$
$\vdots$
$v'_\pi(s) = v'_\pi(s) + \underbrace{\frac{c}{1-\gamma}}_{v_c}, \quad \forall s$
— by adding c, shifts every state value by v_c

Adding c to ALL rewards in Episodic

$\vdots$
$G'_t = G_t + c \cdot (T - t)$
no longer a harmless shift, depends on the length of episodes.

Optimal Policies and optimal value Functions

$$v_*(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]$$
best achievable value; take action a now, then optimal

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a]$$
next best move

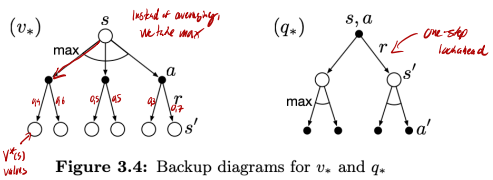


Figure 3.4: Backup diagrams for v_* and q_*

Bellman equations in real problems?

We'd need to know: - exact model of the world
- unlimited compute
- perfect Markov state

instead, we approximate! (to be continued)
- accept that somewhere it might be suboptimal

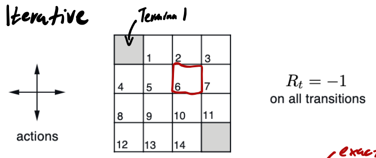
④ Dynamic Programming

– if have MDP and enough computation, we can turn Bellman Functions into iterative algorithms that converge to v^* and q^*

$$v_*(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]$$
$$= \max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_*(s')\Big], \text{ or}$$
$$q_*(s,a) = \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\Big|\; S_t = s, A_t = a\Big]$$
$$= \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big],$$

Policy evaluation (Prediction)

Iterative



– equal prop.
– $\gamma = 1$

exact B.eq.

$$v_\pi(6) = \frac{1}{4}[(-1 + V(2)) + (-1 + V(10)) + (-1 + V(6)) + (-1 + V(7))]$$

but for iterative:

$$V_{k+1}(6) = \frac{1}{4} \sum_{\{2,10,6,7\}} (-1 + V_k(s'))$$

Iterations:

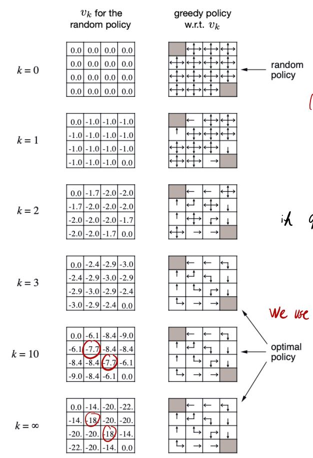
0. $V_0(s) = 0$; 1. $V_1(6) = \frac{1}{4}(-1+0) + (-1+0) \cdots = \frac{1}{4} \cdot (-4) = -1$

2. $V_2(6) = \frac{1}{4} \cdot (-8) = -2$ 3. $V_3(6) = -3$ ....

eventually they converge to a true $v_\pi(s)$. (some neg number here)

We can use 2 arrays (old, new) or 1 (values propagate faster)
↑ exact number.

Policy Improvement



1. Loop over all states {1...14}
2. For each s, do 1-step lookahead for each of 4 actions
3. Pick a with the highest $q_\pi(s,a)$ and set $\pi'(s) = a$

if $q_\pi(s,a) \geq v_\pi(s)$, then switching to a at s always improves policy

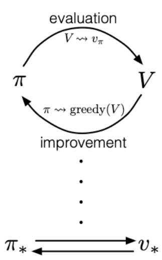
We use that $V(s)$, but pick greedily, to form a better π'

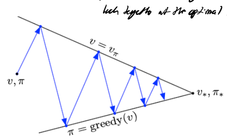
Asynchronous DP

– instead of big sweep over every state, we only pick certain ones, that are 'hot'.

Generalized policy iteration

– the big picture; keep nudging value estimates toward v_π and policy toward greedy(V). Eventually they lock together at the optimal solution.

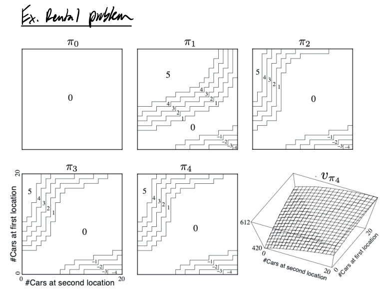




Policy Iteration – evaluate and improve until you can't do better

1. π_0 – random actions
2. repeat for $k = 0, 1, \ldots$
   1. Evaluate $\pi_k \to V^{\pi_k}$ by iterative sweeps
   2. Improve $\pi_k \to \pi_{k+1}$ by greedy one-step lookahead
   3. if $\pi_{k+1} = \pi_k$, converged, optimal ✓
3. Output π^* and V^*

Ex. Rental problem



Value Iteration – simply plugs the Bellman optim. equ. directly into update sweeps until convergence

$$v_{k+1}(s) \doteq \underbrace{\max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a]}_{\text{bellman}}$$
$$= \underbrace{\max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_k(s')\Big]}_{\text{value it. update}},$$

– Faster than full policy

ex. gambler problem demo

Goal: reach \$5 (win +1)
States = 1...5
actions = 1, 2... min(s, 5−s)
coin: heads = 0.4, on heads we go s+a (if that 5, we get +1 win), tails s−a
$\gamma = 1$

0. $V_0(s) = 0$
1. s=1 a=1 $V_1(s)$
   $Q(1,1) = 0.4(V_0(2)) + 0.6(V_0(0)) + 0 = 0 \to V_1(1) = 0$
   $Q(2,1) = 0$ $Q(2,2) = 0.4(0) + V_0(0) = 0$
   $V_1(2) = \max(0, 0.4) = 0$

   $V_1(3) = 0.4$
   $V_1(4) = 0.4$ $V_1 = [0, 0, 0.4, 0.4]$
2. we reuse V_1 .... $Q(1,1) = 0.4 \cdot V_1(2) + 0.6 \cdot V_1(0) = 0.16$
   ⋮
   $V_2 = [0.16, 0.4, 0.4, 0.4]$
3. ...
   ⋮

slowly converges to true V_* → $\pi_*(s) = \arg\max_a Q(s,a)$

So: s=1: stake 1
s=2: 1
s=3: 2
s=4:

Efficiency of DP

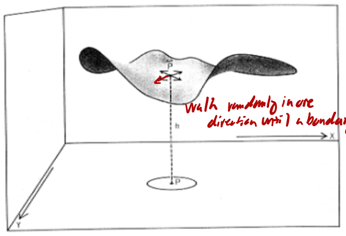
– at first glance hopelessly expensive
but when compared to enumerating every possible policy k^n
– DP's cost is polynomial
– linear programming outperforms DP in theory, but large scale DP better
– real obstacle is how we model and approximate enormous MDPs

⑤ Monte Carlo Methods : value functions and policies from experience

- episodic tasks
- a way to learn exactly Bellman fixed-point solutions v_π and q_π that DP would compute but using only samples of experience

Monte Carlo Prediction ($v_\pi(s)$)

Idea: run entire episodes under your π. Whenever you see 's' for the first time in an episode, record the full return from that point. Then $V_\pi(s)$ is the average of all these returns.



A bubble on a wire loop.

walk randomly in one direction until a boundary -> record the boundary height and repeat. -> then averaging the heights converges to solution on starting point height.

Monte Carlo estimation of Action Values ($q_\pi(s)$)

- same as $v_\pi(s)$ but with (s,a) pairs

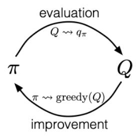
↳ For each episode, first time you saw $(S_t = s, A_t = a)$, return is: $G_t = R_{t+1} + R_{t+2} + \dots + R_T$

↳ append G_t to list of returns $\hat{q}_\pi(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s,a)} G^{(i)}$

↳ average over them.

Number of first visits so far

Monte Carlo Control



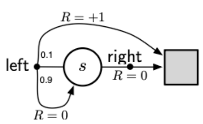
1. init $Q(s,a)$, $N(s,a) = 0$, policy π
2. begin each episode in rand. chosen (s_0, a_0) with prop.
3. $(s_0, a_0) \to r_1 \to (s_1, s_2) \to \dots \to$ terminal
4. For each first-visit (s_t, a_t) compute return G_t and update
   $N(s_t,a_t) = 1 \quad Q(s_t,a_t) \mathrel{+}= \frac{1}{N(s_t,a_t)} [G_t - Q(s_t,a_t)]$
5. For every visited state s_t:
   $\pi(s) \leftarrow \mathrm{argmax}_a Q(s,a)$
6. repeat ; $Q \to q$, $\pi \to \pi$

MC Control without exploring starts

- dropping unrealistic "exploring start" assumption
- in real life, we can't start agents in every state, so how do we ensure we try all actions?

ε-greedy exploration

- same as MC Control but with $\frac{\varepsilon}{|A(s)|}$ prop. to explore.



$\pi(\text{left}|s) = 1$

$b(\text{left}|s) = \frac{1}{2}$

$(0.5)^{k+1} \cdot 0.1 \qquad 1^{k+1} \cdot 0.1 = 0.1$

$\rho = \frac{0.1}{(0.5)^{k+1} \cdot 0.1} = 2^{k+1}$ but since these exponents can be large, ρ is infinite; sometimes we can get huge numbers that off-throw the averages,

$\hat{v} = \frac{1}{N} \sum_{i=1}^{N} \rho_i G_i = \frac{1}{N} \sum 2^{k_i+1}$

So: weighted $\hat{v} = \frac{\sum_i \rho_i G_i}{\sum_i \rho_i}$

- any episode that ever picks right has $\rho = 0$ ⇒ doesn't contribute to anything
- any ep. that only does lefts has $\rho > 0$ any $G = 1$
- every non-zero contribution is exactly 1

$\hat{v} = \frac{1}{1} = 1$ no huge swings.

Weighted IS is almost always the practical choice in MC

Off-policy prediction via Importance sampling

GOAL: learn about π we really care about, while under more exploratory b

Importance sampling

coverage condition: $\pi(a|s) > 0 \Rightarrow b(a|s) > 0$ – b must sometimes pick any action π might pick

importance sampling ratio: $\rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$ – if π picks actions twice as much as b, the ratio = 2 ...

ordinary IS estimator

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{|\mathcal{T}(s)|}.$$

G_t: full return from time t; $\mathcal{T}(s)$: all time steps we visited state s

(for zero bias, high variance)

weighted IS estimator

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1}},$$

(low variance, small bias that vanishes with more data)

- so we re-weigh each return G_t by ρ

Incremental Implementation

- For weighted off-policy MC maintains, for each state (or state-action pair):
  1. a running total of weights C, and
  2. the current weighted average Q, and on each new return G with weight W is:

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

'batch' weighted average:

$$V_n \doteq \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}, \quad n \geq 2,$$

each return G is worth a weight W

estimation of true value V_n by averaging all returns

- Instead of resumming at each step, we keep track of:

$C_n = \sum_{k=1}^{n-1} W_k$; V_n

- When we get next return, we update:

$$V_{n+1} \doteq V_n + \frac{W_n}{C_n}\left[G_n - V_n\right], \quad n \geq 1,$$

and

$$C_{n+1} \doteq C_n + W_{n+1},$$

Off-policy MC Control

- learn optimal policy π^* (deterministic, greedy wrt Q)
- act under separate soft policy b
- use weighted importance sampling – so we get info about π, and do little policy improvement toward greediness every Q update.

Alg:
1. gather eps. under any b
2. weight each return by how much more/less likely π would have taken those actions.
3. incrementally update Q and make greedy wrt the new Q
4. but stop early in each ep when you hit an action π wouldn't have taken.

*Discounting-aware IS

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \left((1-\gamma) \sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1} \rho_{t:h-1} \bar{G}_{t:h} + \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \bar{G}_{t:T(t)} \right)}{|\mathcal{T}(s)|}, \quad (5.9)$$

and a weighted importance-sampling estimator, analogous to (5.6), as

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \left((1-\gamma) \sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1} \rho_{t:h-1} \bar{G}_{t:h} + \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \bar{G}_{t:T(t)} \right)}{\sum_{t \in \mathcal{T}(s)} \left((1-\gamma) \sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1} \rho_{t:h-1} + \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \right)}. \quad (5.10)$$

*Per-decision IS

- it avoids dragging the entire π/b ratio, massively cuts variance.

estimator: $\hat{v}(s) = \frac{1}{|\mathcal{T}(s)|} \sum \tilde{G}_t$

each per-decision return is

$$\tilde{G}_t = \rho_{t:t} R_{t+1} + \gamma \rho_{t:t+1} R_{t+2} + \gamma^2 \rho_{t:t+2} R_{t+3} + \cdots + \gamma^{T-t-1} \rho_{t:T-1} R_T.$$

(truncated ratios $\rho_{t:h} = \prod_{k=t}^{h} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$)

Summary

- learns from full episodes
- estimates $V(s)$ by averaging returns G, observed after visiting a state
- each update depends only on empirical returns, not on other value estims.

2. Core Algorithms

Setting	Key Idea	Policy Improvement
On-policy MC Control	ε-greedy episodes, first-visit MC on Q	Make π ε-greedy wrt current Q
Off-policy Prediction	Importance sampling of returns under b to learn v	(No improvement—π is fixed)
Off-policy Control	Weighted IS + GPI: evaluate π by IS then greedify Q	Stop each episode at first non-π

3. Exploration Guarantees

- **Exploring Starts** (MC-ES): force random (s_0, a_0) at episode start.
- **ε-soft / ε-greedy**: keep $b(a|s) > 0 \; \forall a$ ⇒ automatic coverage.
- **Off-policy**: behavior b explores, target π is refined toward optimal.

4. Importance Sampling Variants

Estimator	Bias	Variance	Use When
Ordinary IS (Eq 5.5)	unbiased	potentially ∞	theoretic analysis, small M
Weighted IS (Eq 5.6)	small bias →0 as $N \to$	bounded	practical off-policy MC
Per-decision IS (Eq 5.15)	unbiased	much lower	long episodes / γ<1
Discount-aware IS (Eqs 5.9/5.10)	unbiased or small bias	lower for γ<1	deep discounts, reduce variance

⑥ Temporal-Difference Learning

Monte carlo + DP

directly from raw experience ← updates using current estimates instead of waiting for episode to end

after each step we get 'partial return' $R_{t+1} + \gamma V(S_{t+1})$ — one step lookahead of the return

TD Prediction

Monte Carlo baseline (waits after visit and uses G_t, full return):

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

TD(0) update: one step bootstrapped target, we learn as we step

$R_{t+1} + \gamma V(S_{t+1})$

and update immediately:

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big]$$

TD error: signed difference between one-step target and old estimate

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

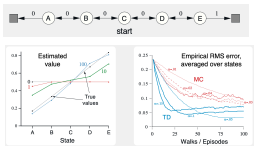
Alg:

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\text{terminal}) = 0$
Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  until S is terminal

Advantages of TD prediction methods

- Updates immediately rather than waiting for ep. to end
- Works on continuing problems (no episodes)
- Updates bootstraps, propagates info faster through state-value estimates
- Model-free
- proven to converge, learns faster than MC

Ex Random walk



Optimality of TD(0)

Under batch updating (stepping until convergence)

- with limited data TD(0) converges to the correct values faster than MC

MC batch - finds the sample-mean solution for state

TD(0) batch - finds solution consistent with maximum-likelihood model of MRP, generates reward to follow chain

Sarsa: ON-policy TD Control

- learn action value estimates and update policy at the same time using:

$S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$

Update Rule: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big]$.

(stay ε-greedy, move ε-greedy; it uses the next action by ε-greedy; TD error)

Update Rule:

Iteration: - evaluate the current policy via the Sarsa update
- improve by making π more greedy, with the updated q
- repeat until $\pi = \pi^*$

- if we ε-greedy and gradually reduce to 0, Sarsa will converge to optimal Q^*

Q-Learning: OFF-policy TD Control

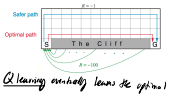
Update

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big].$$

(it doesn't use next ε-greedy action. It always assume the next action is optimal)

exactly as Sarsa, but greedy pick actions.

Ex. Cliff Walking



Q learning eventually learns the optimal path, but ε-greedy we still explore down the cliff. ⅛ of the time. → so cumulative reward is low

Sarsa: takes the safer route, bagging cliff takes huge penalties under ε-greedy → better reward

Expected Sarsa

- On policy, but replaces next-action value with expected value under policy

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t)\Big] \\ &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\Big], \end{aligned}$$

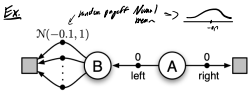
- averages over all next-actions under ε-greedy policy

Figure 6.4: The backup diagrams for Q-learning and Expected Sarsa.

+ lower Variance than Sarsa, on-policy, more stable and safer than others 2
- extra computation

Maximization Bias and Double Learning

Ex



- Q learning picks left far more than expected ε-greedy then, it thinks some action B is better than 0 ← this is maximization

Solve: Double Q learning - decouple action-selection value from value estimate by maintaining 2 independent estimates.

On each step: - flip coin
- if heads: $Q_1(s,a) \leftarrow Q_1(s,a) + \alpha\big[r + \gamma Q_2(s', \arg\max_b Q_1(s',b)) - Q_1(s,a)\big]$.
- if tails: $Q_2(s,a) \leftarrow Q_2(s,a) + \alpha\big[r + \gamma Q_1(s', \arg\max_b Q_2(s',b)) - Q_2(s,a)\big]$.

- over time Q_1 Q_2 are uncorrelated → bias cancels out

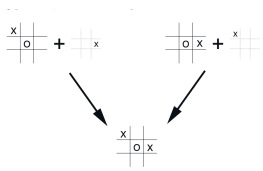
Games, Afterstates, special cases.

- in many domains you know exactly what the immediate effect of your action will be but you don't know what happens after that (opponents move..)

Afterstates - many (s,a) pairs may lead to same s', we can collapse the

- faster learning
- clearer backups

its just: $V_{new}(s') \leftarrow V_{old}(s') + \alpha\big[r + \gamma V_{old}(s'') - V_{old}(s')\big]$



Summary

- one-line update, fully online, model-free
- converge under broad conditions
- works for prediction, control, planning, continuing domains, layered games

7 n-step Bootstrapping

— lets you see n steps future rewards before bootstrapping —

n-step TD Prediction

n-step Return

$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$

if $n=1$ → $R_{t+1} + \gamma V(S_{t+1})$

$n=\infty$ → becomes full G_t return

Update rule

- once collected n rewards and arrived at S_{t+n}, update the old S_t:

$V(S_t) \leftarrow V(S_t) + \alpha[G_t^{(n)} - V(S_t)]$ $\quad = \sum_{k=0}^{n-1} \gamma^k \delta_{t+k}$

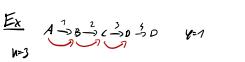
- at the end of ep. we 'flush' out pattern the rest of n updates

- small n → low variance, higher bias
- large n → low bias, higher variance
- often looking for n in between, which learns faster than both.

error reduction property - in worst case the error shrinks by at least a factor of γ^n

$\max_s \left| \mathbb{E}_\pi[G_{t:t+n} | S_t = s] - v_\pi(s) \right| \leq \gamma^n \max_s |V_{t+n-1}(s) - v_\pi(s)|,$

Ex



$\gamma=1$

n=3

$G_{0:3} = R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 V(S_3) = 1 + 2 + 3 - 6 = 6$

$V(A) \leftarrow V(A) + \alpha[6 - V(A)]$

then: $[3-3+7-1] \rightarrow 1 \geq 0 \rightarrow$ yes → update $V(S_1 : 6)$

$G_{1:4} = R_2 + R_3 + R_4 + V(S_4) = 2 + 3 + 4 + 0 = 9$

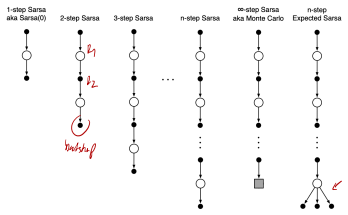
$V(B) \leftarrow V(B) + \alpha[9 - V(B)]$

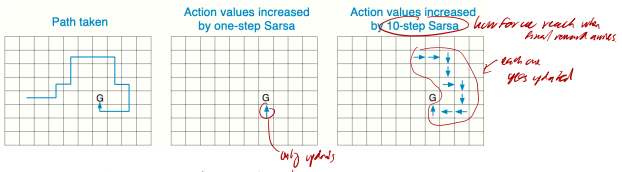
then: $[4-3+7-2.1] \Rightarrow$ yes → update $V(C)$

$G_{2:5} = R_3 + R_4 = 3 + 4 = 7$

↖ We run out of rewards, we sum the remaining

n-step Sarsa





n-step off-policy Learning

- under explor. b and learning abt π, IS ratio

Return

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \begin{cases} \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}) & \text{if } t+n < T, \\ 0 & \text{if } t+n \geq T \end{cases}$$

IS ratio

$$\rho_{t+1:t+n-1} \doteq \prod_{k=t+1}^{\min(t+n-1, T-1)} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)}.$$

SARSA update:

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha \rho_{t+1:t+n} [G_{t:t+n} - Q_{t+n-1}(S_t, A_t)].$$

* per-decision Methods with Control Variates

— to see as we hit a step where $\rho = 0$, that drive that return to 0, just skip that update

$G_{t:h} = R_{t+1} + \gamma G_{t+1:h}$

$$G_{t:h} \doteq \rho_t (R_{t+1} + \gamma G_{t+1:h}) + (1 - \rho_t) V_{h-1}(S_t), \qquad t < h < T,$$

↖ $\rho = 0 \rightarrow G_{t:h} = V(S_t)$

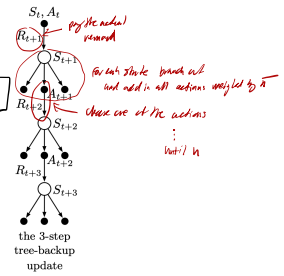
off-policy Learning without IS: n-step tree backup alg.

return

$$G_{t:t+n} = R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+n-1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+n}$$

update

$Q_{t+n}(S_t, A_t) \leftarrow Q_{t+n-1}(S_t, A_t) + \alpha[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)]$



Unifying Algorithm: n-step Q(σ)

- all part of unified family of n-step algs called Q(σ), which interpolate between

$\sigma = 1$ → pure sampling at every step → recovers n-step Sarsa

$\sigma = 0$ → pure expectation at every step → n-step tree-backup

$\sigma \in (0,1)$ → sample a fraction of σ of the time and take the policy expectation $1-\sigma$ of the time

→ blend bias/variance

Let the horizon be $h = t + n$. Define the **n-step $Q(\sigma)$ return** recursively as

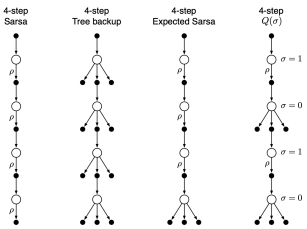
$$G_{t:h} = R_{t+1} + \gamma\Big(\sigma_{t+1}\rho_{t+1} + (1-\sigma_{t+1})\,\pi(A_{t+1} \mid S_{t+1})\Big)\,(G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1})) + \gamma\,\bar{V}_{h-1}(S_{t+1}),$$

where

$$\bar{V}_{h-1}(S_{t+1}) = \sum_a \pi(a \mid S_{t+1})\, Q_{h-1}(S_{t+1}, a),$$

and at the "leaf" (when $t+1 \geq T$ or $t+n \geq T$) you bootstrap to R_T or to 0 in the usual way.

$$Q_{t+h}(S_t, A_t) \leftarrow Q_{t+h-1}(S_t, A_t) + \alpha\,[G_{t:h} - Q_{t+h-1}(S_t, A_t)].$$



⑧ Planning and Learning with Tabular Methods

Models and Planning

What is a model? – any mechanism by which an agent can predict the consequences of its actions.
- outputs prediction of R and S'
- if it returns full probability over all possible (S', R) → distribution model
- if single sample (S', R), according to those probabilities → sample model

Simulating experience via models – by feeding states and actions into a model repeatedly

What's planning? model →(planning) policy

- state-space planning – search directly over states and val func.
- plan-space planning – search over entire action-sequences or partial plans.

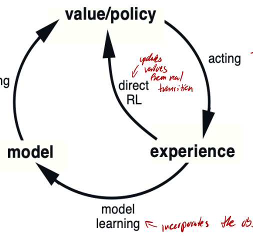
Unified structure

model → simulated experience →(backups) values → policy

Random-sample one-step tabular Q-planning

Loop forever:
1. Select a state, $S \in \mathcal{S}$, and an action, $A \in \mathcal{A}(S)$, at random
2. Send S, A to a sample model, and obtain a sample next reward, R, and a sample next state, S'
3. Apply one-step tabular Q-learning to S, A, R, S':
$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$

Dyna: Integrated planning, Acting, and Learning



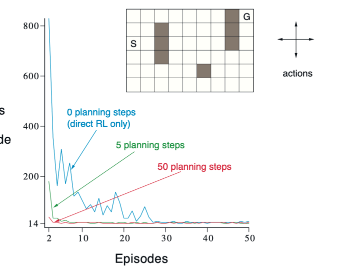
- acting – selects and executes action in real env.
- direct RL – updates values from real transition
- planning – draws simul. transitions from model and applies the same RL update to them
- model learning ← incorporates the observed (s, a, r, s') into its learned model

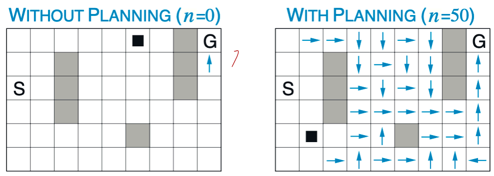
Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \varepsilon\text{-greedy}(S, Q)$
- (c) Take action A; observe resultant reward, R, and state, S'
- (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$ – direct RL
- (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment) – learning
- (f) Loop repeat n times: – planning
  - $S \leftarrow$ random previously observed state
  - $A \leftarrow$ random action previously taken in S
  - $R, S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$

rehearsing past transitions.

Ex. maze





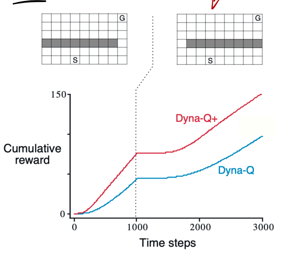
its a lookup table of past outcomes – 'memory'

When the Model is Wrong

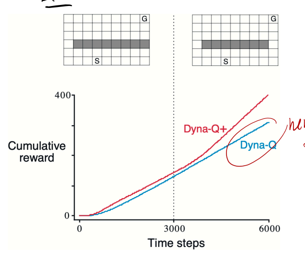
– when model is incorrect, can produce policies that exploit "phantom" opportunities.

EX1

after 1000 steps.



EX2



never finds the shortcut

– improving env – Dyna might never reach

Dyna-Q+

– exploration bonus heuristic

if trans. hasn't been tried in τ steps, then add it to reward during planning

$r + k\sqrt{\tau}$ ← small constant

Expected vs Sample Updates

↳ compute exact Bellman backup by summing over all possible next events

Distribution model tabular Q-function update:

$$Q(s,a) \leftarrow \sum_{s',r} \hat{p}(s',r \mid s,a)\left[r + \gamma \max Q(s',a')\right]$$

– unbiased zero variance updates, but costs b of possible successor states
+ exact but expensive O(b)

↳ (classic TD/Q backup) draws single (S', R) from sample model or real env and do:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[R + \gamma \max Q(s',a') - Q(s,a)\right]$$

– each update costs constant time, but reduces sampling noise
+ noisy but cheap O(1)

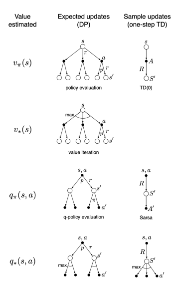


Figure 8.6: Backup diagrams for all the one-step updates considered in this book.

Quantifying the tradeoff

$$\sqrt{\frac{b-1}{bt}}$$

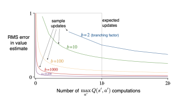


Figure 8.7: Comparison of efficiency of expected and sample updates.

– In large stochastic problems, when limited planning time, sample updates more efficient.

Prioritized Sweeping

Core Idea (in planning) – when a step produces large error Δ, we assign a priority P = Δ. We maintain a priority queue of all (s, a) pairs with P > 0

it computes the most drastic steps in the queue and puts it into the step value function.

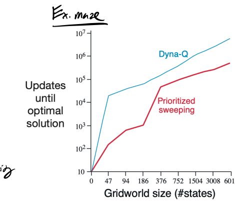
Then repeatedly:
1. pop highest P pair
2. apply standard one-step Q-backup to (s, a) using stored model
3. for each predecessor $(\bar{s}, \bar{a})$ that (in the model) can transition to s, compute its potential priority

$$P_{\bar{s}\bar{a}} = \left|\bar{r} + \gamma \max_{a'} Q(s,a') - Q(\bar{s},\bar{a})\right|$$

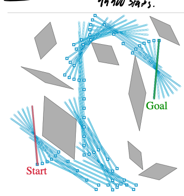
– if $P_{\bar{s}\bar{a}} > 0$, insert to queue

– this efficiently propagates large value changes to their predecessors.

Ex. maze



Ex. 14400 states.



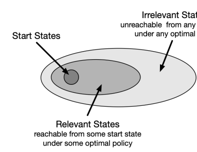
– prioritized homes in on major transitions

Trajectory Sampling

2 ways of distributing planning updates
1. Exhaustive sweeps – cycle through every state, compute exact Bellman backup
2. Trajectory sampling – instead of blind sweeps, simulate entire episodes from model following policy.

Real-time Dynamic Programming

– instead of exhaustive sweeps over all states, RTDP updates only the relevant states



Planning at decision time.

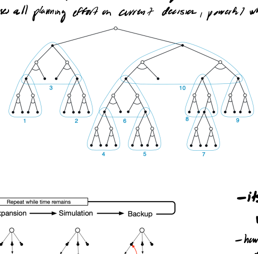
Background – dyna style, runs inbetween real steps.

Decision time – uses model at moment of choice to conduct a mini search, and picks the best action

Can be done by:
- Rollouts – from s_t under some default or learned policy averaging returns (MCTS) – easy implem, high variance, many rollouts for confidence
- Mini value-iteration – s_t sweeping over the subtree of the model reachable from s_t up to some depth – low variance, need explicit p(s', r | s, a) and must touch every successor in local subtree

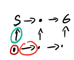
Heuristic Search

decision time planning that:
- builds a tree of possible continuations from current state
- uses one-step Bellman backups to propagate leaf-node evaluations back to root
- discards backed up actions after selecting the best one
- focuses all planning effort on current decision, powerful when deeper lookahead for single choice.



Rollout Algorithms

decision time planning method that use Monte Carlo control on model to choose action



eval right:
- Roll 1:
  1. right → (1,1), r = -1
  2. random π picks up ↑ (0,1), r = -1
  ⇒ $G^{(1)} = -2$
- Roll 2
  1. → (1,1)
  2. → (1,2)
  ⇒ $G^{(2)} = -2$
- Avg $\frac{-2-2}{2} = -2$

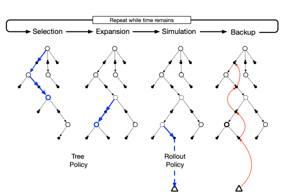
eval up
⋮
Avg = -2

we greedy choose the better one.

Monte Carlo Tree Search

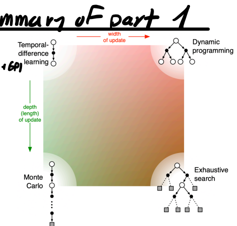
pinnacle of decision-time planner that marries rollout algs with incremental search tree.

1. Select: start at root (curr state). Descend the tree by repeatedly choosing the child action that balances "high average so far" and "haven't tried it much yet"
2. Expand: when you hit a leaf node (one you haven't tried all actions from) add a new child for one as-yet-untried action
3. Simulate: rollout from that new child state to the end, using cheap policy, compute G
4. Backup: trace back up the tree, adding G to statistics for each action you took on the path.



Summary of part 1

– its all just value + backups + GPI
– how wide, how deep, on/off policy, model free/model ...



Part 2: Approximate Solution Methods

- From small, tabular problems to real world, enormous and continuous
- For robotics, vision, games - number of possible states explodes
- parameterized function instead of exact values - supervised learning

⑨ On-policy Prediction with approximation

Value function approximation

- performs each Bellman-style update as a supervised learning example and plugs in a general function approximator in place of lookup table.

Every RL update is: $s \mapsto u$ (will like to estimate state s to some value to target u)

- We treat (s,u) as a training pair for function approximator.

Weight vector $w \in \mathbb{R}^d$ and mapping $\hat{v}(s,w) \approx v_\pi(s)$

these choices are: - linear functions
- neural networks
- decision trees, nearest neighbors...

Generalization vs tabular

update at s changes $\hat{v}(s,w)$ for all s', whose values depends on the same weights - one example "teaches" a whole region of state space

Unlike standard supervised learning: - data arrive incrementally (on the fly)
- targets in often change over time
- approximators must support online learning nonstationary targets

The prediction objective (VE)

- changes to parameter vector w affects many states, we must choose the most important ones.

Choosing state-weighting distribution $\mu(s) \geq 0$ and $\sum_s \mu(s) = 1$ to express how much we care about accuracy in each state

(Continuing with David Silver's RL course slides)

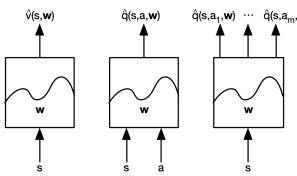
The problem:

We represented V by lookup tables → for real problems impossible → too many states (MDP)

Solution: estimate value function with approximation:

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

$\hat{q}(s,a,w) \approx q_\pi(s,a)$



← With action Q-network
← single forward pass → values for all actions at once

Gradient descent

Goal: find parameter vector w minimizing mean-squared error between approximate value fn $\hat{v}(s,w)$ and true value $v_\pi(s)$

Loss Function "how bad our prediction is"

$J(w) = \mathbb{E}_\pi[(v_\pi(S) - \hat{v}(S,w))^2]$ ← diff Function of par w

Gradient of $J(w)$

$\nabla_w J(w) = \begin{pmatrix} \frac{\partial J(w)}{\partial w_1} \\ \vdots \\ \frac{\partial J(w)}{\partial w_n} \end{pmatrix}$

$\Delta w = -\frac{1}{2}\alpha \nabla_w J(w)$

"move the weights a bit against the gradient"

Ex. $J(w) = 3w^2 + 5$

$J' = 6w$

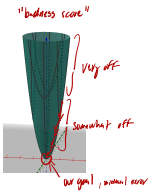
GD update: $w \leftarrow w - \alpha J(w) = (1-6\alpha)w$

Ex2: $J(w_1, w_2) = 3w_1^2 + 5w_2^2$ ← in real scenarios we don't know this

$J'_{w_1} = 6w_1 \quad J'_{w_2} = 10w_2$

$\nabla J(w) = \begin{pmatrix} 6w_1 \\ 10w_2 \end{pmatrix}$

$\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} \leftarrow \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} - \alpha \begin{pmatrix} 6w_1 \\ 10w_2 \end{pmatrix} = \begin{pmatrix} (1-6\alpha)w_1 \\ (1-10\alpha)w_2 \end{pmatrix}$



Value-Function approximation by (stochastic) Gradient descent

Full gradient descent step

$\Delta w = -\frac{1}{2}\alpha \nabla_w J(w) = \alpha \mathbb{E}[(v_\pi(S) - \hat{v}(S,w)) \nabla_w \hat{v}(S,w)]$

Stochastic GD step - sampling one of the states

$\Delta w = \alpha (v_\pi(S) - \hat{v}(S,w)) \nabla_w \hat{v}(S,w)$

we don't know this! instead, we plug sampled returns $\begin{bmatrix} MC: G_t \\ TD(0): R_{t+1} + \gamma \hat{v}(S_{t+1}, w) \end{bmatrix}$

One-step TD update: $\Delta w = \alpha (R_{t+1} + \gamma \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)) \nabla_w \hat{v}(S_t, w)$

Feature Vectors

$x(S) = \begin{pmatrix} x_1(S) \\ x_2(S) \\ \vdots \\ x_n(S) \end{pmatrix}$ real-valued summary or measurement of one aspect of the state

Linear VFA

- represent $\hat{v}(s,w)$ as linear combination of this features:

$\hat{v}(S,w) = x(S)^\top w = \sum_{j=1}^{n} x_j(S) w_j$

$J(w) = \mathbb{E}_\pi\left[(v_\pi(S) - x(S)^\top w)^2\right]$ quadratic in w → 1 global minimum, no 'traps'

because $\hat{v}(S,w)$ is linear, trivially: $\nabla_w \hat{v}(S,w) = x(S)$

Stochastic gradient update:

$\Delta w = \alpha (v_\pi(S) - \hat{v}(S,w)) \nabla_w \hat{v}(S,w) = \alpha (v_\pi(S) - x(S)^\top w) x(S)$ replace by sample target

Incremental Prediction Algorithms

- in practice, substitute $v_\pi(S)$ for sample target

$\Delta w = \alpha (\text{target} - \hat{v}(S,w)) \nabla_w \hat{v}(S,w)$

(target − prediction) = error

MC $G_t = R_{t+1} + \gamma R_{t+2} \dots$

We apply supervised learning: treat each $\langle S_t, G_t \rangle$ as training example, goal is to $\hat{v}(S_t,w) \approx G_t$

Linear MC policy eval: $\Delta w = \alpha (G_t - \hat{v}(S_t,w)) x(S_t)$

→ converges to local optimum, even with non-linear FA

TD same thing, $\langle S_1, R_2 + \gamma \hat{v}(S_2,w) \rangle, \dots$

$\Delta w = \alpha (R + \gamma \hat{v}(S',w) - \hat{v}(S,w)) \nabla_w \hat{v}(S,w) = \alpha \delta x(S)$

TD(λ) $\langle S_1, G_1^\lambda \rangle, \dots \langle S_{T-1}, G_{T-1}^\lambda \rangle$

Forward view lin.: $\Delta w = \alpha (G_t^\lambda - \hat{v}(S_t,w)) \nabla_w \hat{v}(S_t,w) = \alpha (G_t^\lambda - \hat{v}(S_t,w)) x(S_t)$

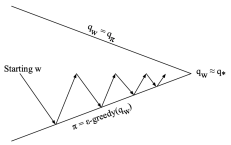
Backward view lin.: $\begin{bmatrix} \delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1},w) - \hat{v}(S_t,w) \\ E_t = \gamma\lambda E_{t-1} + x(S_t) \\ \Delta w = \alpha \delta E_t \end{bmatrix}$

} equivalent

Control with V.FA

goal: Find w so that $\hat{q}(S,A,w) \approx q_\pi$

- (Policy Eval) accurately estimate value of current policy π: $\hat{q} \approx q_\pi$
- (Policy Improv), when acting ε-greedily with respect to $\hat{q}$, we get better policy that moves closer to q_*



Action-Value FA

$\hat{q}(S,A,w) \approx q_\pi(S,A)$

- minimize error: $J(w) = \mathbb{E}_\pi[(q_\pi(S,A) - \hat{q}(S,A,w))^2]$
- use stochastic GD: $\Delta w = \alpha(q_\pi(S,A) - \hat{q}(S,A,w)) \nabla_w \hat{q}(S,A,w)$

Linear Action-value FA

$\hat{q}(S,A,w) = x(S,A)^T w = \sum_{j=1}^{N} x_j(S,A) w_j$

GD update: $\Delta w = \alpha(q_\pi(S,A) - \hat{q}(S,A,w)) x(S,A)$

Incremental Control Algorithms

- For MC, the target is the return G_t

$$\Delta \mathbf{w} = \alpha(G_t - \hat{q}(S_t, A_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

- For TD(0), the target is the TD target $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$

$$\Delta \mathbf{w} = \alpha(R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

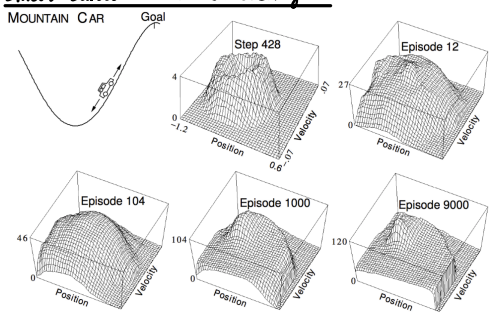
- For forward-view TD(λ), target is the action-value λ-return

$$\Delta \mathbf{w} = \alpha(q_t^\lambda - \hat{q}(S_t, A_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

- For backward-view TD(λ), equivalent update is

$$\delta_t = R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})$$
$$E_t = \gamma \lambda E_{t-1} + \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$
$$\Delta \mathbf{w} = \alpha \delta_t E_t$$

Linear Sarsa with Coarse coding



On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD(0)	✓	✓	✗
	TD(λ)	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD(0)	✓	✗	✗
	TD(λ)	✓	✗	✗

Batch methods

- GD is simple and appealing, but NOT sample efficient
- batch methods seek to find best fitting value-function given the agent's experience ("training data")

Least squares prediction

- we have $\hat{v}(s,w)$ that estimates $v_\pi(s)$
- we collected dataset of experience D, of state-value pairs $\langle S_t, v_t^\pi \rangle$ (actual return or bootstrapped target)

Goal: Find vector w that best fits these examples in least squares sense

$$LS(w) = \sum_{t=1}^{T} (v_t^\pi - \hat{v}(S_t, w))^2$$
$$\updownarrow$$
$$\mathbb{E}_{\langle S,v^\pi \rangle \sim D}[(v^\pi - \hat{v}(S,w))^2]$$

Stochastic GD with experienced replay

- as agent interacts we store transitions in a buffer

$$D = \{(S_i, v_i^\pi)\}_{i=1}^{N}$$
(MC/TD..)

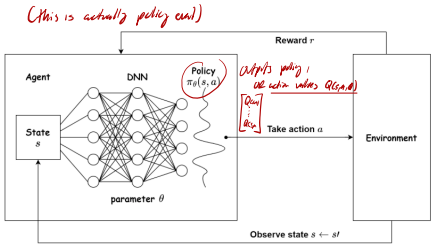
- repeat until converged:
  1. sample at random from D: $\langle S, v^\pi \rangle \sim D$
  2. compute prediction error: $\delta = v^\pi - \hat{v}(S,w)$
  3. compute gradient: $\nabla_w \hat{v}(S,w)$
  4. take SGD step: $w \leftarrow w + \alpha \delta \nabla_w \hat{v}(S,w)$

~ converges to $w^T = \arg\min_w LS(w)$

Experience Replay with Deep Q-networks

Q-network? - neural network that learns $Q^\pi(S,A) \approx Q(S,A,\theta)$ (weights)

(This is actually policy eval)

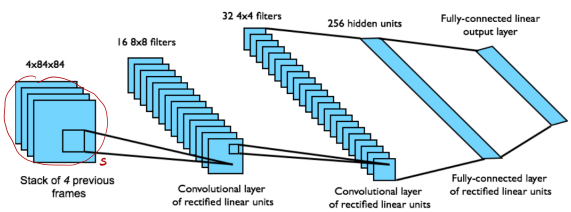


1. take action a_t to ε-greedy policy
2. store transition $(S_t, a_t, r_{t+1}, s_{t+1})$ in replay memory D
3. sample random mini batch of (S_i, a_i, r_i, s_i') from D
4. compute Q-learning targets for fixed w^-
5. optimize

$$J_i(w_i) = \mathbb{E}_{s,a,r,s' \sim D}[(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s,a,w_i))^2]$$

- using variant of SGD

DQN in Atari



Linear Least Squares prediction

- if we assume linear $\hat{v}(s,w) = x(s)^T w$, then instead of gradually arriving with many SGD, we can directly find best-fit solution
  1. collect our data: $\{(x(s_t), v_t^\pi)\}_{t=1}^{T}$ (target (TD/MC))
  2. form equations: $\sum_{t=1}^{T} x(s_t) x(s_t)^T w = \sum_{t=1}^{T} x(s_t) v_t^\pi$ (A / b)
  3. solve for w: $w = A^{-1} b$

- using Sherman-Morrison, solution in $O(N^2)$

Convergence

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	LSMC	✓	✓	-
	TD	✓	✓	✗
	LSTD	✓	✓	-
Off-Policy	MC	✓	✓	✓
	LSMC	✓	✓	-
	TD	✓	✗	✗
	LSTD	✓	✓	-

Least squares Q-Learning

- we approximate $\hat{q}(S,A,w) = x(S,A)^T w$ (Feature vector for state-action)
- we assume Δw is zero - we get linear system:

$$\underbrace{\sum_t x(S_t, a_t)[x(S_t, a_t) - \gamma x(S_{t+1}, \pi(S_{t+1}))]^T}_{A} w = \underbrace{\sum x(S_t, a_t) r_{t+1}}_{b}$$

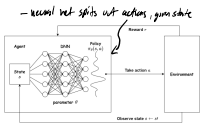
$$w = A^{-1} b$$

Policy Gradient

Policy based RL

- Instead of going through value function, we directly represent the policy itself with parameters θ

$$\pi_\theta(s,a) = \mathbb{P}[a|s,\theta]$$



- Neural net spits out action, given state
- We stay model-free

Advantages of policy RL

+ better convergence behaviour - no max operator w bootstrapping
+ effective in continuous or high dimensional action spaces
+ learns stochastic policies naturally
− typically converge to local rather than global optimum
− inefficient evaluating policy in high variance

Policy objective Functions

Goal: given parametric policy $\pi_\theta(a|s)$ Find best parameters θ^*

- We need obj. f. $J(\theta)$ - tells us "how good" a candidate policy is
- Then we do gradient ascent on $J(\theta)$ to improve policy π_θ

Episodic: Start-state value (compute G from s_1 following policy)

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}[G_1]$$

(we always begin in some start state s_1)

- update ∇_θ policy

1. Continuous: average-value (discounted future value)

- We care about how policy does on average over time
- Under π_θ, the Markov chain reduced over state until settle into stationary distribution $d^{\pi_\theta}(s)$

$$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$$

($d^\pi(s)$ is a long-run fraction of time the agent spends in each state s)

Objective: $J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$ ← $\mathbb{E}_{\pi_\theta}[\sum \gamma^t r_{t+1} | s]$

expected discounted average return

for each state s: (weigh in how much we visit s $d^{\pi_\theta}(s)$, multiply it by value we get, while we are there)

2. Continuous: long-run average reward per time-step (Undiscounted immediate reward rule)

$$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(a|s) R_s^a$$

← expected immediate reward we get after $a|s$

↑ probability of picking a in s

- Why? we want a steady performance rather than discounted sum
- We use no discount here

Policy Optimisation

Goal: Find θ^* that maximises $J(\theta)$ (remind)

Gradient-Free: Hill Climbing, Simplex, Genetic Algs..

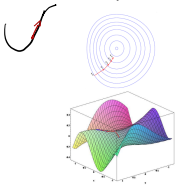
Gradient-based: - Stochastic gradient descent/ascent
- Conjugate Gradient
- Quasi-newton

Simple recipe

1. Pick policy objective $J(\theta)$ - start state, avg reward... - and view it as function of policy parameter θ
2. Compute policy gradient $\nabla_\theta J(\theta) = \begin{pmatrix} \frac{\partial J}{\partial \theta_1} \\ \vdots \\ \frac{\partial J}{\partial \theta_n} \end{pmatrix}$
3. Take small ascent step: $\Delta\theta = \alpha \nabla_\theta J(\theta)$

update: $\theta \leftarrow \theta + \Delta\theta$

repeat.



Computing gradients by Finite distances

- We have policy π_θ and objective $J(\theta)$ that we can only evaluate (eg running episodes) not differentiate analytically
We still want $\nabla_\theta J(\theta)$

Finite distance approximation: 1) For each k: (add tiny ε)

$$\theta^{(k)} = \theta + \varepsilon e_k$$

(e_k unit vector in dimension k)

2) Use Monte Carlo to estimate $J(\theta^{(k)})$ similarly estimate $J(\theta)$

3) Estimate approx: $\frac{\partial J}{\partial \theta_k} \approx \frac{J(\theta+\varepsilon e_k) - J(\theta)}{\varepsilon}$

$\vdots$

$\nabla_\theta J(\theta)$

- Simple, noisy, inefficient, sometimes effective

Score Function

$$\nabla_\theta \pi_\theta(s,a) = \pi_\theta(s,a) \frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)} = \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a)$$

(score function - how does the log probability of choosing a in s change if I tweak θ?)

Softmax policy

- We use probability of action proportional to

$$\pi_\theta(s,a) \propto e^{\phi(s,a)^T \theta}$$ (natural under)

Score function is

$$\nabla_\theta \log \pi_\theta(s,a) = \phi(s,a) - \mathbb{E}_{\pi_\theta}[\phi(s,\cdot)]$$

$$\sum_a \pi_\theta(a|s)\, \phi(s,a)$$

Gaussian Policy

$$\mu_\theta(s) = \phi(s)^T \theta$$

- Sample log rule: $a \sim N(\mu_\theta(s), \sigma^2)$
- continuous spaces
- easy to sample and differentiable

Score function: $\nabla_\theta \log \pi_\theta(a|s) = \frac{(a - \mu(s))\phi(s)}{\sigma^2}$

One-step MDPs

1. start in s due to distribution $d(s)$
2. pick $a \sim \pi_\theta(a|s)$
3. receive $R_{s,a}$, episode ends

$$J(\theta) = \mathbb{E}_{\pi_\theta}[r] = \sum_s d(s) \sum_a \pi_\theta(s,a) R_{s,a}$$

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

Policy Gradient Theorem

- multi-step MDPs

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

(replaces r, w/ long term value)

- discrete, finite

Monte Carlo Policy Gradient (REINFORCE)

- simplest actor-only method; Full episode returns multiplied by score-function gradients
- In discrete/continuous; but high variance!

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

Actor-Critic methods (Lower variance)

- an actor - policy $\pi_\theta(a|s)$
- a critic - learned action-value estimate $Q_w(s,a)$, for lower variance estimate $Q_w(s,a) \approx Q^{\pi_\theta}(s,a)$

(this is just policy eval (MC, TD), least squares...)

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

- Simple actor-critic algorithm based on action-value critic
- Using linear value fn approx. $Q_w(s,a) = \phi(s,a)^\top w$
  - Critic Updates w by linear TD(0)
  - Actor Updates θ by policy gradient

function QAC
Initialise s, θ
Sample $a \sim \pi_\theta$
for each step **do**
Sample reward $r = \mathcal{R}_s^a$; sample transition $s' \sim \mathcal{P}_{s,\cdot}^a$
Sample action $a' \sim \pi_\theta(s', a')$
$\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$
$\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s,a) Q_w(s,a)$
$w \leftarrow w + \beta \delta \phi(s,a)$
$a \leftarrow a', s \leftarrow s'$
end for
end function

Reducing Variance with Baseline

- G_t s can be very noisy from sample to sample, gradients estimates fingers crossed
- We can subtract any function of state $B(s)$ from our return, as long as it's expectation, then expected gradient remains unchanged.

$$\mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) B(s)] = 0$$

So we can safely do:

$$\Delta\theta \propto \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)(G_t - B(s))]$$

The best baseline - is state-value Function $B(s) = V^{\pi_\theta}(s)$

The advantage Function: $A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$ (critic estimate)

and rewrite policy gradient theorem:

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

Estimating Advantage Function

1) learn Q and V, then $\hat{A}(s,a) = Q_w(s,a) - V_v(s)$

2) (cheaper better, common way: TD error as advantage:

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

has expectation:

$$\begin{aligned} \mathbb{E}_{\pi_\theta}[\delta^{\pi_\theta}|s,a] &= \mathbb{E}_{\pi_\theta}[r + \gamma V^{\pi_\theta}(s')|s,a] - V^{\pi_\theta}(s) \\ &= Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s) \\ &= A^{\pi_\theta}(s,a) \end{aligned}$$

So we can just:

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

Critics at different time scales

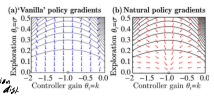
- For MC, the target is the return v_t (Gt)
$$\Delta\theta = \alpha(v_t - V_\theta(s))\phi(s)$$
- For TD(0), the target is the TD target $r + \gamma V(s')$
$$\Delta\theta = \alpha(r + \gamma V(s') - V_\theta(s))\phi(s)$$
- For forward-view TD(λ), the target is the λ-return v_t^λ
$$\Delta\theta = \alpha(v_t^\lambda - V_\theta(s))\phi(s)$$
- For backward-view TD(λ), we use eligibility traces
$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$
$$e_t = \gamma\lambda e_{t-1} + \phi(s_t)$$
$$\Delta\theta = \alpha\delta_t e_t$$

Actors at different time scales

- The policy gradient can also be estimated at many time-scales
$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\, A^{\pi_\theta}(s,a)]$$
- Monte-Carlo policy gradient uses error from complete return (Gt)
$$\Delta\theta = \alpha(v_t - V_v(s_t))\nabla_\theta \log \pi_\theta(s_t, a_t)$$
- Actor-critic policy gradient uses the one-step TD error
$$\Delta\theta = \alpha(r + \gamma V_v(s_{t+1}) - V_v(s_t))\nabla_\theta \log \pi_\theta(s_t, a_t)$$

Natural Policy Gradient

- replace ∇_θ with $G_\theta^{-1}\nabla_\theta J$, giving you ascent direction that
- really measures 'true' changes in policy, not just in parameters
- is invariant to how we write down the



Summary of Policy Gradient Algorithms

- The policy gradient has many equivalent forms

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\, v_t] && \text{REINFORCE} \\ &= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\, Q^w(s,a)] && \text{Q Actor-Critic} \\ &= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\, A^w(s,a)] && \text{Advantage Actor-Critic} \\ &= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\, \delta] && \text{TD Actor-Critic} \\ &= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\, \delta e] && \text{TD}(\lambda)\text{ Actor-Critic} \\ G_\theta^{-1}\nabla_\theta J(\theta) &= w && \text{Natural Actor-Critic} \end{aligned}$$

- Each leads a stochastic gradient ascent algorithm
- Critic uses policy evaluation (e.g. MC or TD learning) to estimate $Q^\pi(s,a)$, $A^\pi(s,a)$ or $V^\pi(s)$