1) Markov Process

Tuple (S, P_ss') = (set of states, state transition proba. matrix)

$$\mathcal{P}_{ss'} = P\left[S_{t+1} = s' | S_t = s\right]$$

Generates chain of **Markov states** governed by prob. transitions

A state s_t is Markov if and only if $P\left[s_{t+1}|s_t\right] = P\left[s_{t+1}|s_1, \dots s_t\right]$

Transient states (round) vs terminal states (square)

Stationary/homogeneous chain: P_ss' only depends on s,s' (not t)

MRP: Markov chain which emits rewards (S, P, R, gamma)

$\mathcal{R}_s = \mathbb{E}\left[r_{t+1}|S_t = s\right]$ Expected immediate reward collected upon departing s, at t+1

$\gamma \in [0,1]$ Discount factor (see advt.)

$R_t = r_{t+1} + \gamma r_{t+2} + \dots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$ Return = total discounted reward from t

The factor $\gamma \in [0,1]$ is how we discount the present value of future rewards.
The value of receiving reward r after $k+1$ time-steps is $\gamma^k r$.
The discount values immediate reward higher than delayed reward:

- γ close to 0 leads to "myopic" (short-sighted) evaluation.
- γ close to 1 leads to "far-sighted" evaluation.

State-value fn = expected return R starting from state s at time t

$$v(s) = \mathbb{E}\left[R_t|S_t = s\right]$$

Bellman Eq for MRPs (start from above)

$$\begin{aligned} v(s) &= \mathbb{E}\left[r_{t+1} + \gamma v(S_{t+1})|S_t = s\right] \\ &= \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s') \end{aligned}$$ If |S|=n, n eqs

$$\mathbf{v} = \mathcal{R} + \gamma \mathcal{P} \mathbf{v}$$ If |S|=n, n-dim vector **v**

Bellman Eq = linear, self-consistent => can solve **v** directly if small MRP (matrix inv.). **Iterative Methods** for larger MRPs: DP/MC/TD

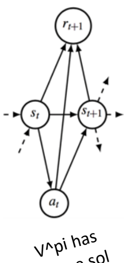
Probabilistic/stochastic or deterministic policy

A policy $\pi_t(a,s) = P\left[A_t = a|S_t = s\right]$ is the conditional probability distribution to execute an action $a \in \mathcal{A}$ given that one is in state $s \in \mathcal{S}$ at time t.

Deterministic: $\pi_t(s) = a$ vs stochastic: $\boldsymbol{a} \sim p_\pi(\boldsymbol{a}|\boldsymbol{s})$

$p_\pi(\boldsymbol{a}|\boldsymbol{s}) = \pi(\boldsymbol{a}|\boldsymbol{s}) \equiv \pi(a,s)$

2) MDP

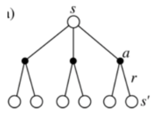


V^pi has unique sol

Value fn (self-consistent, linear)

$$V^\pi(s) = \mathbb{E}_\pi\left[R_t|S_t = s\right] = E\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|S_t = s\right]$$

Backup Diagram: transfer state value info from all s' to s (update/backup op)



To compute v(s): average over all possible traces and their reward

Another version of Bellman Eq (BEq)

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'}\left(\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right)$$

Value fns satisfy a set of recursive consistency eqs.

Policy Evaluation (PE): "the prediction prblm" = *how* to compute v fn for arbitrary policy pi

Iterative PE Algo: apply BEq to obtain better Vi(s)…Vk(s) estimates iteratively (until conv.): better approx. across steps + termination cond (largest diff in in v fn btw 2 iterations < threshold). Does full backup (all s' considered).

```
Input $\pi$, the policy to be evaluated
Initialize $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
    $\Delta \leftarrow 0$
    For each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx V^\pi$
```

State-Action value fn (cost-to-go)

$$Q^\pi(s,a) = \mathbb{E}\left[R_t|S_t = s, A_t = a\right] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|S_t = s, A_t = a\right]$$

relationship btw Q and V

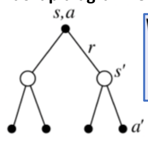
$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(s,a) Q^\pi(s,a)$$

Ordering of policies: $\pi \geq \pi'$ iff $V^\pi(s) \geq V^{\pi'}(s)$ for all $s \in \mathcal{S}$

Optimal V*: $V^*(s) = \max_\pi V^\pi(s), \forall s \in \mathcal{S}$ **Optimal pi*:** Therefore, the policy π^* that maximises the value function is the optimal policy. There is always at least one optimal policy. There may be more than one optimal policy.

Optimal Q*: $Q^*(s,a) = \max_\pi Q^\pi(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A} = \mathbb{E}\left[r_{t+1} + \gamma V^*(s_{t+1})|S_t = s, A_t = a\right]$

Backup diagram for Q^pi(s,a)



B. Optimality Eqs for V* and Q*

$$\begin{aligned} V^*(s) &= \max_a \sum_{s'} P\left[s'|s,a\right]\left(r(s,a,s') + \gamma V^*(s')\right) \\ &= \max_a \sum_{s'} \mathcal{P}^a_{ss'}\left(\mathcal{R}^a_{ss'} + \gamma V^*(s')\right) \end{aligned}$$

if |S|=n, there are n V* eqs; unique sol V* indpt of pi.

$$\begin{aligned} Q^*(s,a) &= \mathbb{E}\left[r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a')|S_t{=}s, A_t{=}a\right] \\ &= \sum_{s'} \mathcal{P}^a_{ss'}\left(\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^*(s', a')\right) \end{aligned}$$

BOEqs: *direct* sol to find pi* but hard + relies on assumptions: know env dynamics; comp resources; Markov property.

BOEq Convergence Theorem =>

For an MDP with a finite state and action space

1. *The Bellman (Optimality) equations have a unique solution.*
2. *The values produced by value iteration converge to the solution of the Bellman equations.*

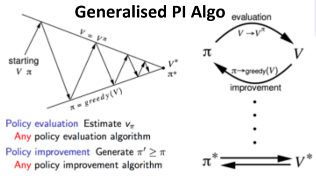
3) DP (solve/find pi* known MDPs)

Assume: finite MDP + know perfect model of env (P & R) + prblm w/ optimal substructure and overlapping subprblms. Simplify prblm by breaking it down into simpler subprblms recursively (Principle of Optimality; BEq as relation btw value of larger prblm & values of subprblms).

Advt:
- can do synchr (all states backed up in \\: 2 copies of V) or asynchr (1 copy of V, in-place; sig. reduced computation + still conv. guaranteed) updates
- bootstrapping: efficient use of data (thx optimal substructure

Disv:
- model-based
- curse of dimensionality (>mil. states)
- even 1 backup too costly

Generalised PI Algo


starting V π
V = Vπ
π = greedy(V)
V*
π*
evaluation
V → Vπ
π
V
π→greedy(V)
improvement
π*
V*
Policy evaluation Estimate v_π
Any policy evaluation algorithm
Policy improvement Generate $\pi' \geq \pi$
Any policy improvement algorithm

Policy Improvement Theorem

Let π and π' be any two deterministic policies such that $\forall s \in \mathcal{S}$: $Q^\pi(s, \pi'(s)) \geq V^\pi(s)$.
Then π' must be as good or better than π: $V^{\pi'}(s) \geq V^\pi(s), \forall s \in \mathcal{S}$.

Policy Iteration (PI): conv. guaranteed; eval⇔imprv

Once a policy, π , has been improved using V^π to yield a better policy π', we can compute $V^{\pi'}$ and improve it again to π'', to yield an even better $V^{\pi''}$. We can thus obtain a sequence of monotonically improving policies and value functions. This way of finding an optimal policy is called policy iteration.

Principle of Optimality

A policy $\pi(a|s)$ achieves the optimal value from state s, $V^\pi(s) = V^*(s)$, if and only if

1. *For any state s' reachable from s, i.e. $\exists a : p(s', s, a) > 0$*
2. *π achieves the optimal value from state s', $V^\pi(s') = V^*(s')$*

One drawback to policy iteration is that each iteration involves a full policy evaluation (which can be a protracted iterative computation requiring multiple sweeps through the state set). If policy evaluation is done iteratively, then convergence exactly to V^π occurs only in the limit. Do we need to wait for policy evaluation to converge to V^π?

We can introduce a stopping condition:

- ϵ-convergence of value function (e.g. $\forall s : V_{100}(s) - V_{101}(s) \leq \epsilon$)
- Stop after k iterations of iterative policy evaluation

k = 1 => VI

Value Iteration (VI): PI w/ 1 iteration only of pi eval

BOEq = update rule applied iteratively (1-step look ahead)

$$V^*(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^*(s')]$$

Unlike PI, no explicit policy pi: Output a deterministic policy, π, such that $\pi(s) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$

4) MC (for V estimation)

Learn from complete episodes of sample traces (no bootstrapping). V(s) = mean over empirical returns observed after visits to s (*not* expected R; will converge to it) = MC Policy Evaluation by sampling values for V. Incremental updates MC.

First-visit vs Every-visit MC: record return of episode E from 1st occurrence vs every occurrence of s in E, for all s.

Vanilla MC: update pi *once* after X episodes. **Batch MC**: update pi every *batch_size* episodes. **Online MC**: update pi *every* episode

We can now update value functions without having to store sample traces:

1. Update $V(s)$ incrementally after step $s_t, a_t, r_{t+1}, s_{t+1}$
2. For each state s_t with return R_t (up to this point) and $N(s)$ the visit counter to this state:

$$\begin{aligned} N(s_t) &\leftarrow N(s_t) + 1 \\ V(s_t) &\leftarrow V(s_t) + \frac{1}{N(s_t)}(R_t - V(s_t)) \end{aligned}$$

Moreover, if the world is non-stationary, it can be useful to track a running mean, i.e. by gradually forgetting old episodes.

MC update at t $V(s_t) \leftarrow V(s_t) + \alpha(R_t - V(s_t))$ **Alpha = lr**

The parameter α controls the rate of forgetting old episodes

Advt:
- model-free (no MDP knowledge needed: R/P) since R sampled
- fights curseOfDim through sampling (sample backups)
- cost of backups = constant (not exp.) + indept of N=|S|
- zero bias
- good convergence properties (even w/ FA)
- not v sensitive to initial value
- v simple to understand and use

Disv:
- only episodic MDPs w/ terminal states
- high variance: sampled R depends on many random A,P,R

Usually more effective in non-Markov env (no Markov assumed); TD exploits Markov property so TD more efficient in Markov env.

Model-free prediction (policy eval): **estimate** v fn of **unknown** MDP
Model-free control (policy improv): **optimise** v fn **=> see 6)**

5) TD (for V estimation)

Advt:
- combines desirable properties of DP & MC: bootstrapping & sampling
- model-free
- learn directly from episodes of exp; works for non-episodic episodes (incomplete) too
- learn from incomplete episodes or w/o terminal state or before terminal state (online after every step).
- low variance: TD target (depends on one random A,P,R) is much lower variance than the sampled return (used in MC) that depends on *many* random A,P,R
- usually more efficient than MC
- TD, esp. TD(0), converges to V^pi(s)

Incremental update after each t

$$V(s_t) \leftarrow V(s_t) + \alpha\left(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right)$$

We update the value of $V(s_t)$ towards the estimated return $r_{t+1} + \gamma V(s_{t+1})$ Note, how we are combining a measurement r_{t+1} with an estimate $V(s_{t+1})$ to produce a better estimate $V(s_t)$.

$r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ is the Temporal Difference Error

$r_{t+1} + \gamma V(s_{t+1})$ is the Temporal Difference Target

Disv:
- some bias: TD target is biased estimate of V^pi(s) as it relies on estimate of state s_t+1 (bootstrapping)
- convergence not guaranteed w/ FA
- more sensitive to initial value than MC

Bootstrapping = update involves an estimate
Sampling = update does not involve an expected value

TD control: on-policy (**SARSA**) vs off-policy (**Q-learning**)

Apply TD to Q(S,A) + use e-greedy policy improv. + update every t

SARSA: Use GPI like in DP/MC, but eval./pred. w/ TD on Q (conv.)

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left(r + \gamma Q(S',A') - Q(S,A)\right)$$

Q-learning (w/ assumpt. of coverage): improve pi (greedy wrt Q(s,a)) **and** pi' (e-greedy wrt Q(s,a)). Q-learning update (*R* = immediate r):

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left(R + \gamma \max_{a'} Q(S',a') - Q(S,A)\right)$$

No explicit policies in algo: pi implicit in greedy term; pi' = e-greedy version of pi. Both updated every *t* bc Q updated every *t*.
TD(0) here: immediate r + 1-step-look-ahead with Q(s',a')

6) Model-Free Control

V/Q learned by MC/TD & follow GPI Algo w/ approximate V/Q and pi converging to pi*

MC Policy Improvement: make pi (e-)greedy wrt current **Q fn** (**no** model needed to build greedy pi since Q, not V)

For any action-value function $Q(s,a)$, the corresponding greedy policy is the one that, for each $s \in \mathcal{S}$ deterministically chooses an action with maximal action-value: $\pi(s) = \arg\max_a Q(s,a)$

Policy improvement then can be done by constructing each π^{k+1} as the greedy policy with respect to Q^{π_k}.

For the approx. V/Q -> true V/Q and the Policy Improv. Theorem w/ MC to work, assume: inf. num of episodes + exploring starts (random s0). Then, MC can find pi* given only sample episodes + no knowledge of env dynamics

=> **avoid exploring starts assumption**: ensure agent continues to select them => on/off policy methods

On-policy (learn pi from exp sampled from pi) vs **Off-policy** (learn **target pi** from exp sampled from **behavior pi'**)

on-policy method -> soft policies -> epsilon-greedy policies)

Soft policies have in general $\pi(a,s) > 0 \ \forall s \in \mathcal{S}, \forall a \in \mathcal{A}$. I.e. we have a finite probability to explore all actions.

e.g., on-policy first-visit MC control (for e-soft policies), batch or iterative learning for control

ϵ-greedy policy with $\epsilon \in [0,1]$.

$$\pi(s,a) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A(s)|}, & \text{if } a^* = \underset{a}{\text{argmax}} Q(s,a) \\ \frac{\epsilon}{|A(s)|}, & \text{if } a \neq a^* \end{cases}$$

7) Function Approx. (FA)

Prblm (tabular RL): too many states/actions (CurseOfD) to be storing values of V(S) and Q(S,A) as lookup tables.
SOL: estimate V/Q fn w/ FA. Generalise from seen states to unseen states. Update param w using MC/TD learning.

$$V^\pi(s) \approx \hat{V}(s,\mathbf{w}) \qquad Q^\pi(s,a) \approx \hat{Q}(s,a,\mathbf{w})$$

FA: ANN, decision tree, nearest-neighbour…

Assumpt: differentiable fn, non-stationary & non-iid data
GD goal: find **w** minimizing MSE btw approx. Vhat(s,w) and true V^pi(s) fn

$$J(\mathbf{w}) = \mathbb{E}\left[\left(V^\pi(s) - \hat{V}(s,\mathbf{w})\right)^2\right]$$

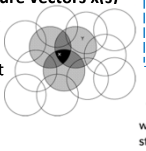
$$\Delta\mathbf{w} = -\frac{1}{2}\alpha\nabla_\mathbf{w}J(\mathbf{w}) = \alpha\mathbb{E}\left[\left(V^\pi(s) - \hat{V}(s,\mathbf{w})\right)\nabla_\mathbf{w}\hat{V}(s,\mathbf{w})\right]$$

SGD samples the gradient & the average update = full update

$$\Delta w = \alpha\left(V^\pi(s) - \hat{V}(s,\mathbf{w})\right)\nabla_\mathbf{w}\hat{V}(s,\mathbf{w})$$

States represented as hand-engineered feature vectors x(s)

1) Coarse coding: represent s w/ overlapping binary features (if s in a circle, 1, else 0).
e.g. **tile coding** (suited for computer, efficient on-line learning)



2) Radial Basis Fns (RBF): generalise coarse coding to continuous-valued features ([0,1]).

$$\phi_s(i) = \exp\left(-\frac{|s-c_i|^2}{2\sigma_i^2}\right)$$

where each $\phi_s(i)$ is the i-th basis function that maps a continuous state space variable s. Each RBF is centered at location c_i and has width σ_i.

MC/TD for evaluation (V/Q FA)

Pseudocode for MC learning **w**

```
Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, S_1, A_1, \dots, R_T, S_T$ using $\pi$
    Loop for each step of episode, $t = 0, 1, \dots, T-1$:
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha[G_t - \hat{v}(S_t,\mathbf{w})]\nabla\hat{v}(S_t,\mathbf{w})$
```

Return R_t is an unbiased, noisy sample of true value $V^\pi(s_t)$
We apply supervised learning to "training data" of state return trace:

$$(s_1, r_1), (s_2, r_2), \dots, (s_T, r_T) \quad (7)$$

For example, using linear Monte-Carlo policy evaluation

$$\Delta\mathbf{w} = \alpha(R_t - \hat{V}(s,\mathbf{w}))\nabla_\mathbf{w}\hat{V}(s_t,\mathbf{w}) \quad (8)$$
$$= \alpha(R_t - \hat{V}(s,\mathbf{w}))\mathbf{x}(s_t) \quad (9)$$

Monte-Carlo evaluation converges to a local optimum, even when using non-linear value function approximation (provable)

MC/TD FA = **linear** V fn approx.:

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

Update **rule** = lr x pred error x feature value **x**

The TD-target $R_{t+1} + \gamma\hat{V}(s_{t+1},\mathbf{w})$ is a biased (single) sample of the true value $V^\pi(s_t)$
We still perform supervised learning on "digested" training data:

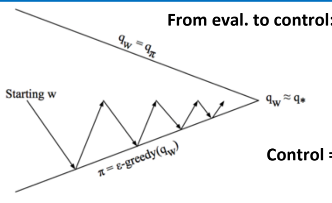
$$(s_1, r_2 + \gamma\hat{V}(s_2,\mathbf{w})), (s_2, r_3 + \gamma\hat{V}(s_3,\mathbf{w})), \dots, (s_T, r_T) \quad (10)$$

For example, using linear TD

$$\Delta\mathbf{w} = \alpha(r + \gamma\hat{V}(s',\mathbf{w}) - \hat{V}(s,\mathbf{w}))\nabla_\mathbf{w}\hat{V}(s_t,\mathbf{w}) \quad (11)$$
$$= \alpha(r + \gamma\hat{V}(s',\mathbf{w}) - \hat{V}(s,\mathbf{w}))\mathbf{x}(s_t) \quad (12)$$

Linear TD converges "close" to the global optimum (provable). This does not extend to non-linear TD (see Sutton & Barto, 2018).

From eval. to control: FA in GPI



Control = find v/q*

1. Policy evaluation: Approximate policy evaluation $\hat{Q}(s,a,\mathbf{e}) \approx Q^pi(s,a)$
2. Policy improvement: ϵ-greedy policy improvement

8) Deep-Q Learning

Use DL to replace hand-engineering of state space features w/ learning features from state data directly.
DQN (Mnih et al., 2015) Atari: Q(S,A) approx. by CNN w/ raw pixel inputs and discrete action outputs

Given our TD Q-learning update:

$$Q(s_t,a) \leftarrow Q(s_t,a) + \alpha[r_{t+1} + \gamma\max Q(s_{t+1},a') - Q(s_t,a)]$$

we want to learn $Q(s_{t+1},a';w)$ as a parametrised function (a neural network) with parameters w.
We can define the TD error as our learning target that we want to reduce to zero.

$$\text{TD error}(w) = r_{t+1} + \gamma\max_{a_{t+1}} Q(s_{t+1},a';w) - Q(s_t,a;w)$$

Taking the gradient of E wrt w we obtain:

$$\Delta w = \alpha[r + \gamma\max_{a_{t+1}} Q(s_{t+1},a_{t+1};w) - Q(s_t,a;w)]\nabla_w Q(s_t,a;w)$$

Engineering needed to solve Atari games w/ DQN ("regular Deep-Q learning"):

1) Experience Replay: CNN overfitting latest experienced episodes. Inefficient use of interactive experience + highly correlated training samples (agent's recent actions generated from recent policy outputted). **SOL**: experiences (traces) stored and replayed in mini-batches to train on more than just last episode (makes data more iid); instead of running Q-learning on each s-a pair as they occur, experience replay buffer stores the traces sampled. Exp reuse (random sampled) => data-efficient (higher learning speed) + avoid catastrophic forgetting of R associated w/ replayed transitions

2) Target Network: unstable training (e.g. resonance effect, divergence) bc bootstrapping a continuous state space repr. SOL: slow learning down (resonance dampener) using target network Q'. Initialise main network Q and target network Q'. Use Q' to calculate TD-error. Infrequently set Q'=Q (params). Gives highly fluctuating Q time to settle (relaxation time) before updating Q'

3) Clipping of Rewards: all positive rewards = +1, all negative rewards = -1 to avoid diff reward scales making training unstable.

4) Skipping of Frames: reduce comp. cost and accelerate training time + make the game run at speed comparable to human RT by only using every 4 video game frames as input (60Hz => 15Hz).

DQN config (49 Atari 2600 games)
Input: 84x84x4 preprocessed image. 3 conv. => 2 FC (ReLU) => FC output layer RMSProp, minibatch=32, e-greedy behav. pi (e=1->0.1)/Trained for 50million frames (38days game exp) + replay memory = 1million

Prblm w/ **regular Deep-Q learning**: **maximisation bias** = when taking max over all actions w/ (very) finite data, we may always **overestimate** values. **SOL: Double Q-learning (DDQN)**

Complication: The maximum Q value may be overestimated (variance-bias problem), because an unusually high value from the main network Q does not mean that there is an unusually high value from the target network Q'.

The problem with (Deep) Q-Learning is that the same samples (i.e. the Q network) are being used to decide which action is the best (highest expected reward), and the same samples are also being used to estimate that specific action-value.

DDQN (van Hasselt et al., 2017, AAAI): Use target network Q' for estimating best action selection; regular Q for estimating Q-value of s-a pair (or converse). Reduces frequency by which max Q-value may be overestimated bc less likely that both networks are overestimating the same action.

DDQN learning: more realistic (closer to final values), more stable, less biased (far less systematic overshooting).
Also see **Double Dueling Q networks** for Double-Q learning

9) Policy Gradients

9) Policy Gradients. Policy-based methods: find pi* without V/Q fns. Faster convergence and often better for continuous and stochastic envs than **value-based methods** 1)-8).
= look directly at parametrised pi_theta w/ params theta. Optimise by looking at traces that a policy pi_theta rolls out to correlate them w/ the rewards they incur.

Probability to observe a trace tau depends on (the policy weights theta) of pi:

Hard to measure — Hard to model

$$p(\tau) = p(s_1,a_1,\dots,s_T,a_T) = p(s_1)\pi_\theta(a_2|s_1)p(s_2|s_1,a_2)p_\theta(s_2,a_2,\dots,s_T,a_T) = p(s_1)\prod_{t=1}^{T}\pi_\theta(a_t|s_t)p(s_{t+1}|s_t,a_t)$$

Finite horizon
$$\theta^* = \arg\max_\theta \mathbb{E}\left[\sum_{t=1}^{T} r(s_t,a_t)\right]_{\tau\propto p_\theta(\tau)}$$

Infinite horizon
$$\theta^* = \arg\max_\theta \mathbb{E}\left[\sum_{t} r(s_t,a_t)\right]_{\tau\sim p_\theta(\tau)}$$

$\nabla_\theta J(\theta)$

pi* obtained from theta* instead of V*/Q*, where theta* = theta giving maximum average return.
Rationale: for continuous envs, infinite states/actions to estimate => value-based method = comp. expensive, esp. w/ GPI where policy improv step needs full scan of action space (max over A). A value fn may be used to learn theta but is not required for action selection.

Can use any parametric supervised ML model to learn pi(a|s; learned theta).
Use **gradient ascent** as we want to max performance.

Performance measure (episodic case)

$$J(\theta) = V^\pi(s_0) = \mathbb{E}\left[\sum_{t=1}^{T} r(s_t,a_t)\right]_{\tau\propto p_\theta(\tau)}$$

Difficulty of computing policy gradient:
Depends on traces tau: meaning we need derivatives on action selection and the stationary dist of states p(s), both determined by pi(a|s, theta). Given env is also generally unknown, difficult to estimate the effect of a policy update on state dist.

Approach 1: Finite Difference Grad *Approx.* (numerical)

$$\frac{\partial J}{\partial\theta_k} \approx \frac{J(\boldsymbol{\theta} + \boldsymbol{u}_k\epsilon) - J(\boldsymbol{\theta})}{\epsilon}$$

Simple, noisy, inefficient, sometimes effective; works for arbitrary pi (even non-diff ones)

Approach 2: Direct Policy Gradients (using log trick)

1. average return approx. by empirical mean over N traces

$$J(\theta) = \mathbb{E}\left[\sum_{t=1}^{T} r(s_t,a_t)\right]_{\tau\propto p_\theta(\tau)} \approx \frac{1}{N}\sum_{i=i}^{N}\sum_{t} r(s_{i,t},a_{i,t})$$

$$\underline{\pi_\theta(\tau)\nabla_\theta\log\pi_\theta(\tau)} = \pi_\theta(\tau)\frac{\nabla_\theta\pi_\theta(\tau)}{\pi_\theta(\tau)} = \underline{\nabla_\theta\pi_\theta(\tau)}$$

2. log trick to derive gradient

$$\nabla_\theta J(\theta) = \int \underline{\nabla_\theta\pi_\theta(\tau)}r(\tau)d\tau = \int \underline{\pi_\theta(\tau)\nabla_\theta\log\pi_\theta(\tau)}r(\tau)d\tau = E_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta\log\pi_\theta(\tau)r(\tau)]$$

$$\nabla_\theta\log\pi_\theta(\tau) = \nabla_\theta\left[\cancel{\log p(\mathbf{s}_1)} + \sum^{T}\log\pi_\theta(\mathbf{a}_t|\mathbf{s}_t) + \cancel{\log p(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t)}\right]$$

PG = trial-and-error like MC learning

$$\nabla_\theta J(\theta) = E_{\tau\sim\pi_\theta(\tau)}\left[\left(\sum_{t=1}^{T}\nabla_\theta\log\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)\right)\left(\sum_{t=1}^{T} r(\mathbf{s}_t,\mathbf{a}_t)\right)\right]$$

= one version of the **Policy Gradient Theorem (policy-centered equivalent of Bellman Theorem.** Implemented in **REINFORCE**

REINFORCE: insightful first-shot at PGs (popularising it)

1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run the policy)
2. $\nabla_\theta J(\theta) \approx \sum_i\left(\sum_t\nabla_\theta\log\pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right)\left(\sum_t r(\mathbf{s}_t^i,\mathbf{a}_t^i)\right)$
3. $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$ In the limit of large amounts of data => model will converge to theta*

No bias but high variance in the sampled trajectories => difficult to stabilise theta. Any erratic journey can cause suboptimal shift in the policy dist. Reduce variance (being smarter correlating rewards with trajectories) by subtracting a baseline from reward term (keep values smaller) or using an advantage term (Schuman et al., 2016) (**see 10)**).

10) Actor-Critic Methods

10) Actor-Critic Methods => model-learning to improve the model (!= direct RL improving V/Q/pi, as in 1-9)). A-C = improving V/Q/pi via model = **indirect/model-based RL => get better pi w/ fewer interactions**. Can have both indirect and direct methods => Split RL model into an **actor** (compute a based on s) and **critic** (produce Q-values of s,a = "model").
Actor: input=s, output=a. Controls how agent behaves by learning pi* (policy-based learning). Critic: evaluates a (value-based learning). The two models "compete" and each gets better at its role: key point = **combined arch learns better than the 2 separate networks would individually**.

A simple Q-driven policy-gradient actor-critic.

Algorithm 1 Q Actor Critic

```
Initialize parameters $s, \theta, w$ and learning rates $\alpha_\theta, \alpha_w$; sample $a \sim \pi_\theta(a|s)$.
for $t = 1 \dots T$: do
    Sample reward $r_t \sim R(s,a)$ and next state $s' \sim P(s'|s,a)$
    Then sample the next action $a' \sim \pi_\theta(a'|s')$
    Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s,a)\nabla_\theta\log\pi_\theta(a|s)$; Compute
    the correction (TD error) for action-value at time t:
        $\delta_t = r_t + \gamma Q_w(s',a') - Q_w(s,a)$
    and use it to update the parameters of Q function:
        $w \leftarrow w + \alpha_w\delta_t\nabla_w Q_w(s,a)$
    Move to a $\leftarrow a'$ and s $\leftarrow s'$
end for
```

Advantage A-C (A2C): learn A, not Q: $Q(s,a) = V(s) + A(s,a)$
Eval. of action a based on how much it *improves* s value. A2C reduces high var. of actor (=PG method) w/o adding bias, stabilises model in training
Asynchronous (A3C): multiple indpt agents (networks) interact w/ diff env copy in // => explore bigger part of S-A space in much less time. Trained in // and update periodically (asynchronously) a global network holding shared params. After each update, each agent copies global params + resume indpt exploring until next update (cool if large scale simulators). Info flow btw agents and global network, and btw agents (given param reset).

DDPG: deterministic policy, model-free, off-policy, A-C. Continuous actions version of DQN. Uses bootstrapping to learn Q fn, and to learn pi from estimated Q fn. Explore in continuous space using Gaussian policy (~= discrete action e-greediness). Replay buffer, minibatch, target network (=DQN) control learning variability. MSBE. PG-based actor. Smooth updates to A&C target networks $\theta^{t+1} = \beta\theta^t + (1-\beta)\theta'^t, 0 < \beta \ll 1$