Reinforcement Learning

Deep Reinforcement Learning

Based on An Introduction to Deep Reinforcement Learning**
+ Sutton/Barto* Sections 9.7 and 15.7

Michael Hahsler, SMU

*Sutton and Barto, Reinforcement Learning: An Introduction, 2nd edition, MIT Press, Cambridge, MA, 2018

**Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare and Joelle Pineau (2018), "An Introduction to Deep Reinforcement Learning", *Foundations and Trends in Machine Learning*: Vol. 11, No. 3-4. DOI: 10.1561/2200000071.

Topics of this Course

- Introduction to reinforcement learning
- Markov decision processes
- Part I: Tabular Methods
    - Dynamic programming
    - Monte Carlo methods
    - Temporal-difference learning
    - Multi-step bootstrapping
    - Planning and learning with tabular methods
- Part II: Approximate Solution Methods
    - Prediction and Control using Approximation
    - Eligibility Traces
    - Policy Gradient Methods
- **Part III: Modern RL Methods**
    - **Deep Reinforcement Learning**
    - Current Applications

Summary of Notation

General

X	capital letters: random variables
x, p	lower-case letters: realizations of random variables or scalar functions
$\boldsymbol{w}$	Bold lower-case letters: real-valued vectors (even if random variables)
$\boldsymbol{W}$	bold capitals: matrices
α	Greek letters: parameters (vectors if in bolt)
$\Pr\{X = x\}$	probability that a random variable X takes on the value x
$X \sim p$	random variable X selected from distribution $p(x) = \Pr\{X = x\}$
$\mathbb{E}[X]$	expectation of a random variable X, i.e., $\mathbb{E}[X] = \sum_x p(x)x$
$\mathrm{argmax_a}\ f(a)$	a value of action a at which $f(a)$ takes its maximal value

Value Function

G_t	return (cumulative reward) following time t
$G_{t:h}$	return from t to h (discounted and corrected)
$v_\pi(s)$	value of state s under policy π (expected return)
$v_*(s)$	value of state s under the optimal policy
$q_\pi(s, a)$	value of taking action a in state s under policy π
$q_*(s, a)$	value of taking action a in state s under the optimal policy
V, V_t	array estimates of state-value function v_π or v_*
Q, Q_t	array estimates of action-value function q_π or v_*

MDP

s, s'	states
a	an action
r	a reward
$\mathcal{S}$	set of all (nonterminal) states, $\mathcal{S}^+$ are all states
$\mathcal{A}(s)$	set of all actions available in state s
γ	discount-rate parameter
t	discrete time step
T	final time step of an episode (a.k.a. horizon)
A_t	random variable for the action at time t
S_t	random variable for the state at time t
R_t	random variable for the reward at time t
$p(s', r \mid s, a)$	probability of transition to state s' and receiving reward r, from state s taking action a.
$p(s' \mid s, a)$	probability of transition to state s' fom state s taking action a.
$r(s, a)$	expected immediate reward from state s after action a.
$r(s, a, s')$	expected immediate reward from state s to s' with action a.
$\pi(a \mid s)$	probability of taking action a in state s under stochastic policy π
$\pi(s)$	action taken in state s under deterministic policy π

From Tabular RL to Deep RL

	Tabular Methods	Linear Functions Approximation	Deep RL (DRL)
Representation (value function or policy)	Table	Linear combination of basis functions	Deep ANN
Methods	Q-learning, Sarsa	Stochastic Gradient Descent (SGD) used in Sarsa	Stochastic Gradient Descent (SGD) used in DQN and Actor-Critic methods.
Convergence	With sufficient exploration, appropriate learning rates.	Can converge to a good approximation of the optimal value function or policy.	Non-linearity makes convergence challenging.
Limitations	Table size for big state/action spaces.	Performance depends on choice of basis functions.	• Converges to local optima • Instability • Hyperparameter sensitivity

Tabular Methods → Linear Functions Approximation:

+ scales to large state/action spaces

+ generalizes over states

Linear Functions Approximation → Deep RL (DRL):

+ more powerful approximation for high-dimensional and continuous state and action spaces

Value-based DRL Methods

Use ANNs to represent a parameterized value function that is updated via gradient descent.

Remember: Value Functions

- Immediate reward: $r_t = \mathbb{E}_{a \sim \pi(s_t, \cdot)}\, r(s_t, a, s_{t+1})$
- Value functions:
  - $V^\pi(s) = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid S_t = s\,, \pi]$
  - $Q^\pi(s, a) = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid S_t = s, A_t = a\,, \pi]$
- Bellman equation:

$$Q^\pi(s, a) = \sum_{s' \in \mathcal{S}} p(s, a, s') \Big(r(s, a, s') + \gamma Q^\pi\big(s', a = \pi(s')\big) \Big)$$

- Optimality condition:
  $V^*(s) = \max_{\pi \in \Pi} V^\pi(s)$ and $Q^*(s, a) = \max_{\pi \in \Pi} Q^\pi(s, a)$
- Optimal policy: $\pi^*(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q^*(s, a)$

New: Advantage Function

- The "advantage" of choosing action a over the baseline given by the state value is

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

- Positive advantage means action a is better than $\pi(s)$
- For the optimal π^*:

$$A^{\pi^*}(s,a) = \begin{cases} 0 & \text{if } s = s^* \\ < 0 & \text{otherwise.} \end{cases}$$

- V, Q and A can be estimated using Monte Carlo methods (unbiased).
- V, Q and A can be used to calculate a TD-error.

Remember: Tabular Q-Learning

- An extremely influential method developed by Watkins (1989).
- Tabular update rule only uses Sars:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[\underbrace{\overbrace{R_{t+1} + \gamma \max_a Q(S_{t+1}, a)}^{\text{target}} - \overbrace{Q(S_t, A_t)}^{\text{prediction}}]}_{\text{TD-error}}$$

- Q directly approximates q_* independent of the behavior policy being followed! → Off-policy
- Converges to the optimal value function:
  - State-action pairs are represented discretely (in a table).
  - All action are repeatedly sampled in all states (keeps exploring).

- **Issue**: $Q(s, a)$ can often not be represented in a table due to too many action-value pairs. Use a parameterized approximation $Q(s, a; \boldsymbol{\theta})$

Fitted Q-Learning

- Use a parameterized approximation $Q(s, a; \boldsymbol{\theta})$
- Initial setting for $\boldsymbol{\theta}_0$ such that all Q-value are close to 0 (helps with learning).
- Target value at iteration k:

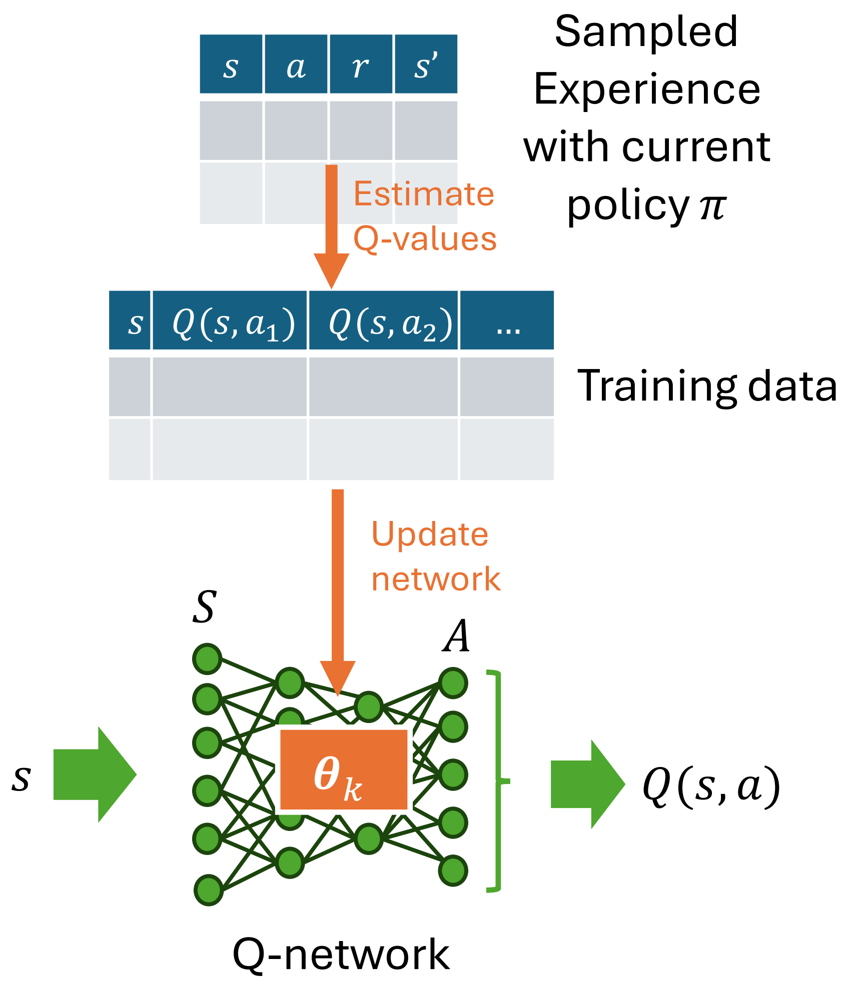
$$U_k = r + \gamma \max_{a'} Q(s', a'; \boldsymbol{\theta}_k)$$

- Update:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha \Big(\underbrace{\overset{\text{target}}{U_k} - \overbrace{Q(s, a, \boldsymbol{\theta}_k)}^{\text{prediction}}}_{\text{TD-error}} \Big) \nabla Q(s, a, \boldsymbol{\theta}_k)$$

Neural fitted Q-learning (NFQ; Riedmiller, 2005)

- Use an ANN called the **Q-network** as the approximator for the Q-function.
- **Issue**: Online learning does not work well.
- **Solution**:
  - Learn the Q-network offline: **batch learning** with experience replay from a sample set of experience containing (s, a, r, s') tuples.
  - Use **supervised learning (regression)** with the sample as the training set.
  - The regression target Q-values are estimated from the tuples using the Bellman equation: $Q(s,a) = r + \gamma \max_{a'} Q(s', a')$
- **More issues**:
  - May not converge or become unstable (uses approximation).
  - Q-values tend to be overestimated due to the max operator in the target.



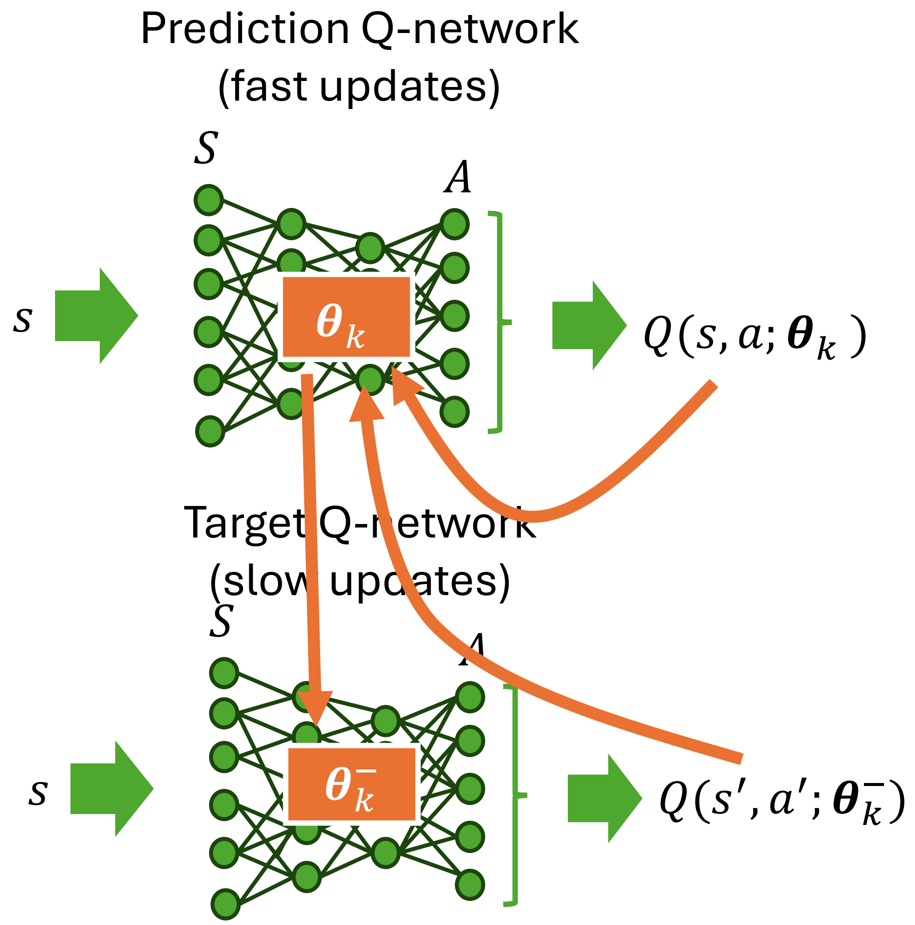
Deep Q-Network (DQN; Minh et al, 2015)

- DNQ addresses the **instability** of NFQ.
- **Idea**: reduce instability by calculating the target from a second Q-network that is updated slowly:

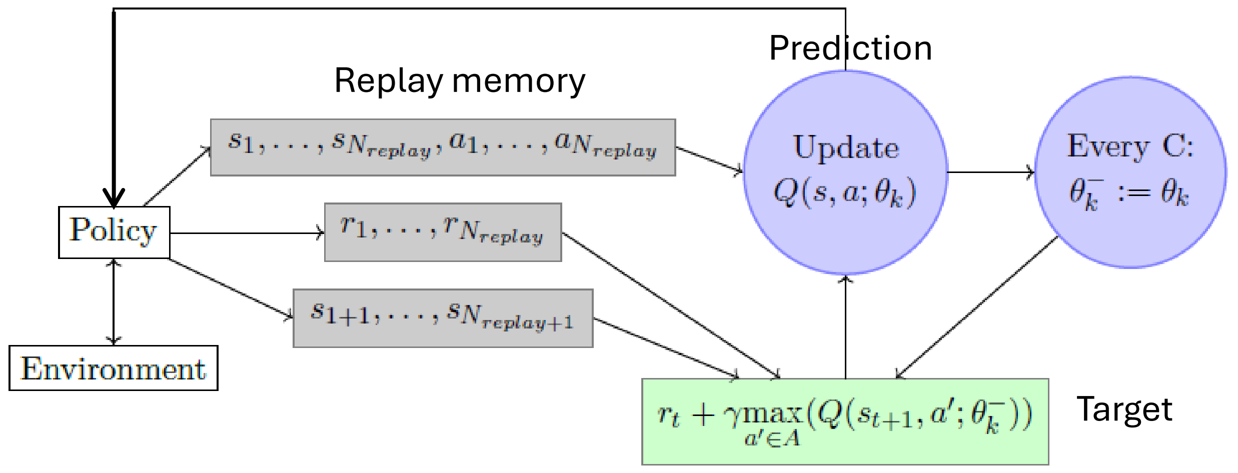
$$U_k = r + \gamma \max_{a'} Q(s', a'; \boldsymbol{\theta}_k^-)$$

- **Updates**:
  - $\boldsymbol{\theta}_k$ is updated every iteration:
    $$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha\big(U_k - Q(s, a, \boldsymbol{\theta}_k)\big)\nabla Q(s, a, \boldsymbol{\theta}_k)$$
  - $\boldsymbol{\theta}_k^-$ is updated only every C iterations by $\boldsymbol{\theta}_k^- = \boldsymbol{\theta}_k$.

- **Effect**: Target U_k estimates are kept constant for C iterations, instabilities cannot propagate quickly.



DQN in Online Settings


Prediction
Replay memory
$s_1, \ldots, s_{N_{replay}}, a_1, \ldots, a_{N_{replay}}$
Update $Q(s,a;\theta_k)$
Every C: $\theta_k^- := \theta_k$
Policy
$r_1, \ldots, r_{N_{replay}}$
$s_{1+1}, \ldots, s_{N_{replay}+1}$
Environment
$r_t + \gamma \max_{a' \in A}(Q(s_{t+1}, a'; \theta_k^-))$
Target

1. Initialize $\boldsymbol{\theta}_0$ and $\boldsymbol{\theta}_0^-$ so that all $Q(s,a;\boldsymbol{\theta_0})$ are close to 0.
2. Use an ϵ-greedy policy to collect experience in the form of N_{replay} many (s,a,r,s') tuples.
3. Mini-batches (e.g., 32 elements) are taken randomly from the replay memory to update $\boldsymbol{\theta}_k$.
4. $\boldsymbol{\theta}_k^-$ is only set to $\boldsymbol{\theta}_k$ every C iterations.
5. Go back to collecting more experience with a new policy ϵ-greedy policy extracted from the prediction network.

Double DQN (DDQN; Van Hasselt, 2016)

- **Issue**: Q-learning and DQN use one function in the target to choose the action and evaluate its value. The max creates an **overestimation bias**.

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha \underbrace{\left(\overbrace{r + \gamma \boxed{\max_{a'} Q(s', a'; \boldsymbol{\theta}_k)}}^{\text{target}} - \overbrace{Q(s, a, \boldsymbol{\theta}_k)}^{\text{prediction}}\right)}_{\text{TD-error}} \nabla Q(s, a, \boldsymbol{\theta}_k)$$

- DDQN **decoupling the action selection and evaluation** by using different parameters:

$$U_k = r + \gamma Q(s', \operatorname*{argmax}_{a} Q(s', a'; \boldsymbol{\theta}_k); \boldsymbol{\theta}_k^-)$$

Action evaluation uses more stable $\boldsymbol{\theta}_k^-$

Action selection uses $\boldsymbol{\theta}_k$

Dueling Network Architecture (Wang, 2015)

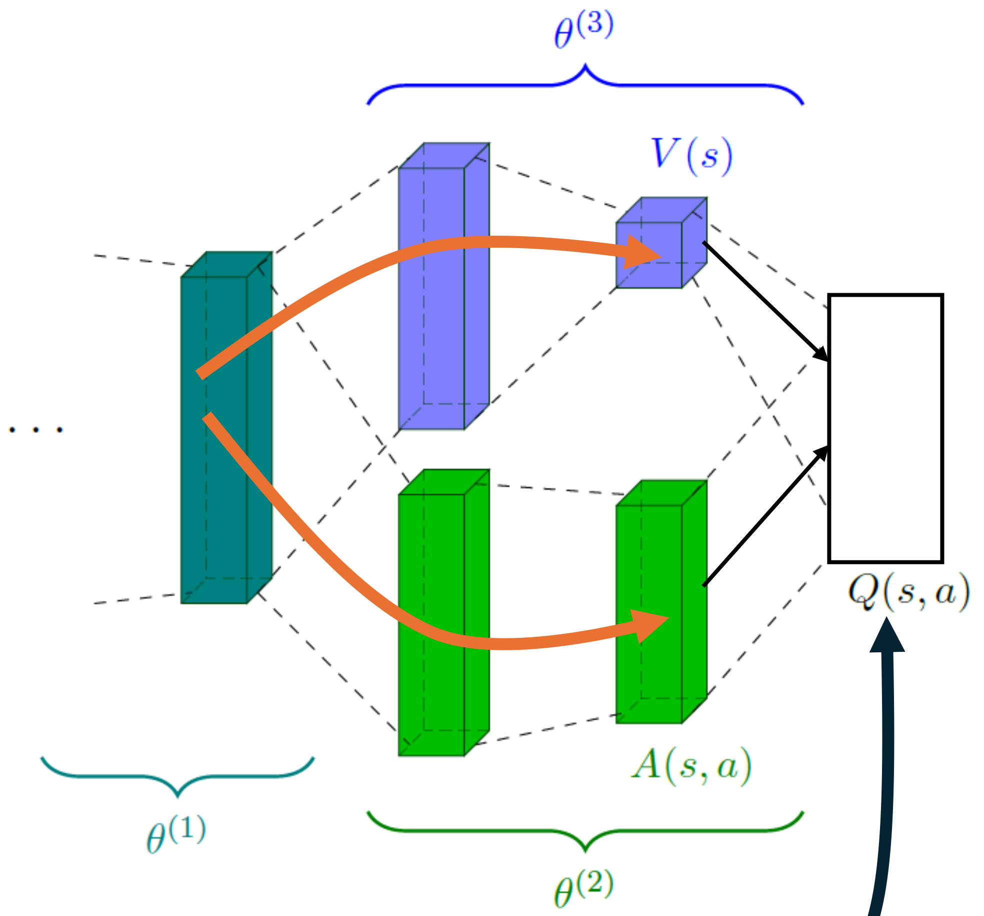
- Dueling Networks Architecture only changes the ANN architecture. It learns two components:
  - Value function $V(s; \boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(3)})$
  - Advantage function $A(s, a'; \boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(3)})$
- $Q(s,a)$ is calculated using the equation below.

- Reminder: Advantage function

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

  gives the "advantage" of choosing action a over the state value.

- This approach typically leads to improved performance.



$$\boxed{Q(s,a;\boldsymbol{\theta}^{(1)},\boldsymbol{\theta}^{(2)},\boldsymbol{\theta}^{(3)}) = V(s;\boldsymbol{\theta}^{(1)},\boldsymbol{\theta}^{(3)}) + \underbrace{\left(\mathrm{A}(\mathrm{s},\mathrm{a};\boldsymbol{\theta}^{(1)},\boldsymbol{\theta}^{(2)}) - \max_{a'\in\mathcal{A}} \mathrm{A}(\mathrm{s},\mathrm{a}';\boldsymbol{\theta}^{(1)},\boldsymbol{\theta}^{(2)})\right)}_{\text{Becomes 0 for } a = a^{*}}}$$

n-step Methods

Bootstrapping (one-step)

- Instability with function approximation (uses its own prediction).
+ Can use off-policy learning.

Less Bootstrapping (n-step)

+ Less instability
+ Quicker information propagation for delayed rewards.

MC (no bootstrapping)

+ Stable
- Only on-policy learning leads to sample inefficiency (cannot use replay memory).

• ***n*-step** methods are between Q-learning (one-step) and MC, which does not bootstrap.

• Use n-step target for DQN-type algorithms:

$$U_k^n = \sum_{t=0}^{n-1} \gamma^t r_t + \gamma^n \max_{a' \in \mathcal{A}} Q(s_n, a'; \boldsymbol{\theta}_k)$$

• **Traces** can also be used.

Policy Gradient DRL Methods

Use ANNs to represent and learn a parameterized policy.

Note: θ is called w!

Remember: Policy Gradient Theorem

- We define the performance measure as the true value of following π_θ from the start state s_0

$$J(\boldsymbol{\theta}) \overset{\text{def}}{=} v_{\pi_\theta}(s_0)$$

- The **policy gradient theorem** gives us an analytical expression for the gradient of the with respect to the policy parameters:

$$\nabla J(\boldsymbol{\theta}) = \nabla v_{\pi_\theta}(s_0) \propto \sum_s \mu(s) \sum q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta})$$

On-policy distribution under π

Depends on estimates for q

Derived from the def. of the policy.

- The theorem also provides a **strong convergence guarantee**!

Policy: $\pi(a|s,\boldsymbol{\theta}) = \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}}$

Gradient: $\nabla \ \ln \pi(s|a,\boldsymbol{\theta}) = h(s,a) - \sum'_a h(s,a')\pi(s|a',\boldsymbol{\theta})$

Policy Gradient

- Following the ideas of REINFORCE.
- Gradient of a stochastic policy

Needs a value function estimate

$$\nabla_w v_{\pi_w}(s_0) = \mathbb{E}_{s\sim\mu_{\pi_w},a\sim\pi_w}[\nabla_w\ (\log\pi_w(s,a))Q^{\pi_w}(s,a)]$$

Sample state from the on-policy distribution and action from the policy

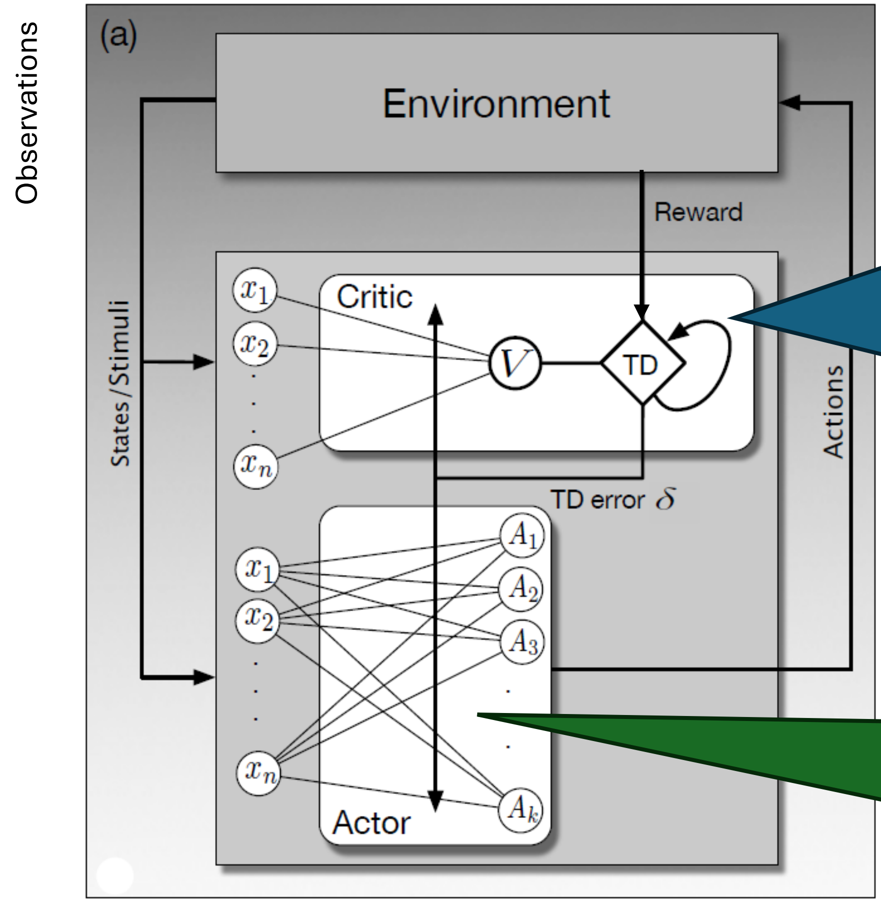
- Gradient of a differentiable deterministic policy

$$\nabla_w v_{\pi_w}(s_0) = \mathbb{E}_{s\sim\mu_{\pi_w}}\ [\nabla_w(\pi_w)\ \nabla_a\ (Q^{\pi_w}(s,a))\ |a = \pi_w(s)]$$

Also requires this gradient!

Neural Actor-Critic Method

- **Remember:** Estimate the value function and a parameterized policy at the same time.



Section 15.7 in Sutton and Barto (2018)

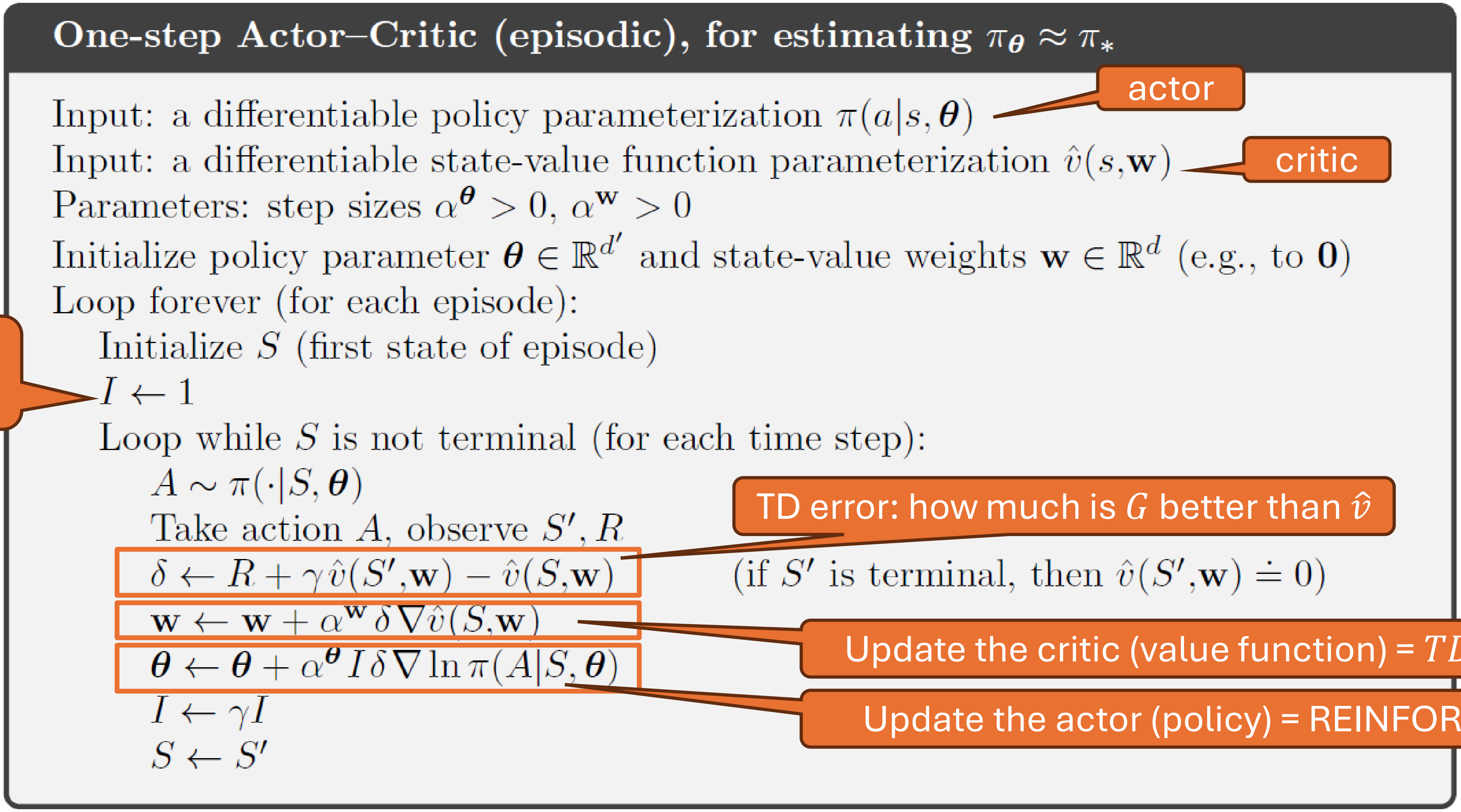
Critic

- Represents value function $Q^{\pi_w}(s,a) \approx Q(s,a;\boldsymbol{\theta})$
- Produces the TD error from the reward signal using its network's state-value prediction.
- Adjusts state-value network using the TD error.
- Does not select actions!

Actor

- Represents policy π_w
- Selects actions with a policy network.
- Adjusts a policy network based on the TD error
- No access to reward signal!

Neural Actor-Critic Method: One-Step Algorithm



One-step Actor–Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

```
Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
    Initialize $S$ (first state of episode)
    $I \leftarrow 1$
    Loop while $S$ is not terminal (for each time step):
        $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
        Take action $A$, observe $S'$, $R$
        $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$        (if $S'$ is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\nabla\hat{v}(S,\mathbf{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I\delta\nabla\ln\pi(A|S,\boldsymbol{\theta})$
        $I \leftarrow \gamma I$
        $S \leftarrow S'$
```

actor

critic

Takes care of discounting γ^t

TD error: how much is G better than $\hat{v}$

Update the critic (value function) = $TD(0)$

Update the actor (policy) = REINFORCE

Possible improvements:

- Sample efficiency: Use a replay memory with a set of (s, a, r, s') tuples.
- Stability and faster reward propagation: Multi-step methods

Natural Policy Gradients (NPG)

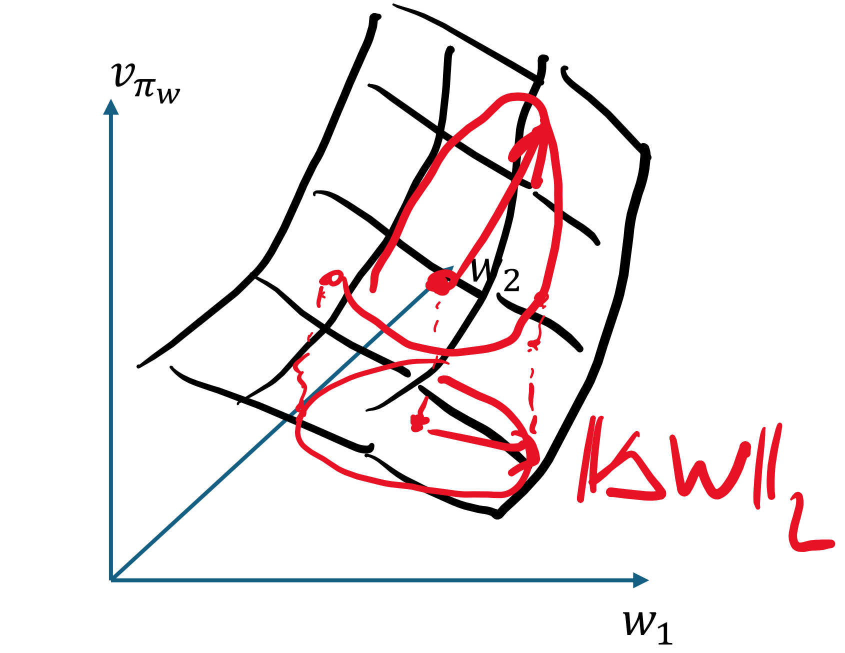
- Regular policy gradient methods update policy parameters in the direction of the steepest ascent of the performance measure $J(\boldsymbol{w}) \overset{\text{def}}{=} v_{\pi_w}(s_0)$ given a small change of the parameter vector $\boldsymbol{w}$.
- This is achieved by an update $\Delta\boldsymbol{w} \propto \nabla_w J(\boldsymbol{w})$ which maximizes $J(\boldsymbol{w}) - J(\boldsymbol{w} + \Delta\boldsymbol{w})$ under a constraint on $\|\Delta\boldsymbol{w}\|_2$
- **Issue**: The same amount of change in the parameter space $\boldsymbol{w}$ can lead to very different magnitudes of change in the actual policy π_w (the probability distribution over actions) resulting in instability.

- **Idea**: Natural gradients define the steepest direction by the largest change in the value function given a small change in the probability distribution of the policy π_w.
- This requires redefining the constraint on $\Delta\boldsymbol{w}$. Kullback-Leibler (KL) divergence measures the difference between two policies' action probability distributions $D_{KL}(\pi_w || \pi_{w+\Delta w})$.
- A local approximation, the Fisher information criterion, can be used to adjust the gradient for policy changes:
$$\Delta\boldsymbol{w} \propto F_w^{-1} \nabla_w J(\boldsymbol{w})$$
- F_w is the Fisher information matrix that depends

- **Problem**: Calculating the Fisher information matrix of deep learning models is too expensive.
- The idea inspired several state-of-the-art methods:
  - Trust Region Optimization (TRPO)
  - Proximal Policy optimization (PPO)

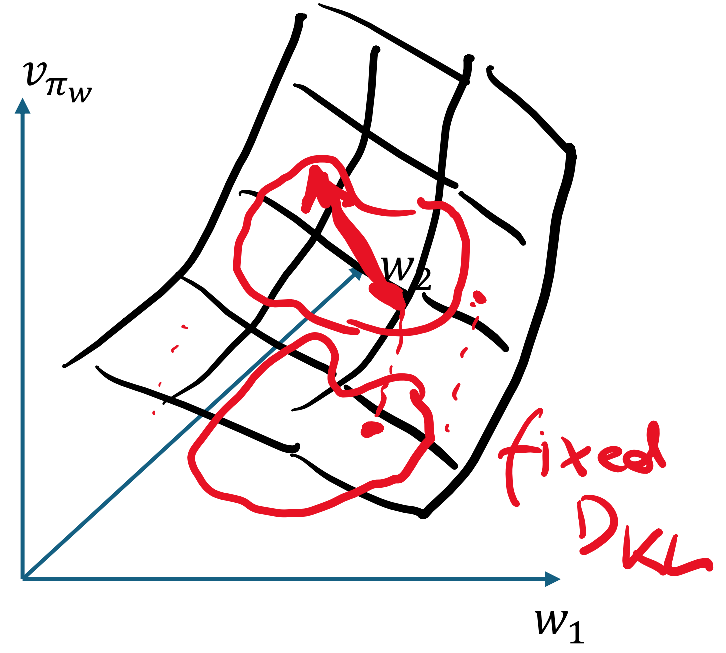
Standard vs. Natural Policy Gradient

Standard Gradient

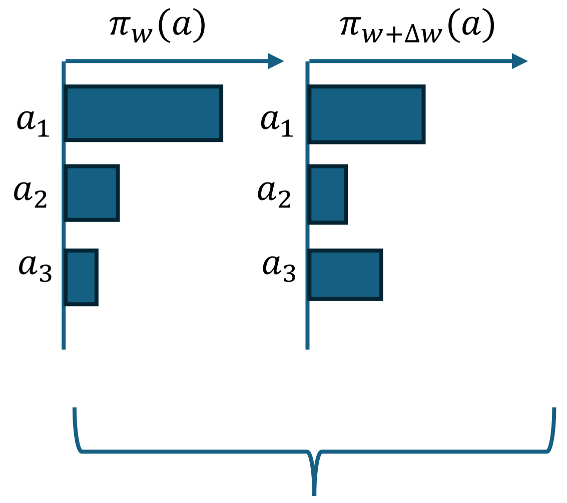


Move a fixed distance in the **parameter space**. This can lead to large policy changes.

Natural Policy Gradient



Move a fixed distance in the **policy space**.



$D_{KL}(\pi_w || \pi_{w+\Delta w})$

Trust Region Policy Optimization (TRPO; Schulman et al, 2015)

- Simplification of NPG.
- Finds the update that improves the advantage function the most

$$\max_{\Delta w} \mathbb{E}_{s\sim\mu^{\pi_w}, a\sim\pi_w}\left[\frac{\pi_{w+\Delta w}(s,a)}{\pi_w(s,a)} A^{\pi_w}(s,a)\right]$$

Prefer new policies that pick high-advantage actions.

- constrained by an explicitly bound the policy change per update step.

$$\mathbb{E}\left[D_{KL}(\pi_w(s,\cdot)||\pi_{w+\Delta w}(s,\cdot))\right] \leq \delta \qquad \text{where } \delta \in \mathbb{R}$$

- δ is a tuneable hyperparameter specifying the size of the “trust region.”

Proximal Policy Optimization (PPO; Schulman et al, 2017)

- A popular and more practical variant of TRPO.
- Instead of KL-divergence, it uses a clipped surrogate objective function to penalize overly large policy changes with at probability ratio outside of $[1 - \epsilon, 1 + \epsilon]$

$$r_t(w) = \frac{\pi_{w+\Delta w}(s,a)}{\pi_w(s,a)}$$

$$\max_{\Delta w} \mathbb{E}_{s\sim\mu^{\pi_w}, a\sim\pi_w}\left[\min\left(r_t(w)A^{\pi_w}(s,a), \operatorname{clip}(r_t(w), 1-\epsilon, 1+\epsilon)A^{\pi_w}(s,a)\right)\right]$$

- ϵ is a tunable hyperparameter
- Achieves much of the stability of NPG methods with the computational efficiency of first-order optimization.

Generalization

Generalization in RL

a) Perform well in an environment after limited data is gathered
(= sample efficiency)

b) Perform well in a new, but related environment
(~ transfer learning)