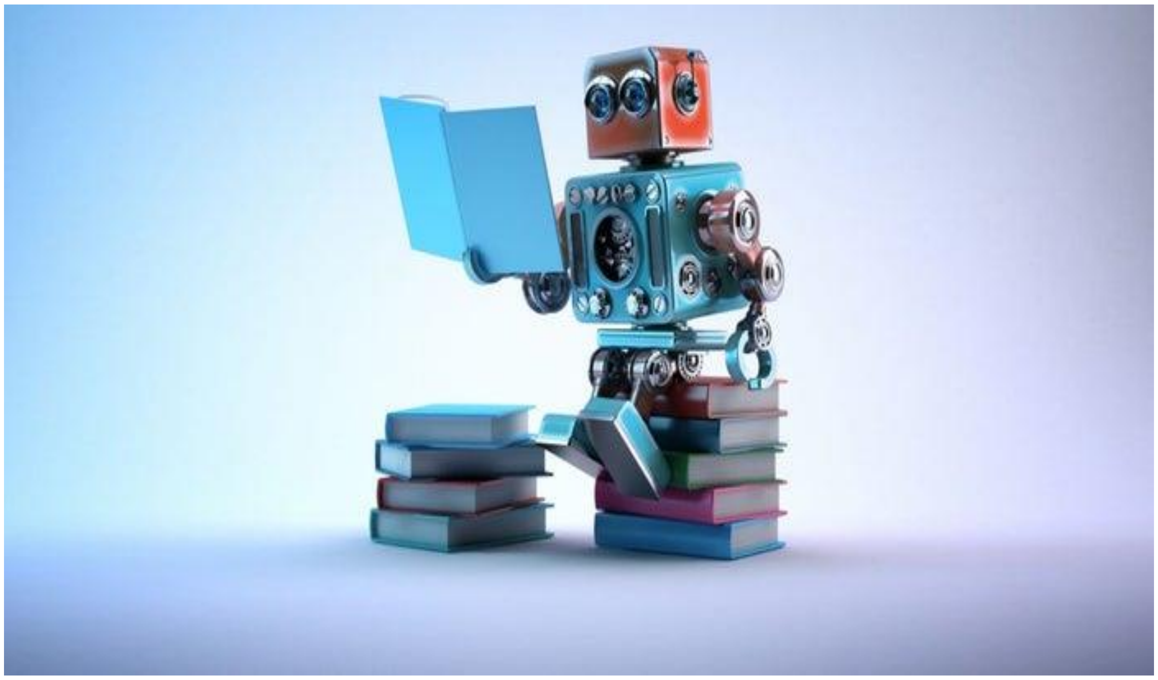


Reinforcement Learning

Artificial Intelligent

Iheb Bouariche | Reinforcement Learning | 29.07.2022

❖ Introduction

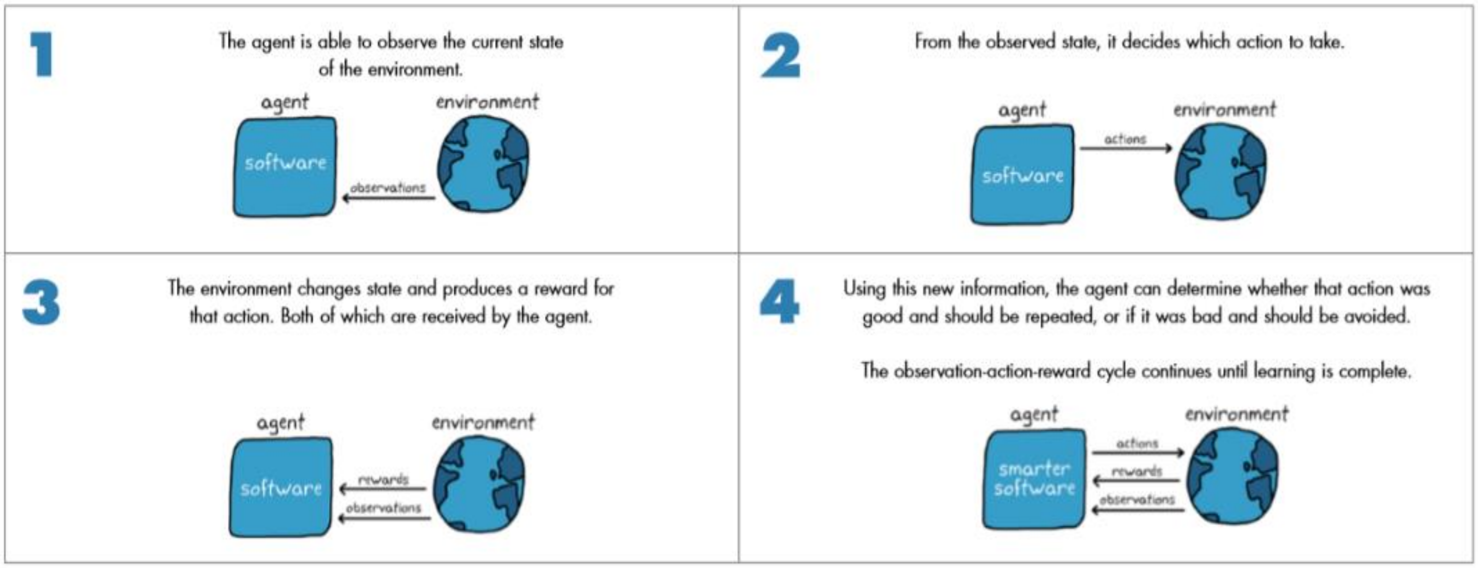
Reinforcement learning (RL) has successfully trained computer programs to play games at a level higher than the world's best human players.

These programs find the best action to take in games with large state and action spaces, imperfect world information, and uncertainty around how short-term actions pay off in the long run.

Engineers face the same types of challenges when designing controllers for real systems. Can reinforcement learning also help solve complex control problems like making a robot walk or driving an autonomous car?

What is Reinforcement Learning?

Reinforcement learning is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning. It works with data from a dynamic environment. The goal of RL is to find the best sequence of actions that will generate the optimal outcome. The way reinforcement learning solves this problem is by allowing a piece of software called an agent to explore, interact with, and learn from the environment.



1 The agent is able to observe the current state of the environment.

2 From the observed state, it decides which action to take.

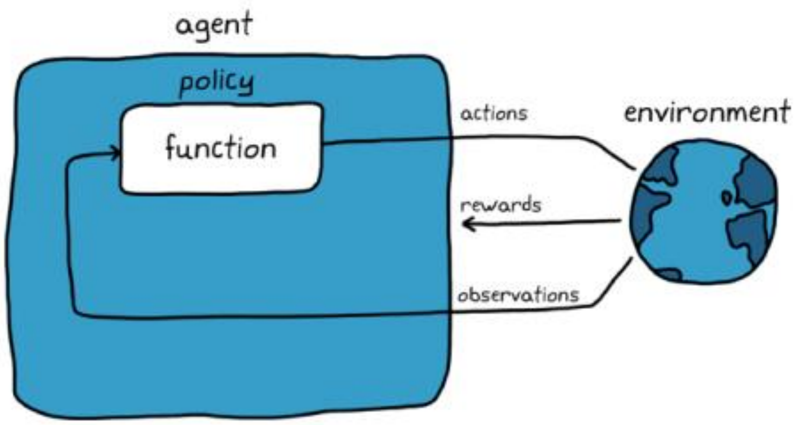
3 The environment changes state and produces a reward for that action. Both of which are received by the agent.

4 Using this new information, the agent can determine whether that action was good and should be repeated, or if it was bad and should be avoided.

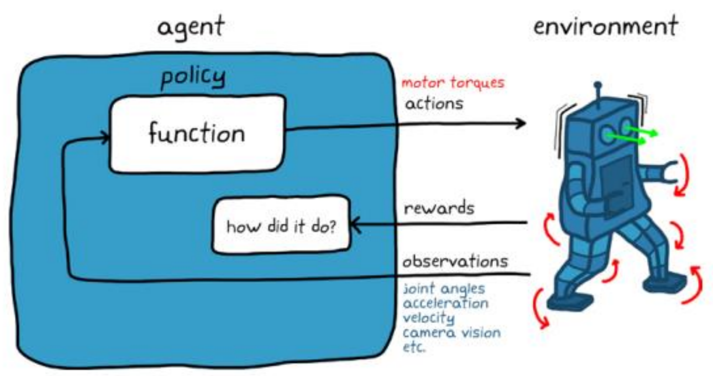
The observation-action-reward cycle continues until learning is complete.

Anatomy of Reinforcement Learning

Within the agent, there is a function that takes in state observations (the inputs) and maps them to actions (the outputs). In the RL nomenclature, this function is called the policy. Given a set of observations, the policy decides which action to take.

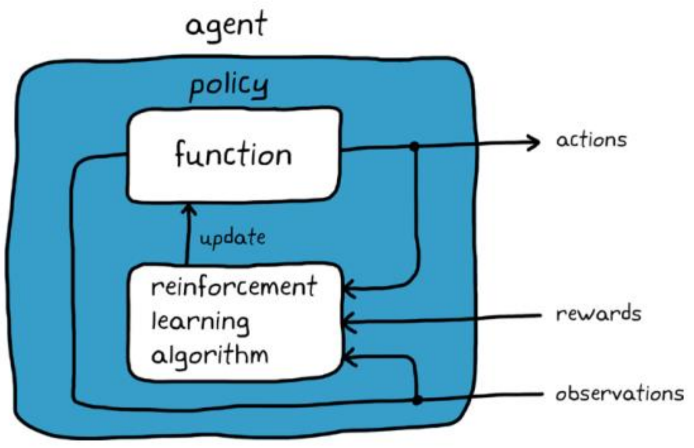


For example, a walking robot, the observations would be the angle of every joint, the acceleration and angular velocity of the robot trunk, and the thousands of pixels from the vision sensor. The policy would take in all of these observations and output the motor commands that will move the robot's arms and legs. The environment would then generate a reward telling the agent how well the very specific combination of actuator commands did. If the robot stays upright and continues walking, the reward will be higher than if the robot falls to the ground.



Learning the Optimal Policy

If you were able to design a perfect policy that would correctly command the right actuators for every observed state, then your job would be done. Of course, that would be difficult to do in most situations.



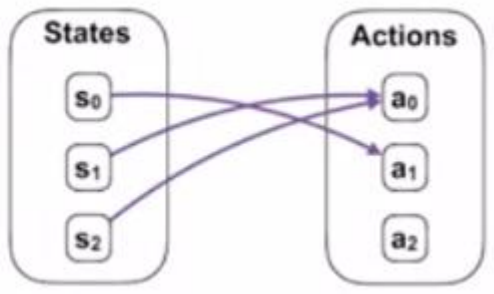
Even if you did find the perfect policy, the environment might change over time, so a static mapping would no longer be optimal. This brings us to the reinforcement learning algorithm. It changes the policy based on the actions taken, the observations from the environment, and the amount of reward collected.

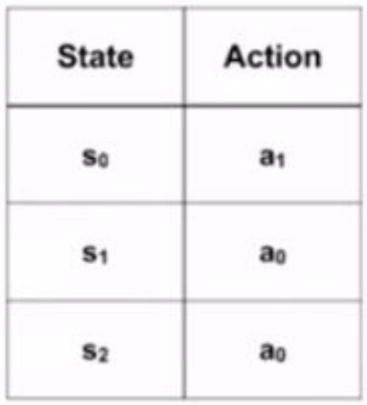
The goal of the agent is to use reinforcement learning algorithms to learn the best policy as it interacts with the environment so that, given any state, it will always take the most optimal action, the one that will produce the most reward in the long run.

❖Fundamentals of Reinforcement Learning

1. **Deterministic policy** is policy that maps state to actions with no uncertainty. it's used in deterministic environments.

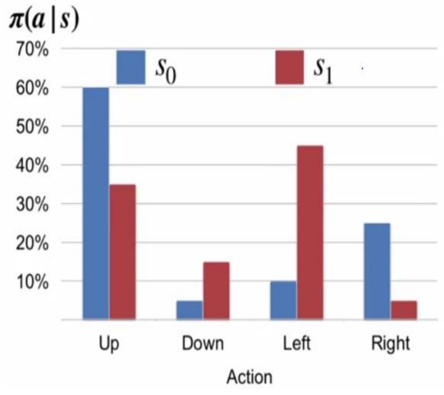
$$\pi(s) = a$$





State	Action
s_0	a_1
s_1	a_0
s_2	a_0

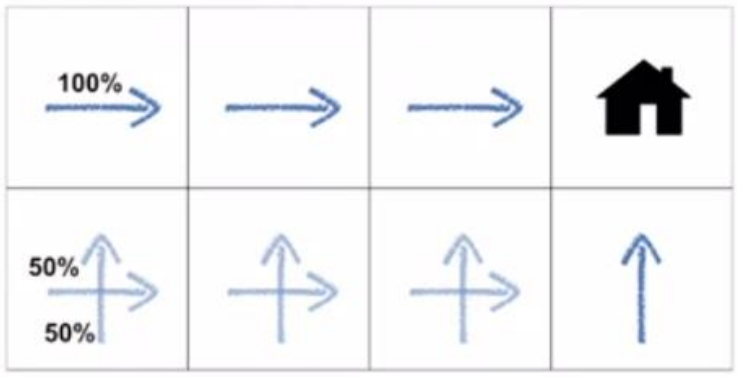
2. **Stochastic policy** allows the agent to map each state to a probability distribution of actions.



$$\pi(a|s)$$

$$\sum_{a \in \mathcal{A}(s)} \pi(a|s) = 1$$

$$\pi(a|s) \geq 0$$



- **What is the difference between a deterministic and a stochastic policy?**

A deterministic policy can be interpreted as a stochastic policy that gives the probability of 1 to one of the available actions (and 0 to the remaining actions), for each state.

3. Markov decision process Almost all problems in Reinforcement Learning are theoretically modelled as maximizing the *return* in a Markov Decision Process, or simply, an MDP. An MDP is characterized by 4 things:

1. S: The set of states that the agent *experiences* when interacting with the environment. The states are assumed to have the Markov property.
2. A: The set of legitimate actions that the agent can execute in the environment.
3. P: The transition function that governs which probability distribution of the states the environment will transition into, given the initial state and the action applied to it.
4. R: The reward function that determines what reward the agent will get when it transitions from one state to another using a particular action.

A Markov decision process is often denoted as M=⟨S, A, P, R⟩.

4. State-value function Vπ(s) describes the value of a state when following a policy. It is the expected return when starting in state "s" and acting according to our policy.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t | S_t = s]$$

5. Action-value function Q(s,a) describes the value of taking an action in some state when following a policy. It is the expected return given a state "s" and an action "a" under a policy.

$$q_p i \doteq \mathbb{E}[G_t | S_t = s, A_t = a]$$

- **Gt** is the sum of futur reward of taking actions following a policy π.
- $\boldsymbol{\delta}$ is a discount factor.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

6. State-value Bellman equation

It defines the relationship between the value of a state and the value of future possible states.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t | S_t = s]$$

$$= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a) \big[r + \gamma \mathbb{E}[G_{t+1} | S_{t+1} = s'] \big]$$

7. Action-value Bellman equation :

$$q_p i \doteq \mathbb{E}[G_t | S_t = s, A_t = a]$$

$$= \sum_{s'} \sum_{r} p(s', r | s, a) \left[r + \gamma \sum_{a'} \pi(a' | s') q_\pi(s', a') \right]$$

- We can only solve small MDPs directly from state-value and action-value Bellman equations.
- But Bellman Equations will factor into the solutions of large MDPs. We see that later
- Bellman equations define a relationship between the value of a state, or state-action pair, and its possible successors.

8. Optimal Policy

- in these 3 cases we suppose that we calculated the values of V(s) and q(s,a) of an MDP. So we try to find the optimal policy.

- **Bellman Optimality Equation for state-value function**
  The state-value function for the optimal deterministic policy is defined by :

$$v*(s) = \max_a \sum_{s'} \sum_{r} p(s', r | s, a[r + \gamma v*(s')])$$

- **Bellman Optimality Equation for action-value function**
  The action-value function for the optimal deterministic policy is defined by :

$$q*(s, a) = \sum_{s'} \sum_{r} p(s', r | s, a) \left[r + \gamma \max_{a'} q*(a' | s') \right]$$

- **The Optimal Policy**
  It's more simple to find the state-value function or the action-value function and calculte the optimal deterministic policy, then do calculation directly of the optimal determinitic policy.So for that we used argmax of the calculated value function to find the optimal policy.

$$\pi_*(s) = \arg\max_a \sum_{s'} \sum_{r} p(s', r, | s, a)[r + \gamma v_*(s')]$$

$$\pi_* = \arg\max_a q*(s, a)$$

9. Iterative value function for policy evaluation:

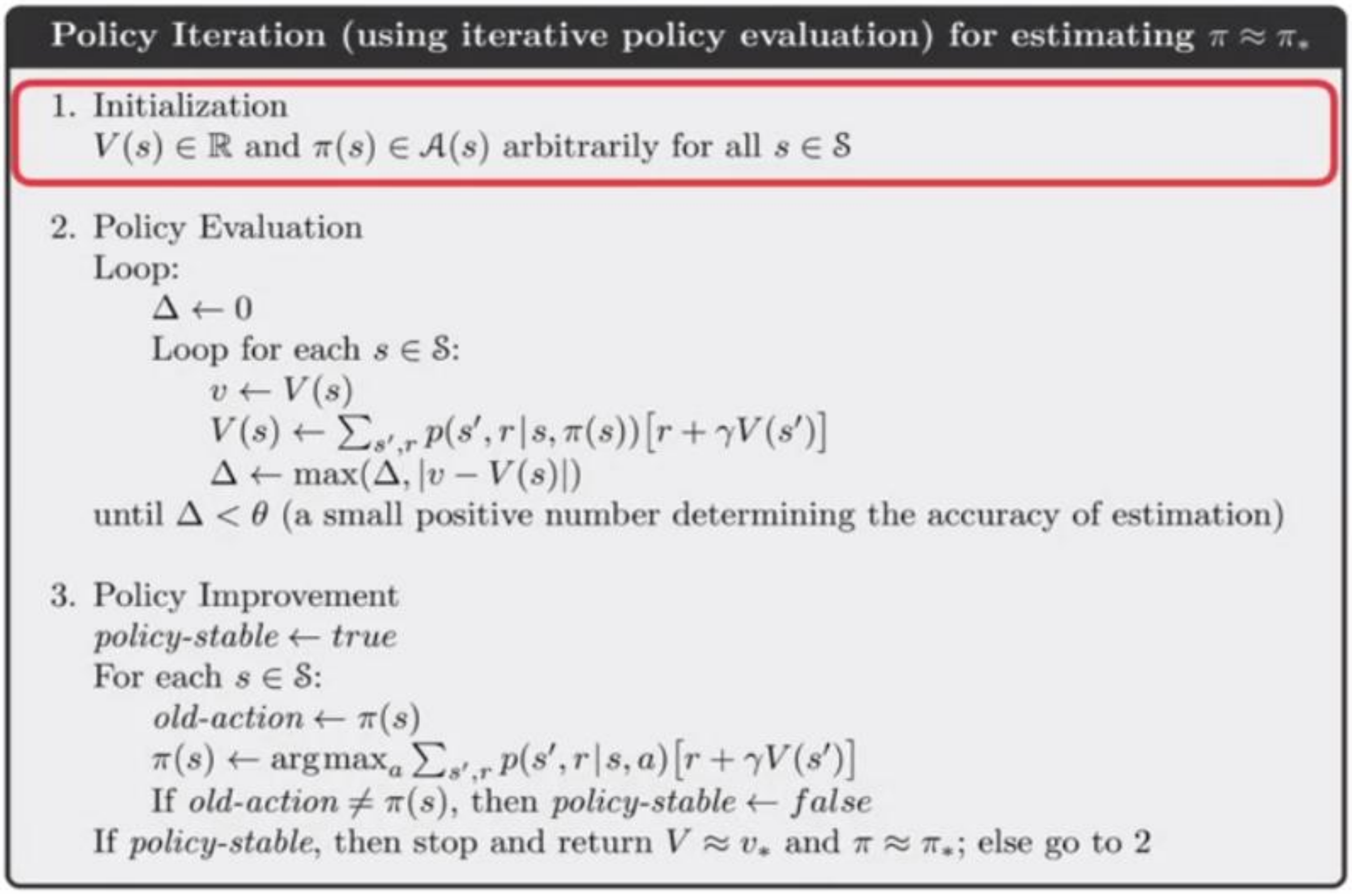
- This iterative method is used to calculate values from a stochastic policy. it's used for policy evaluation.

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a)\left[r + \gamma v_\pi(s')\right]$$

$$\Big\downarrow$$

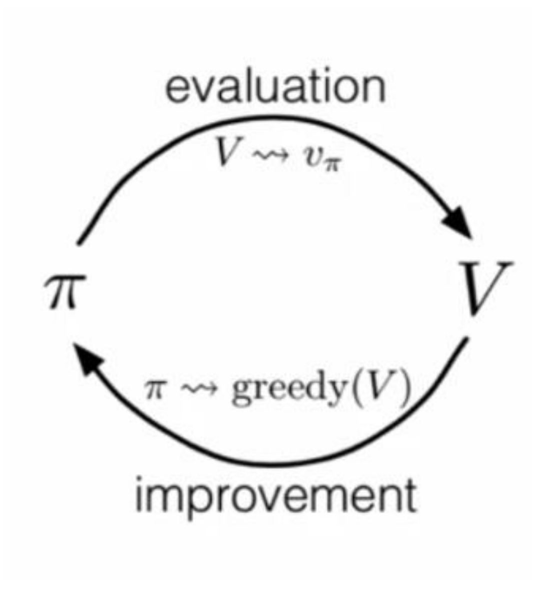
$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a)\left[r + \gamma v_k(s')\right]$$

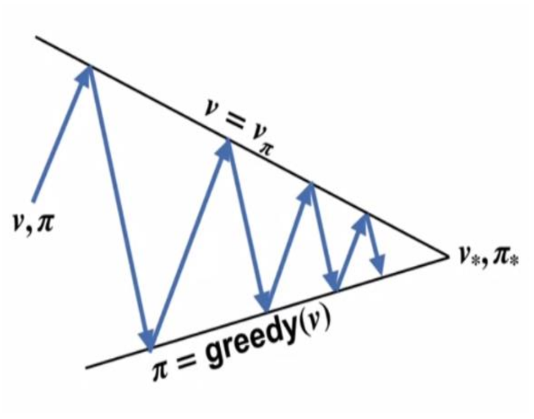
10. Policy Greedy Iteration (Policy evaluation and improvement (Algorithme 1))



Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

```
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
       $\Delta \leftarrow 0$
       Loop for each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))\left[r + \gamma V(s')\right]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ true
   For each $s \in \mathcal{S}$:
       old-action $\leftarrow \pi(s)$
       $\pi(s) \leftarrow \mathrm{argmax}_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma V(s')\right]$
       If old-action $\neq \pi(s)$, then policy-stable $\leftarrow$ false
   If policy-stable, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2
```


evaluation
$V \rightsquigarrow v_\pi$
π
V
$\pi \rightsquigarrow \text{greedy}(V)$
improvement



$$\begin{tikzcd} \pi_* \arrow[r, leftrightarrow] & v_* \end{tikzcd}$$

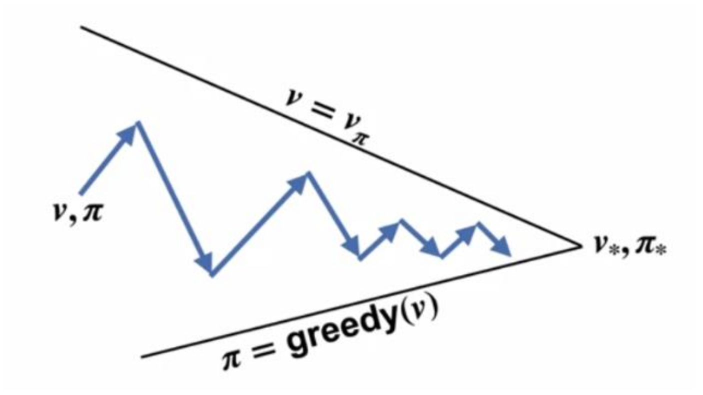
11. Generalized Policy Iteration (Algorithme 2)

- Generalized Policy Methode unifies classical dynamique programming methods, value iteration, and asynchronous dynamique programming.
- This algorithme allows us to combine policy evaluation and improvement into a single update.

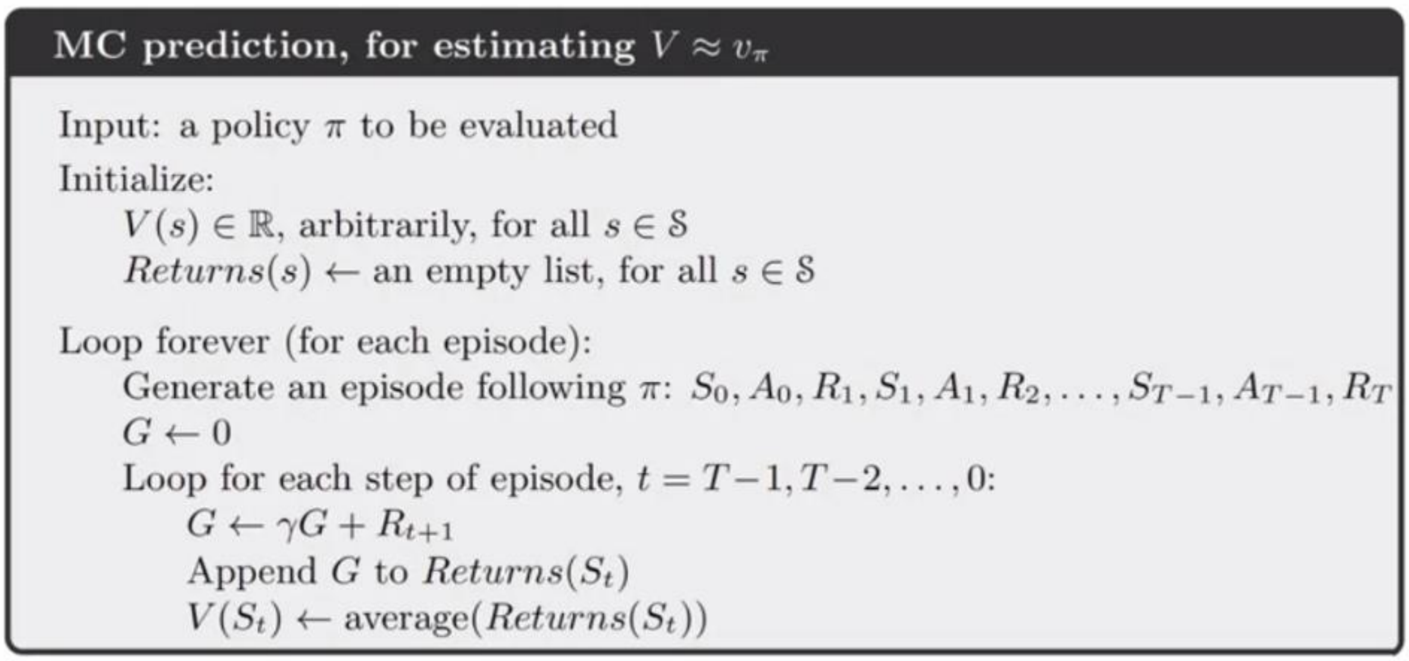
Value Iteration, for estimating $\pi \approx \pi_*$

```
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
|   $\Delta \leftarrow 0$
|   Loop for each $s \in \mathcal{S}$:
|       $v \leftarrow V(s)$
|       $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
|       $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
    $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
```

- The last two algorithms require a model for the environement. In the next topic we will try to solve MDPs without a model for environement. (Monte Carlo and TD algorithm)



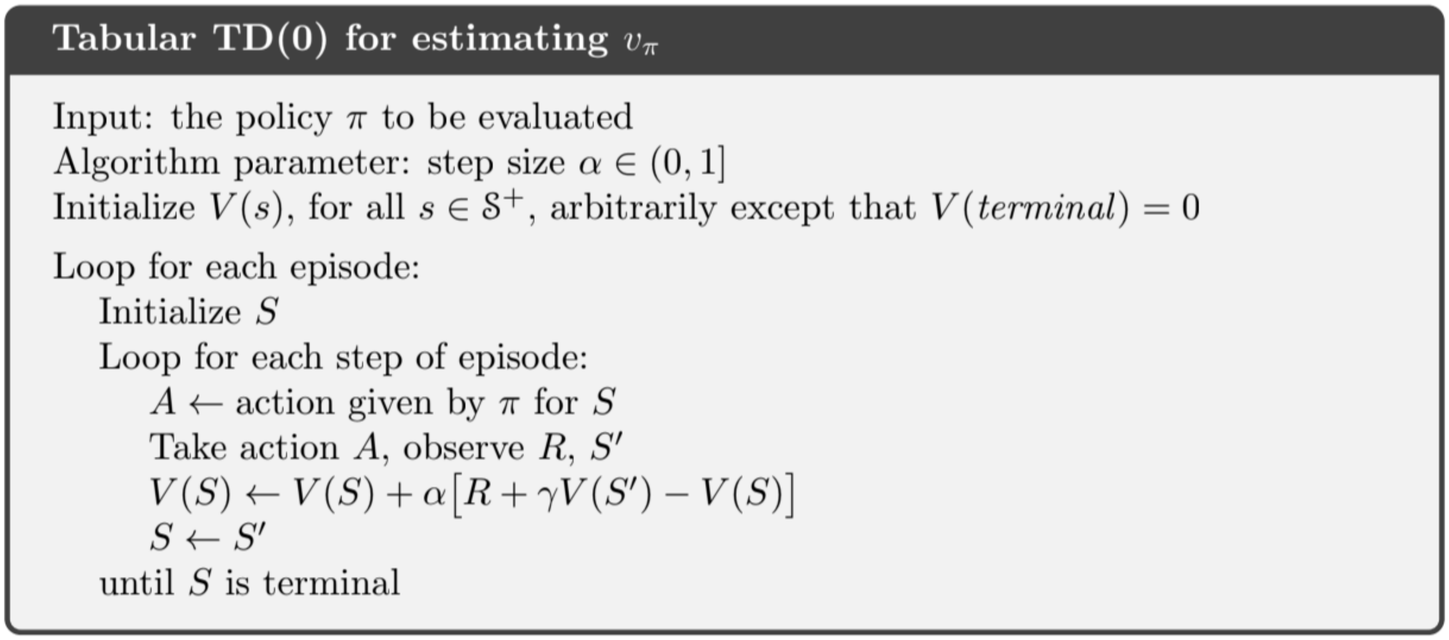
12. **Monte Carlo (Algorithm)**



```
MC prediction, for estimating $V \approx v_\pi$

Input: a policy $\pi$ to be evaluated
Initialize:
    $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
    Generate an episode following $\pi$: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Append $G$ to $Returns(S_t)$
        $V(S_t) \leftarrow$ average($Returns(S_t)$)
```

13. **Tabular Temporel Difference() (Algorithm)**

- The third iterative method to calculate values functions of a stochastic policy. This method is used for evaluation of a policy.
- α is the learning rate.



```
Tabular TD(0) for estimating $v_\pi$

Input: the policy $\pi$ to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        $A \leftarrow$ action given by $\pi$ 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
```

- **What is the difference between MC and TD algorithms and value iteration algorithms?**

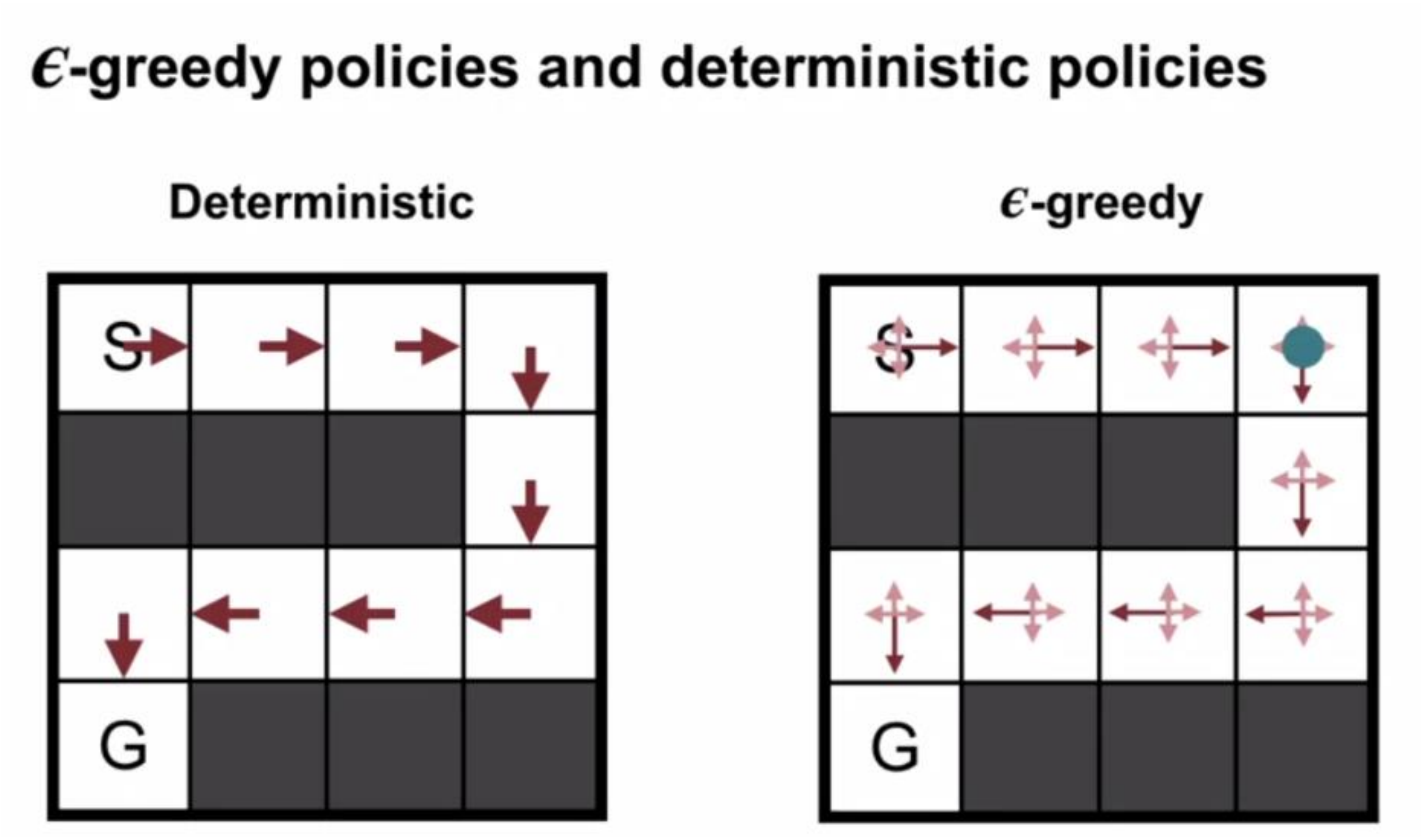
MC and TD are based on exploration and exploitation. They do not require a model for the environement and calculation for all the values.

- **What is the difference between MC and TD?**

Both algorithms are used on the calculation of the value function. But TD can converge faster than Monte Carlo method and does not require a model for the environment and it's online and incremental.

14. Epsilon Greedy policy:

- E-greedy policies used to solve exploration probleme. All actions have a probality of at least epsilon over the number of actions.

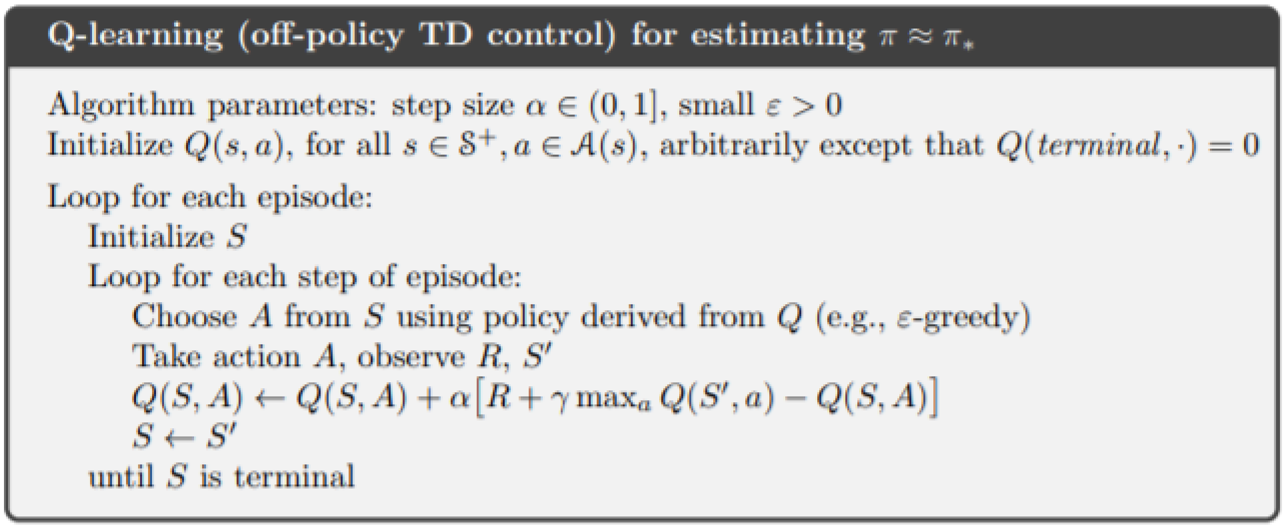


There is others algorithms based on TD algorithme:

- Sarsa.
- The expected Sarsa.
- Q-learning.
- Dyna-Q. (Q-learning + Q-planning)
- Dyna-Q+. (reward bonus in planning updates)

But we will focus only on Q-Learning.

15. Q-Learning (off-policy)(Algorithme 3)



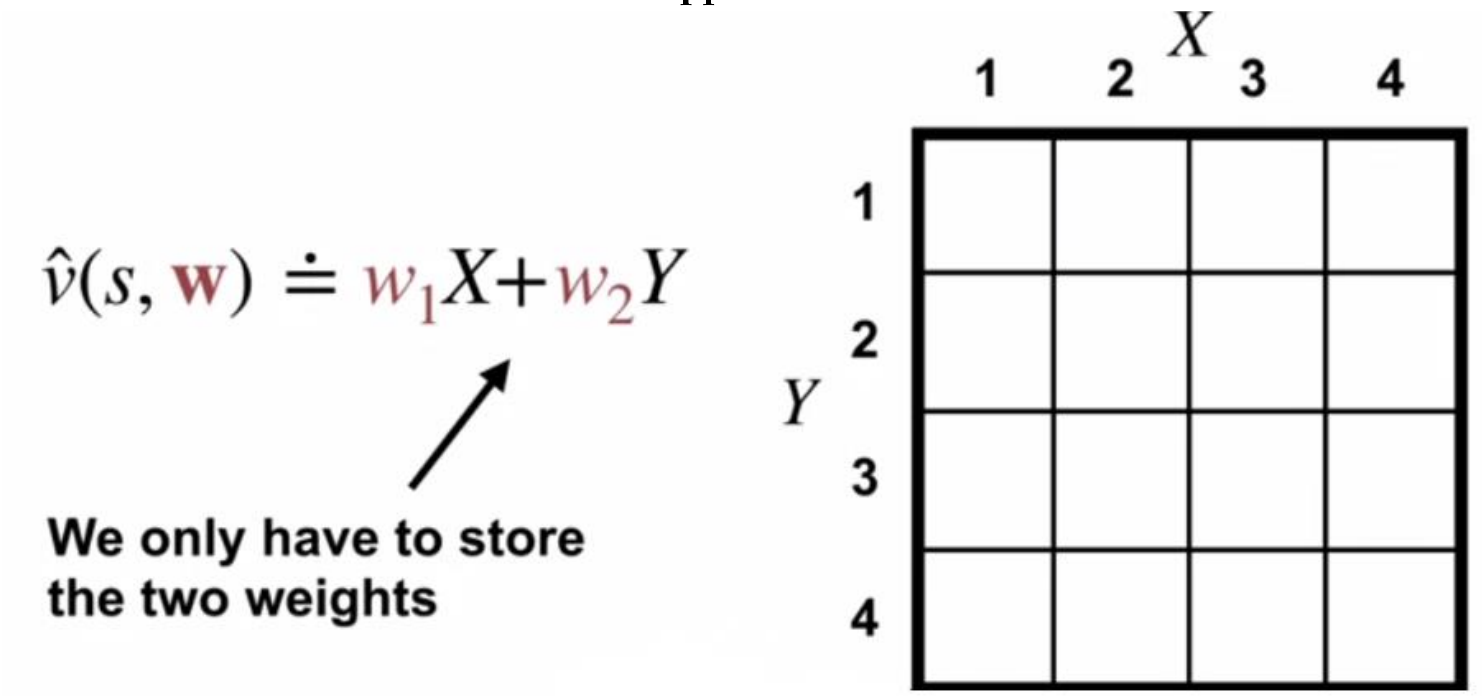
Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
   Initialize $S$
   Loop for each step of episode:
      Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
      Take action $A$, observe $R$, $S'$
      $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
      $S \leftarrow S'$
   until $S$ is terminal
```

- On-Policy: improve and evaluate the policy being used to select actions.
- Off-Policy: improve and evaluate the policy from the one used to select actions.
- Off-Policy learning allows learning an optimal policy from suboptimal behavior.
- The policy that we are learning is the target policy.
- The policy that we are choosing actions from is the behavior policy.
- Q-Learning Algorithme can solve large MDPs and it’s based on TD algorithme.

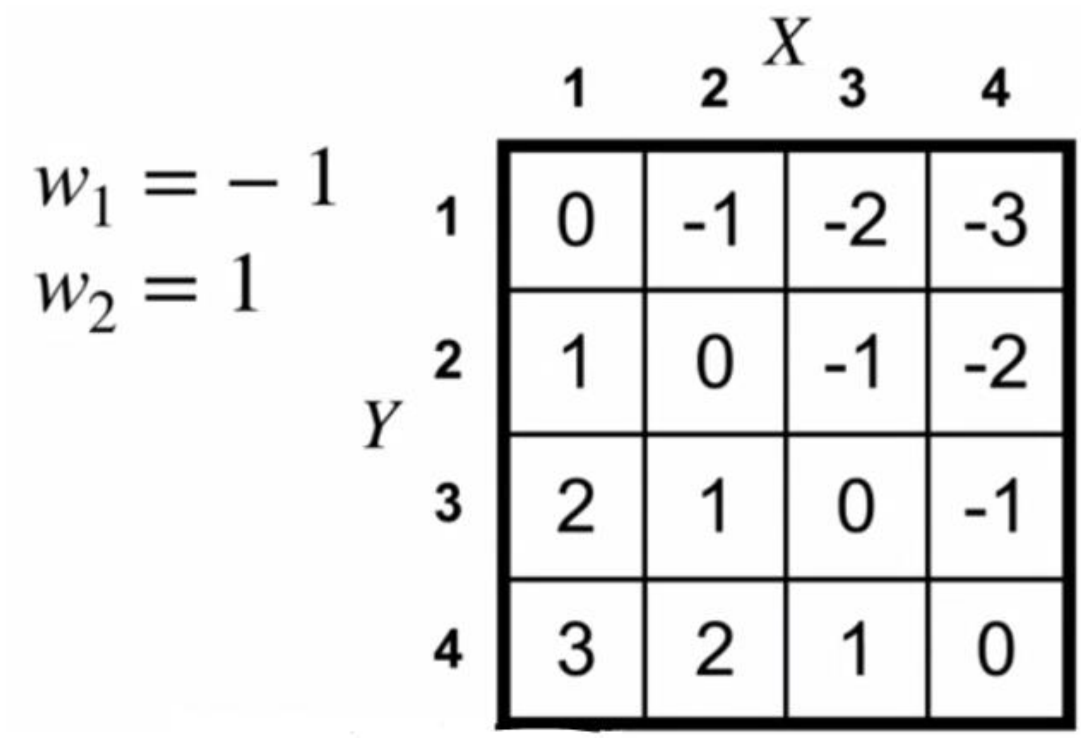
1. Parametrized value function

- **Linear Value Function Approximation:**



$$\hat{v}(s, \mathbf{w}) \doteq w_1X+w_2Y$$

- For example:

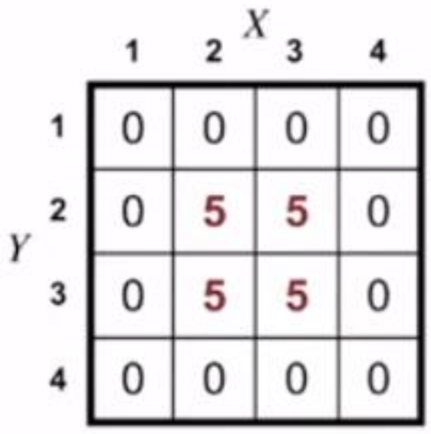


$$w_1 = -1$$
$$w_2 = 1$$

$$\begin{array}{c|cccc} & X=1 & 2 & 3 & 4 \\ \hline Y=1 & 0 & -1 & -2 & -3 \\ 2 & 1 & 0 & -1 & -2 \\ 3 & 2 & 1 & 0 & -1 \\ 4 & 3 & 2 & 1 & 0 \end{array}$$

➢ Limitation of Linear Value Function Approximation :

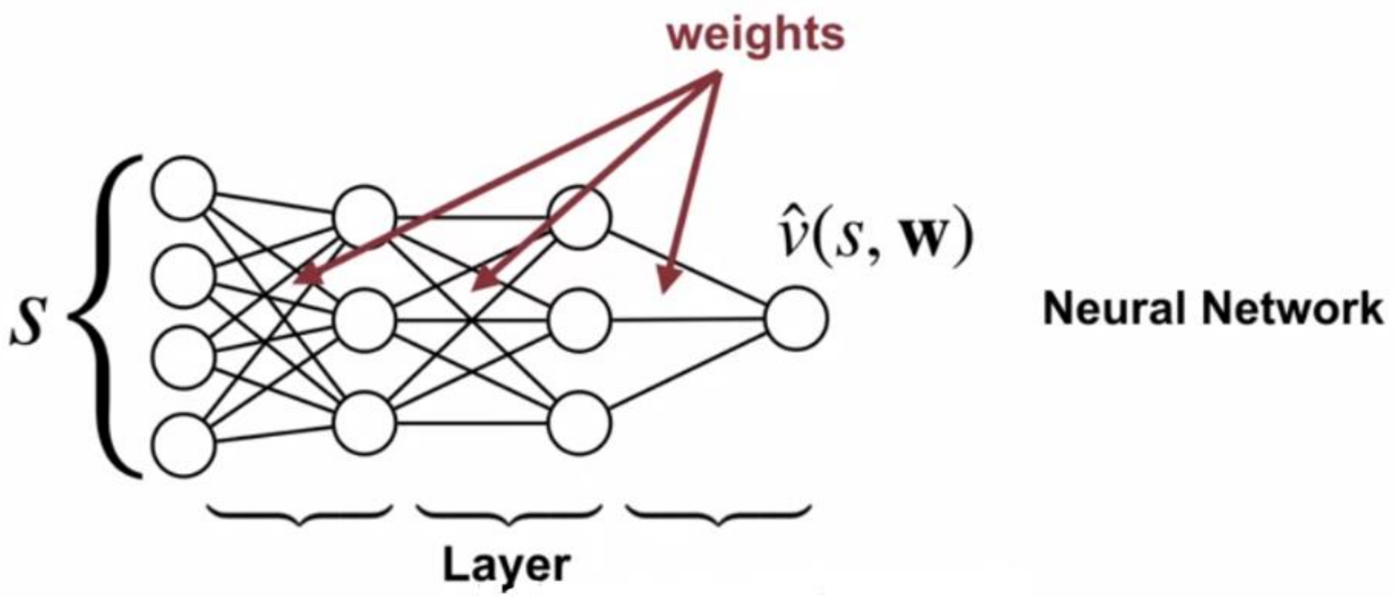
$$\hat{v}(s, \mathbf{w}) \doteq \sum w_i x_i(s)$$



Y \ X	1	2	3	4
1	0	0	0	0
2	0	5	5	0
3	0	5	5	0
4	0	0	0	0

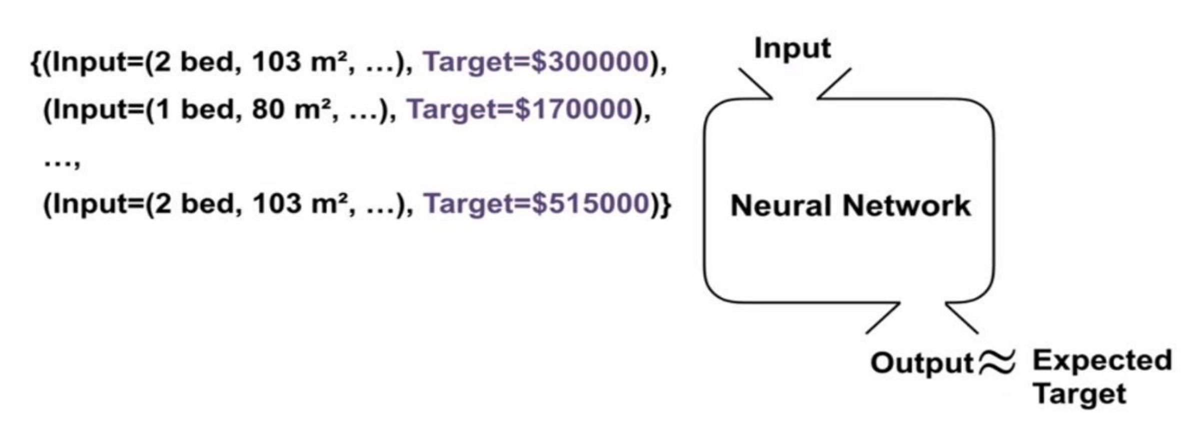
✓ Neural network solve this limitation problem.

- **Non Linear Value Function Approximation**

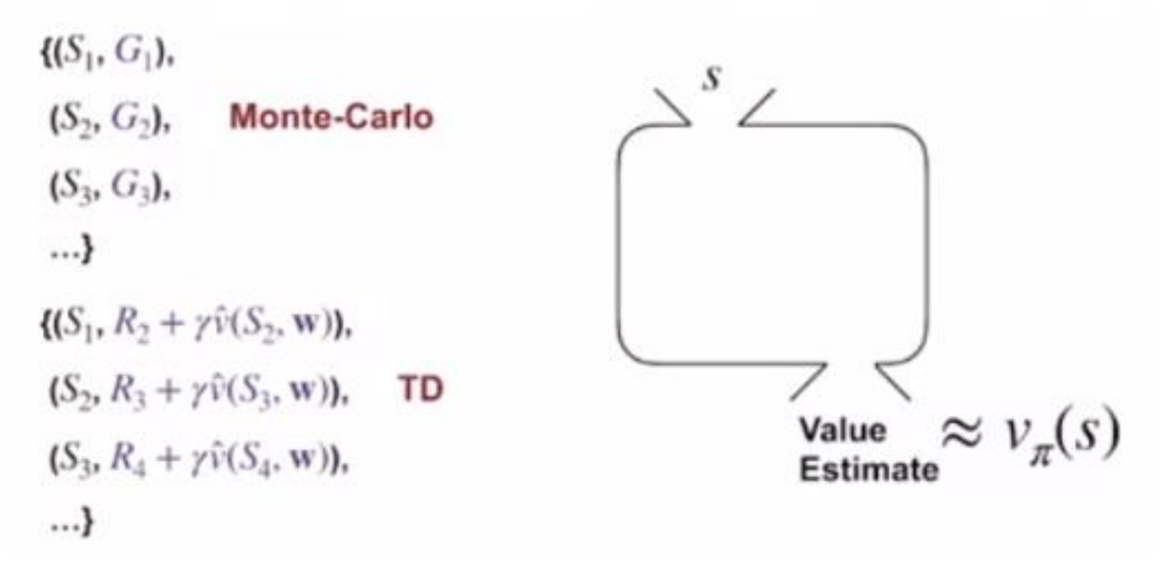


- **Value function ≈ Suppervised Learning :**

○ Value function can be framed as a supervised leaning problem.



{(Input=(2 bed, 103 m², …), Target=\$300000),
(Input=(1 bed, 80 m², …), Target=\$170000),
…,
(Input=(2 bed, 103 m², …), Target=\$515000)}



$\{(S_1, G_1),$
$(S_2, G_2),$ Monte-Carlo
$(S_3, G_3),$
$\dots\}$

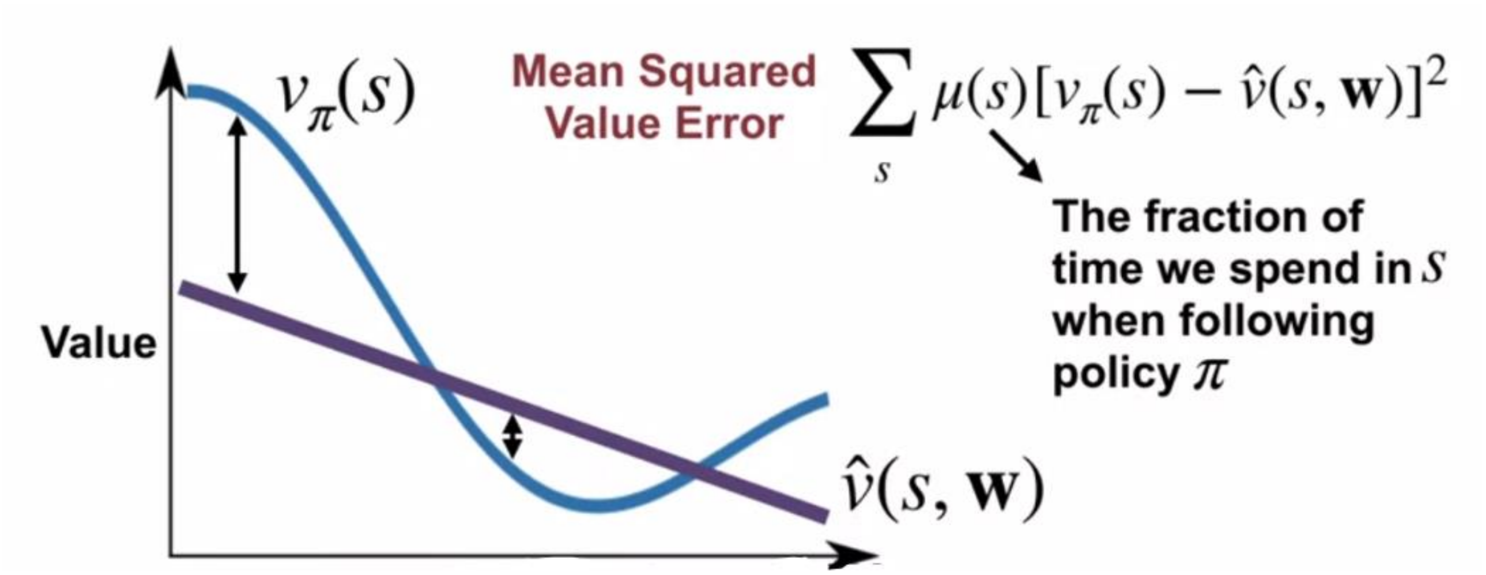
$\{(S_1, R_2 + \gamma\hat{v}(S_2, \mathbf{w})),$
$(S_2, R_3 + \gamma\hat{v}(S_3, \mathbf{w})),$ TD
$(S_3, R_4 + \gamma\hat{v}(S_4, \mathbf{w})),$
$\dots\}$

➢ The function approximator should be compatible with online updates.

$$(S_1, G_1),\ (S_2, G_2),\ (S_3, G_3),\ (S_4, G_4),\ (S_5, G_5),\ \dots$$

2. Mean squared Value Error

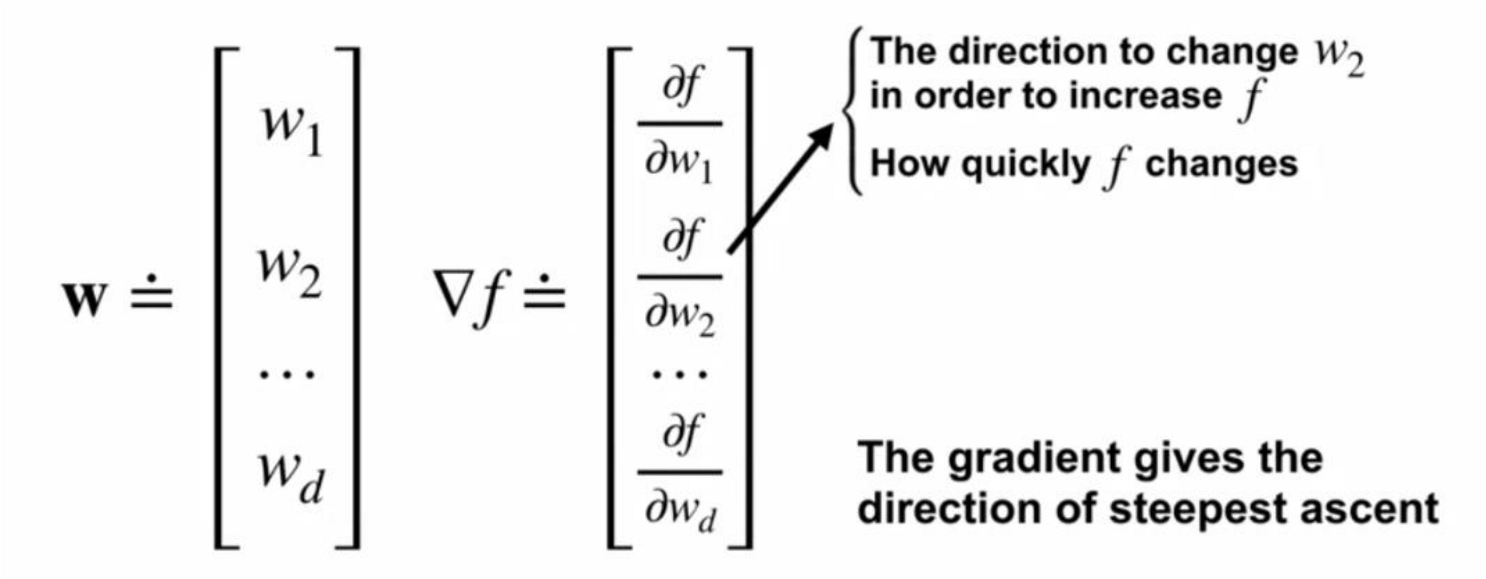
- **Mean squared Value Error**

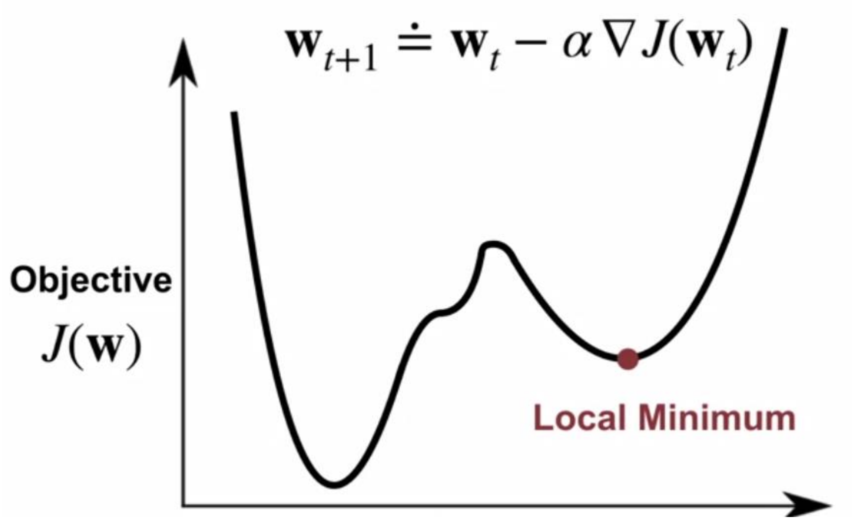

$v_\pi(s)$
Mean Squared Value Error
$\sum_s \mu(s)[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$
The fraction of time we spend in S when following policy π
Value
$\hat{v}(s, \mathbf{w})$

➢ Adapting the weights to minimize the mean squared value error objective

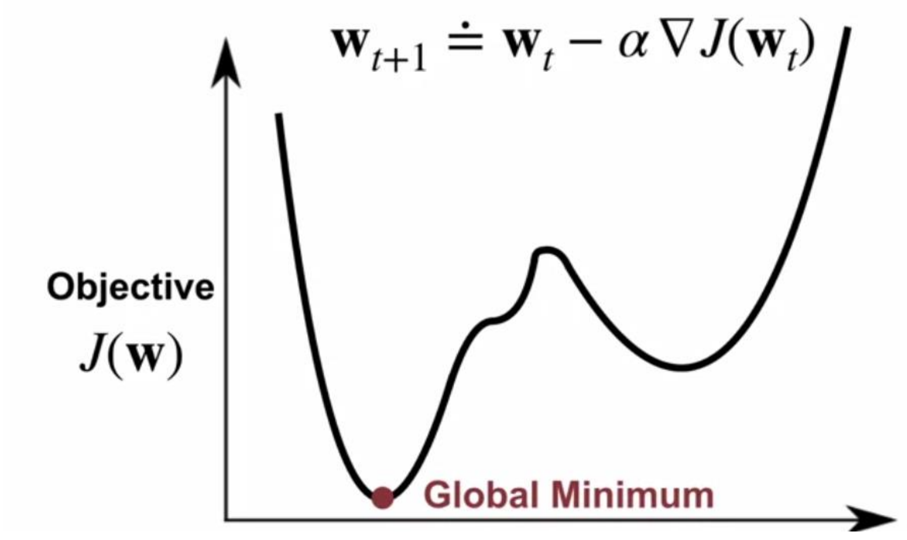
$$\overline{VE} = \sum_s \mu(s)[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$$

- **Gradient Descent**

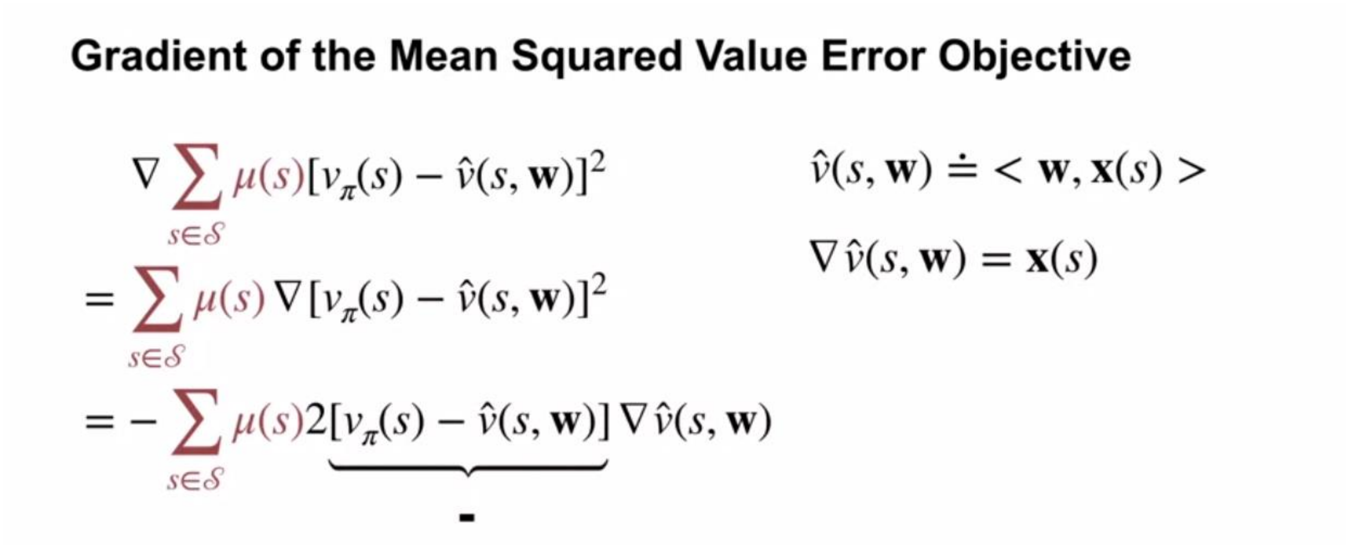

$\mathbf{w} \doteq \begin{bmatrix} w_1 \\ w_2 \\ \cdots \\ w_d \end{bmatrix}$
$\nabla f \doteq \begin{bmatrix} \frac{\partial f}{\partial w_1} \\ \frac{\partial f}{\partial w_2} \\ \cdots \\ \frac{\partial f}{\partial w_d} \end{bmatrix}$
The direction to change w_2 in order to increase f
How quickly f changes
The gradient gives the direction of steepest ascent


$\mathbf{w}_{t+1} \doteq \mathbf{w}_t - \alpha \nabla J(\mathbf{w}_t)$
Objective
$J(\mathbf{w})$
Local Minimum

Weights


$\mathbf{w}_{t+1} \doteq \mathbf{w}_t - \alpha \nabla J(\mathbf{w}_t)$
Objective
$J(\mathbf{w})$
Global Minimum

➢ We use the gradient descent to find the weights of the approximative function that gives us the minimal mean squared value error.



Gradient of the Mean Squared Value Error Objective

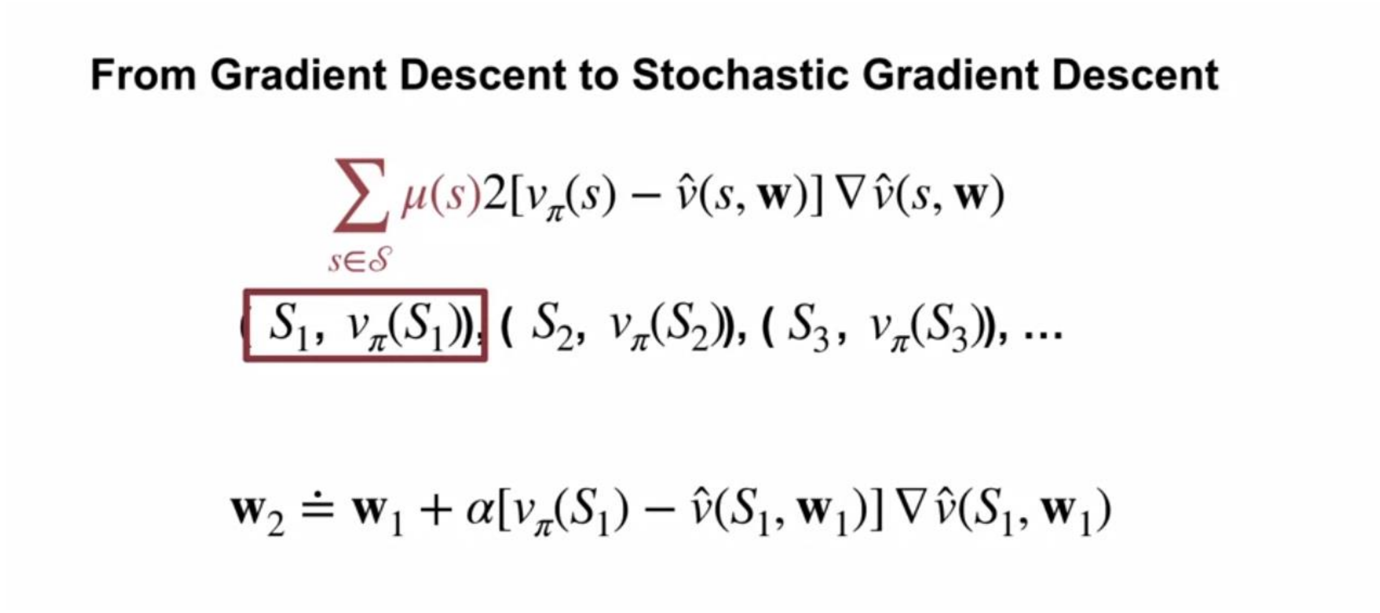
$$\nabla \sum_{s\in\mathcal{S}} \mu(s)[v_\pi(s) - \hat{v}(s,\mathbf{w})]^2$$

$$= \sum_{s\in\mathcal{S}} \mu(s)\,\nabla[v_\pi(s) - \hat{v}(s,\mathbf{w})]^2$$

$$= -\sum_{s\in\mathcal{S}} \mu(s) 2\underbrace{[v_\pi(s) - \hat{v}(s,\mathbf{w})]}_{-}\,\nabla\hat{v}(s,\mathbf{w})$$

$$\hat{v}(s,\mathbf{w}) \doteq <\mathbf{w},\mathbf{x}(s)>$$

$$\nabla\hat{v}(s,\mathbf{w}) = \mathbf{x}(s)$$



From Gradient Descent to Stochastic Gradient Descent

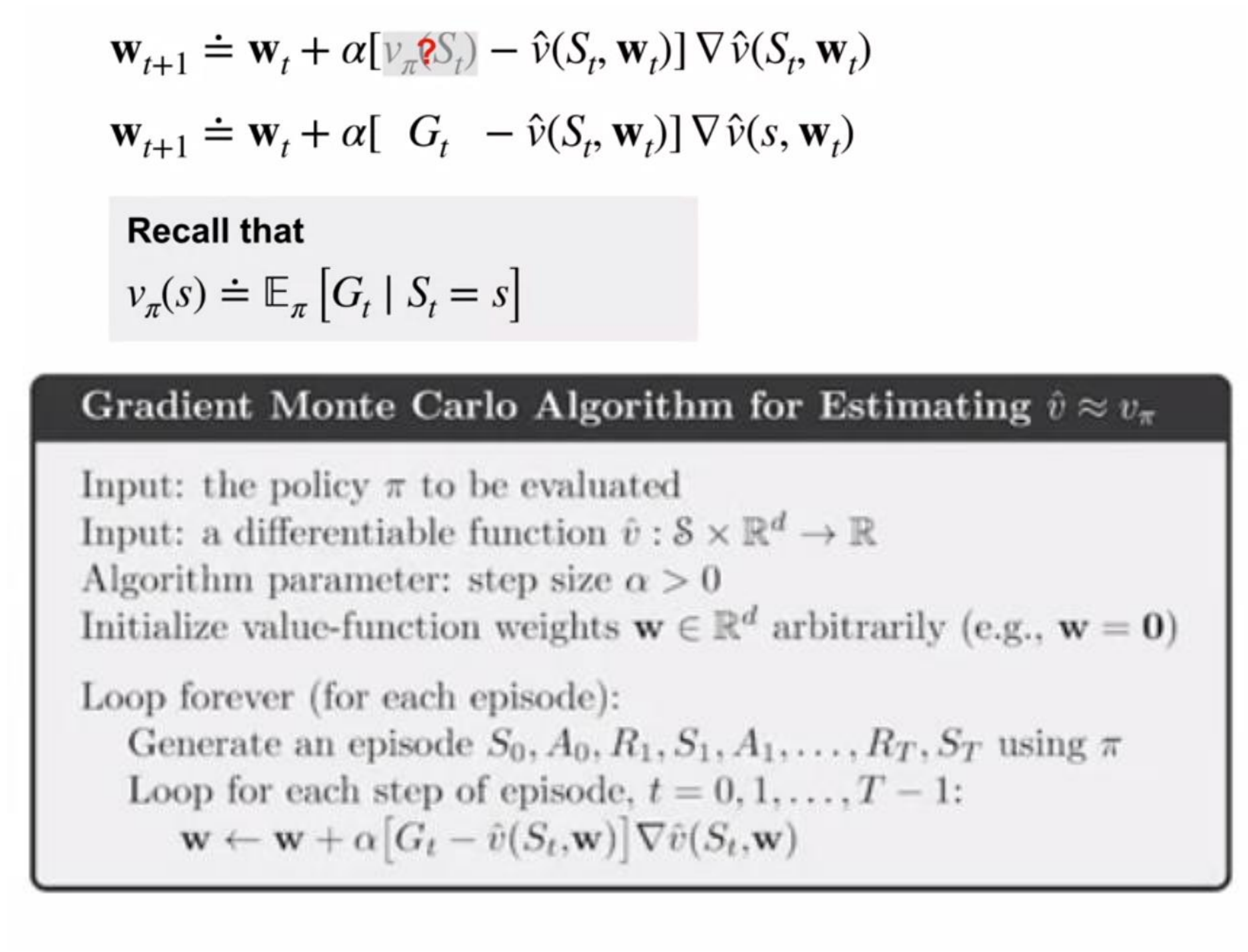
$$\sum_{s\in\mathcal{S}} \mu(s) 2[v_\pi(s) - \hat{v}(s,\mathbf{w})]\,\nabla\hat{v}(s,\mathbf{w})$$

$$\boxed{(S_1,\ v_\pi(S_1)),}\ (S_2,\ v_\pi(S_2)),\ (S_3,\ v_\pi(S_3)),\ \dots$$

$$\mathbf{w}_2 \doteq \mathbf{w}_1 + \alpha[v_\pi(S_1) - \hat{v}(S_1,\mathbf{w}_1)]\,\nabla\hat{v}(S_1,\mathbf{w}_1)$$

3. Algorithms:

- **Algorithm 01: Gradient Monte Carlo**



$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)] \nabla \hat{v}(S_t, \mathbf{w}_t)$$

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha[\ G_t \ - \hat{v}(S_t, \mathbf{w}_t)] \nabla \hat{v}(s, \mathbf{w}_t)$$

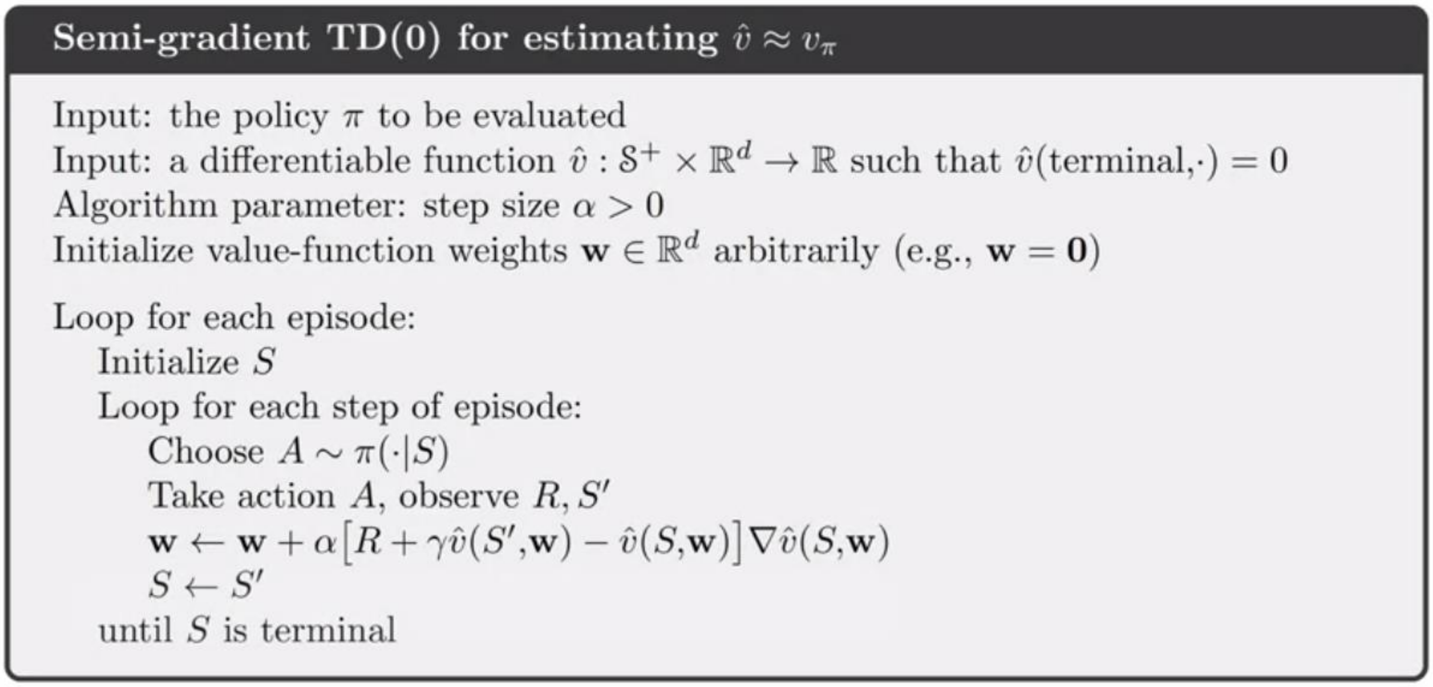
Recall that

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

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

```
Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \rightarrow \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

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

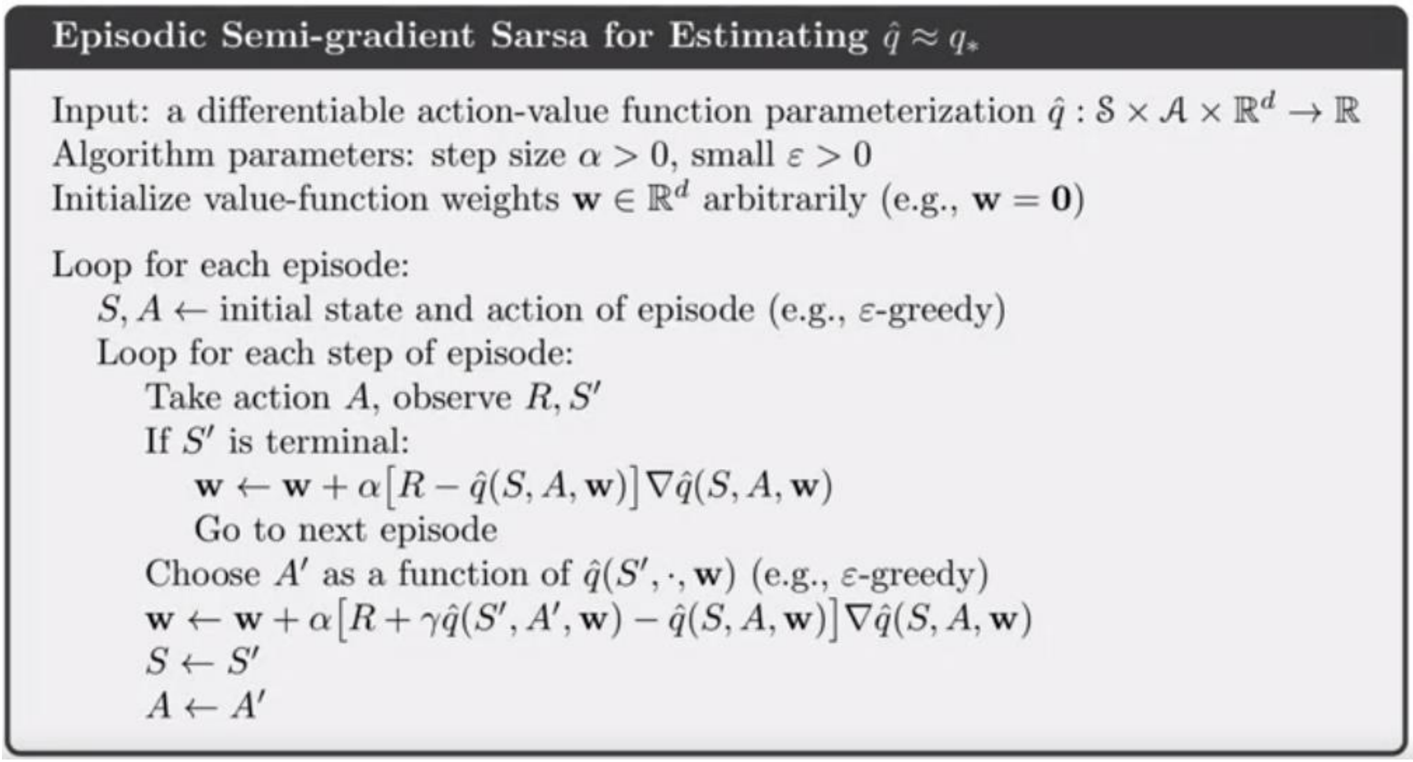
- **Algorithm 2 : Semi-gradient TD (0)**



```
Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
   Initialize $S$
   Loop for each step of episode:
      Choose $A \sim \pi(\cdot|S)$
      Take action $A$, observe $R, S'$
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
      $S \leftarrow S'$
   until $S$ is terminal
```

- We often prefer TD learning over Monte Carlo anyway because it can converge quickly

- **Algorithm 3: Episodic Sarsa with function approximation**



```
Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
   $S, A \leftarrow$ initial state and action of episode (e.g., $\varepsilon$-greedy)
   Loop for each step of episode:
      Take action $A$, observe $R, S'$
      If $S'$ is terminal:
         $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
         Go to next episode
      Choose $A'$ as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., $\varepsilon$-greedy)
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      $S \leftarrow S'$
      $A \leftarrow A'$
```

- **Expected Sarsa and Q-learning with function approximation**

Expected Sarsa:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\Big(R_{t+1} + \gamma \sum_{a'} \pi(a' \mid S_{t+1})\hat{q}(S_{t+1}, a', \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})\Big) \nabla \hat{q}(S_t, A_t, \mathbf{w})$$

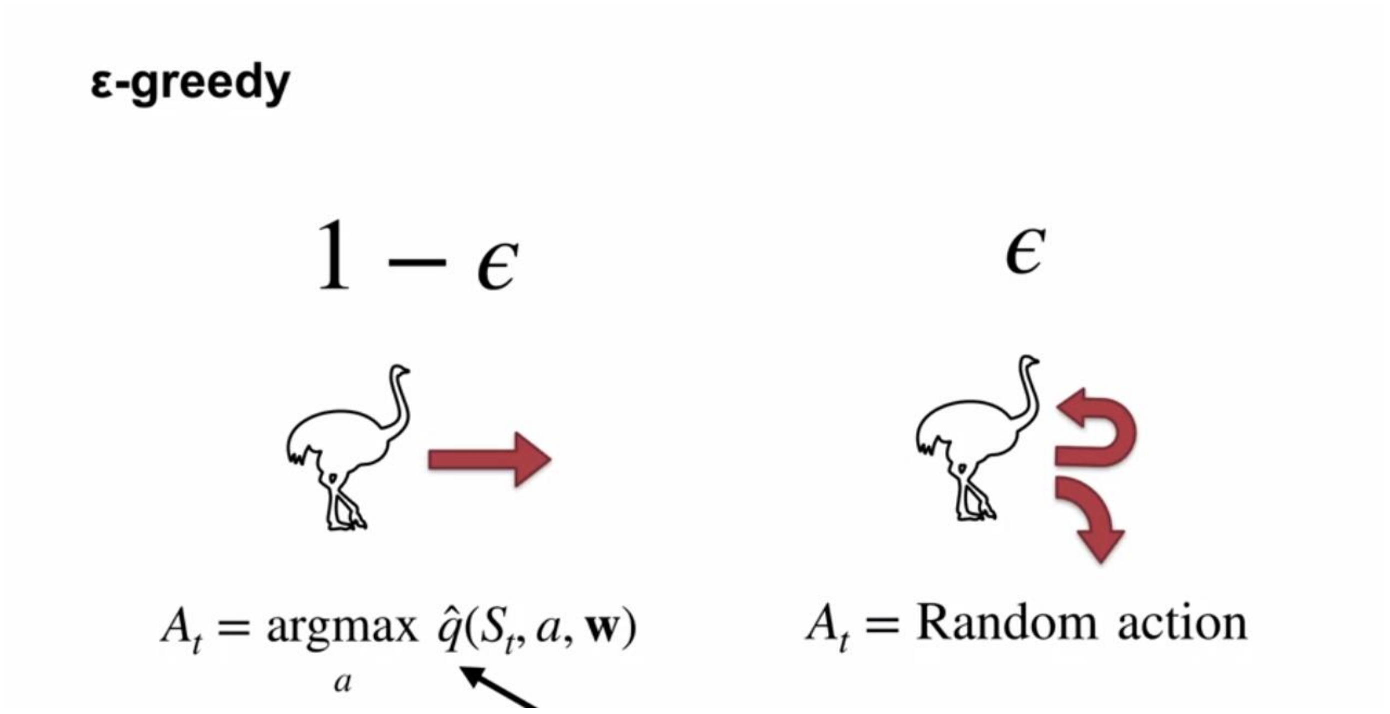
Q-learning

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\Big(R_{t+1} + \gamma \max_{a'} \hat{q}(S_{t+1}, a', \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})\Big) \nabla \hat{q}(S_t, A_t, \mathbf{w})$$

$$q_\pi(s, a) \approx \hat{q}(s, a, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s, a)$$

- **Epsilon greedy for function approximation (exploration and exploitation)**

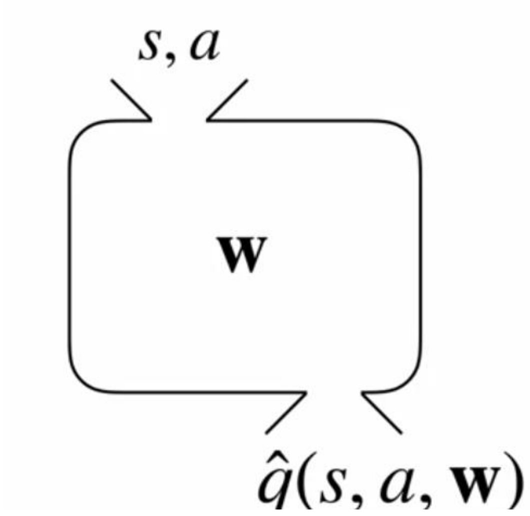
ε-greedy

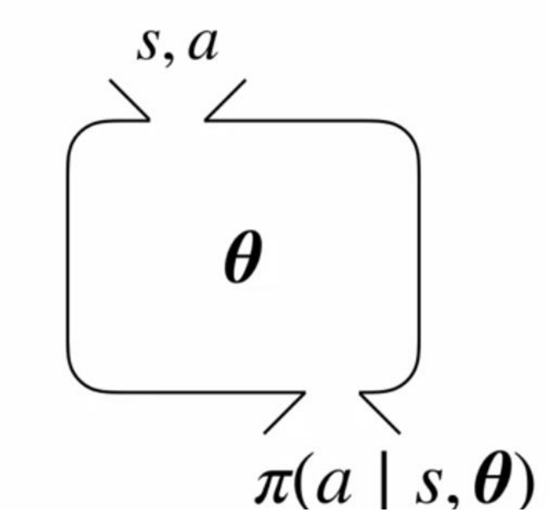


$1 - \epsilon$
ϵ
$A_t = \underset{a}{\text{argmax}}\ \hat{q}(S_t, a, \mathbf{w})$
$A_t = \text{Random action}$

4. Policy Gradient

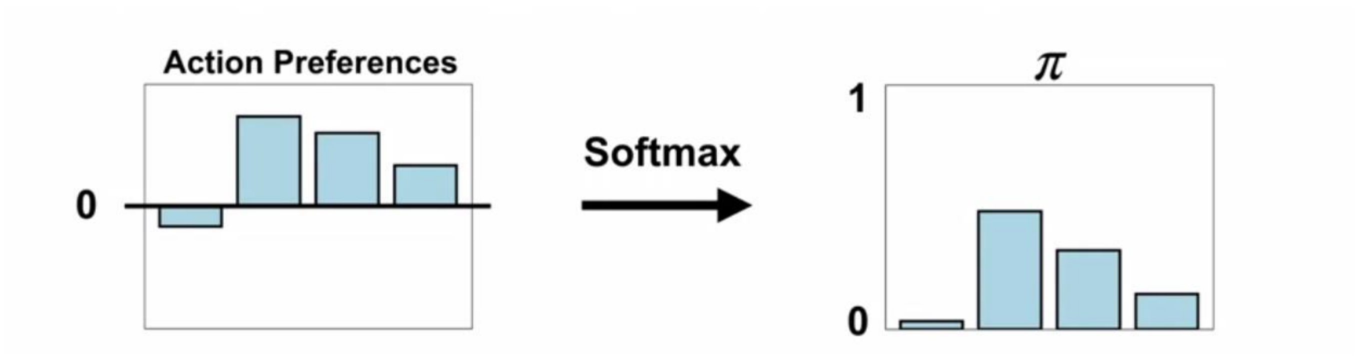
- **Parameterizing Policies Directly**





Value approximation function **Policy approximation function**

- **Softmax function**



- **Average Reward**

$$r(\pi) = \sum_{s} \mu_\pi(s) \sum_{a} \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) r$$

$$G_t = \boxed{R_{t+1} - r(\pi)} + \boxed{R_{t+2} - r(\pi)} + \boxed{R_{t+3} - r(\pi)} + \dots$$

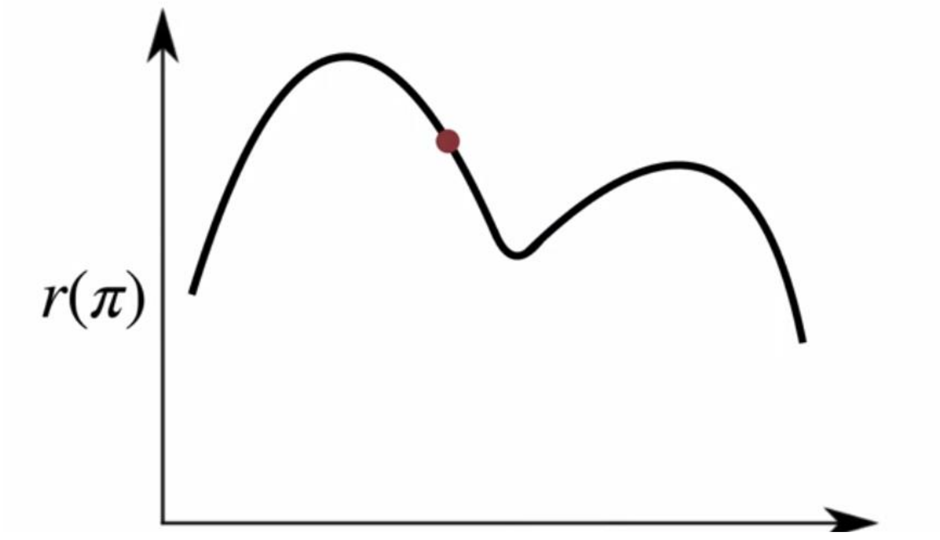
$$q_\pi(s, a) = \sum_{s',r} p(s', r \mid s, a) \Big(r - r(\pi) + \sum_{a'} \pi(a' \mid s') q_\pi(s', a') \Big)$$

- **The goal is to optimizing the average reward**

$$\nabla r(\pi) = \nabla \sum_{s} \mu(s) \sum_{a} \pi(a \mid s, \boldsymbol{\theta}) \sum_{s',r} p(s', r \mid s, a) r$$

Policy-Gradient Method

- **Gradient Ascent to maximaze the average reward**



- **Getting stochastic samples of the gradient**

$$\nabla r(\pi) = \sum_{s} \mu(s) \sum_{a} \nabla \pi(a \mid s, \boldsymbol{\theta}) q_\pi(s, a)$$

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \sum_{a} \nabla \pi(a \mid S_t, \boldsymbol{\theta}_t) q_\pi(S_t, a)$$

$$S_0, A_0, R_1, S_1, A_1, \ldots, S_t, A_t, R_{t+1}, \ldots$$

- **Getting stochastic samples with one action**

$$\sum_a \nabla \pi(a \mid S, \boldsymbol{\theta}) q_\pi(S, a)$$
$$= \sum_a \pi(a \mid S, \boldsymbol{\theta}) \boxed{\frac{1}{\pi(a \mid S, \boldsymbol{\theta})} \nabla \pi(a \mid S, \boldsymbol{\theta}) q_\pi(S, a)}$$
$$= \mathbb{E}_\pi[\frac{\nabla \pi(A \mid S, \boldsymbol{\theta})}{\pi(A \mid S, \boldsymbol{\theta})} q_\pi(S, A)]$$

- **Stochastic Gradient Ascent for policy parameters**

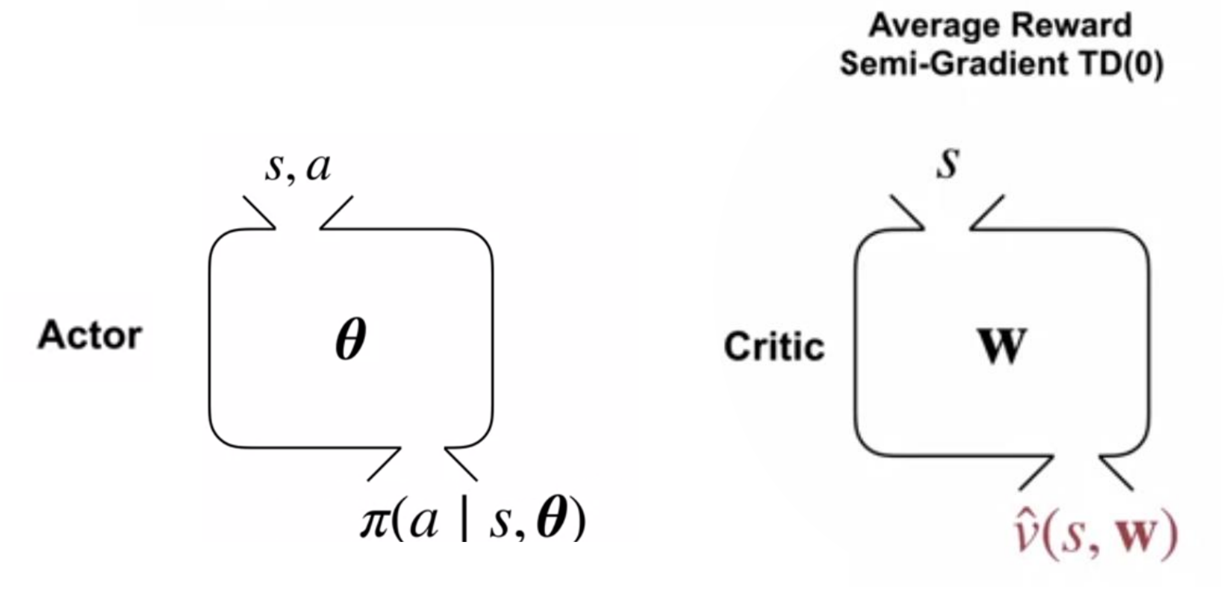
$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \frac{\nabla \pi(A_t \mid S_t, \boldsymbol{\theta}_t)}{\pi(A_t \mid S_t, \boldsymbol{\theta}_t)} q_\pi(S_t, A_t)$$

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \nabla \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_t) q_\pi(S_t, A_t)$$

5. Actor critic

- **Approximating the action value in the policy update**

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \nabla \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_t)[R_{t+1} - \bar{R} + \hat{v}(S_{t+1}, \mathbf{w})]$$



- **Subtracting the current state's value estimate**

TD Error δ

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \nabla \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_t)[\boxed{R_{t+1} - \bar{R} + \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})}]$$

Does not affect the Expected Update

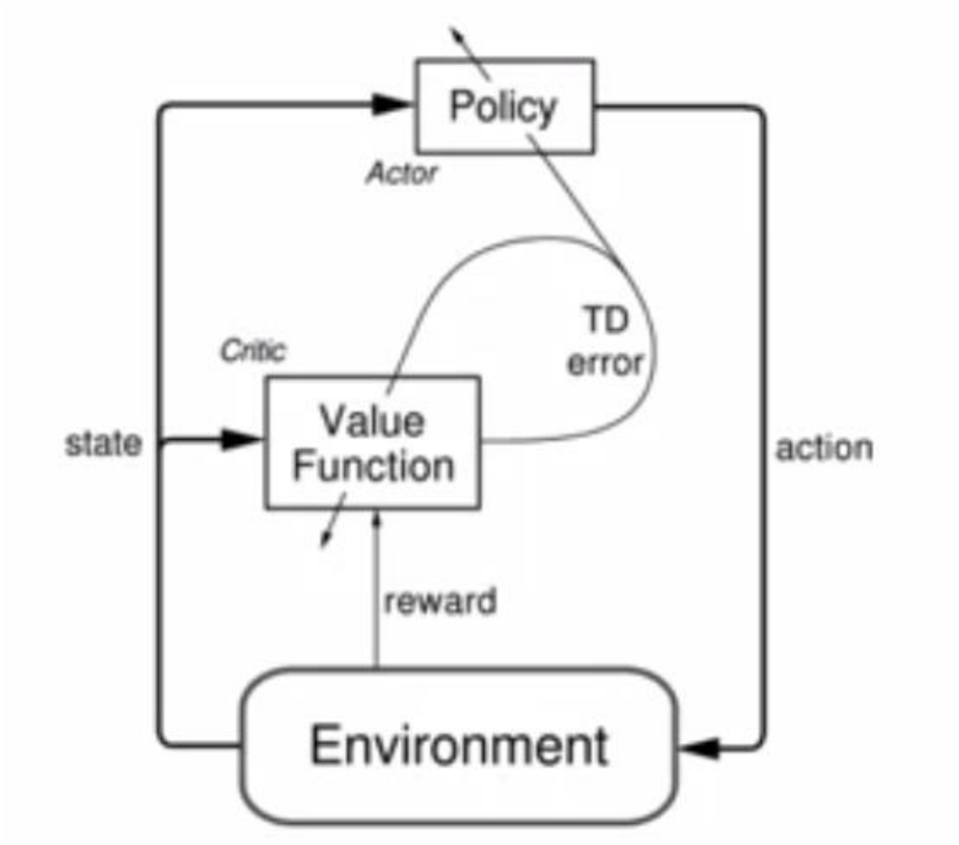
- **Adding a Baseline to reduce the variance**

$$\mathbb{E}_\pi\left[\nabla \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_t)[R_{t+1} - \bar{R} + \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})] \mid S_t = s\right]$$

$$= \mathbb{E}_\pi\left[\nabla \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_t)[R_{t+1} - \bar{R} + \hat{v}(S_{t+1}, \mathbf{w})] \mid S_t = s\right]$$

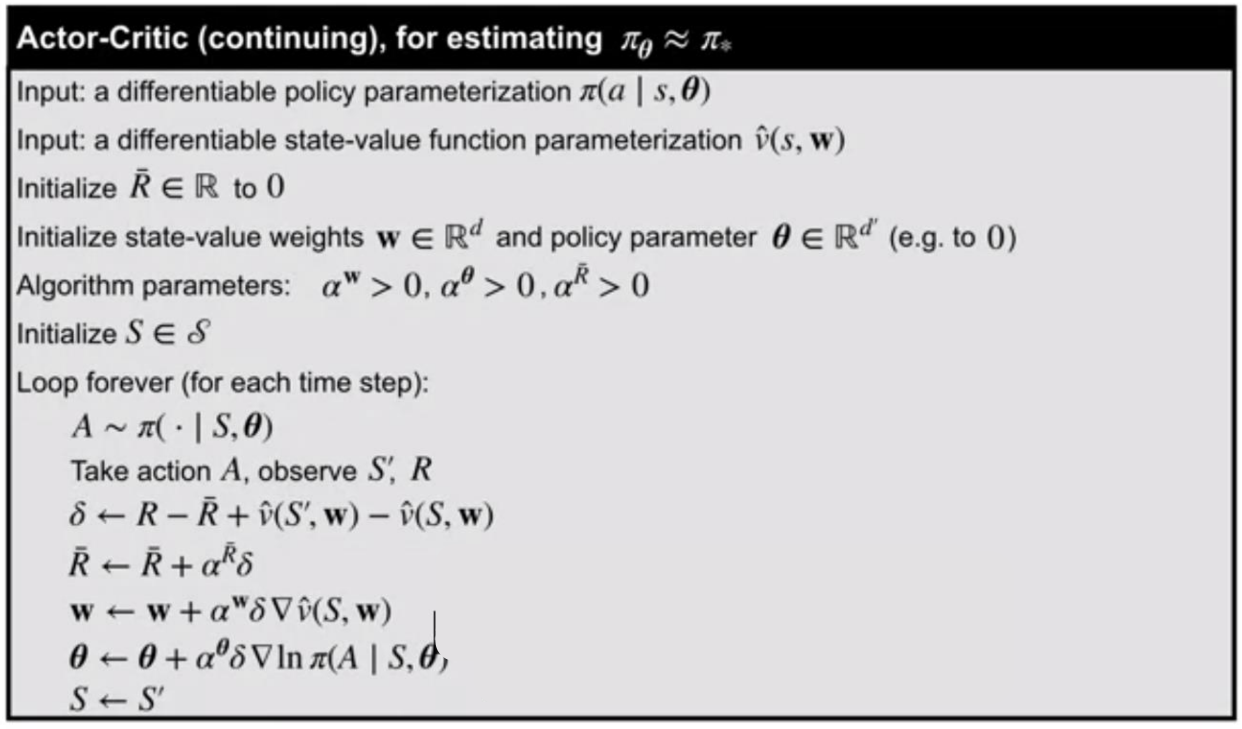
$$-\underbrace{\boxed{\mathbb{E}_\pi\left[\nabla \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_t)\hat{v}(S_t, \mathbf{w}) \mid S_t = s\right]}}_{0}$$

- **How the actor-critic interact**



- **Actor-Critic Algotithm**

Actor-Critic is a Temporal Difference (TD) version of Policy gradient. It has two networks: Actor and Critic. The actor decided which action should be taken and critic inform the actor how good was the action and how it should adjust. The learning of the actor is based on policy gradient approach. In comparison, critics evaluate the action produced by the actor by computing the value function.

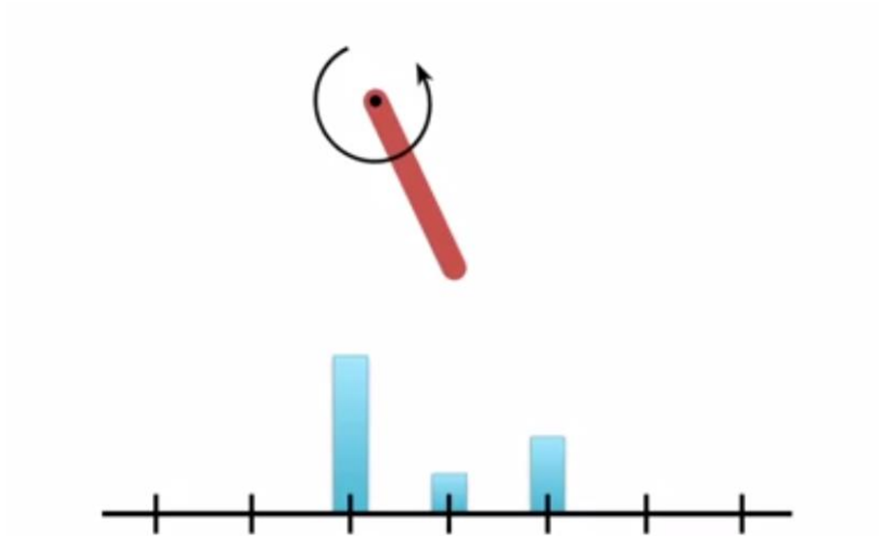


Actor-Critic (continuing), for estimating $\pi_\theta \approx \pi_*$

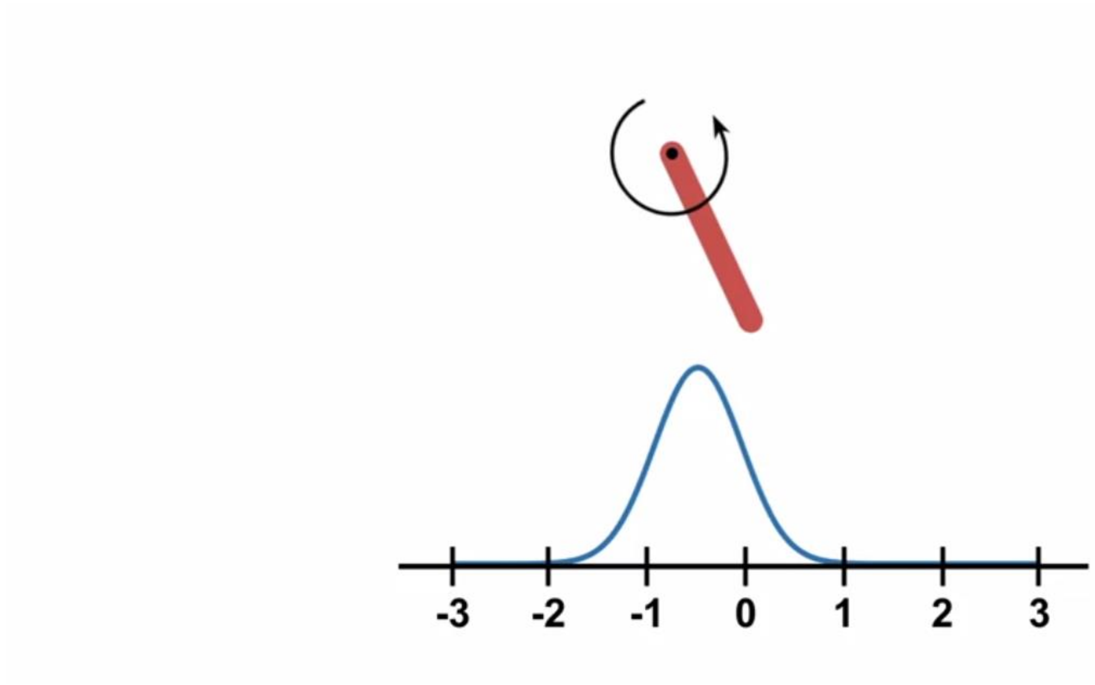
```
Input: a differentiable policy parameterization $\pi(a \mid s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s, \mathbf{w})$
Initialize $\bar{R} \in \mathbb{R}$ to $0$
Initialize state-value weights $\mathbf{w} \in \mathbb{R}^d$ and policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g. to $0$)
Algorithm parameters: $\alpha^{\mathbf{w}} > 0, \alpha^{\boldsymbol{\theta}} > 0, \alpha^{\bar{R}} > 0$
Initialize $S \in \mathcal{S}$
Loop forever (for each time step):
    $A \sim \pi(\cdot \mid S, \boldsymbol{\theta})$
    Take action $A$, observe $S'$, $R$
    $\delta \leftarrow R - \bar{R} + \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$
    $\bar{R} \leftarrow \bar{R} + \alpha^{\bar{R}} \delta$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S, \mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \nabla \ln \pi(A \mid S, \boldsymbol{\theta})$
    $S \leftarrow S'$
```

- **Continious space action**

  From discerte action space

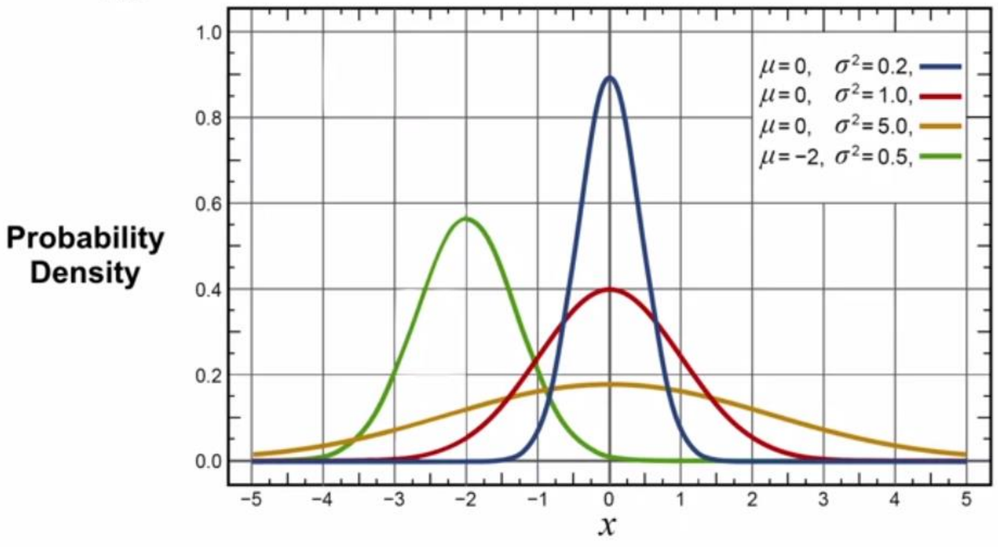


To continuious action distrubution



- **Gaussian Distrubution**

$$p(x) \doteq \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$



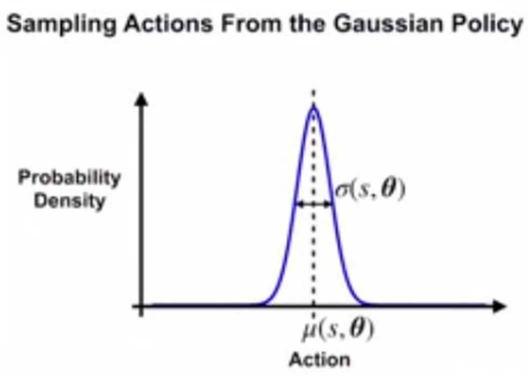
- **Gaussian Policy**

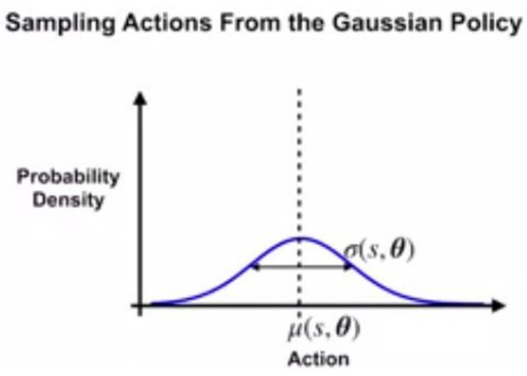
$$\pi(a \mid s, \boldsymbol{\theta}) \doteq \frac{1}{\sigma(s,\boldsymbol{\theta})\sqrt{2\pi}} \exp\left(-\frac{(a-\mu(s,\boldsymbol{\theta}))^2}{2\sigma(s,\boldsymbol{\theta})^2}\right)$$

$$\mu(s,\boldsymbol{\theta}) \doteq \boldsymbol{\theta}_\mu^T \mathbf{x}(s)$$

$$\boldsymbol{\theta} \doteq \begin{bmatrix} \boldsymbol{\theta}_\mu \\ \boldsymbol{\theta}_\sigma \end{bmatrix}$$

$$\sigma(s,\boldsymbol{\theta}) \doteq \exp\left(\boldsymbol{\theta}_\sigma^T \mathbf{x}(s)\right)$$


Sampling Actions From the Gaussian Policy
Probability Density
$\sigma(s,\boldsymbol{\theta})$
$\mu(s,\boldsymbol{\theta})$
Action


Sampling Actions From the Gaussian Policy
Probability Density
$\sigma(s,\boldsymbol{\theta})$
$\mu(s,\boldsymbol{\theta})$
Action

- **Gradient the log of the Gaussian Policy**

$$\nabla \ln \pi(a \mid s, \boldsymbol{\theta}_\mu) = \frac{1}{\sigma(s,\boldsymbol{\theta})^2}(a - \mu(s,\boldsymbol{\theta}))\mathbf{x}(s)$$

$$\nabla \ln \pi(a \mid s, \boldsymbol{\theta}_\sigma) = \left(\frac{(a - \mu(s,\boldsymbol{\theta}))^2}{\sigma(s,\boldsymbol{\theta})^2} - 1\right)\mathbf{x}(s)$$

- **Action-Critic for continious actions and discrete actions**

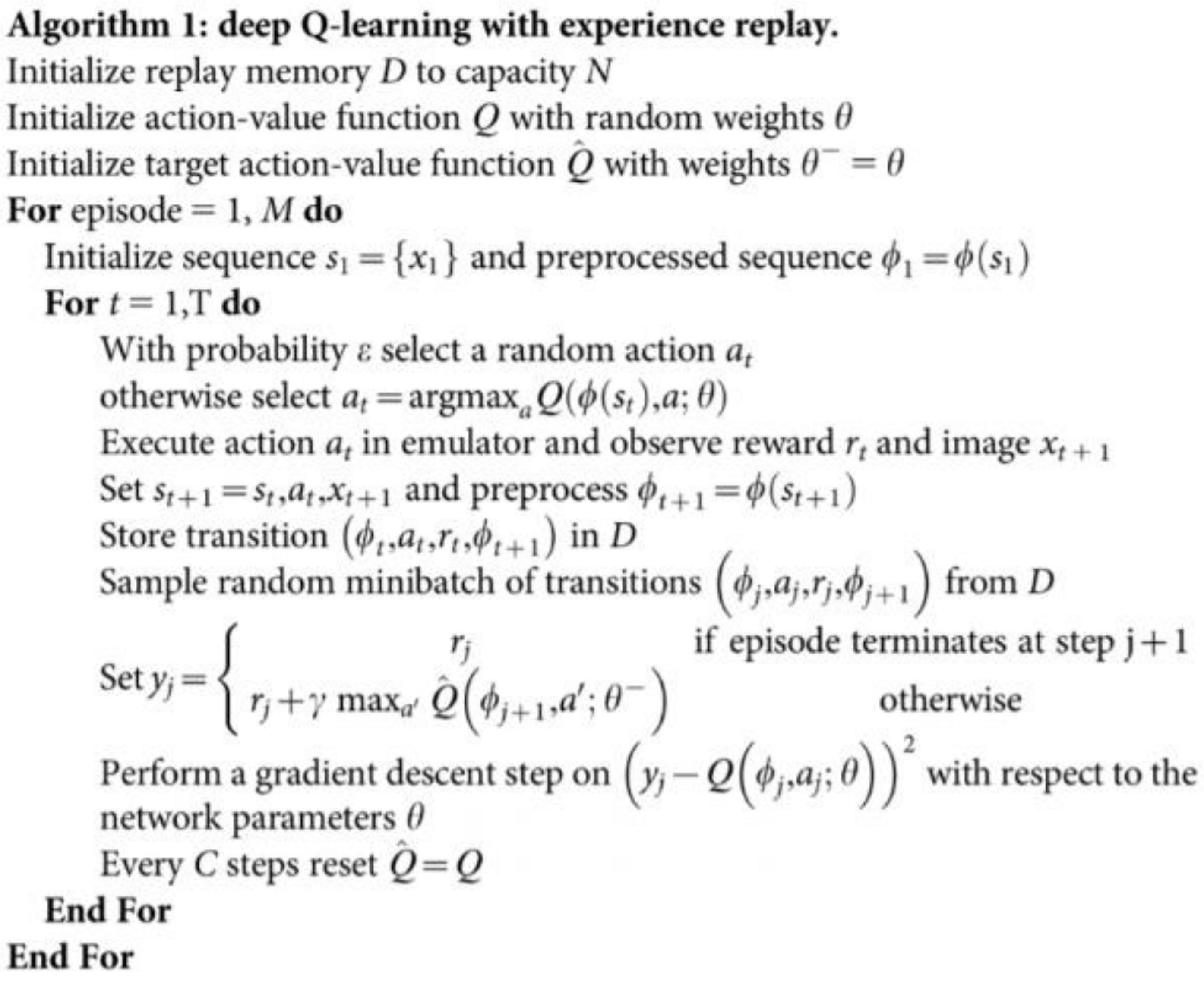
Discrete $$\pi(a \mid s, \boldsymbol{\theta}) \doteq \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_{b\in\mathcal{A}} e^{h(s,b,\boldsymbol{\theta})}}$$

Continuous $$\pi(a \mid s, \boldsymbol{\theta}) \doteq \frac{1}{\sigma(s,\boldsymbol{\theta})\sqrt{2\pi}} \exp\left(-\frac{(a - \mu(s,\boldsymbol{\theta}))^2}{2\sigma(s,\boldsymbol{\theta})^2}\right)$$

❖Deep Reinforcement Learning

1. Deep Q-Learning (DQN)(Discrete action space)

During the training process, the Agent, interacts with the environment and receives data, which is used during the learning the Q neural network. The Agent explores the environment to build a complete picture of transitions and action outcomes. At the beginning the Agent decides about the actions randomly which over time becomes insufficient. While exploring the environment the Agent tries to look on Q network (approximation) in order to decide how to act. We called this approach (combination of random behavior and according to Q network) as an **epsilon-greedy** method, which just means changing between random and Q policy using the probability hyper parameter epsilon.
During the learning process we use two separate Q networks (Q Network local and Q Network target) to calculate the predicted value (weights θ) and target value (weights θ’). The target network is frozen for several time steps and then the target network weights are updated by copying the weights from the actual Q network. Freezing the target Q Network for a while and then updating its weights with the actual Q network weights stabilizes the training.



```
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For $t$ = 1,T do
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
```

2. Deep Deterministic Policy Gradient (DDPG) (Continious action space)

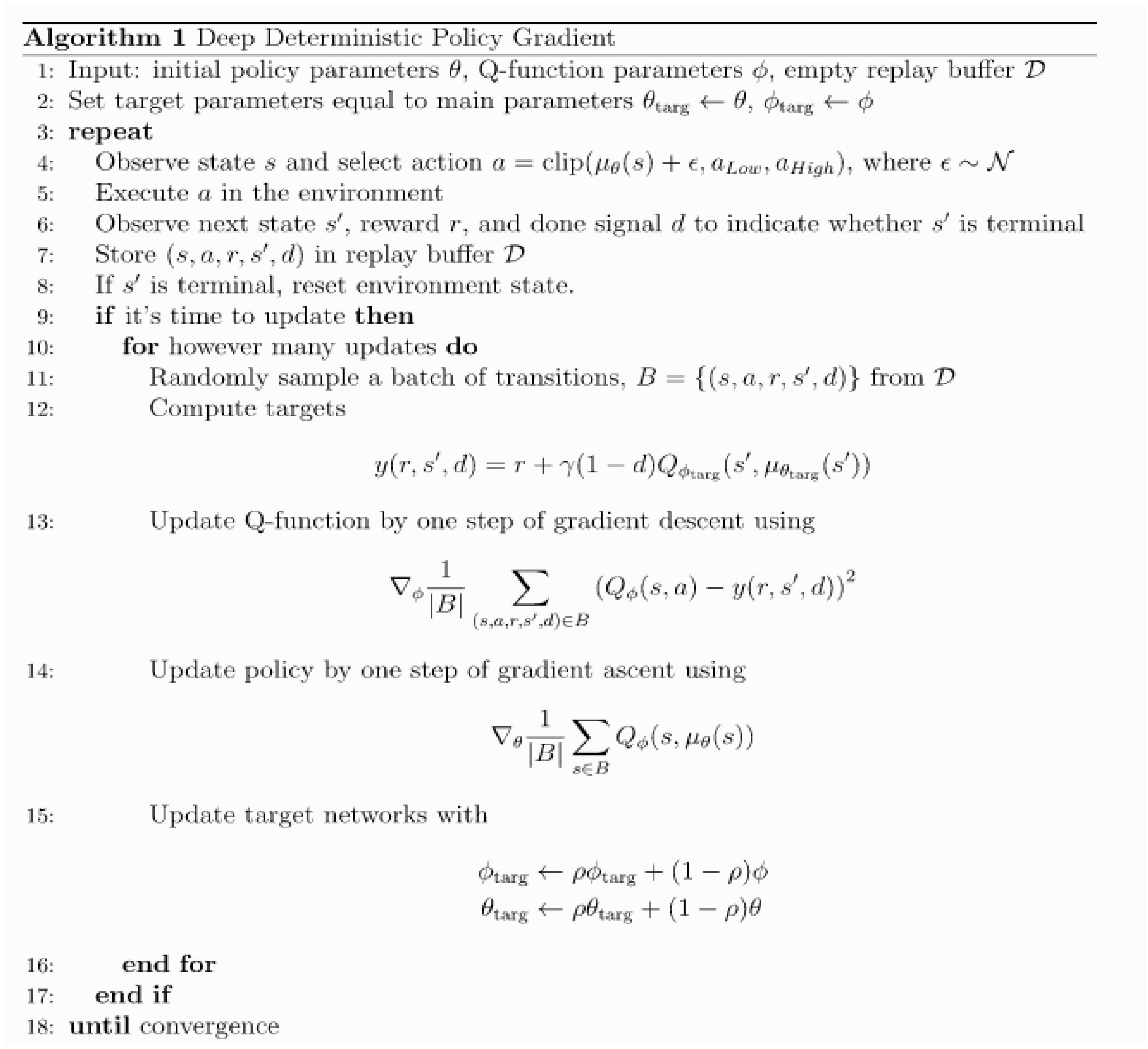
DDPG is a model-free off-policy actor-critic algorithm that combines Deep Q Learning (DQN) and DPG. Original DQN works in a discrete action space and DPG extends it to the continuous action space while learning a deterministic policy. As it is an off-policy algorithm, it uses two separate policies for the exploration and updates. It uses a stochastic behavior policy for the exploration and deterministic policy for the target updated is an actor-critic algorithm; it has two networks: actor and critic. Technically, the actor produces the action to explore. During the update process of the actor, TD error from a critic is used. The critic network gets updated based on the TD error similar to Q-learning update rule.

<u>Algorithm 01:</u>

Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $s_1$
    for t = 1, T do
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
        Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
        Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:

            $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$

        Update the target networks:
            $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
            $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$

    end for
end for
```

Algorithm 02:



Algorithm 1 Deep Deterministic Policy Gradient

```
1: Input: initial policy parameters $\theta$, Q-function parameters $\phi$, empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3: repeat
4:    Observe state $s$ and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
5:    Execute $a$ in the environment
6:    Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
7:    Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
8:    If $s'$ is terminal, reset environment state.
9:    if it's time to update then
10:       for however many updates do
11:          Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12:          Compute targets
                $y(r, s', d) = r + \gamma(1 - d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$
13:          Update Q-function by one step of gradient descent using
                $\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_\phi(s, a) - y(r, s', d))^2$
14:          Update policy by one step of gradient ascent using
                $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$
15:          Update target networks with
                $\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1 - \rho)\phi$
                $\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$
16:       end for
17:    end if
18: until convergence
```