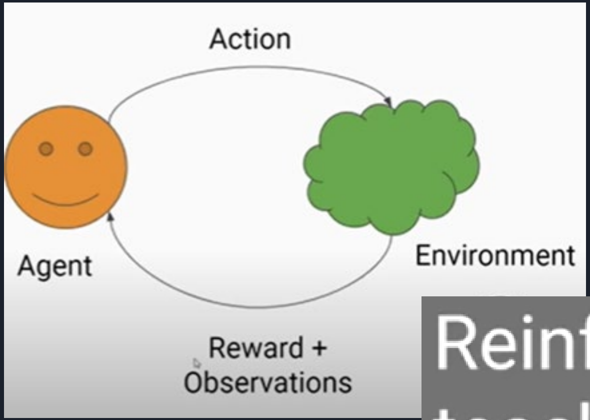
Reinforcement Learning



Reinforcement learning focuses on teaching agents through trial and error.

The agent is **the component that makes the decision of what action to take**

environment contains information about all possible **actions that an agent can take**

State is **the current board position**,

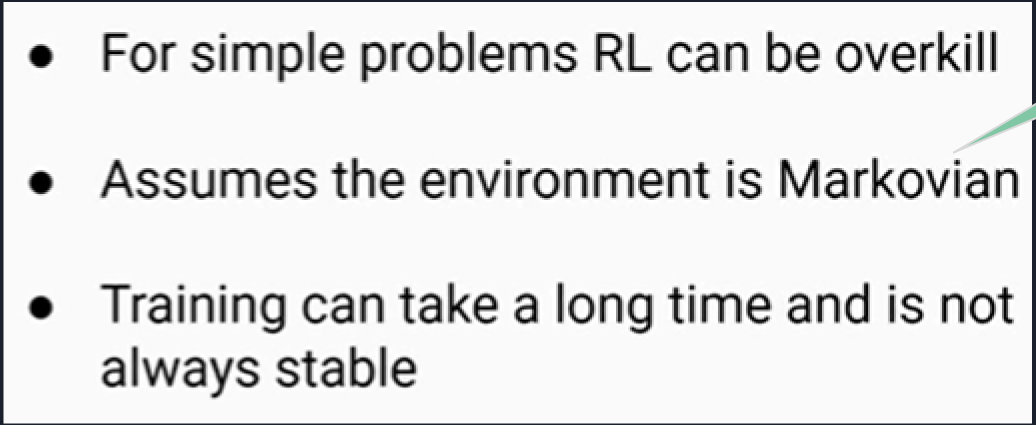
Action **the mechanism by which the agent transitions between states of the environment**.

Reward Function is an incentive mechanism that tells the agent what is correct and what is wrong using **reward** and punishment.

policy is, therefore, **a strategy that an agent uses in pursuit of goals**.

Observation So after it performs that specific action the agent observes the **state of the environment** (Observation) and a goodness score (or Reward) based on its last action. These feedback information are then utilized by the RL algorithm to upgrade/ improve the existing strategy (or policy), for better performance in the future.

limitations



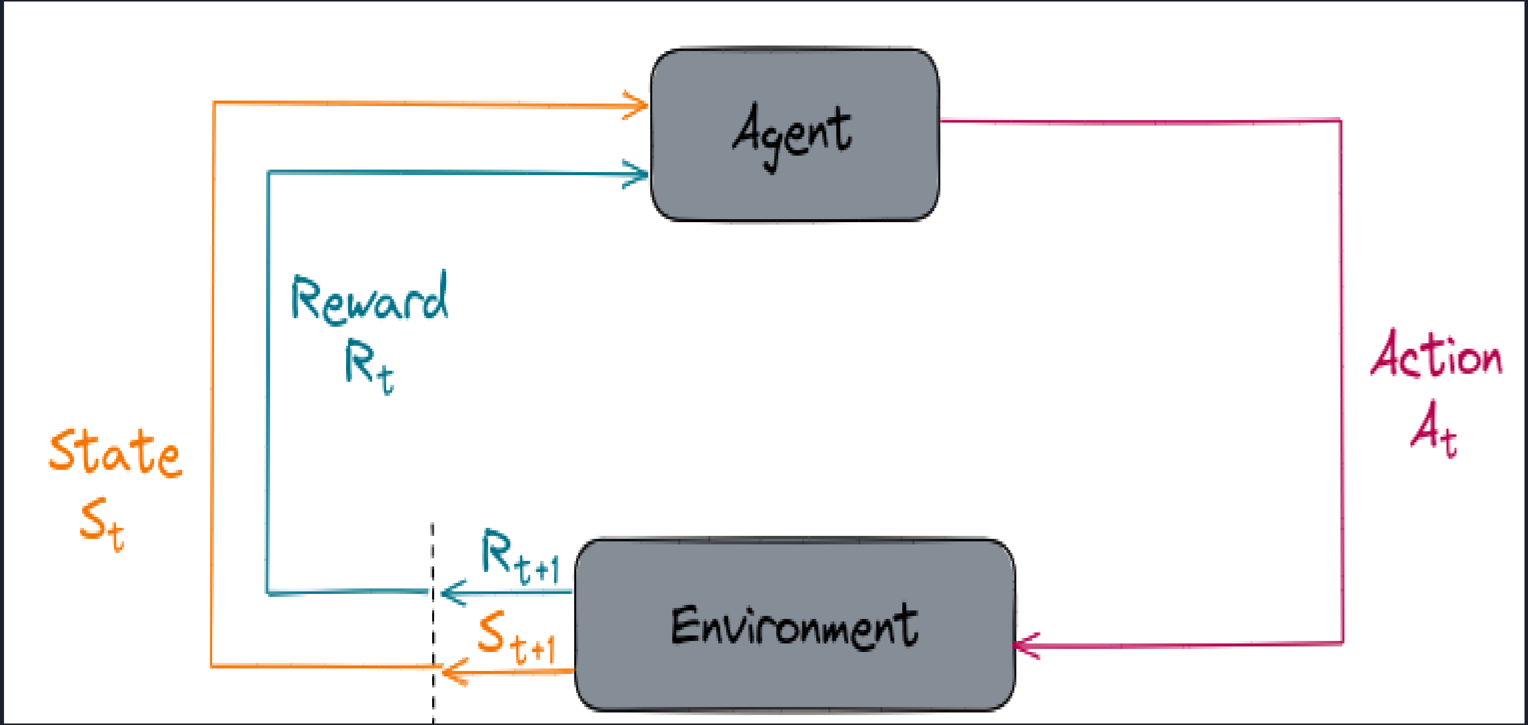
- For simple problems RL can be overkill
- Assumes the environment is Markovian
- Training can take a long time and is not always stable

No random perturbation

applications

- autonomous driving
- securities trading
- neural network architecture search // choose your best deep learning architecture for your problem
- simulated training of robots

Markovian Decision Processes



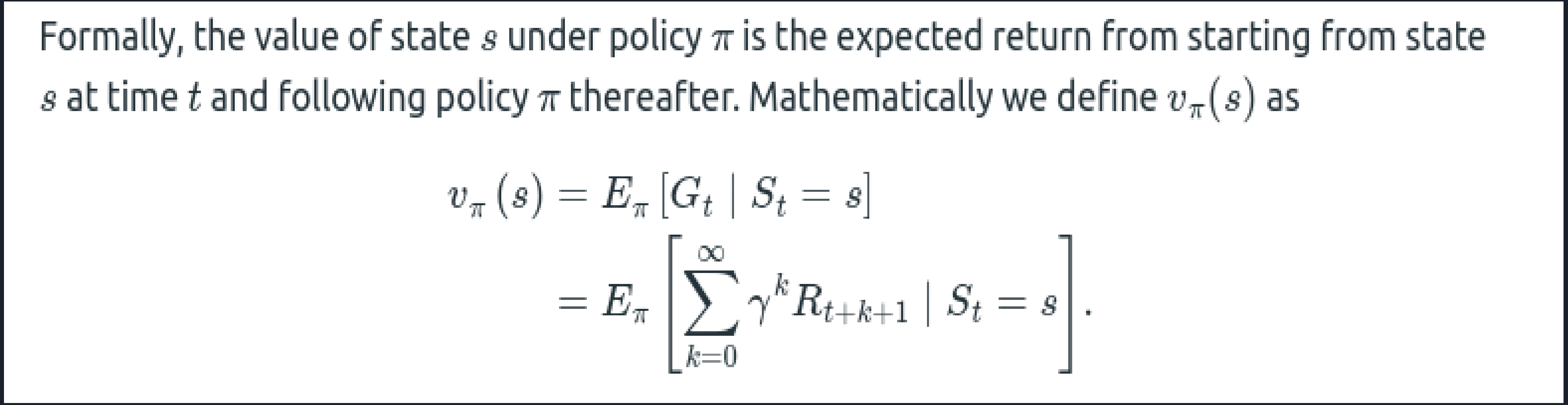
Expected Return

the agent's goal is to maximize the expected return of rewards.

$$
\begin{aligned}
G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \\
&= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}.
\end{aligned}
$$

Gamma : Discount Factor 0< and <1

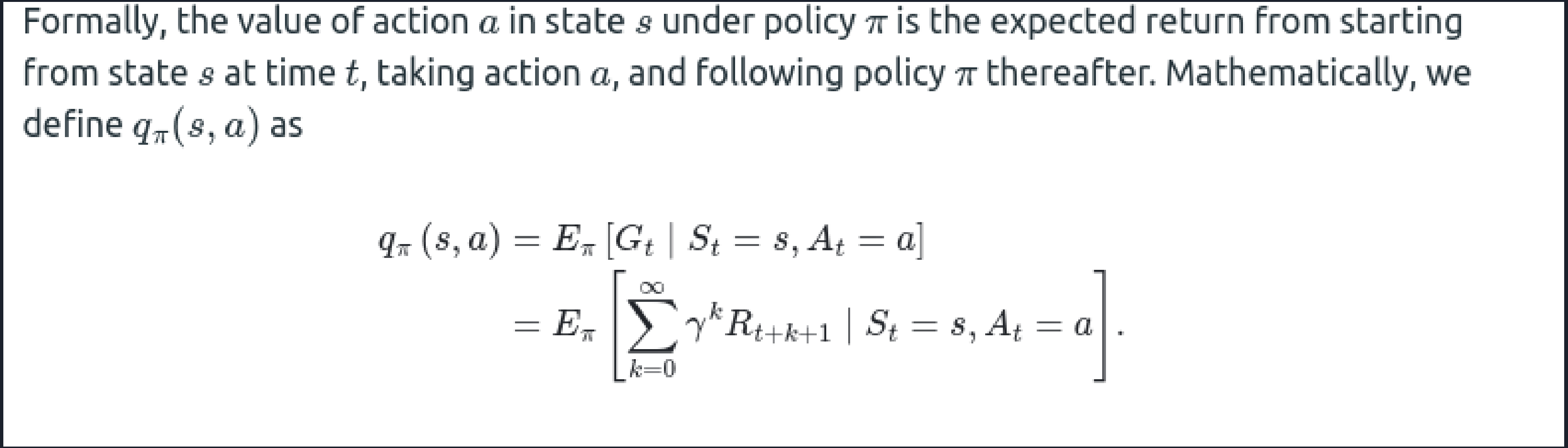
State-Value Function



Formally, the value of state s under policy π is the expected return from starting from state s at time t and following policy π thereafter. Mathematically we define $v_\pi(s)$ as

$$\begin{aligned} v_\pi(s) &= E_\pi\left[G_t \mid S_t = s\right] \\ &= E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right]. \end{aligned}$$

Action-Value Function



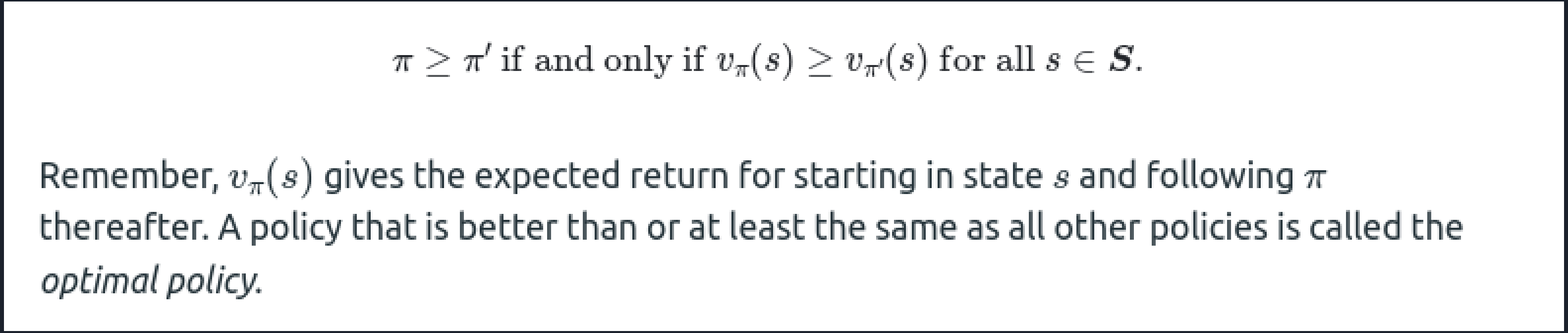
Formally, the value of action a in state s under policy π is the expected return from starting from state s at time t, taking action a, and following policy π thereafter. Mathematically, we define $q_\pi(s,a)$ as

$$\begin{aligned} q_\pi(s,a) &= E_\pi\left[G_t \mid S_t = s, A_t = a\right] \\ &= E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right]. \end{aligned}$$

Optimality

Reinforcement learning algorithms seek to find a policy that will **yield more return to the agent than all other policies.**

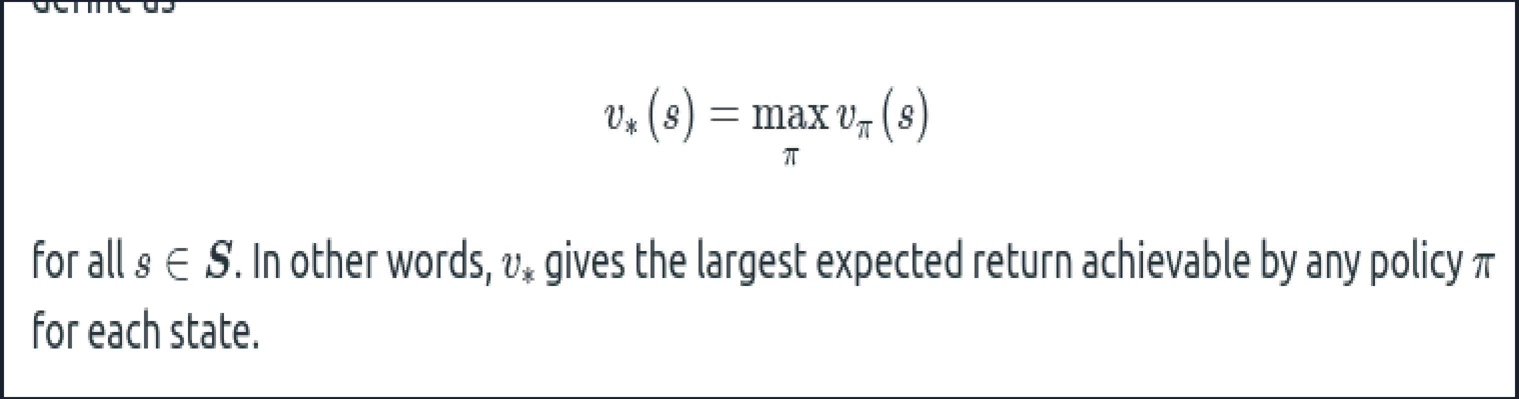
Optimal Policy



$$\pi \geq \pi' \text{ if and only if } v_\pi(s) \geq v_{\pi'}(s) \text{ for all } s \in \boldsymbol{S}.$$

Remember, $v_\pi(s)$ gives the expected return for starting in state s and following π thereafter. A policy that is better than or at least the same as all other policies is called the *optimal policy*.

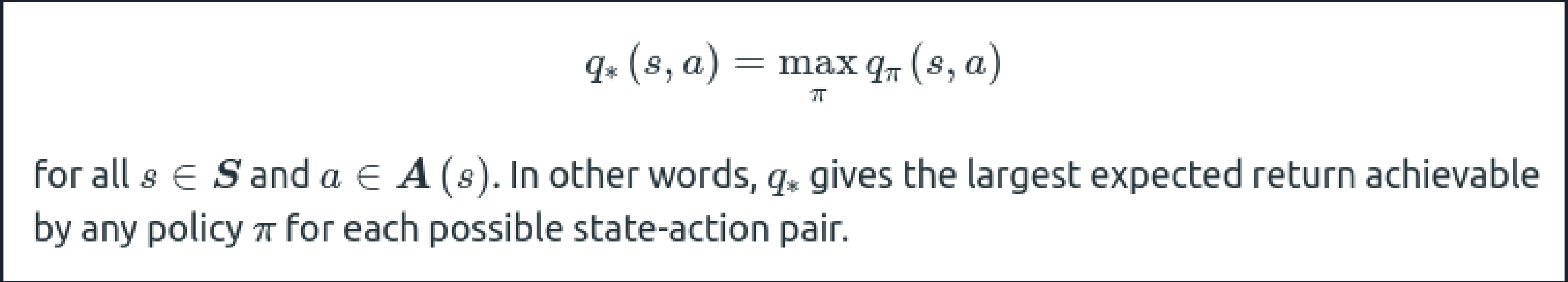
Optimal State-Value Function



$$v_*(s) = \max_{\pi} v_\pi(s)$$

for all $s \in \boldsymbol{S}$. In other words, v_* gives the largest expected return achievable by any policy π for each state.

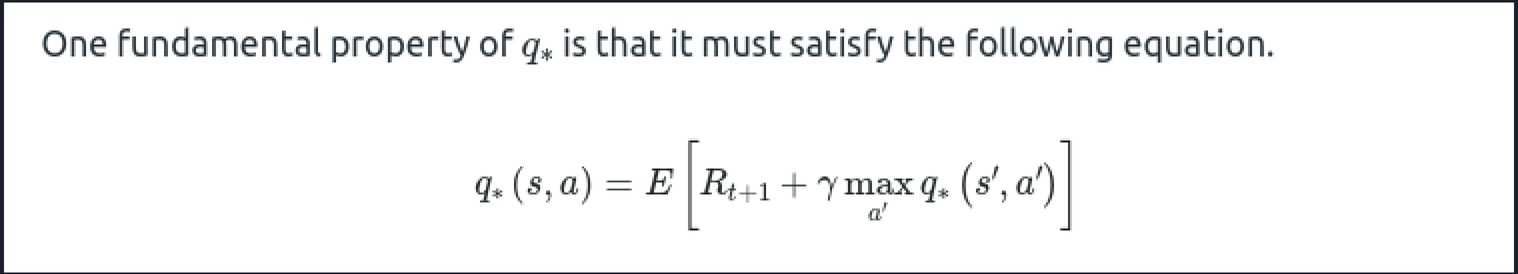
Optimal Action-Value Function



$$q_* (s, a) = \max_{\pi} q_\pi (s, a)$$

for all $s \in \boldsymbol{S}$ and $a \in \boldsymbol{A}(s)$. In other words, q_* gives the largest expected return achievable by any policy π for each possible state-action pair.

Bellman Optimality Equation



One fundamental property of q_* is that it must satisfy the following equation.

$$q_*\left(s,a\right) = E\left[R_{t+1} + \gamma \max_{a'} q_*\left(s',a'\right)\right]$$

Q-Learning

Q-learning is a technique that can solve for the optimal policy in an MDP.

the goal of Q-learning is to find the optimal policy by learning the optimal Q-values for each state-action pair.

Other Algorithms

- Q-Learning
- Deep Q-Learning
- Deep Deterministic Policy Gradient
- Soft Actor-Critic
- State–action–reward–state–action (SARSA)

General Policy iteration (GPI)

The algorithm works by first initializing a policy, and then iteratively improving it. In each iteration, two main steps are performed:

Policy Evaluation: The value function for the current policy is estimated. This is typically done by iteratively updating the value function until it converges to the true value function.

Policy Improvement: A new policy is derived from the value function by choosing actions that have higher values. This new policy is then used in the next iteration of the algorithm.

These two steps are repeated until the policy converges to an optimal policy.

Importance Sampling

Importance sampling is a technique used in Reinforcement Learning (RL) to estimate the value of an expected reward or return under a given policy when only samples from a different policy are available. This technique is used when the agent has to learn from experience and the environment is not fully observable. Importance sampling allows the agent to estimate the expected value of the reward, which is necessary for making decisions that maximize the expected reward.

Monte carlo VS one-step TD

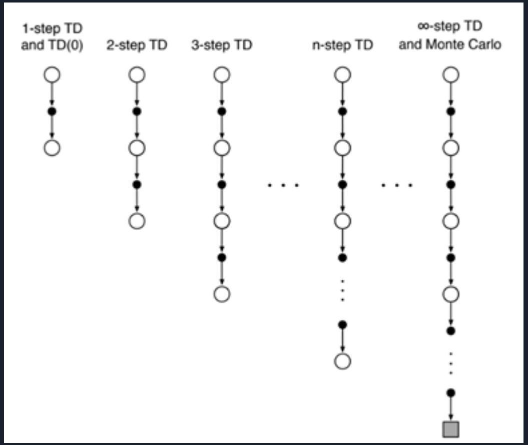
- Monte Carlo methods rely on the complete episode of interaction
- Monte Carlo methods require the agent to wait until the end of an episode to update the value function
- Monte Carlo methods are guaranteed to converge to the optimal policy in finite time
- Monte Carlo methods can have high variance due to the dependence on the random rewards observed during the episode

- TD methods update the value function incrementally based on the observed reward at each time step
- TD methods update the value function at each time step
- whereas TD methods do not have such guarantees
- TD methods have low variance due to the update based on a single observed reward at each time step

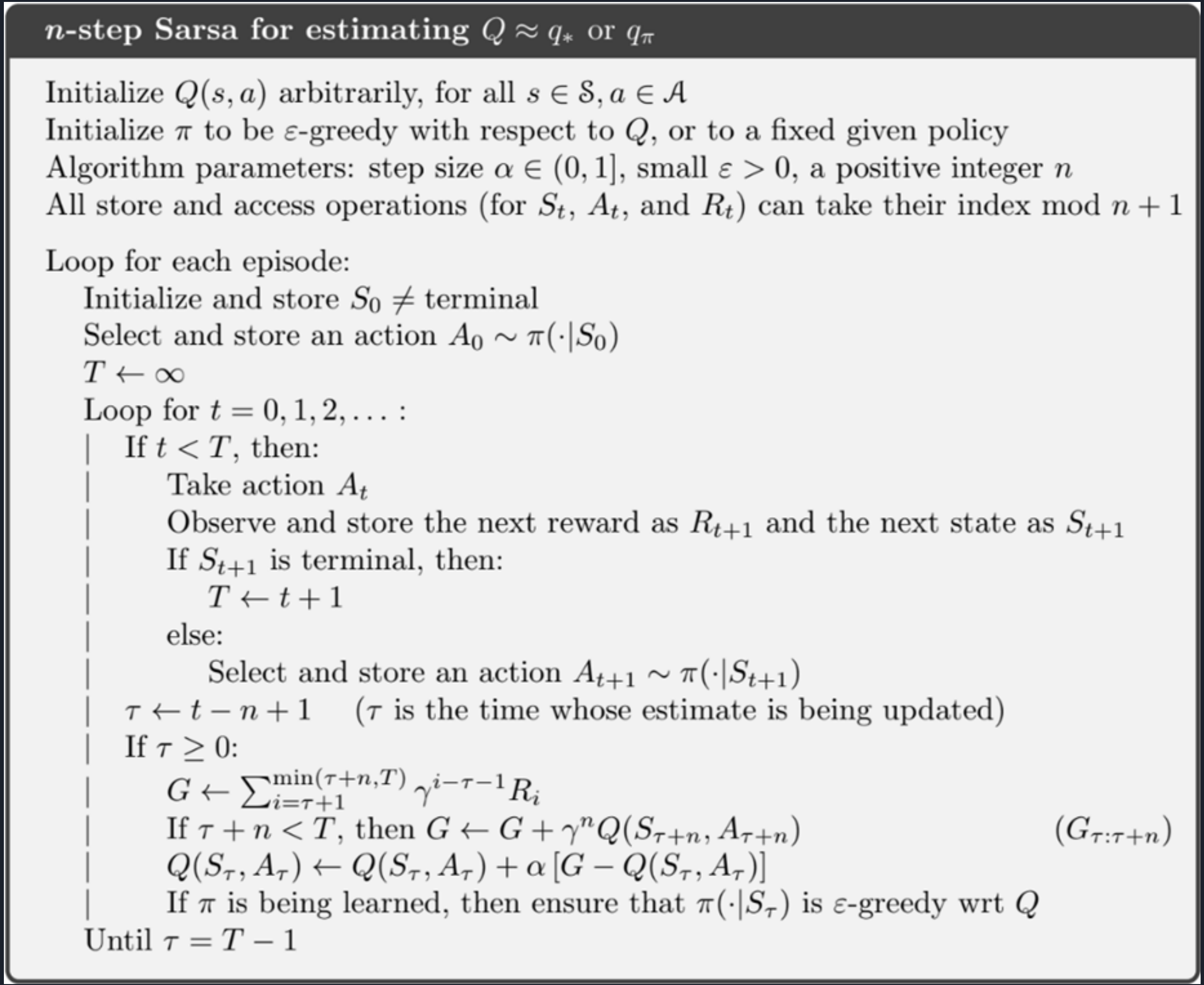
Neither MC methods nor one-step TD methods are always the best.

n-step TD

- generalize both methods so that one can shift from one to the other smoothly as needed
- n-step methods span a spectrum with MC methods at one end and one-step TD methods at the other. The best methods are often intermediate between the two extremes.
- n-step methods enable bootstrapping to occur over multiple steps, freeing us from the tyranny of the single time step.
- enable bootstrapping over multiple time intervals simultaneously.
- One kind of intermediate method,
- then, would perform an update based on an intermediate number of rewards: more than
- one, but less than all of them until termination.



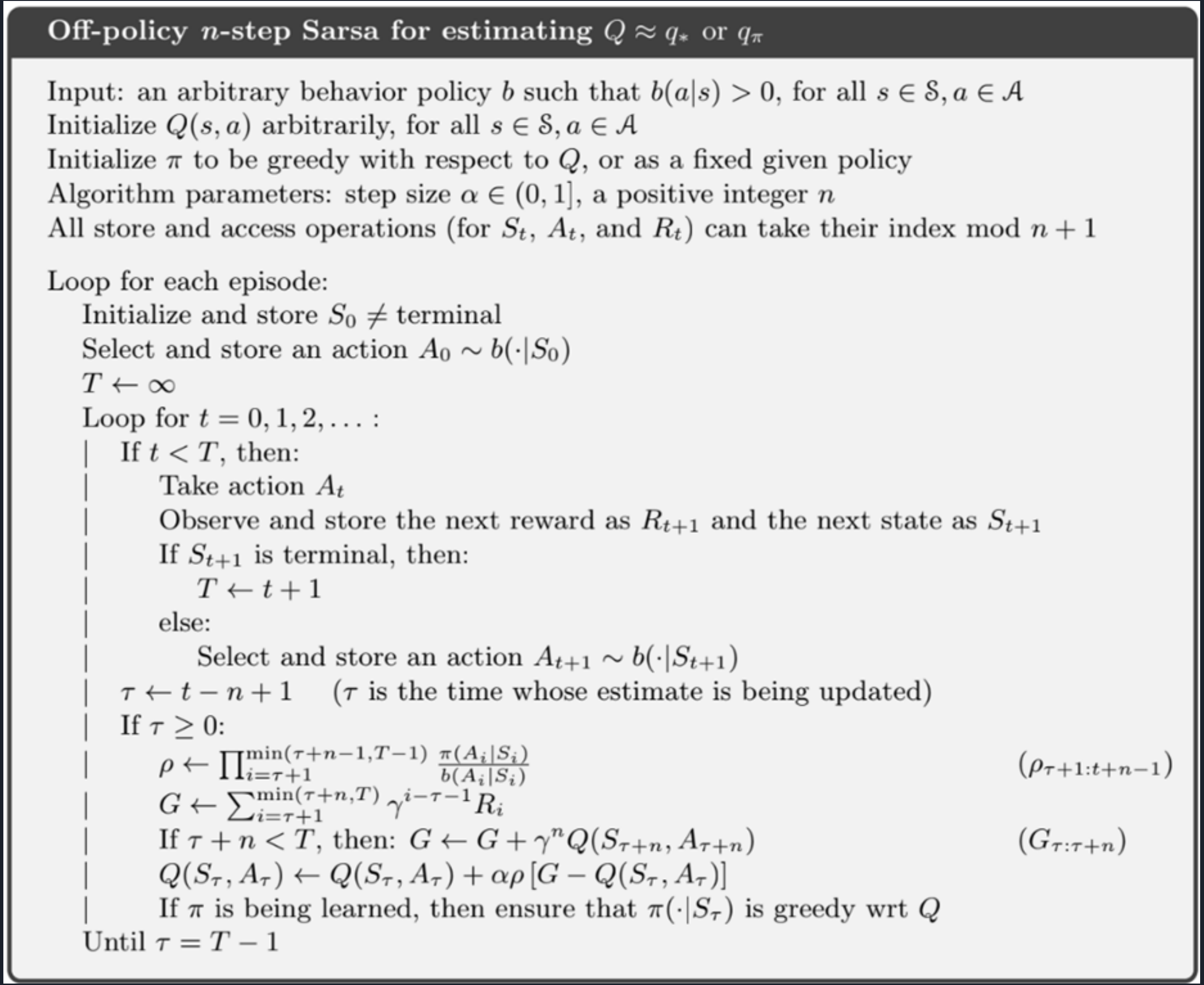
n-step on-policy learning



n-step Sarsa for estimating $Q \approx q_*$ or q_π

```
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $\pi$ to be $\varepsilon$-greedy with respect to $Q$, or to a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer $n$
All store and access operations (for $S_t$, $A_t$, and $R_t$) can take their index mod $n+1$

Loop for each episode:
    Initialize and store $S_0 \neq$ terminal
    Select and store an action $A_0 \sim \pi(\cdot|S_0)$
    $T \leftarrow \infty$
    Loop for $t = 0, 1, 2, \dots$ :
    |   If $t < T$, then:
    |       Take action $A_t$
    |       Observe and store the next reward as $R_{t+1}$ and the next state as $S_{t+1}$
    |       If $S_{t+1}$ is terminal, then:
    |           $T \leftarrow t+1$
    |       else:
    |           Select and store an action $A_{t+1} \sim \pi(\cdot|S_{t+1})$
    |   $\tau \leftarrow t - n + 1$    ($\tau$ is the time whose estimate is being updated)
    |   If $\tau \geq 0$:
    |       $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
    |       If $\tau + n < T$, then $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$        $(G_{\tau:\tau+n})$
    |       $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha [G - Q(S_\tau, A_\tau)]$
    |       If $\pi$ is being learned, then ensure that $\pi(\cdot|S_\tau)$ is $\varepsilon$-greedy wrt $Q$
    Until $\tau = T - 1$
```

n-step off-policy learning



Off-policy n-step Sarsa for estimating $Q \approx q_*$ or q_π

```
Input: an arbitrary behavior policy $b$ such that $b(a|s) > 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $\pi$ to be greedy with respect to $Q$, or as a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, a positive integer $n$
All store and access operations (for $S_t$, $A_t$, and $R_t$) can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim b(\cdot|S_0)$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \dots$ :
  |  If $t < T$, then:
  |      Take action $A_t$
  |      Observe and store the next reward as $R_{t+1}$ and the next state as $S_{t+1}$
  |      If $S_{t+1}$ is terminal, then:
  |          $T \leftarrow t+1$
  |      else:
  |          Select and store an action $A_{t+1} \sim b(\cdot|S_{t+1})$
  |  $\tau \leftarrow t - n + 1$    ($\tau$ is the time whose estimate is being updated)
  |  If $\tau \geq 0$:
  |      $\rho \leftarrow \prod_{i=\tau+1}^{\min(\tau+n-1,T-1)} \frac{\pi(A_i|S_i)}{b(A_i|S_i)}$        $(\rho_{\tau+1:t+n-1})$
  |      $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  |      If $\tau + n < T$, then: $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$        $(G_{\tau:\tau+n})$
  |      $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\rho\,[G - Q(S_\tau, A_\tau)]$
  |      If $\pi$ is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt $Q$
  Until $\tau = T - 1$
```

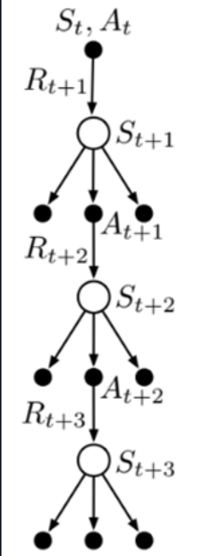
Per-decision Methods with Control Variates

The basic idea behind this technique is to use a control variate to reduce the variance of the estimated value function, and hence improve the learning efficiency.

The control variate is a known function that is correlated with the value function, but has lower variance. In per-decision methods with control variates, the control variate is used to adjust the estimate of the value function at each decision point.

The n-step Tree Backup Algorithm

$$G_{t:t+2} \doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+1}(S_{t+1}, a) \\ + \gamma \pi(A_{t+1}|S_{t+1}) \Big(R_{t+2} + \gamma \sum_{a} \pi(a|S_{t+2}) Q_{t+1}(S_{t+2}, a) \Big)$$



the 3-step tree-backup update

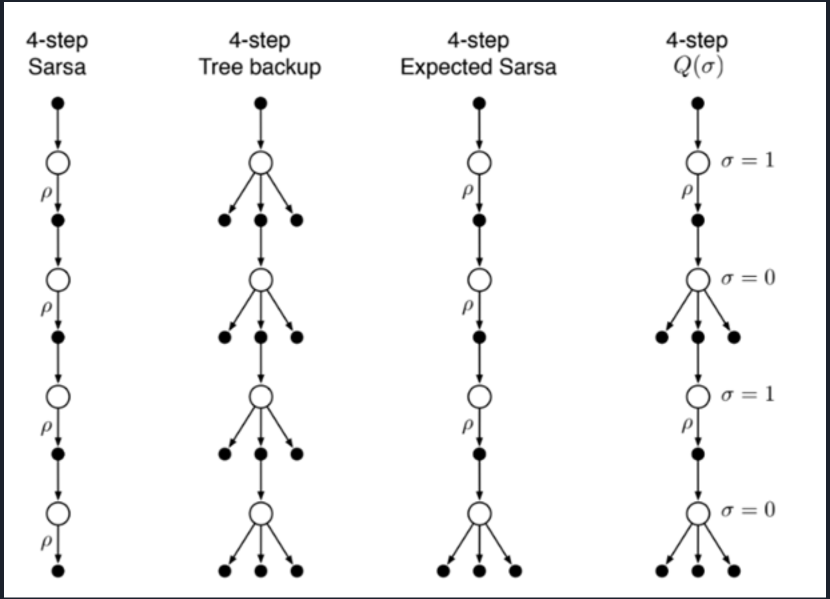
*A Unifying Algorithm:

One idea for unification is suggested by the fourth backup diagram in Figure 7.5. This

is the idea that one might decide on a step-by-step basis whether one wanted to take the

action as a sample, as in Sarsa, or consider the expectation over all actions instead, as in

the tree-backup update.



Approximate methods

In many of the tasks to which we would like to apply reinforcement learning the state space is combinatorial and enormous; the number of possible camera images, for example, is much larger than the number of atoms in the universe. In such cases we cannot expect to find an optimal policy or the optimal value function even in the limit of infinite time and data; our goal instead is to find a good approximate solution using limited computational resources.

The Prediction Objective (VE)

explicit objective for prediction: mean squared value error

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

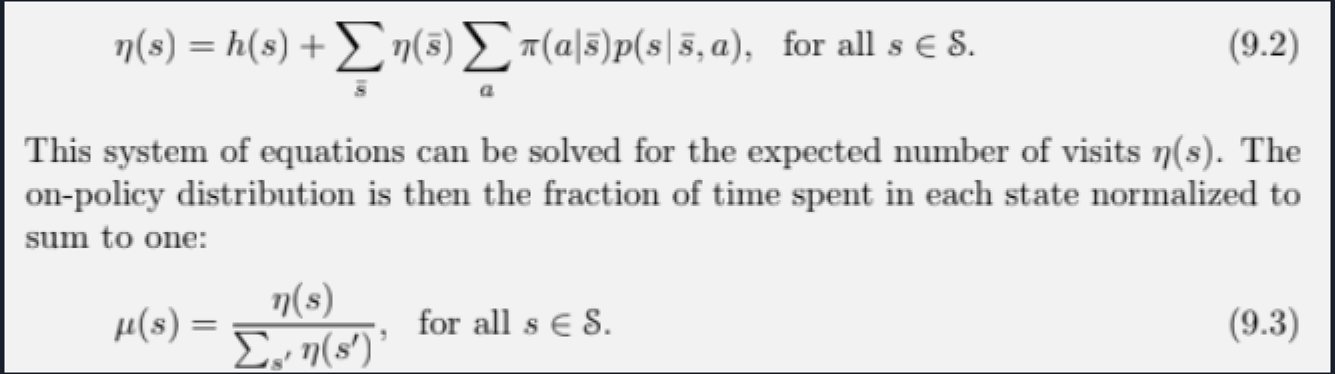
specify most important states (by assumption we have less weights than states)

$$\text{state distribution } \mu(s) \geq 0, \ \textstyle\sum_s \mu(s) = 1$$

Often μ(s) is

chosen to be the fraction of time spent in s.

state distribution



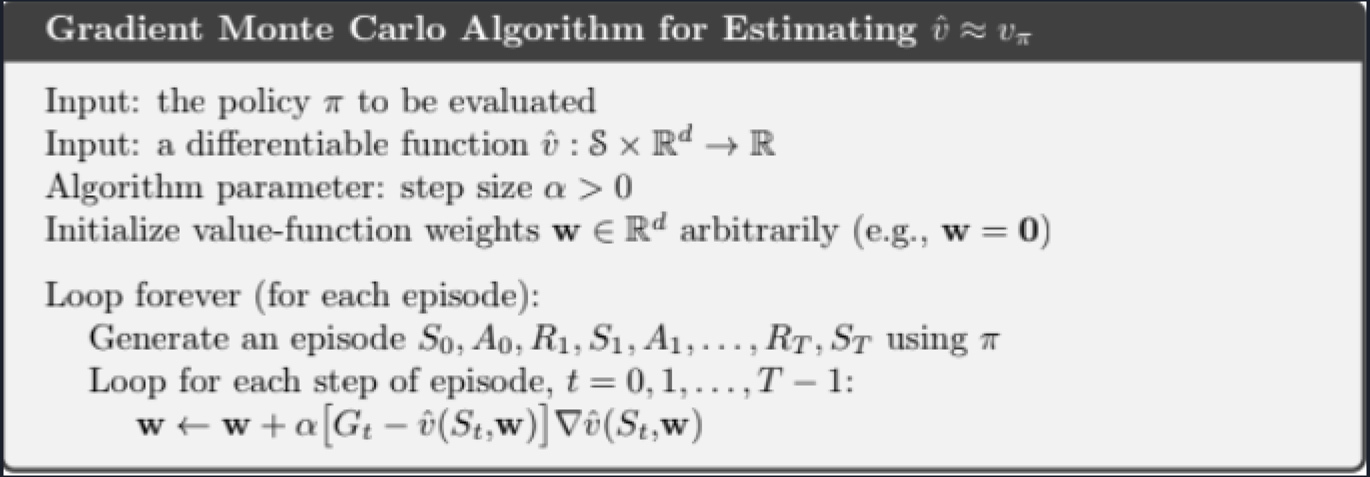
$$\eta(s) = h(s) + \sum_{\bar{s}} \eta(\bar{s}) \sum_a \pi(a|\bar{s}) p(s \,|\, \bar{s}, a), \quad \text{for all } s \in \mathcal{S}. \tag{9.2}$$

This system of equations can be solved for the expected number of visits $\eta(s)$. The on-policy distribution is then the fraction of time spent in each state normalized to sum to one:

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')}, \quad \text{for all } s \in \mathcal{S}. \tag{9.3}$$

Stochastic-gradient and Semi-gradient Methods

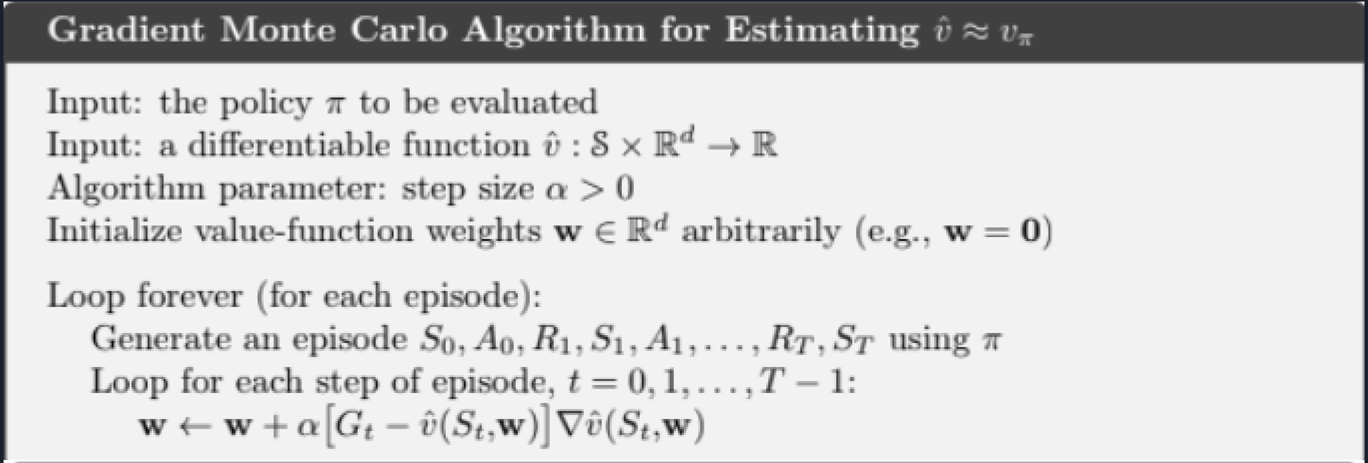
$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t - \frac{1}{2}\alpha\nabla\Big[v_\pi(S_t) - \hat{v}(S_t,\mathbf{w}_t)\Big]^2 \\ &= \mathbf{w}_t + \alpha\Big[v_\pi(S_t) - \hat{v}(S_t,\mathbf{w}_t)\Big]\nabla\hat{v}(S_t,\mathbf{w}_t),\end{aligned}$$



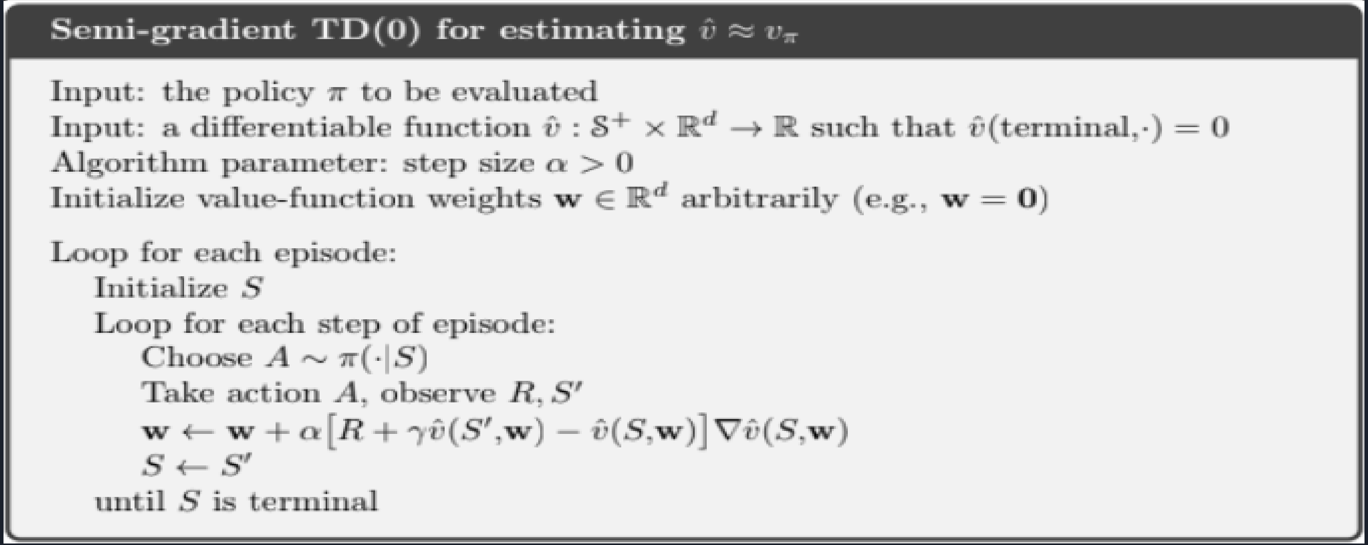
Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

```
Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
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 $S_0, A_0, R_1, S_1, A_1, \dots, R_T, S_T$ using $\pi$
    Loop for each step of episode, $t = 0, 1, \dots, T-1$:
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$
```

U_t is an *unbiased* estimate, that is, if $\mathbb{E}[U_t|S_t = s] = v_\pi(S_t)$



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

Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
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 $S_0, A_0, R_1, S_1, A_1, \dots, R_T, S_T$ using $\pi$
    Loop for each step of episode, $t = 0, 1, \dots, T-1$:
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$
```

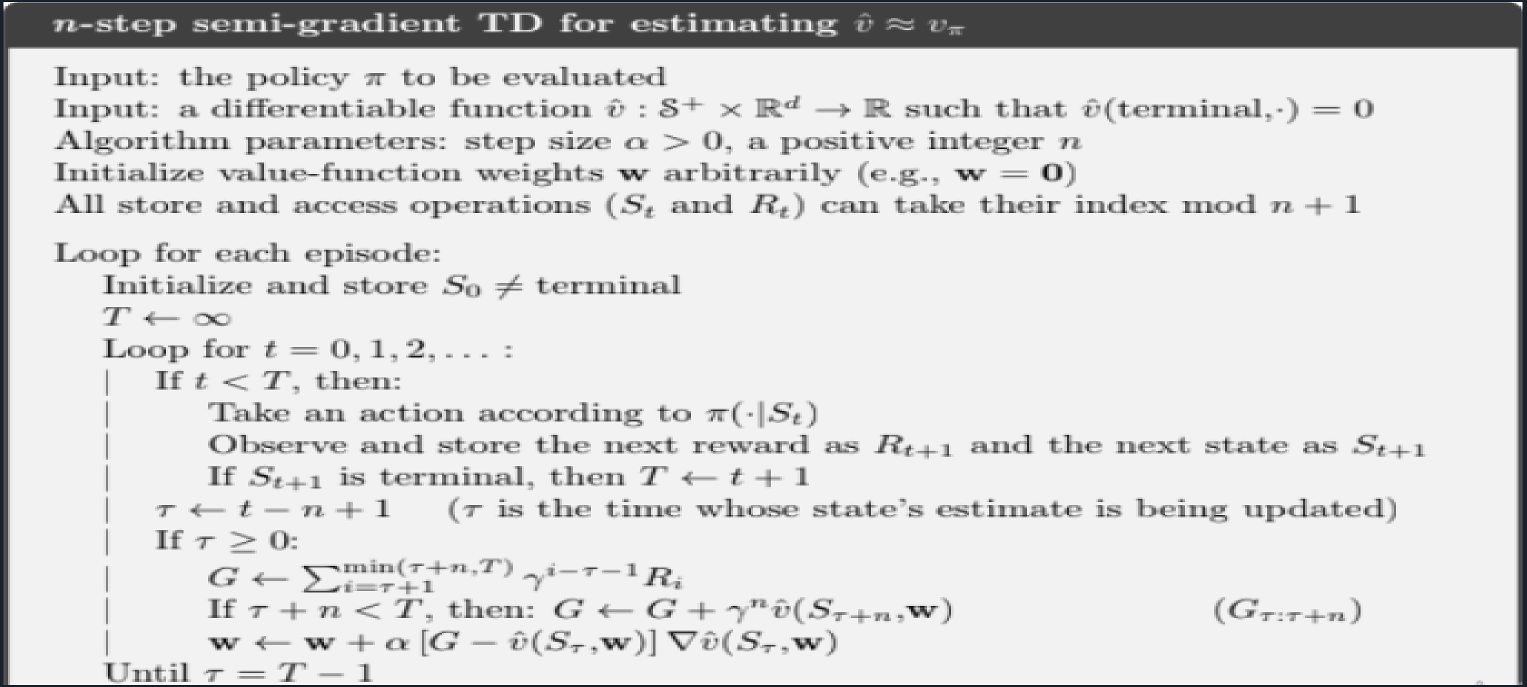


```
Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

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
```

Linear Methods

$$\hat{v}(s,\mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s) \doteq \sum_{i=1}^{d} w_i x_i(s)$$

an approximation of the gradient



n-step semi-gradient TD 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 parameters: step size $\alpha > 0$, a positive integer $n$
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations ($S_t$ and $R_t$) can take their index mod $n + 1$

Loop for each episode:
    Initialize and store $S_0 \neq$ terminal
    $T \leftarrow \infty$
    Loop for $t = 0, 1, 2, \ldots$ :
    |   If $t < T$, then:
    |       Take an action according to $\pi(\cdot|S_t)$
    |       Observe and store the next reward as $R_{t+1}$ and the next state as $S_{t+1}$
    |       If $S_{t+1}$ is terminal, then $T \leftarrow t + 1$
    |   $\tau \leftarrow t - n + 1$    ($\tau$ is the time whose state's estimate is being updated)
    |   If $\tau \geq 0$:
    |       $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
    |       If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n},\mathbf{w})$                ($G_{\tau:\tau+n}$)
    |       $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[G - \hat{v}(S_\tau,\mathbf{w})\right] \nabla \hat{v}(S_\tau,\mathbf{w})$
    Until $\tau = T - 1$
```