Part I: Elementary Reinforcement Learning

0. Introduction to Reinforcement Learning (RL)

Definitions

- **Markov**:
    - a Markov process or markov chain is a memoryless random process, i.e. a sequence of random states $S_1, S_2, ...$ with the Markov property.
    - a Markov Reward Process (MRP) is a Markov chain with values. It is a tuple $< S, P, R, \gamma >$
    - A Markov Decision Process (MDP) is a MRP with decisions. It is an environment where all states are Markov.

Process	Tuple
Markov Process / Chain	<S,P>
Markov Reward Process (MRP)	<S,P,R, γ>
Markov Decision Process (MDP)	<S,A,P,R,γ>
Partially Observable MDP	<S,A,O,P,R,Z,γ>

with:

- S is a finite set of states
- A is a finite set of actions
- O is a finite set of observations
- P is a state transition probability matrix, $P^a_{ss'} = \mathbb{P}[S_{t+1} = s'/S_t = s, A_t = a]$
- R is a reward function, $R^a_s = \mathbb{E}[R_{t+1}/S_t = s, A_t = a]$
- Z is an observation function, $Z^a_{s'o} = \mathbb{P}[O_{t+1} = o|S_{t+1} = s, A_t = a]$
- γ is a discount factor $\gamma \in$ [0, 1]

- **Rewards**:
    - a scalar feedback signal R_t. It indicates how well agent is doing at step t. The agent's job is to maximize cumulative rewards.
    - Reward hypothesis: all goals can be described by the maximization of expected cumulative rewards. Reinforcement learning is based on the reward hypothesis.
- **History**: sequence of observations, actions, rewards $H_t = A_1, O_1, R_1, \dots, A_t, O_t, R_t$
- **Agent** selects the actions and the **environment** selects observations / rewards
- Formally, **the state** is a function of the history - $S_t = f(H_t)$. Many states exist:
    - environement state S_t^e : the environment's private representation. Usually not visible to the agent.
    - agent's state S_t^a : the agent's internal representation
    - a state S_t is **Markov** if and only if $P(S_{t+1}/S_t) = P(S_{t+1}/S_1, S_2, .., S_t)$, i.e. history does not matter
- A **policy** is the agent's behavior, a map from state to action.
    - Deterministic policy: $a = \pi(s)$
    - Stochastic policy: $\pi(a/s) = \mathbb{P}[A_t = a/S_t = s]$ = a probability distribution over actions given states.
- on-policy learning: learn about policy π from experience sampled from policy π
- off-policy learning: learn about policy π f from experience sampled from another policy μ
- ϵ-greedy methods/policy consist in behaving greedily (picking the action with highest value) most of the time but select randomly with a probability ϵ from **among all the actions (including the highest value)** with equal probability (independently of the action-value estimates).
    - $\pi(a/s) = \epsilon/m + 1 - \epsilon$ if $a* = \underset{a \in A}{argmax} Q(s, a)$ Note that the probability of picking the highest value is higher than $1 - \epsilon$ as it can still be picked randomly.
    - $\pi(a/s) = \epsilon/m$ otherwise
- a **model** predicts what the environment will do next: transition from state to state and prediction of the next immediate reward. We say that the agent has an internal representation of the environment, called the model.

- Prediction problems: evaluate the future **given a policy**
- Control problems: optimize the future by **finding the best policy**

1. Sequential Decision Problems

Sequential decision-making problems have the following characteristics:

- **Goal**: select actions to maximise total future rewards
- **Actions**: may have long term consequences
- **Reward**: may be delayed - immediate reward vs. long-term reward trade-off

Two fundamental problems in sequential decision making

- **Reinforcement Learning**:
  - The environment is initially unknown
  - The agent interacts with the environment
  - The agent improves its policy
- **Planning:**
  - A model of the environment is known
  - The agent performs computations with its model (without any external interaction)
  - The agent improves its policy
  - a.k.a. deliberation, reasoning, introspection, pondering, thought, search

A sequential decision problem for a fully observable, stochastic environment with a Markovian transition model and additive rewards is called a **Markov decision process**, or **MDP**, and consists of:

- a set of states (with an initial state s_0);
- a set ACTIONS(s) of actions in each state A(s), A;
- a transition model T (s,a,s') ~ P (s' | s, a);
- a reward function R(s).
- $\gamma \in [0, 1]$ a discount factor

Therefore, a solution must specify what the agent should do for any state that the agent might reach. **A solution of this kind is called a policy**.

It is traditional to denote a policy by π, and $\pi(s)$ is the action recommended by the policy π for state s. If the agent has a complete policy, then no matter what the outcome of any action, the agent will always know what to do next.

An optimal policy is a policy that yields the highest expected utility. We use π_* to denote an optimal policy. Given π*, the agent decides what to do by consulting its current percept, which tells it the current state s, and then executing the action π*(s)

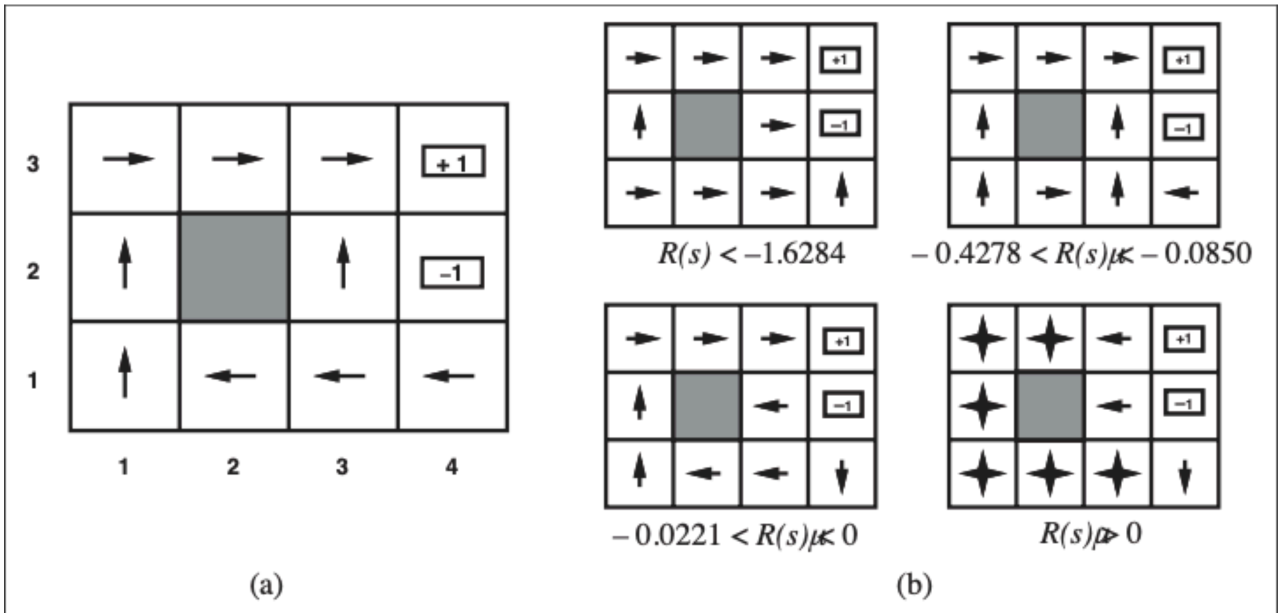


Figure 17.2 (a) An optimal policy for the stochastic environment with $R(s) = -0.04$ in the nonterminal states. (b) Optimal policies for four different ranges of $R(s)$.

Note: a policy does not describe a plan but tells us what to do next in a given state. We can still infer a plan based on a policy.

Utilities over time:

- the optimal policy for a finite horizon is non-stationary, i.e. the optimal action in a given state could change over time
- the optimal policy for a infinite horizon is stationary (no reason to behave differently)

We call stationary preferences the fact that, given 2 utilities of sequences V1 and V2, $V_1 = \sum_{t=0}^{\infty} U(S_t)$ and $V_2 = S_0 + \sum_{t=1}^{\infty} U(S_t')$: if V1 > V2, then $\sum_{t=1}^{\infty} U(S_t) > \sum_{t=1}^{\infty} U(S_t')$

Under stationarity, there are 2 coherent ways to assign utilities to sequences (of states):

- Additive rewards: $U(s_0, s_1, s_2 \ldots) = \sum_{t=0}^{\infty} R(S_t)$
- Discounted rewards $U(s_0, s_1, s_2 \ldots) = \sum_{t=0}^{\infty} \gamma^t R(S_t) <= \sum_{t=0}^{\infty} \gamma^t R_{max} = \frac{R_{max}}{1-\gamma}$

Fundamental principles / Formulas

Utility (U), State value (V), State-Action value (Q)

The optimal policy is the policy that maximizes the expected future rewards (by following the action at each state, returned by the policy)

$$\pi^* = \underset{\pi}{argmax}(\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R(S_t)/\pi])$$

The value (or true utility) of a state given a policy is the expected rewards we'll get by following the policy

$$U^\pi(s) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R(s_t)/\pi, s_0 = s]$$

For each state, the optimal policy returns the action that maximizes the weighted value of the next state

$$\pi^*(s) = \underset{a}{argmax} \sum_{s'} T(s, a, s')U^{\pi^*}(s')$$

The value or true utility of a state is the immediate reward and the discounted future rewards (**Bellman equation**)

$$U(s) = R(s) + \gamma . \underset{a}{max} \sum_{s'} T(s, a, s') . U(s')$$

It can also be written as:

- a **value** function V of a state (or **state-value** function) Formally, the state-value function of an MDP is the expected returm starting from state s, and then following policy π

$$v_\pi(s) = \mathbb{E}[G_t/S_t = s)$$

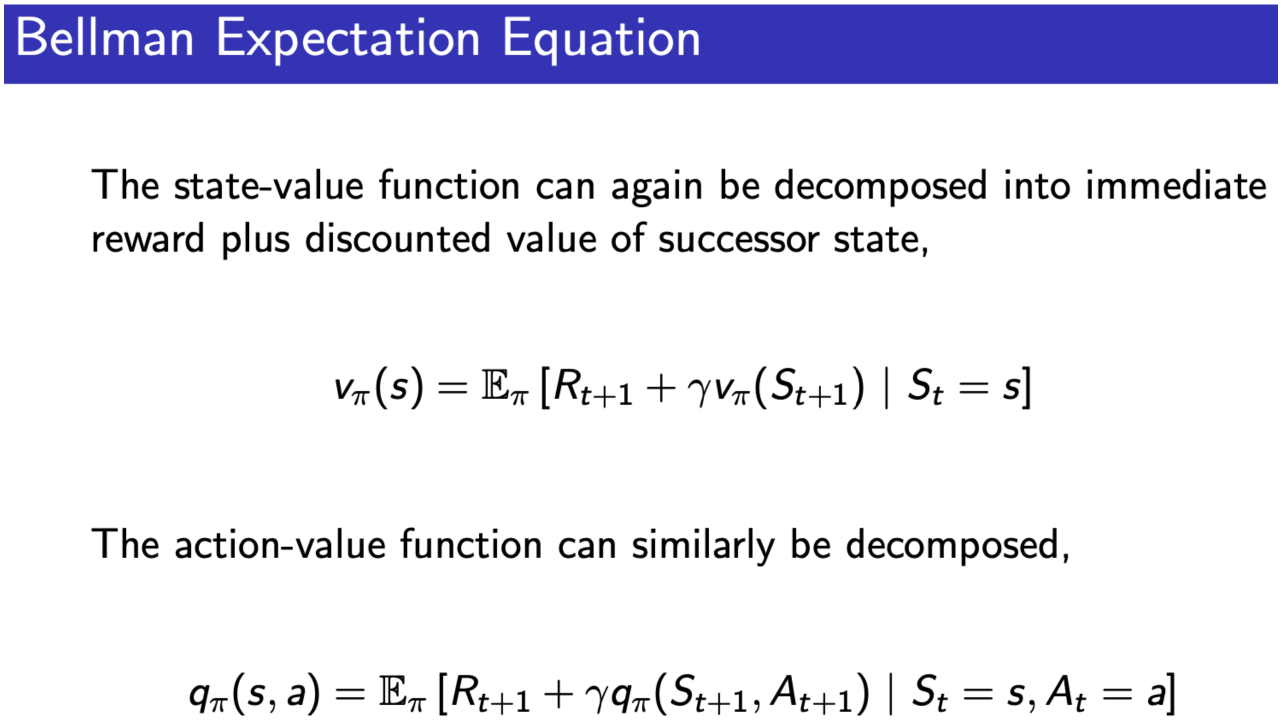
- a **quality** function Q (value of an action) or **action-value** function Formally, the action-value function of an MDP is the expected returm starting from state s, taking action a, and then following policy π

$$q_\pi(s, a) = \mathbb{E}[G_t/S_t = s, A_t = a)$$

- a **continuity** function

1. Bellmann Equations (MDPs only)

Bellmann Expectation Equation



Bellman Expectation Equation

The state-value function can again be decomposed into immediate reward plus discounted value of successor state,

$$v_\pi(s) = \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s \right]$$

The action-value function can similarly be decomposed,

$$q_\pi(s,a) = \mathbb{E}_\pi \left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a \right]$$

$$v_\pi(s) = \sum_{a \in A} \pi(a/s) q_\pi(s,a)$$

$$q_\pi(s,a) = R_s^a + \gamma . \sum_{s' \in S} P(s'/s,a) v_\pi(s')$$

By unrolling the equations above, we obtain:

$$v_\pi(s) = \sum_{a \in A} \pi(a/s) . [R_s^a + \gamma . \sum_{s' \in S} P(s'/s,a) v_\pi(s')]$$

$$q_\pi(s,a) = R_s^a + \gamma . \sum_{s' \in S} P(s'/s,a) . \sum_{a' \in A} \pi(a'/s') q_\pi(s',a')$$

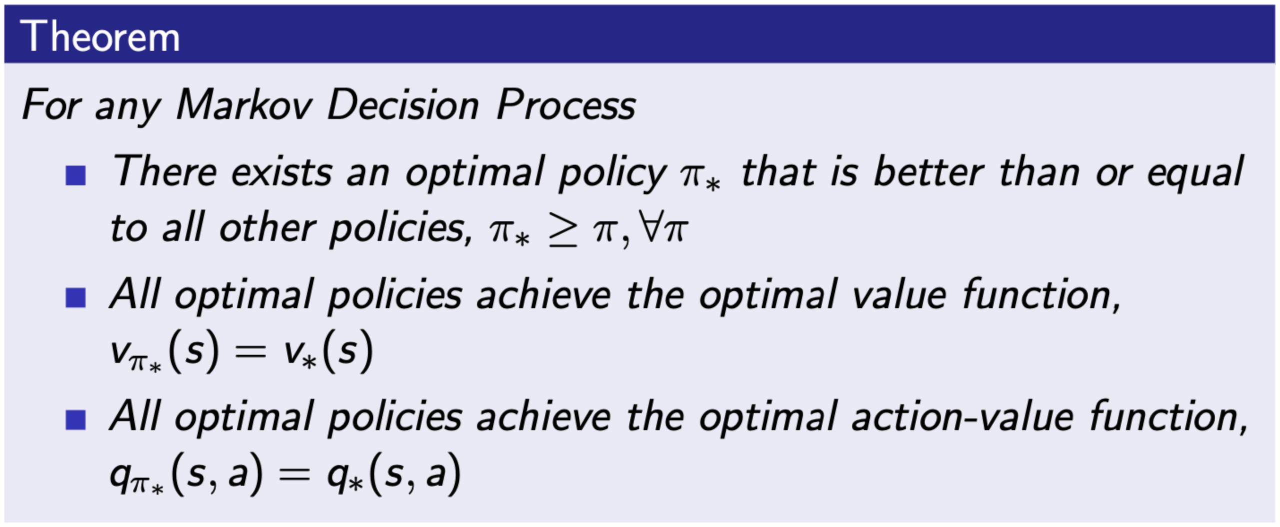
Bellmann Optimality Equation

The optimal state-value function $v_*(s)$ is the maximum value function over all policies
$v_*(s) = \max_\pi v_\pi(s)$

The optimal action-value function $q_*(s,a)$ is the maximum value function over all policies $q_*(s,a) = \max_\pi q_\pi(s,a)$

A MDP is "solved" when we know the optimal value function.

Optimal Policy



Theorem

For any Markov Decision Process

- *There exists an optimal policy π_* that is better than or equal to all other policies, $\pi_* \geq \pi, \forall \pi$*
- *All optimal policies achieve the optimal value function, $v_{\pi_*}(s) = v_*(s)$*
- *All optimal policies achieve the optimal action-value function, $q_{\pi_*}(s,a) = q_*(s,a)$*

Note that there can be more than 1 optimal policy. They all get the same optimal value functions.

An optimal policy can be found by optimizing over $q_*(s,a)$

$\pi_*(a/s)$**= 1** if $a = argmax_{a \in A} q_*(s,a)$

$\pi_*(a/s)$**= 0** otherwise

There is always a deterministic optimal policy for any MDP.

$$v_*(s) = \max_a q_*(s,a)$$

$$q_*(s,a) = R_s^a + \gamma . \sum_{s' \in S} P(s'/s,a) v_*(s')$$

By unrolling the equations above, we obtain:

$$v_*(s) = \max_a [R_s^a + \gamma . \sum_{s' \in S} P(s'/s,a) v_*(s')$$

]

$$q_*(s,a) = R_s^a + \gamma . \sum_{s' \in S} P(s'/s,a) \max_{a'} q_*(s',a')$$

Bellman Optimality Equation is non-linear. No closed form solution (in general) Many iterative solution methods exist:

- Value Iteration (DP)
- Policy Iteration (DP)
- Q-learning (off-policy TD control)
- Sarsa (on-policy TD control)

We usually use Bellmann expectation equation to do policy evaluation and the Belllmann optimality equation for control.

Typology of RL algorithms

From the most to the least supervised and from the least to the most direct learning

- Model-based: from , learns the transition model and calculates Q* through a MDP solver and then a policy
- Value-function based or model-free: from , directly learns the Q*
- policy search: from , directly learns the policy

2. Planning by Dynamic Programming (DP)

The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP). **Classical DP algorithms are of limited utility in reinforcement learning** both because of their assumption of a **perfect model** and because of their great **computational expense**, but they are still important theoretically. More useful methods attempt to achieve much the same effect as DP, only with less computation and without assuming a perfect model of the environment.

Dynamic programming is a very general solution method for problems with 2 properties:

- Optimal substructure
  - Principle of optimality
  - Optimal solution can be decomposed into subproblems
- Overlapping subproblems
  - Subproblems recur many times
  - Solutions can be cached and reused

MDP satisfy both properties:

- Bellman equation gives recursive decomposition
- Value function stores and reuses solutions

Policy evaluation

Evaluating a policy consists in the iterative application of Bellmann expectation equation.

Synchronous backups: at each iteration k+1, for all states, update $v_{k+1}(s)$ from $v_k(s')$ where s' is a successor state of s.

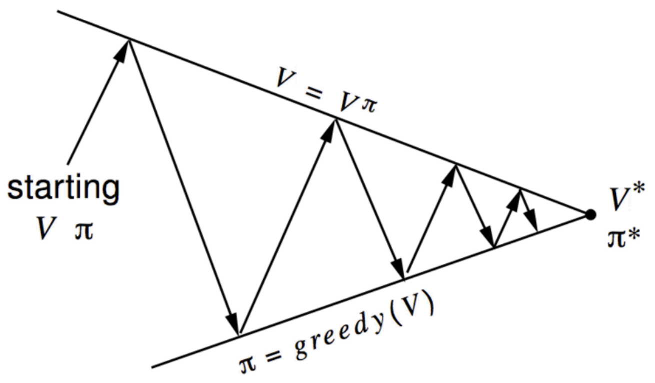
Policy iteration

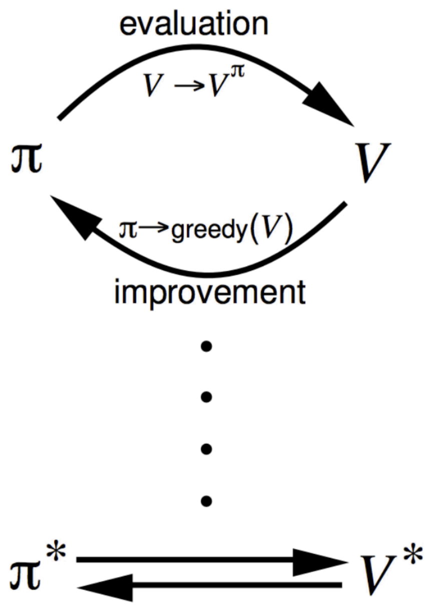
Given a policy π,

- evaluate the policy π (until it converges)
- improve the policy by acting greedily with respect to v_π

This process of policy iteration always converges to $\pi *$

Policy Iteration





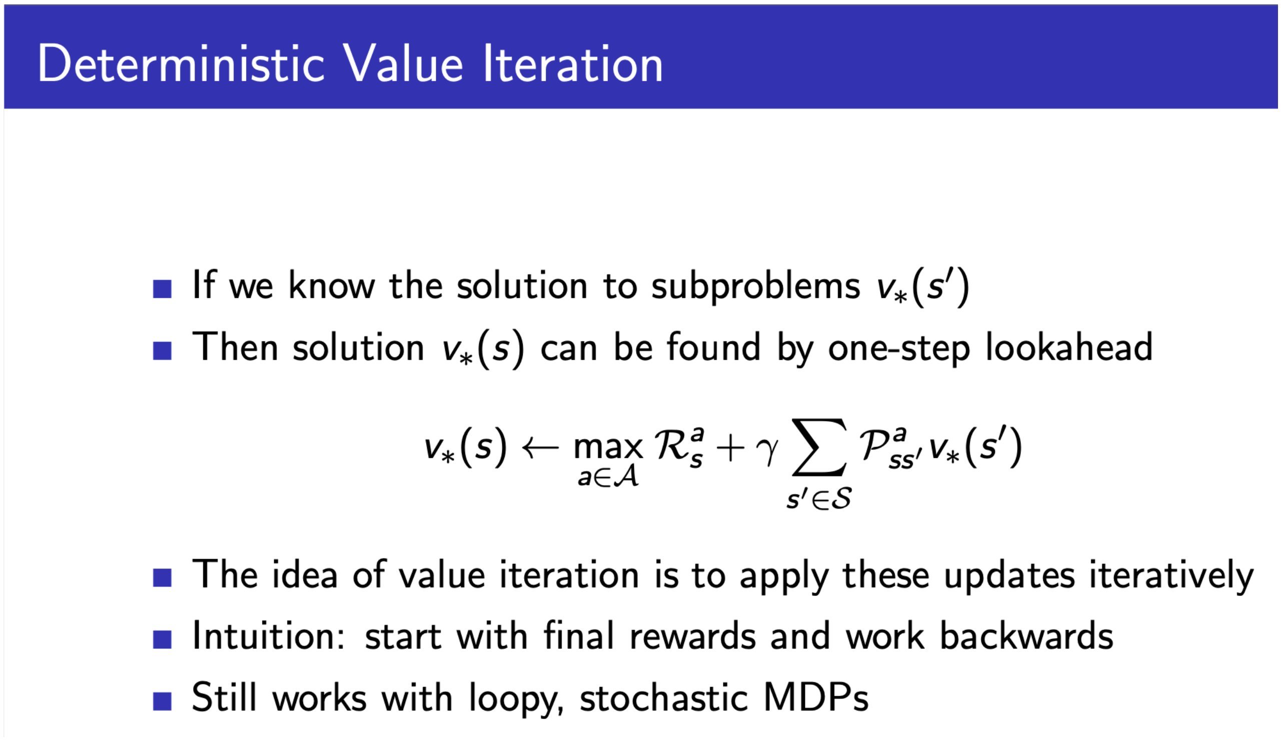
Policy evaluation Estimate v_π
Iterative policy evaluation

Policy improvement Generate $\pi' \geq \pi$
Greedy policy improvement

The generalised policy iteration principle is that any algorithm can be used for policy evaluation and policy improvement.

Modified policy iteration: stop the iteration before convergence. For example, stop after k iterations of policy evaluation. Value iteration consists in stopping after 1 iteration, i.e 1 iteration of policy evaluation, 1 iteration of policy improvement, 1 iteration of policy evaluation...

Value iteration



Deterministic Value Iteration

- If we know the solution to subproblems $v_*(s')$
- Then solution $v_*(s)$ can be found by one-step lookahead

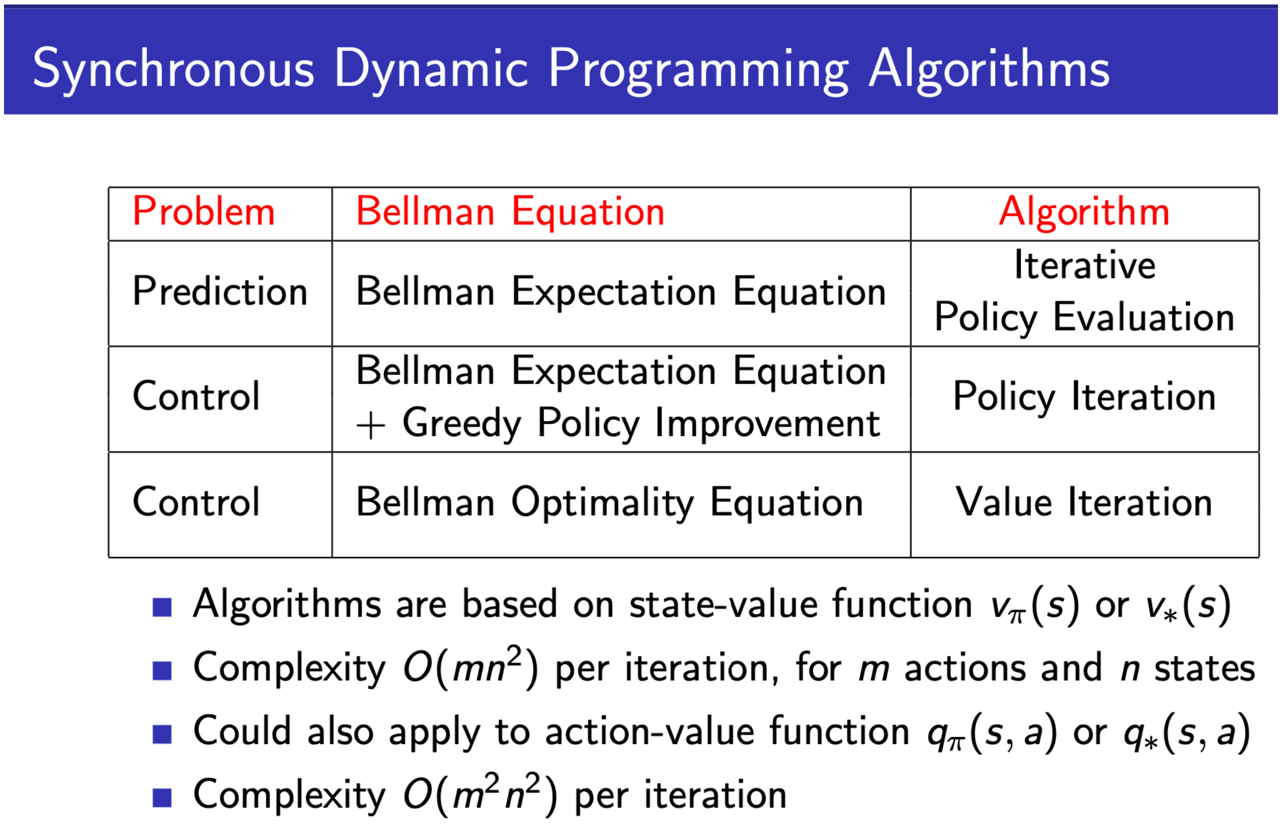
$$v_*(s) \leftarrow \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

- The idea of value iteration is to apply these updates iteratively
- Intuition: start with final rewards and work backwards
- Still works with loopy, stochastic MDPs

The iteration consists in 2 steps:

- calculating the value function for each action in order to extract the best action.
- selecting the maximum [acting greedily]

Intermediate value functions may not correspond to any policy.



Synchronous Dynamic Programming Algorithms

Problem	Bellman Equation	Algorithm
Prediction	Bellman Expectation Equation	Iterative Policy Evaluation
Control	Bellman Expectation Equation + Greedy Policy Improvement	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

- Algorithms are based on state-value function $v_\pi(s)$ or $v_*(s)$
- Complexity $O(mn^2)$ per iteration, for m actions and n states
- Could also apply to action-value function $q_\pi(s,a)$ or $q_*(s,a)$
- Complexity $O(m^2n^2)$ per iteration

Prediction addresses the question of how good is a given policy Control optimizes the rewards and finds the optimal policy. Complexity is higher if we use the Q-value function because we move from state-action pair to state-action pair (nm x nm) whereas with V-value, we move from state to state and consider all the actions (n x m x m)

Asynchronous Dynamic programming

Asynchronous means we update each selected state individually, in any order. If all states continue to be selected, it will converge.

Several methods exist:

- In-place DP (it stores only 1 value function vs. 2 for synchronous)
- Prioritised sweeping (Use magnitude of Bellman error to guide state selection)
- Real-time DP (only states that are relevant to agent)

Convergence

- Iterative policy evaluation converges to v_π - we also say that v_π converges to a unique fixed point of the Bellmann expectation operator T^π

- Iterative policy iteration and iterative value iteration converge to v_π

3. Model-Free Prediction

Dynamic programming is useful to solve a (small) known MDP. Model-Free prediction methods aim at solving unknown MDPs.

Monte-Carlo (MC)

- MC methods learn directly from episodes of experience
- MC is model-free: no knowledge of MDP transitions / rewards
- MC learns from complete episodes: no bootstrapping
- MC uses the simplest possible idea: value = mean return
- Caveat: can only apply MC to episodic MDPs (all episodes must terminate)

Monte-Carlo policy evaluation uses empirical mean return instead of expected return.

V(S) = S(s) / N(s) with S(s)+=G_t , the increment total return for state s and N(s)+=1 the increment counter. V(s) -> $v_\pi(s)$ as N(s) $\rightarrow\ \infty$

- First-visit MC Policy Evaluation: mean return calculated with only 1 visit per episod
- Every-visit MC Policy Evaluation: mean return calculated with several visits per episod

Incremental MC updates consists in tracking a running mean (forget old episodes).

Incremental mean : $\mu_k = \frac{1}{k}\sum_{j=1}^{k} x_k = \mu_{k-1} + \frac{1}{k}.(x_k - \mu_{k-1})$

In general, **new_estimate<-- old_estimate + step_size [target - old_estimate]**

Temporal Difference Learning ($TD(\lambda)$)

TD methods learn directly from episodes of experiences. TD learns from incomplete episodes (contrary to MC that requires full episodes), **by bootstrapping**. TD updates a guess towards a guess.

MC has high variance (because of the high variance of the return), zero bias (unbiased estimate of v_π(s))

- Good convergence properties
- (even with function approximation) Not very sensitive to initial value Very simple to understand and use
- MC does not exploit Markov property (Usually more effective in non-Markov environments)

- MC converges to solution with minimum mean-squared error, i.e. best fit to the observed returns $\sum_{k=1}^{k} \sum_{t=1}^{T_k} (G_t^k - V(s_t^k))^2$

TD has low variance (because only dependent on the variance on one random action, transition, reward), some bias (TD target is biased estimate of $v_\pi(s)$

- Usually more efficient than MC
- TD(0) converges to $v_\pi(s)$
- (but not always with function approximation)
- More sensitive to initial value
- TD exploits Markov property (Usually more efficient in Markov environments)
- TD(0) converges to solution of max likelihood Markov model
    - Solution to the MDP ⟨S, A, P, R, γ⟩ that best fits the data

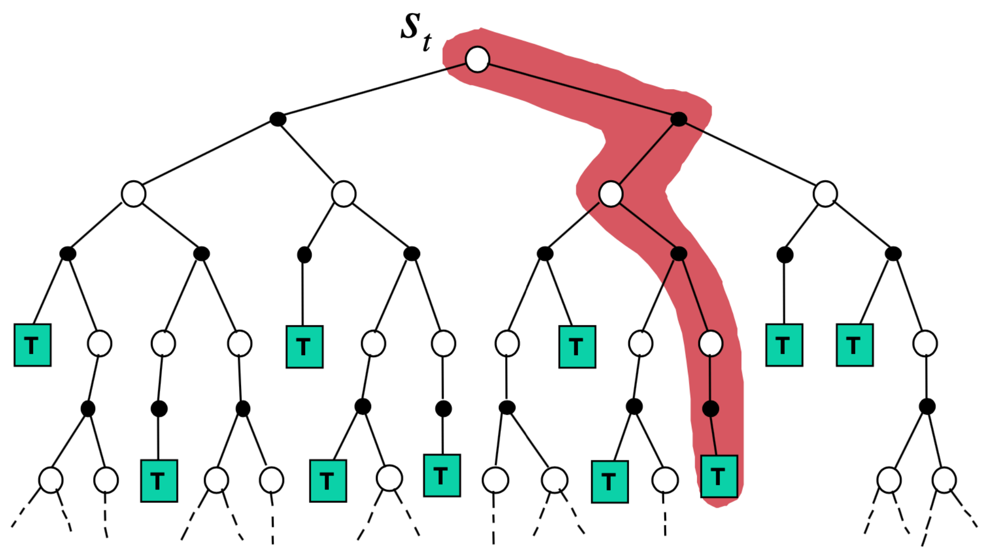
Backup comparisons

Method	Bootstrapping	Sampling
Dynamic Programming	Yes	No
Monte Carlo	No	Yes
Temporal Difference	Yes	Yes

Visually, boostrapping = depth of the tree; sampling = width of the tree

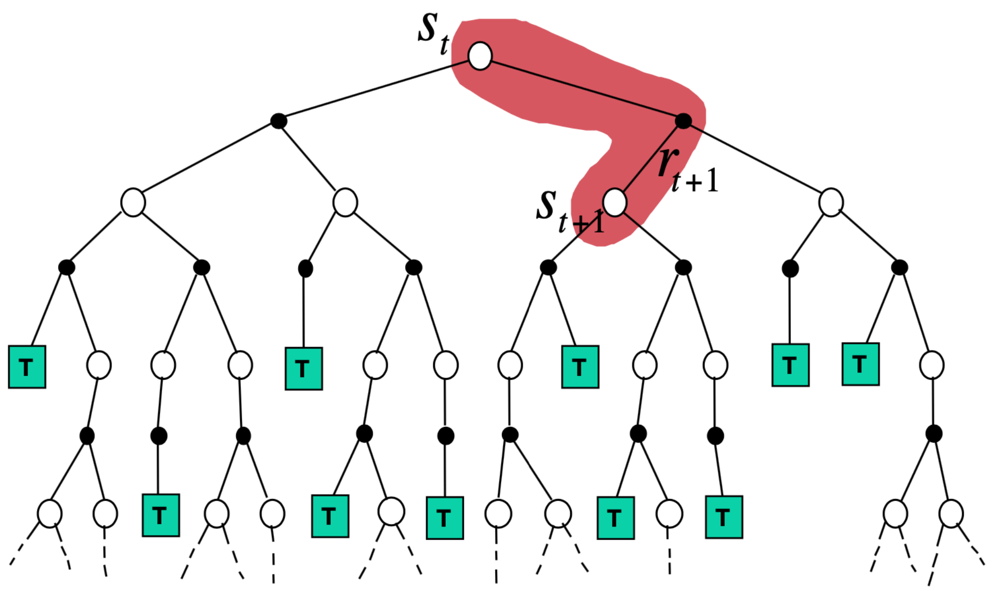
Monte-Carlo Backup

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



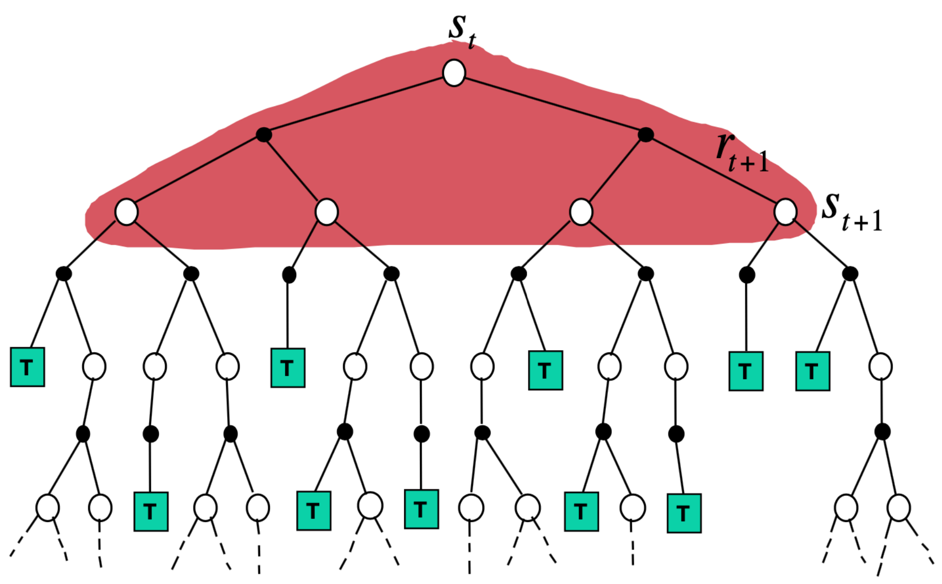
Temporal-Difference Backup

$$V(S_t) \leftarrow V(S_t) + \alpha \left(R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right)$$



Dynamic Programming Backup

$$V(S_t) \leftarrow \mathbb{E}_\pi \left[R_{t+1} + \gamma V(S_{t+1})\right]$$



TD(0) Update Rule

Finds maximum-likelihood estimate (if finite data repeated infinitely often)

Algorithm:

Episode T: For all s, at start of episode, $V_T(s) = V_{T-1}(s)$

For all s = s_k,

$V_T(s_{k-1}) = V_T(s_{k-1}) + \alpha_T.(r_k + \gamma.V_T(s_k) - V_T(s_{k-1}))$

then: $V_T(s_{k-1}) = \mathbb{E}_{s_k}[r + \gamma.V_T(s_k)]$

Properties of learning rates

$\lim_{t->\infty} V_T(s) = V(s)$ if 2 conditions about α_T are met (Robbins-Monro sequence of step-sizes):

- $$\sum_T \alpha_T = \infty$$

- $$\sum_T \alpha_T^2 < \infty$$

TD(1) Update Rule

We assume:

- $s_1, s_2, \dots, s_f$ states with reward r_k between state s_k and s_{k+1}
- learning rate γ
- e(s) is called the eligibility rate

Algorithm:

Episode T

For all s, e(s) = 0 at start of episode, $V_T(s) = V_{T-1}(s)$

After $S_{k-1} - > S_k$, reward r_k: (Step k)

- $e(s_{k-1}) = e(s_{k-1}) + 1$

For all s,

- $V_T(s) = V_T(s) + \alpha_T(r_T + \gamma V_{T-1}(s_T) - V_{T-1}(s_{T-1})). e(s)$
- For all k: $e(s) = \gamma . e(s)$

TD(1) is equivalent to outcome-based updates without repeated states with $V_T(s_k)$ += $\alpha . (\sum_{i=k}^{f-1} \gamma^{i-1} . r_i + \gamma^f . V_{T-1}(S_f) - V_{T-1}(S_k))$

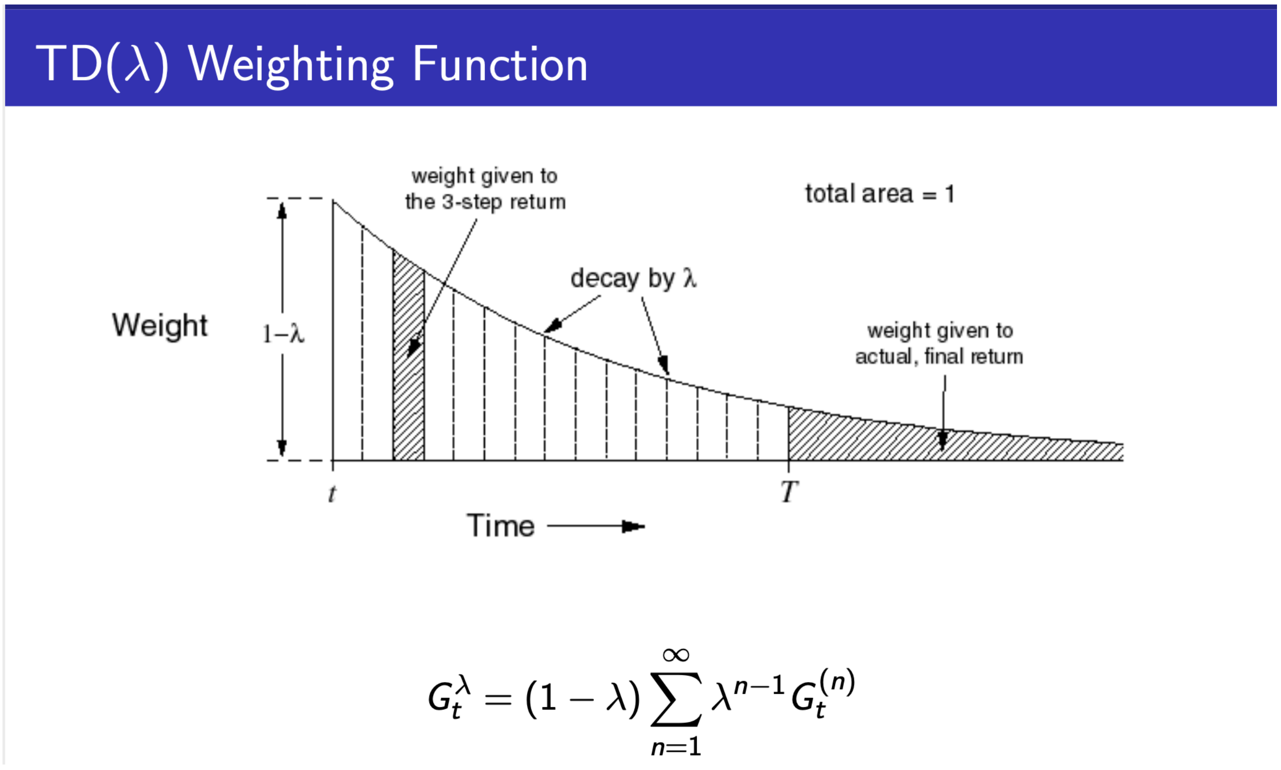
Generalization: TD(λ) Update rule

Forward-view implementation

The principle is to combine different returns with n-step look-ahead, $G_t^{(n)}$ through a weighted sum. The weight is $(1 - \lambda). \lambda^{n-1}$

$$G_t^\lambda = (1 - \lambda). \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

Forward-view $TD(\lambda)$: $V(S_t) \leftarrow V(S_t) + \alpha . (G_t^\lambda - V(S_t))$


TD(λ) Weighting Function
weight given to the 3-step return
total area = 1
Weight
1−λ
decay by λ
weight given to actual, final return
t
T
Time

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^{(n)}$$

Backward view implementation (eligibility traces)

Episode T

For all s, e(s) = 0 at start of episode, $V_T(s) = V_{T-1}(s)$

After $S_{k-1} - > S_k$, reward r_k: (Step k)

- $e(s_{k-1}) = e(s_{k-1}) + 1$

For all s,

- $V_T(s) = V_T(s) + \alpha_T(r_T + \gamma V_{T-1}(s_T) - V_{T-1}(s_{T-1})).e(s)$
- For all k: $e(s) = \lambda.\gamma.e(s)$

K-Step Estimators

- $V(s_t) = V(S_t) + \alpha_t.(r_{t+1} + \gamma.V(S_{t+1}) - V(S_t))$ = TD(0) [1-step estimator]
- $V(s_t) = V(S_t) + \alpha_t.(r_{t+1} + \gamma.r_{t+2} + \gamma^2.V(S_{t+2}) - V(S_t))$ = [2-step estimator]
- ...
- $V(s_t) = V(S_t) + \alpha_t.(r_{t+1} + \gamma.r_{t+2} + ... + \gamma^{k-1}.r_{t+k} + \gamma^k.V(s_{t+k}) - V(S_t))$ = [k-step estimator]
- $V(s_t) = V(S_t) + \alpha_t.(r_{t+1} + \gamma.r_{t+2} + ... + \gamma^{k-1}.r_{t+k} + ... - V(S_t))$ = TD(1) = [infinite-step

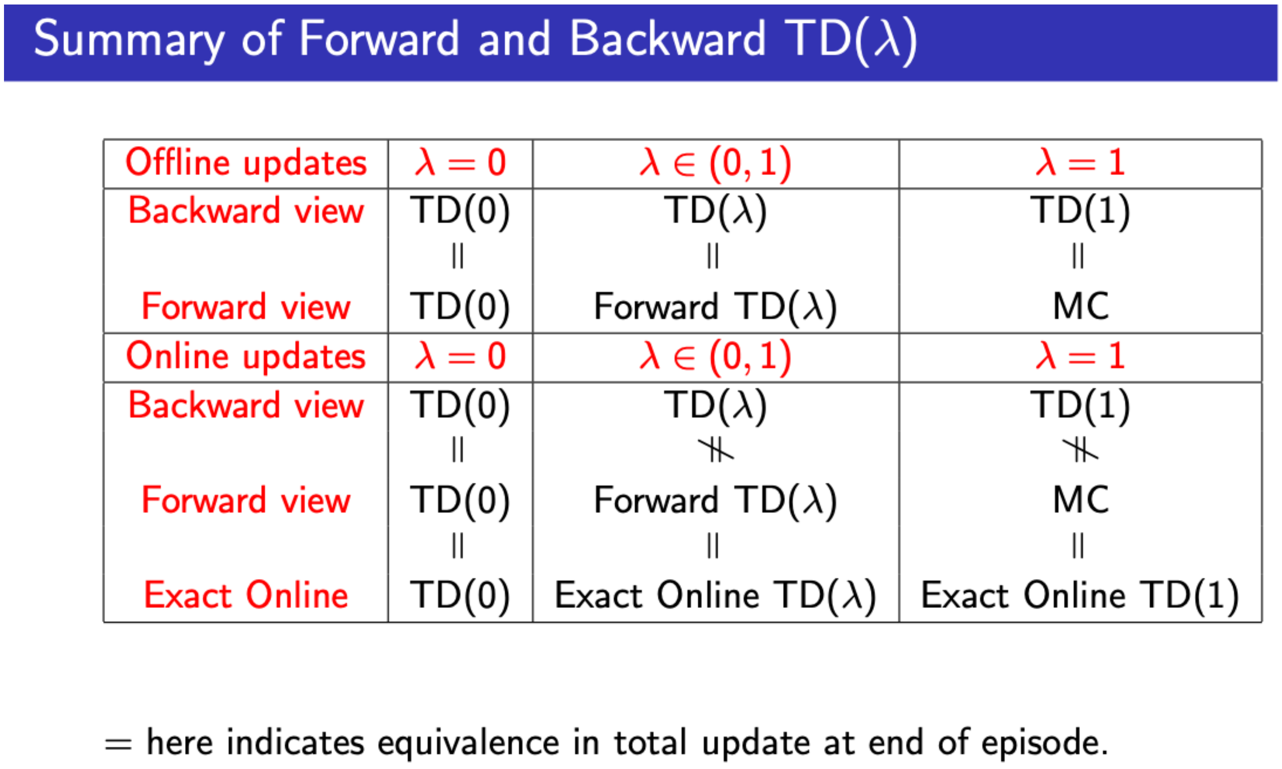
estimator]

K-Step Estimators and $TD(\lambda)$

V(S) estimated as a weighted combination of step estimators, with weights = $\lambda^{t-1}(1-\lambda)$

Empirically, we can observe that error in V after finite data is:

- highest for TD(1) but lower for TD(0)
- lowest for λ between 0.3 and 0.7



Summary of Forward and Backward TD(λ)

Offline updates	$\lambda = 0$	$\lambda \in (0,1)$	$\lambda = 1$
Backward view	TD(0)	TD(λ)	TD(1)
	$\parallel$	$\parallel$	$\parallel$
Forward view	TD(0)	Forward TD(λ)	MC
Online updates	$\lambda = 0$	$\lambda \in (0,1)$	$\lambda = 1$
Backward view	TD(0)	TD(λ)	TD(1)
	$\parallel$	$\nparallel$	$\nparallel$
Forward view	TD(0)	Forward TD(λ)	MC
	$\parallel$	$\parallel$	$\parallel$
Exact Online	TD(0)	Exact Online TD(λ)	Exact Online TD(1)

$=$ here indicates equivalence in total update at end of episode.

4. Model-Free Control

For most of RL problems, either:

- MDP model is unknown but experience can be sampled
- MDP model is known, but is too big to use, except by samples

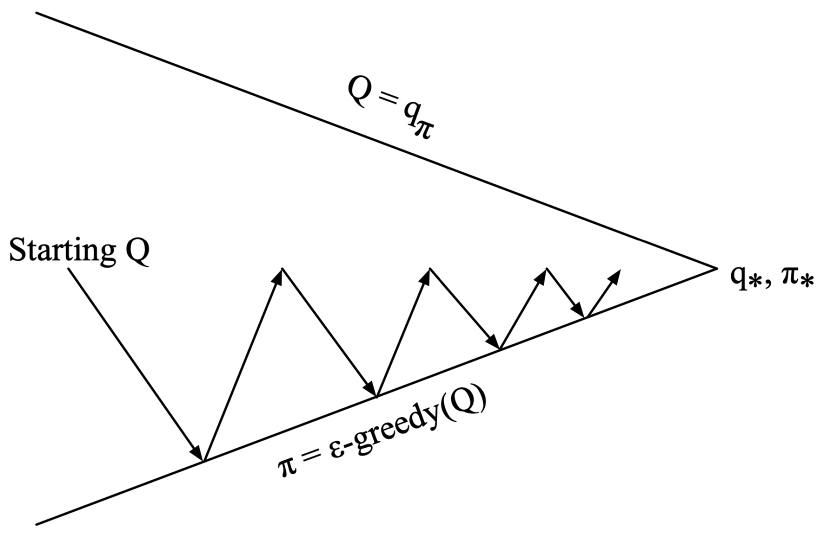
On-policy learning describes learning about policy π from experience sampled from π, unlike **off-policy learning** which learns from experience sampled from another policy μ.

On-policy learning

Generalisated policy iteration (GPI) with Monte-Carlo evaluation: 2 issues:

- We cannot use the greedy policy improvement over V(s) because it requires a model of MDP $\pi'(s) = \underset{a \in A}{argmax}[R_s^a + P_{ss'}^a V(s')]$; So we'll use Q(s,a) instead => $\pi'(s) = \underset{a \in A}{argmax}\ Q(s, a)$
- We need to maintain some level of exploration. So we'll use a ϵ-greedy policy improvement approach.

Monte-Carlo Control



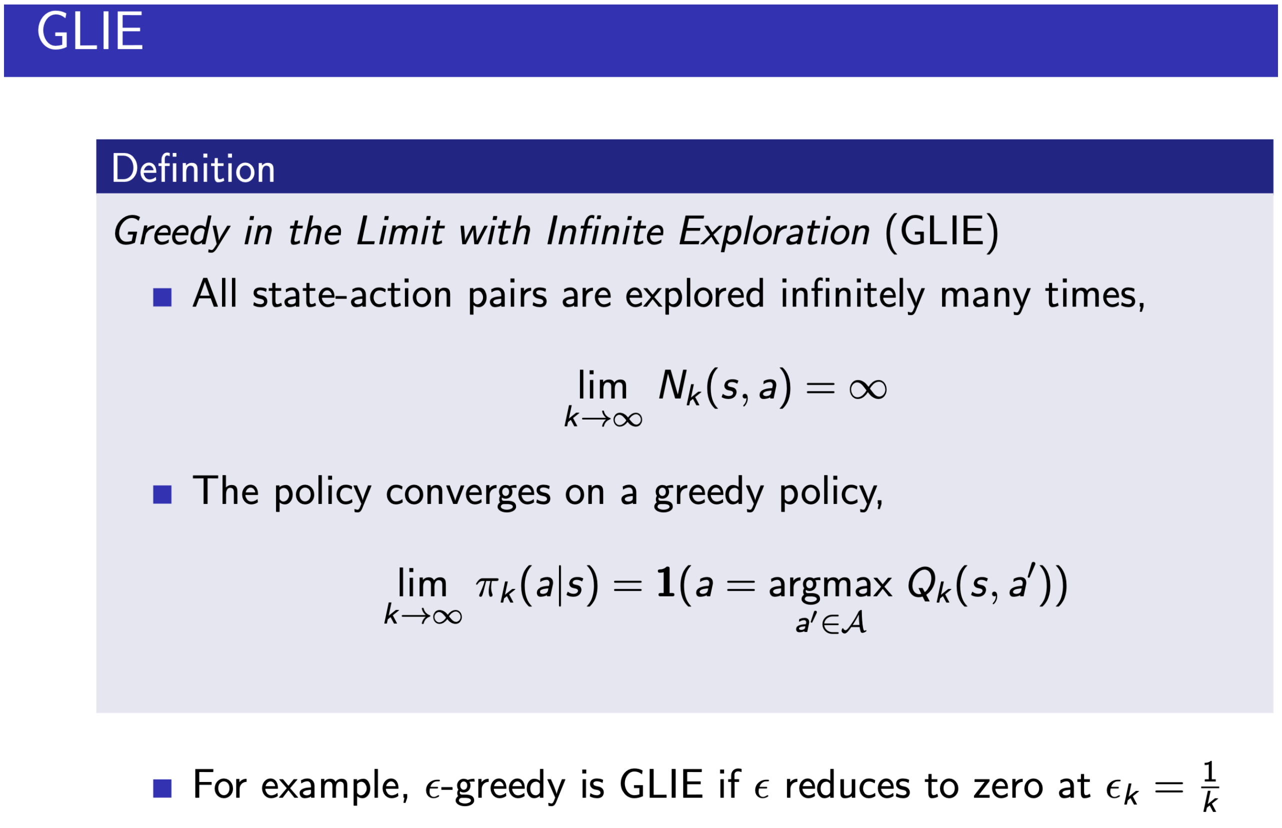
Every episode:

Policy evaluation Monte-Carlo policy evaluation, $Q \approx q_\pi$

Policy improvement ϵ-greedy policy improvement

The question of this approach is how to balance the need for exploration and the fast convergence into a policy that does not explore anymore. GLIE is an idea to come up with a schedule of exploration to manage this balance (e.g. decaying ϵ by 1/k).

GLIE Monte-Carlo Control



GLIE

Definition

Greedy in the Limit with Infinite Exploration (GLIE)

- All state-action pairs are explored infinitely many times,

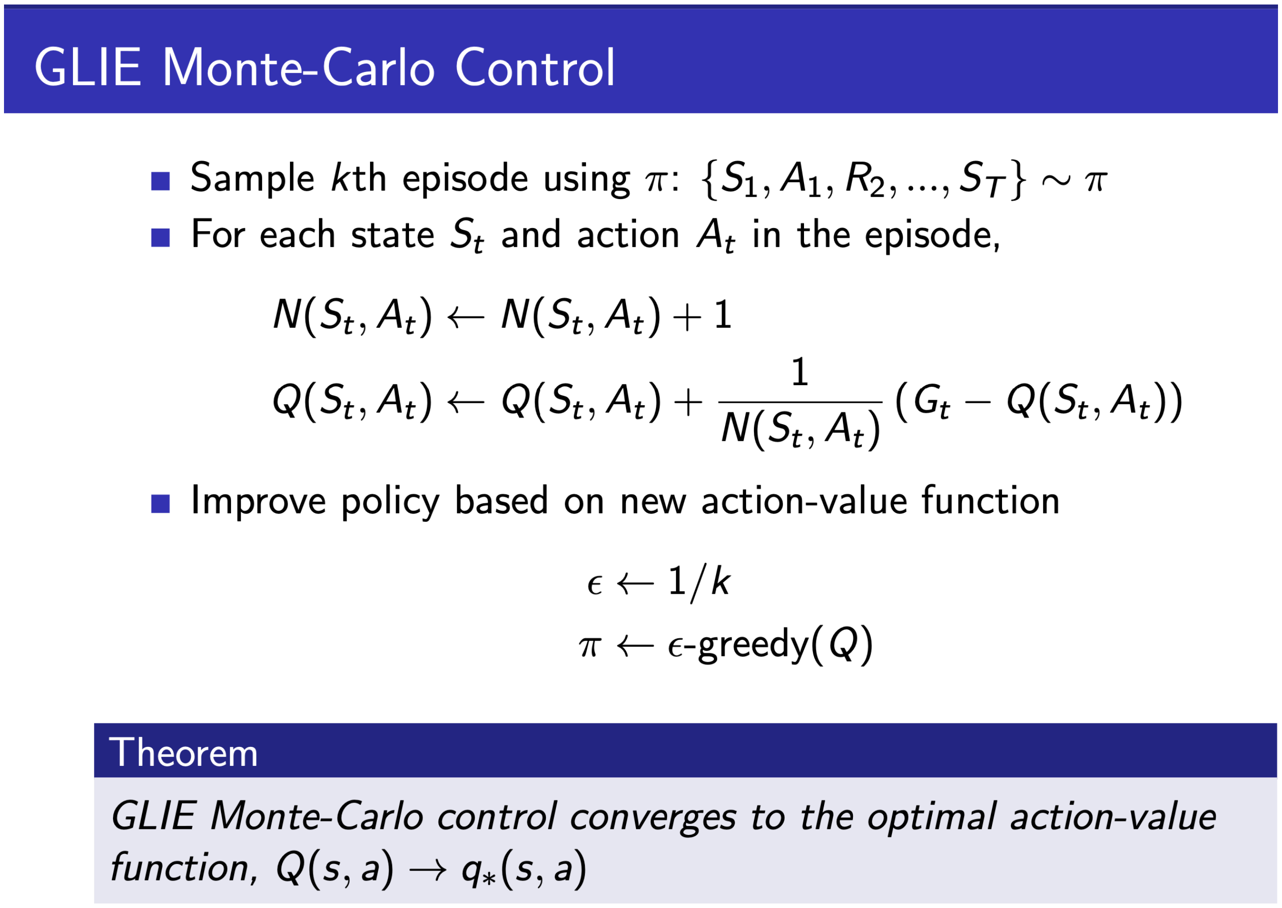
$$\lim_{k\to\infty} N_k(s,a) = \infty$$

- The policy converges on a greedy policy,

$$\lim_{k\to\infty} \pi_k(a|s) = \mathbf{1}(a = \operatorname*{argmax}_{a'\in\mathcal{A}} Q_k(s,a'))$$

- For example, ϵ-greedy is GLIE if ϵ reduces to zero at $\epsilon_k = \frac{1}{k}$

Here's below one algorithm where we iterate, for every episode, evaluation and improvement - note this is much more efficient than generating thousands of episodes before evaluating the policy (i.e. we can even improve the policy with a single episode):



GLIE Monte-Carlo Control

- Sample kth episode using π: $\{S_1, A_1, R_2, ..., S_T\} \sim \pi$
- For each state S_t and action A_t in the episode,

$$\begin{aligned} N(S_t, A_t) &\leftarrow N(S_t, A_t) + 1 \\ Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)} \left(G_t - Q(S_t, A_t)\right) \end{aligned}$$

- Improve policy based on new action-value function

$$\begin{aligned} \epsilon &\leftarrow 1/k \\ \pi &\leftarrow \epsilon\text{-greedy}(Q) \end{aligned}$$

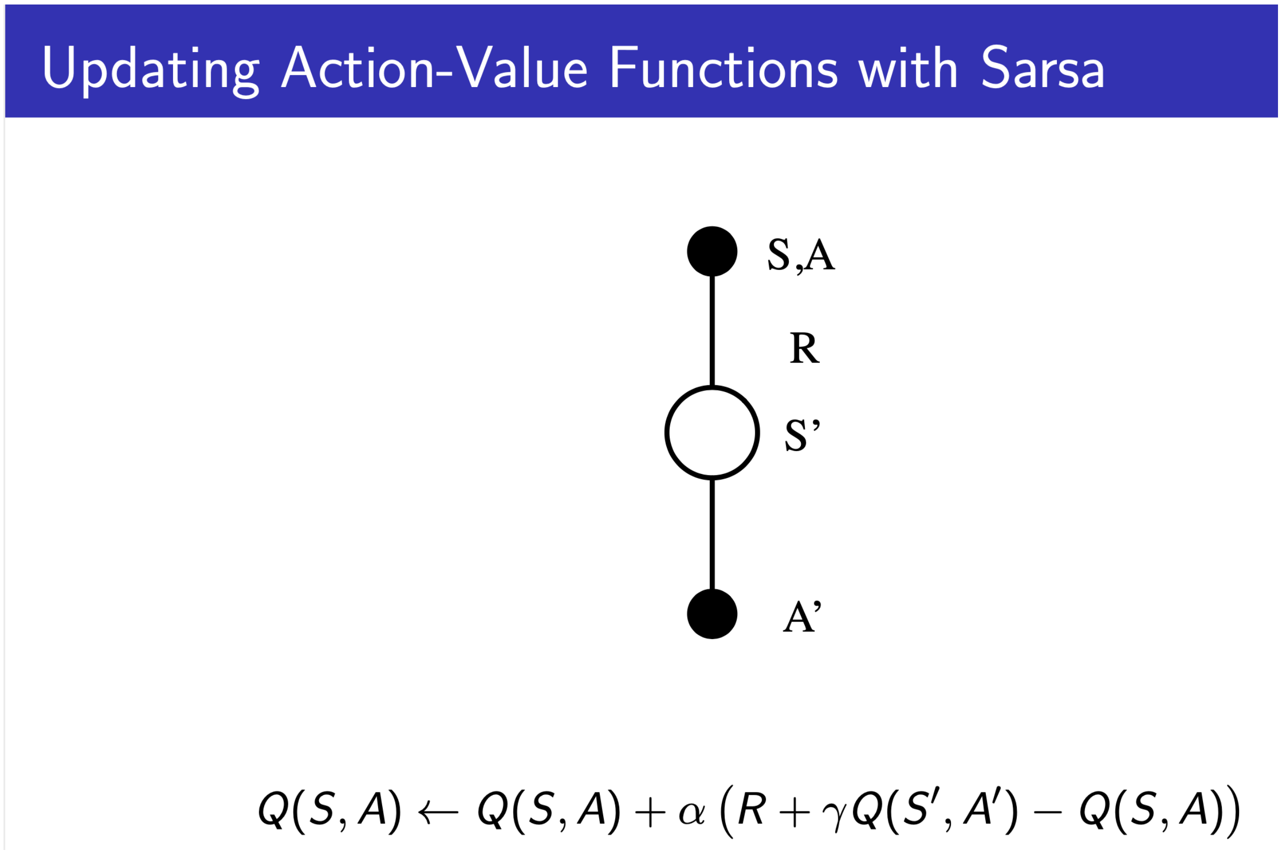
Theorem

GLIE Monte-Carlo control converges to the optimal action-value function, $Q(s, a) \rightarrow q_*(s, a)$

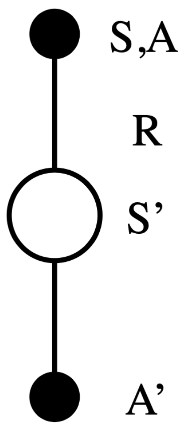
TD learning

The main advantage vs. MC is to imcrease the frequency of the policy improvement - will be done every time-step, instead of at the end of each episode in MC.

The general idea is called SARSA. SARSA name came from the fact that agent takes one step from one state-action value pair to another state-action value pair and along the way collect reward R (so it's the S_t, A_t, R_{t+1}, S_{t+1} and A_{t+1} tuple that creates the term S,A,R,S,A). SARSA is an **on-policy method**. SARSA use action-value function Q and follow the policy π. GPI (Generalized Policy Iteration) is used to take action based on policy π (ϵ-greedy to ensure exploration as well as greedy to improve the policy).



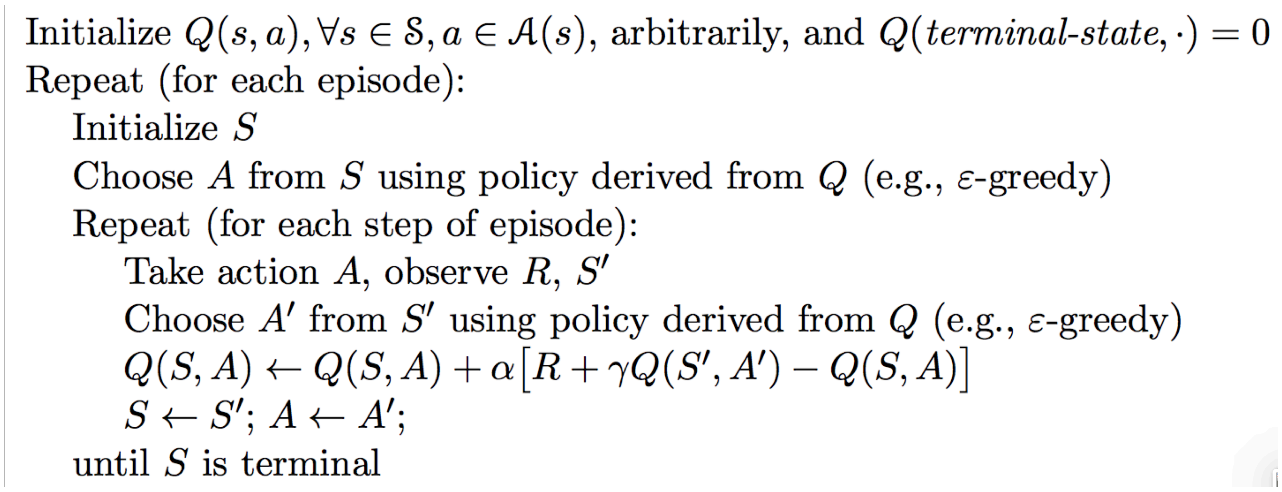
Updating Action-Value Functions with Sarsa



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

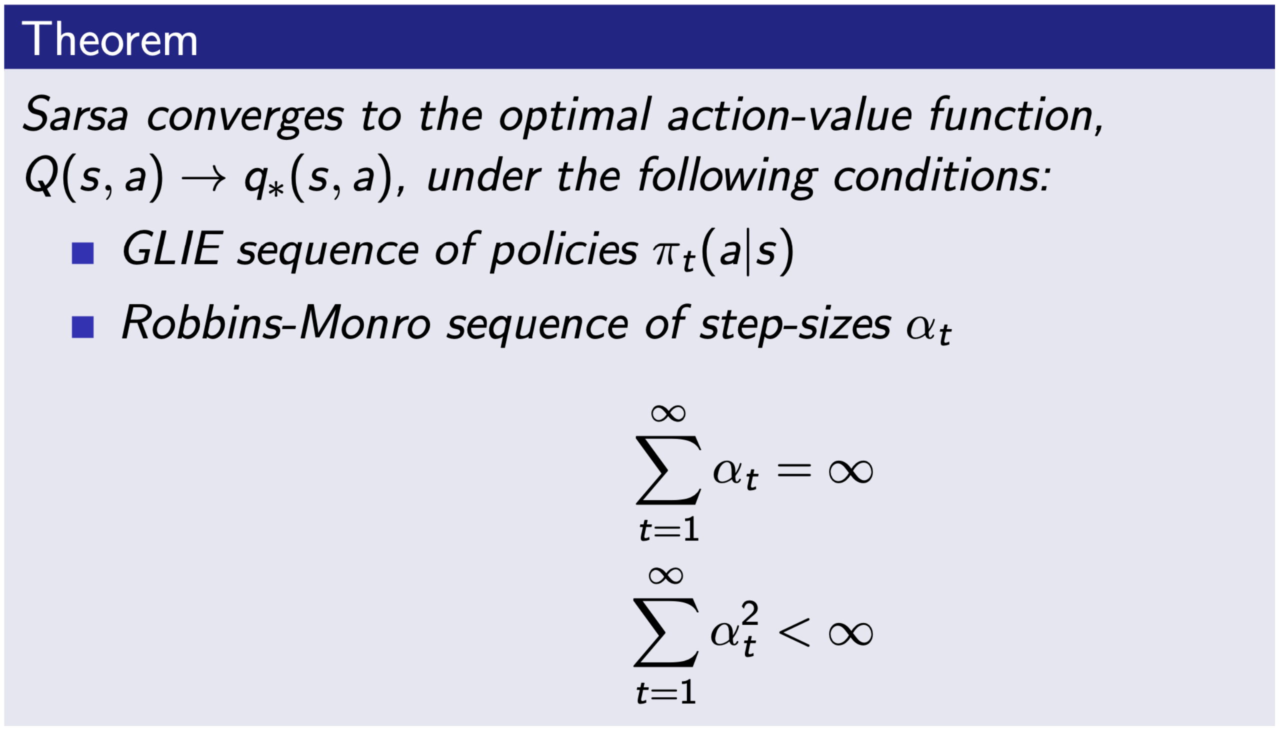
Update Formula:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_{t+1} + \gamma . Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$



```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
   Initialize $S$
   Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
   Repeat (for each step of episode):
      Take action $A$, observe $R$, $S'$
      Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
      $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
      $S \leftarrow S'$; $A \leftarrow A'$;
   until $S$ is terminal
```

Convergence of SARSA



Theorem

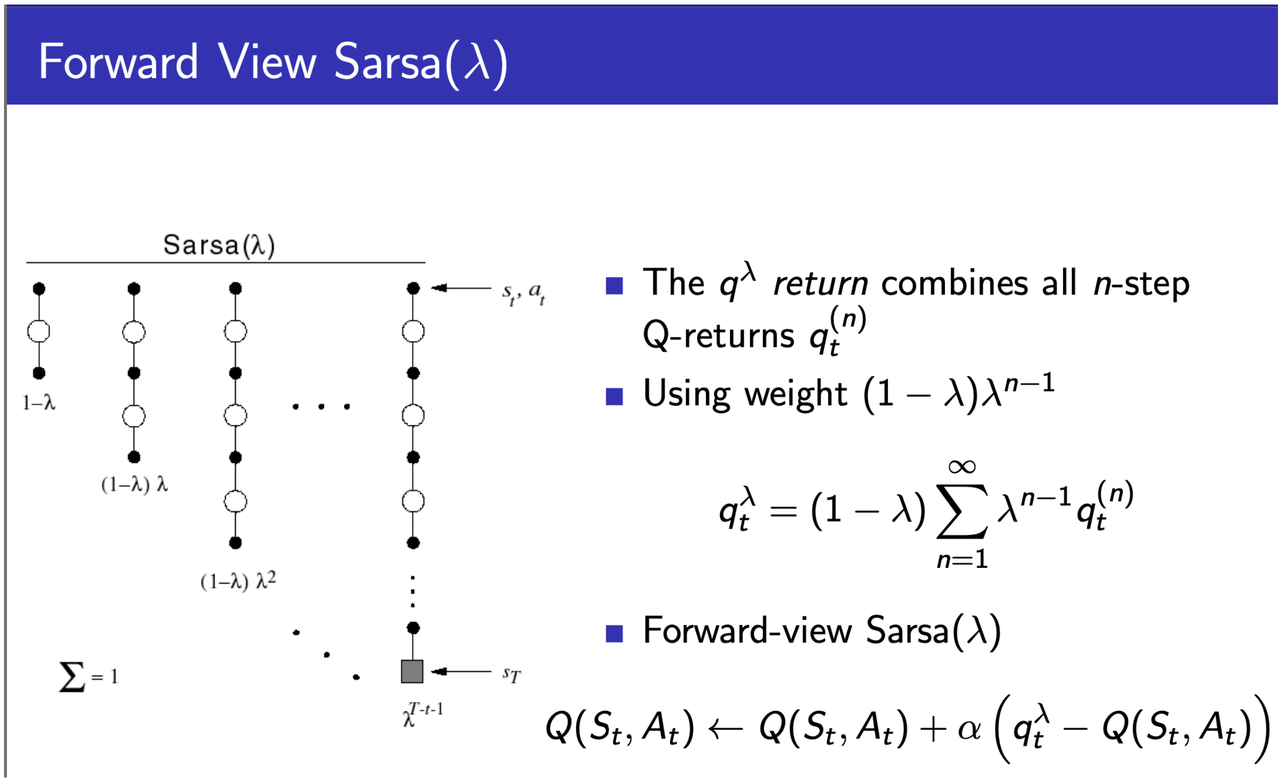
Sarsa converges to the optimal action-value function, $Q(s,a) \rightarrow q_*(s,a)$, under the following conditions:

- *GLIE sequence of policies $\pi_t(a|s)$*
- *Robbins-Monro sequence of step-sizes α_t*

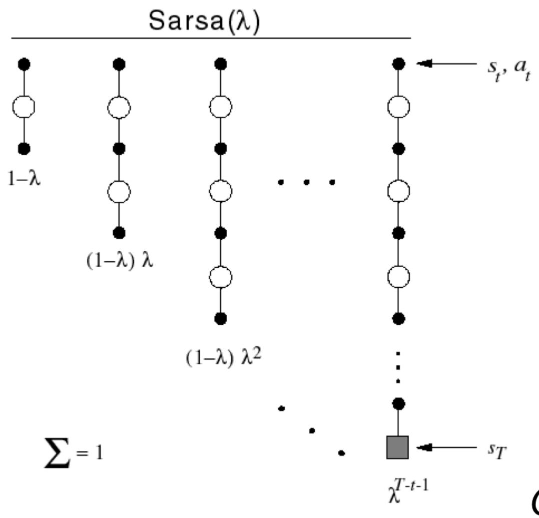
$$\sum_{t=1}^{\infty} \alpha_t = \infty$$

$$\sum_{t=1}^{\infty} \alpha_t^2 < \infty$$

SARSA-λ Forward-view



Forward View Sarsa(λ)



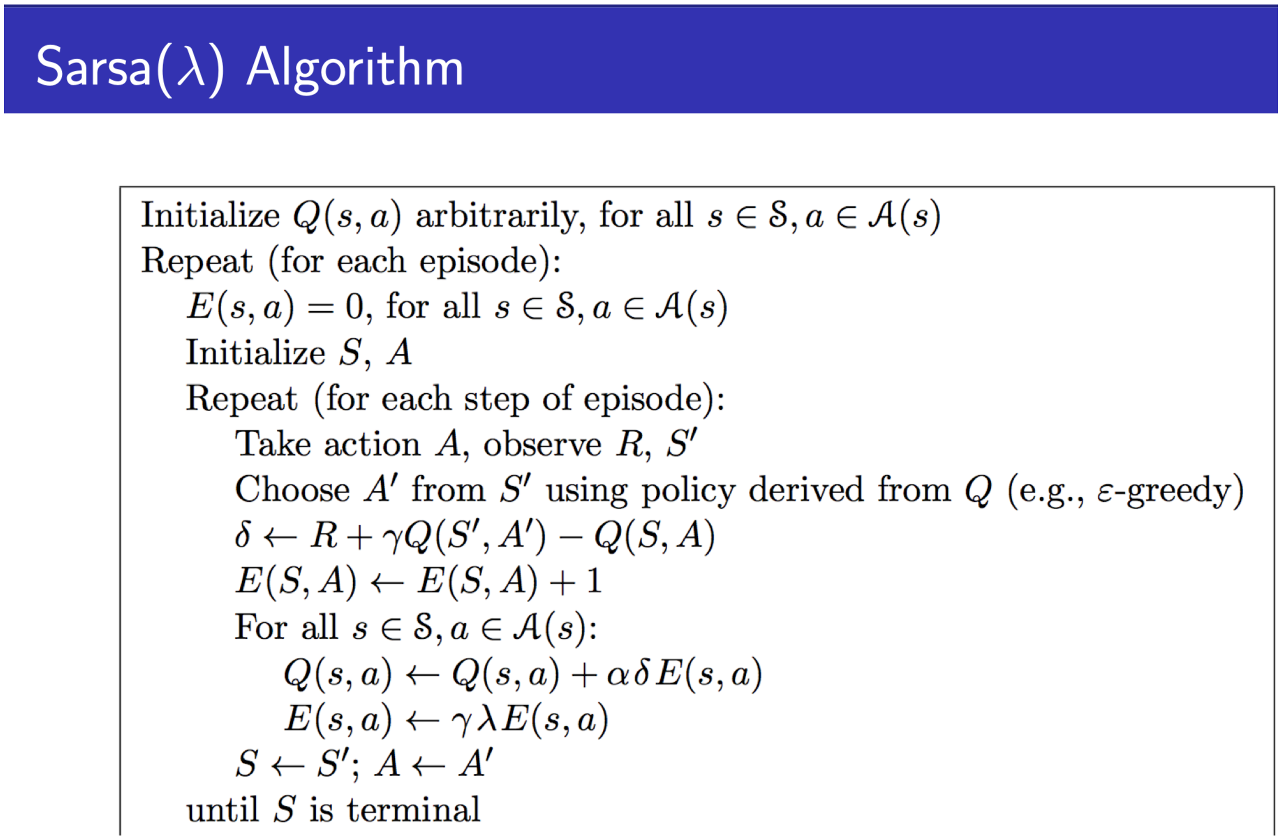
- The q^λ *return* combines all *n*-step Q-returns $q_t^{(n)}$
- Using weight $(1-\lambda)\lambda^{n-1}$

$$q_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} q_t^{(n)}$$

- Forward-view Sarsa(λ)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left(q_t^\lambda - Q(S_t, A_t)\right)$$

SARSA-λ Backward-view



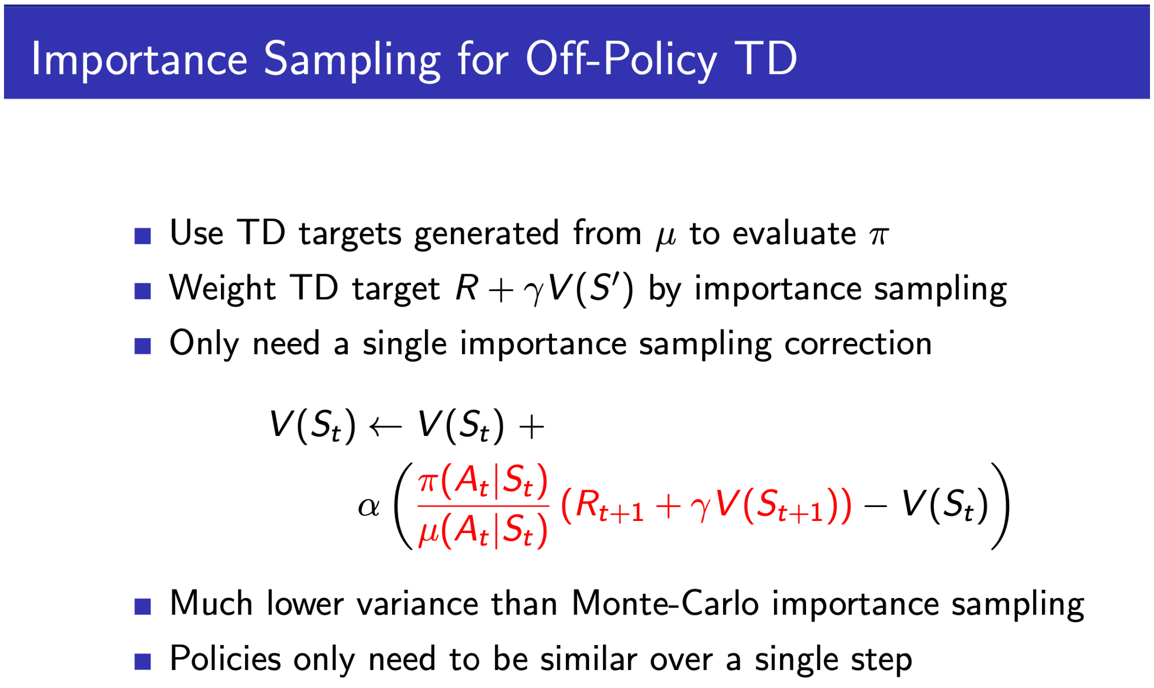
Sarsa(λ) Algorithm

```
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
    $E(s,a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
    Initialize $S$, $A$
    Repeat (for each step of episode):
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $\delta \leftarrow R + \gamma Q(S',A') - Q(S,A)$
        $E(S,A) \leftarrow E(S,A) + 1$
        For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
            $Q(s,a) \leftarrow Q(s,a) + \alpha \delta E(s,a)$
            $E(s,a) \leftarrow \gamma \lambda E(s,a)$
        $S \leftarrow S'$; $A \leftarrow A'$
    until $S$ is terminal
```

Off-policy learning

Evaluate target policy $\pi(a/s)$ to compute $v_\pi(s)$ or $q_\pi(s,a)$ while following behaviour policy $\mu(a/s)$. 2 mechanisms:

- importance sampling:
    - for off-policy MC: extremely high-variance and in practice useless
    - for off-policy TD: works better than MC



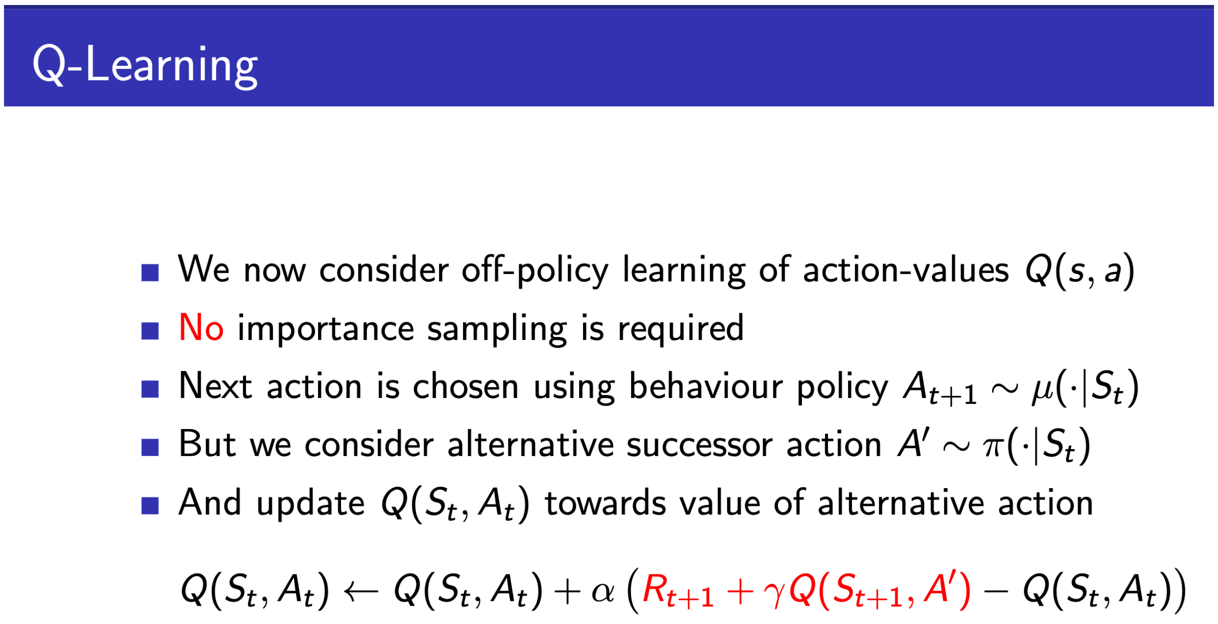
Importance Sampling for Off-Policy TD

- Use TD targets generated from μ to evaluate π
- Weight TD target $R + \gamma V(S')$ by importance sampling
- Only need a single importance sampling correction

$$V(S_t) \leftarrow V(S_t) + \alpha \left(\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} (R_{t+1} + \gamma V(S_{t+1})) - V(S_t) \right)$$

- Much lower variance than Monte-Carlo importance sampling
- Policies only need to be similar over a single step

- Q-learning: this is the method that works the best



Q-Learning

- We now consider off-policy learning of action-values $Q(s, a)$
- No importance sampling is required
- Next action is chosen using behaviour policy $A_{t+1} \sim \mu(\cdot|S_t)$
- But we consider alternative successor action $A' \sim \pi(\cdot|S_t)$
- And update $Q(S_t, A_t)$ towards value of alternative action

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma Q(S_{t+1}, A') - Q(S_t, A_t) \right)$$

Off-Policy Control with Q-Learning

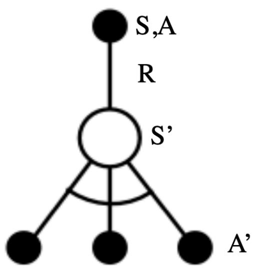
- We now allow both behaviour and target policies to improve
- The target policy π is greedy w.r.t. $Q(s, a)$

$$\pi(S_{t+1}) = \underset{a'}{\operatorname{argmax}}\ Q(S_{t+1}, a')$$

- The behaviour policy μ is e.g. ϵ-greedy w.r.t. $Q(s, a)$
- The Q-learning target then simplifies:

$$\begin{aligned} & R_{t+1} + \gamma Q(S_{t+1}, A') \\ =& R_{t+1} + \gamma Q(S_{t+1}, \underset{a'}{\operatorname{argmax}}\ Q(S_{t+1}, a')) \\ =& R_{t+1} + \max_{a'} \gamma Q(S_{t+1}, a') \end{aligned}$$

Q-Learning Control Algorithm



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

Theorem

Q-learning control converges to the optimal action-value function, $Q(s, a) \to q_*(s, a)$

This Q-learning algorithm for control is also called SARSAMAX as it combines SARSA approach and the greedy policy.

There is also the expected SARSA algorithm where we replace the max operation by the expected value of the q_value of the pair next state, next action.

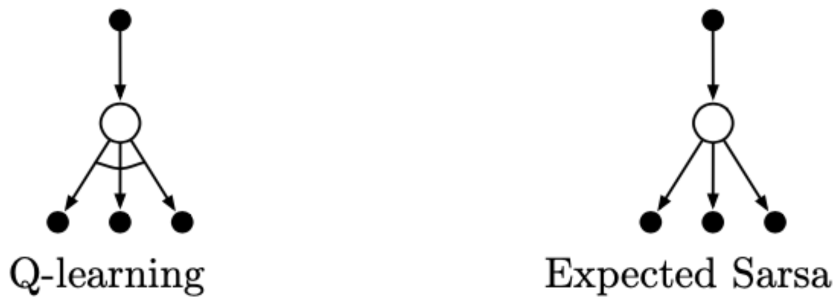
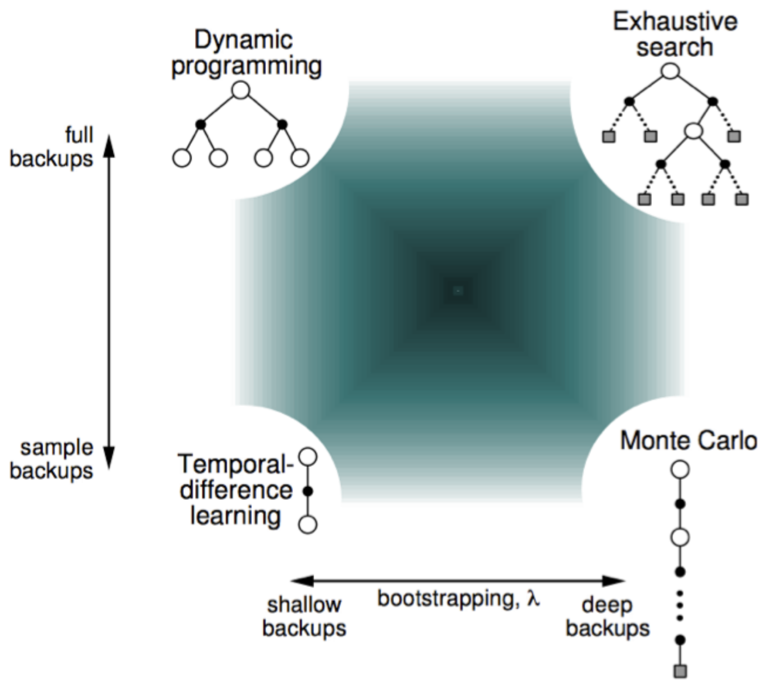


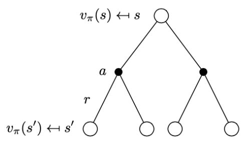
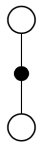
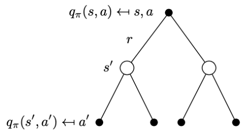
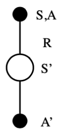
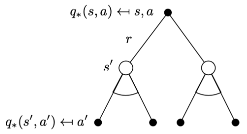
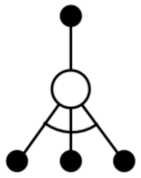
Figure 6.4: The backup diagrams for Q-learning and Expected Sarsa.

Comparison of RL policy evaluation



DP and TD

Relationship Between DP and TD

	Full Backup (DP)	*Sample Backup (TD)*
Bellman Expectation Equation for $v_\pi(s)$	 $v_\pi(s) \leftarrow s$ a r $v_\pi(s') \leftarrow s'$ Iterative Policy Evaluation	 TD Learning
Bellman Expectation Equation for $q_\pi(s,a)$	 $q_\pi(s,a) \leftarrow s,a$ r s' $q_\pi(s',a') \leftarrow a'$ Q-Policy Iteration	 Sarsa
Bellman Optimality Equation for $q_*(s,a)$	 $q_*(s,a) \leftarrow s,a$ r s' $q_*(s',a') \leftarrow a'$ Q-Value Iteration	 Q-Learning

Relationship Between DP and TD (2)

Full Backup (DP)	*Sample Backup (TD)*
Iterative Policy Evaluation $V(s) \leftarrow \mathbb{E}[R + \gamma V(S') \mid s]$	TD Learning $V(S) \xleftarrow{\alpha} R + \gamma V(S')$
Q-Policy Iteration $Q(s,a) \leftarrow \mathbb{E}[R + \gamma Q(S',A') \mid s,a]$	Sarsa $Q(S,A) \xleftarrow{\alpha} R + \gamma Q(S',A')$
Q-Value Iteration $Q(s,a) \leftarrow \mathbb{E}\left[R + \gamma \max_{a' \in \mathcal{A}} Q(S',a') \mid s,a\right]$	Q-Learning $Q(S,A) \xleftarrow{\alpha} R + \gamma \max_{a' \in \mathcal{A}} Q(S',a')$

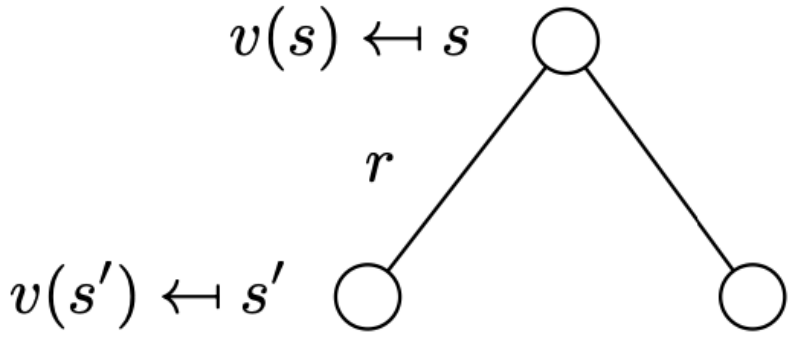
where $x \xleftarrow{\alpha} y \equiv x \leftarrow x + \alpha(y - x)$

Appendix

Bellmann Equations for MRPs

- for **MRPs (NO ACTION YET!)**: $v(s) = \mathbb{E}[G_t / S_t = s] = \mathbb{E}[R_{t+1} + \gamma . v(S_{t+1}) / S_t = s]$

$$v(s) = \mathbb{E}\left[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s\right]$$



$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s')$$

- **Matrix Form**

The Bellman equation can be expressed concisely using matrices,

$$v = \mathcal{R} + \gamma \mathcal{P} v$$

where v is a column vector with one entry per state

$$\begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1 \\ \vdots \\ \mathcal{R}_n \end{bmatrix} + \gamma \begin{bmatrix} \mathcal{P}_{11} & \dots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{11} & \dots & \mathcal{P}_{nn} \end{bmatrix} \begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix}$$

So the Bellmann equation is a linear equation that can be solved directly (matrix inversionn)

$v = R + \gamma P. v$

$v = (1 - \gamma. P)^{-1}. R$

This direct solution is only possible for small MRPs. For large MRPs, there are many iterative methods: Dynamic programming, Monte Carlo, Temporal-Difference learning

k-bandit problem

- stationary environment:

$Q_n = \frac{R_1 + R_2 + \ldots + R_{n-1}}{n-1}$

$Q_{n+1} = \frac{1}{n}. \sum_{i=1}^{n} R_i = Q_n + \frac{1}{n}[R_n - Q_n]$

- non-stationary environment:

$Q_{n+1} = Q_n + \alpha. [R_n - Q_n = (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha (1-\alpha)^{n-i} R^i$

It is a weighted average of past rewards and the initial estimate Q_1 with $\alpha \in (0, 1]$

Comparison with supervised and unsupervised learning

- Supervised learning is equivalent to finding a function f based on x,y pair - y = f(x). Also called function approximation

- Unupervised learning is equivalent to finding a function f based on x input data only. Also called clustering description
- Reinforcement learning consists in finding a function, called an optimal policy π^* , based on a set of input data (s,a) and rewards z = r for each (s,a) pair. y = f(x) <=> $a = \pi^*(s)$. Main differences:
    - There is no supervisor, only a reward signal.
    - Feedback is delayed, not instantaneous
    - Time really matters
    - Agent takes actions and influences its environment

Going Further

- Extensions to MDPs
    - Infinite and continuous MDPs
    - Partially Observable Markov Decision Process
    - Belief States (Lecture 2- UCL (David Silver)
    - Ergodic Markov Process (Lecture 2- UCL (David Silver)
    - Average Reward Value Function (Lecture 2- UCL (David Silver)

References

- repository for RL algorithms:
    - ShangTong Zhang
    - Denny Britz

- David Silver courses: https://www.davidsilver.uk/teaching/