Reinforcement Learning

Part 2

Machine Learning, Fall 2024

Michael Wand

using some material from Paulo Rauber, Faustino Gomez, Jan Koutnik

TA: Dylan Ashley (dylan.ashley@idsia.ch)

Summary of Part 1

- We have covered Dynamic Programming (DP) for solving RL problems.
- Key idea: determine the *value* of each state (or state-action pair).
  - Value iteration, policy iteration
- Major problem: need full access to the model (states, actions, state transitions).
  - If this is possible, DP can be used to solve problems with millions of states, with reasonably fast convergence (often faster than the theoretical guarantee).

Summary of Part 1

- What to remember (non-exhaustive list):
  - states, actions, reward and return, policies
  - Markov Decision Processes
  - (action-)value function, and the optimal value function
  - Policy iteration, value iteration

Outlook

- Remainder of this lecture: How to do RL if the model is intractable or inexistent, or if we cannot directly model state and/or actions (possibly because they are continuous).
  - also includes the case of a state which is not fully observable!
- Today: *tabular model-free algorithms*
  - state space is known, finite, and observable
  - we can make a *table* of state-action values and try to estimate/optimize them
- Next time: *approximative algorithms*
  - in particular, for continuous state space

S	A	Q
1		
2		
3		

Model-free RL:
Learning from Experience

Sampling-based Approximations

- We have gotten to know two “exact” DP methods: *Value Iteration, Policy Iteration*
  - They are exact in the sense that we aim at computing the true value function / optimal value function from the model.
  - Note that knowing the model, we never needed to actually “run” the agent (i.e. generate actual experience)!
- If we do not have access to the model, we need to
  - a) generate actual experience of the environmental dynamics
  - b) adapt our methods such that they work with samples from the environment.

Sampling-based Approximations

- Bellman equation:

$$V^\pi(s) = \sum_a \sum_{s'} \pi(s,a) P^a_{ss'} [R^a_{ss'} + \gamma V^\pi(s')]$$

- Policy iteration:

$$V_k(s) \leftarrow \sum_a \sum_{s'} \pi(s,a) P^a_{ss'} [R^a_{ss'} + \gamma V_{k-1}(s')]$$

- Replace weighted sum by expectation:

$$V_k(s) \leftarrow E_{a,s'} [R^a_{ss'} + \gamma V_{k-1}(s')]$$

where the expectation is taken over the probability distribution of actions and states, derived from policy and environment.

Sampling-based Approximations

- We know that expectations can be replaced by sampling!
- For sampling-based Policy Iteration, let us assume that we generate a series of sampled returns $G_{\text{smp}}(s)$ from some state s (will soon see how).
  - Requires that all states are actually reached.
- Then

$$\hat{V}(s) = \frac{1}{N} \sum_{n=1}^{N} G_{\text{smp}}^{(n)}(s)$$

- or, written as a running average:

$$\hat{V}(s) \leftarrow (1-\alpha)\hat{V}_{\text{old}}(s) + \alpha G_{\text{smp}}(s)$$

where the estimate gets updated whenever a new sample arrives. α can be considered the learning rate.

Sampling-based Approximations

- What about Value Iteration?

$$V_k(s) \leftarrow \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_{k-1}(s')\right]$$

- We cannot sample across the max operator!
- Value iteration can *not* directly be used in a sampling-based framework!

Sampling-based Approximations

- Idea: after having done Policy Evaluation, improve policy (by choosing the “best” action, e.g. greedily).
- But: If we do not have access to the state transitions, sampling state values is not enough!
  - Need to get information about *actions*, given the state.
- What about estimating the action-value function?

$$\hat{Q}(s,a) = \frac{1}{N}\sum_{n=1}^{N} G_{\text{smp}}^{(n)}(s,a)$$

- Issue: depending on the policy, certain state-action pairs *never occur!*
  - This is a stronger requirement than reaching all states.
  - We will see later on how to solve that.

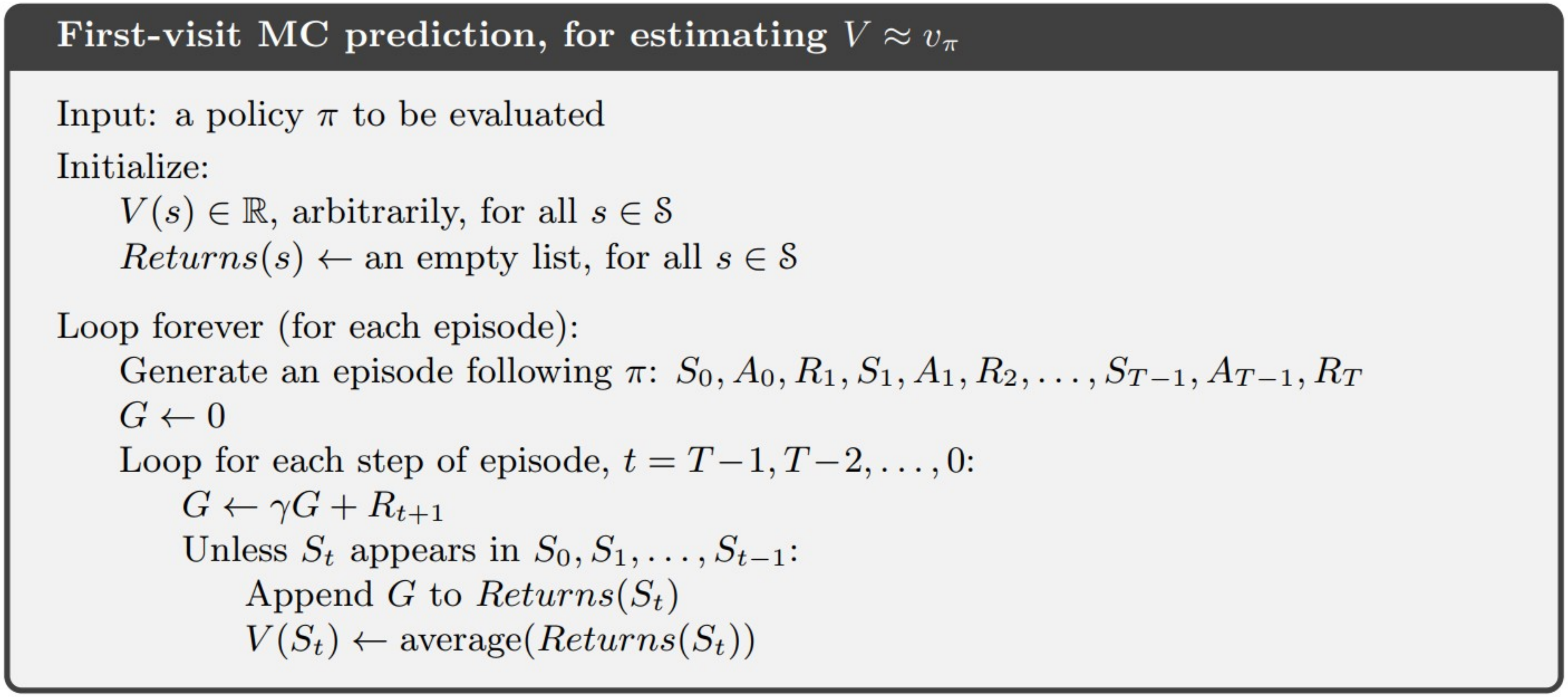
Monte Carlo Methods

Monte Carlo Methods

- MC idea: collect data from entire *episodes.*
  - we start anywhere and let the agent collect rewards
  - assume that agent always reaches terminal state (*episodic task*), so that we get an estimate for the return
  - estimate state values or action-values by averaging over observed returns!
- *First-visit MC*: only consider the first visit to any state
- *Every-visit MC*: consider all visits.
- Conceptually simple, but data-hungry...

Basic Algorithm

(source: Sutton/Barto)



First-visit 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, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless $S_t$ appears in $S_0, S_1, \ldots, S_{t-1}$:
            Append $G$ to $Returns(S_t)$
            $V(S_t) \leftarrow$ average$(Returns(S_t))$
```

MC Policy Evaluation

- Assuming that all states are actually reached, it is easy to show that *V(s)* converges to the true value function.
  - Good convergence guarantees possible (quadratic convergence, follows from the law of large numbers)
- In contrast to DP (where we had a static model of the environment), MC methods rely on generating actual experience!
  - The experience could also be simulated (depends on the task).
- Note that usually, a large number of episodes is necessary to obtain good results.

Blackjack example

- Blackjack: Card-based chance game
- Each player plays against the dealer, goal is to have more points on the hand than the dealer, but *not* more than 21
  - ace = 1 or 11, faces = 10, otherwise as numbered.
- Player and dealer each get two initial cards, one dealer card shown.
- Player draws cards *(hit)* until satisfied *(stick)*, loses immediately *(goes bust)* with >21 points.
  - Assumption: Infinite supply of cards
- The dealer has a fixed strategy.
- Goal: find a good strategy for the player!

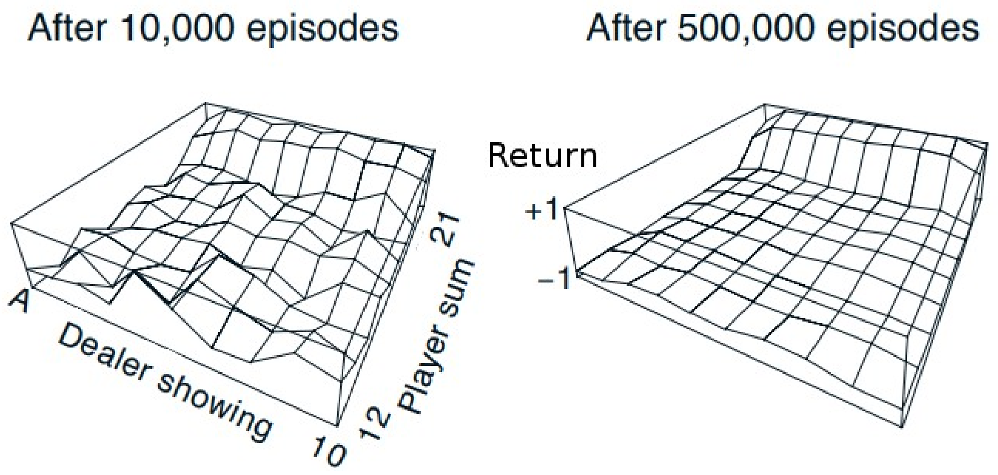
Blackjack example

- State space: ~200 states
  - sum of the player's cards
  - dealer’s visible card
  - does the player have an ace or not.
- Actions: hit or stick.

- Model description ($\rightarrow$DP) is complex, inefficient, and error-prone
- Sampling ($\rightarrow$MC) is straightforward!

Blackjack example

- Assume a policy, e.g. hit until sum >= 20
- rewards: +1 for win, 0 for draw, -1 for loss
- simulate many games, starting from all possible states, and average rewards.



Value function for the example policy, estimated via MC. After 500000 episodes, convergence has been achieved. Do you see why the strategy is not very good?

Monte Carlo on Actions

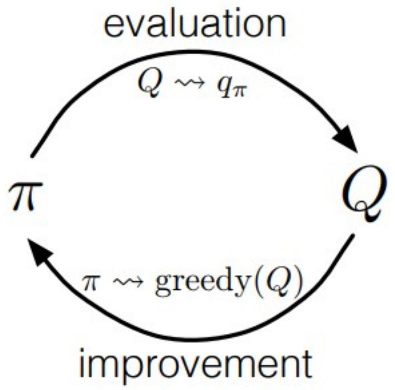
- If we do not have full knowledge of state transitions, estimating state values is not enough.
  - One must explicitly estimate *action values* Q(s,a) in order to suggest a policy.
- Problem: not all state/action pairs are covered!
  - Because the existing policy suggests which action to take; for a given state, could always be the same one (deterministic policy)

Monte Carlo on Actions

- Solution 1: assure random “exploring” starts
  - for the start of each episode, sample a state/action pair, and then follow the policy
  - make sure that all state/action pairs are covered.
- Solution 2: occasionally choose a random action, instead of the best one (ε-greedy policy).
- Solution 3: Make estimates (e.g. value estimates) for a *target* policy, using a different *behavior* policy.
  - These methods are called *off-policy* methods.
    - often converge slower, theoretically more complicated, but much more general than *on-policy* methods.
    - some technical details required to make them work (importance sampling)

Monte Carlo Control

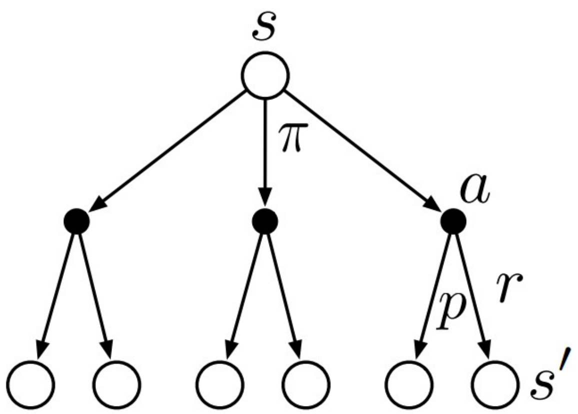
- *Generalized Policy Iteration*:
  - one has both an approximate value function and an approximate policy
  - iteratively improve one or the other.
- It is natural to iterate between computing one episode of experience, and using this experience to update the value function for those states which have been touched.
- Very commonly, one relies on an ε-greedy policy.


evaluation
$Q \rightsquigarrow q_\pi$
π
Q
$\pi \rightsquigarrow \text{greedy}(Q)$
improvement

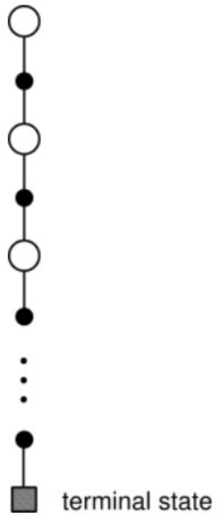
Backup Diagrams

- *Backup diagrams* indicates the origin of value information which is transferred *back* to a state.
- DP: collect information from *all* possible successors, but only 1 step lookahead.
- MC: consider an entire chain, but always just one successor.

backup diagram for DP



backup diagram for MC



Exploration vs Exploitation

- Remember the ε-greedy policy: Occasionally choose a random action, instead of the best (known) one.
- General concept: *Exploitation versus Exploration*
  - *Exploit:* follow the current best (greedy) policy, accumulating rewards
  - *Explore:* try out new actions, even if they seem suboptimal for now.
- Balancing these two is a key task is complex reinforcement learning setups!

Temporal Difference Methods

Temporal Difference Learning

- Combines the ideas of Monte Carlo sampling and Dynamic Programming:
  - instead of running an episode till the end, just do *one step!*
- State value is updated based on existing estimates (*bootstrapping*).
- A major original RL breakthrough, basis for many more sophisticated techniques.
- Standard state-of-the-art approach.

Temporal Difference Learning

We consider Policy Evaluation.

- Consider a running Monte Carlo estimate for *V:*

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[G_t - V(S_t)\Big]$$

  where G_t is the return *after* the episode. α can be considered the learning rate.

- TD-Learning (more exactly TD(0)) uses the following rule:

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big]$$

Temporal Difference Learning

- Both methods update the current estimate with a new estimate, with a certain learning rate α.
- Comparing the TD and MC updates,
  - in MC, the update target is the Return of the entire episode
  - in TD, the update target is the immediate Reward, plus the discounted estimate of the future reward

$$\text{Monte Carlo:} \quad V(S_t) \leftarrow V(S_t) + \alpha[\underbrace{G_t}_{\text{Target}} - V(S_t)]$$

$$\text{Temporal Difference:} \quad V(S_t) \leftarrow V(S_t) + \alpha[\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{Target}} - V(S_t)]$$

Temporal Difference Learning

- In contrast to DP, one does not need values of *all* next states.
  - This is good because in the real world we can only visit one at a time.
  - No model required.
- In contrast to standard MC, one does not have to wait for the end of the episode
  - faster updates
  - method also words for non-episodic tasks.
- Value function updates are computed based on existing estimates: *bootstrapping.*

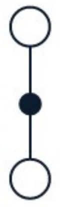
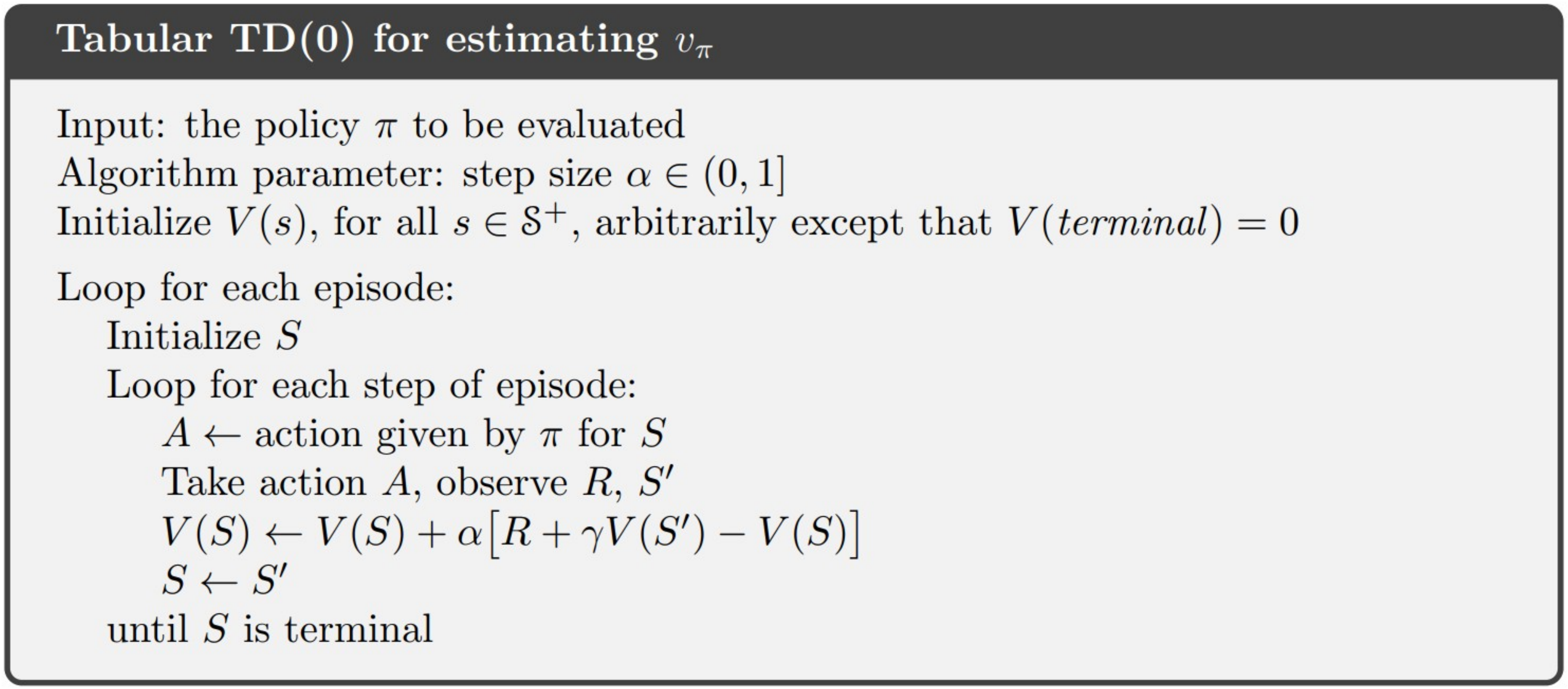


Figure 6.2: The backup diagram for TD(0).

TD Algorithm

- Simple tabular TD(0) (source: Sutton/Barto)



Tabular TD(0) for estimating v_π

```
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 $V(\mathit{terminal}) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        $A \leftarrow$ action given by $\pi$ for $S$
        Take action $A$, observe $R$, $S'$
        $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

TD Control Methods

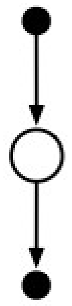
- How to apply TD methods to RL-based control tasks?
- Use the standard pattern of Generalized Policy Iteration.
- As with MC methods: exploitation/exploration tradeoff, on-policy and off-policy methods.

SARSA: On-Policy TD Control

- A straightforward on-policy algorithm: SARSA.
  - (The name derives from the sequence of state-action-reward-...).
- Learns the action-value function by TD learning, i.e.

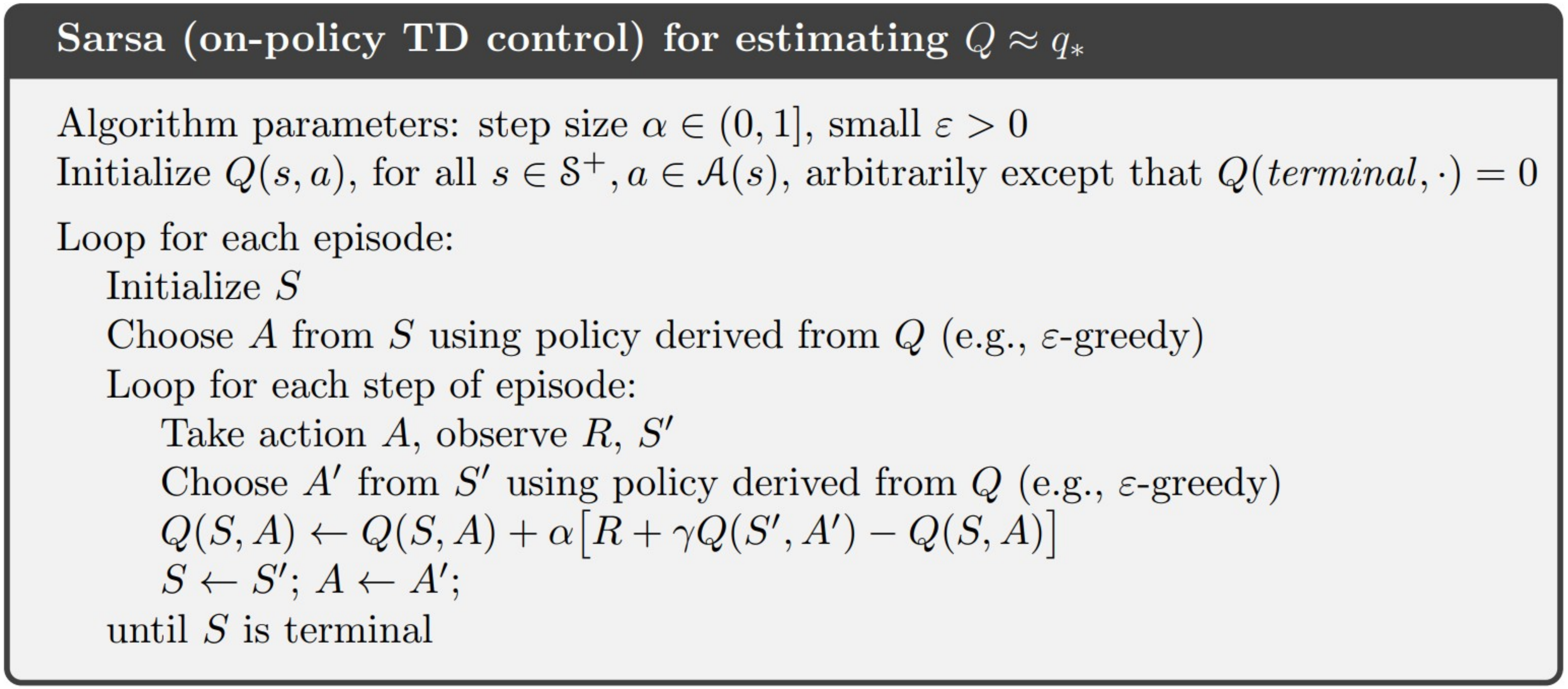
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \Big]$$

- Typically use ε-greedy policy to insure that all state/action pairs are covered.
- Backup diagram starts and ends with an *action.*
- Can deal with probabilistic state transitions.



SARSA: On-Policy TD Control

(source: Sutton/Barto)



Sarsa (on-policy TD control) for estimating $Q \approx q_*$

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

Q-learning: Off-policy TD control

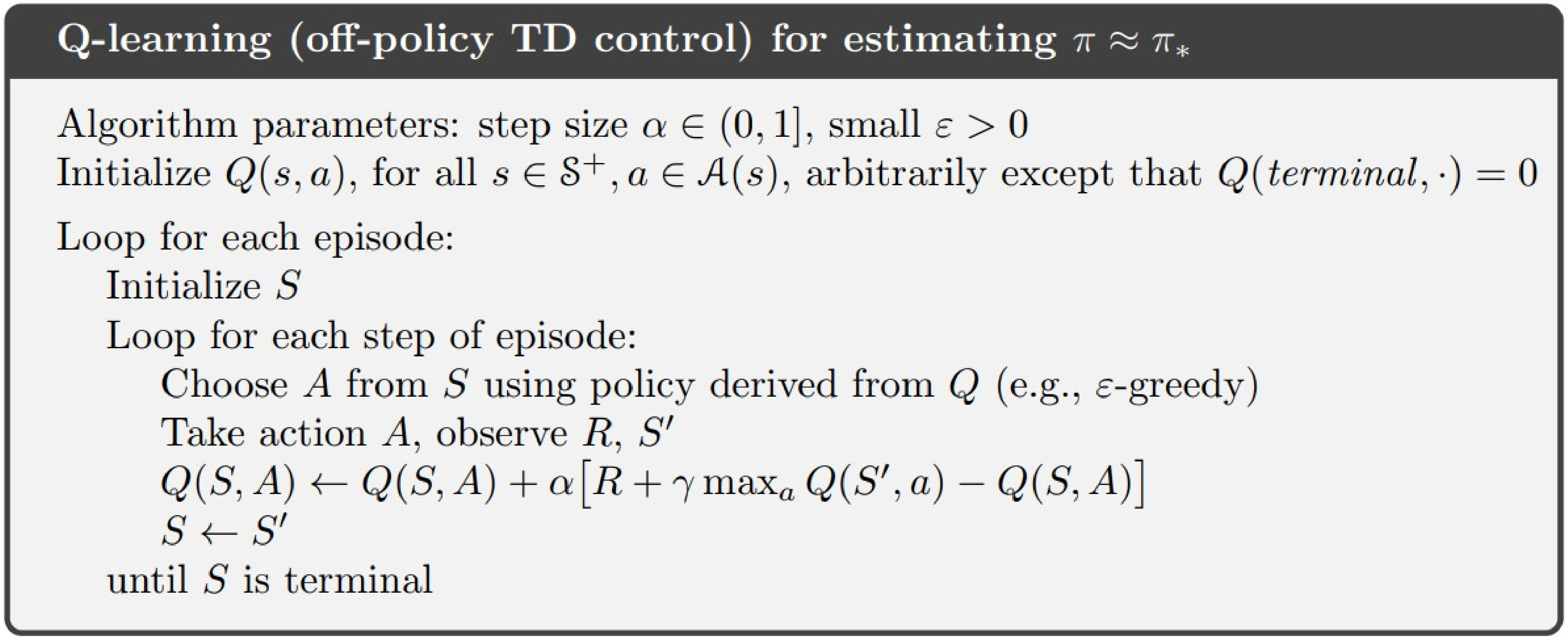
- The SARSA method has the disadvantage of requiring to follow the target policy (on-policy algorithm).
- However, a simple change to the update rule allows to use the idea for different target policies (off-policy learning):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \Big]$$

- For the update, we maximize over all possible future actions.
- One can show that *Q* approximates the optimal action-value function *Q**.
  - Flexibility in choosing underlying behavior policy.

Q-learning: Off-policy TD control

source: Sutton/Barto



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
```

Expected SARSA

- Another variation of the TD update rule consists in taking the *expectation* over possible next actions.

$$
\begin{aligned}
Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t)\Big] \\
&\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\Big],
\end{aligned}
$$

- The resulting algorithm (*Expected SARSA)* is slightly more complex that SARSA, but in general converges better.

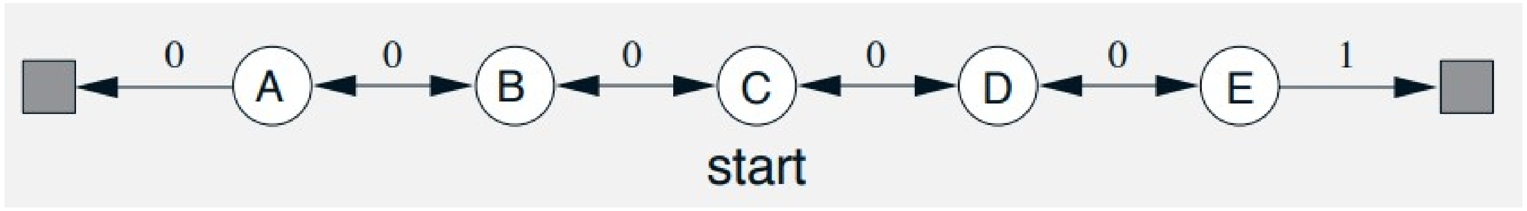
Further Remarks and Conclusion

TD versus MC

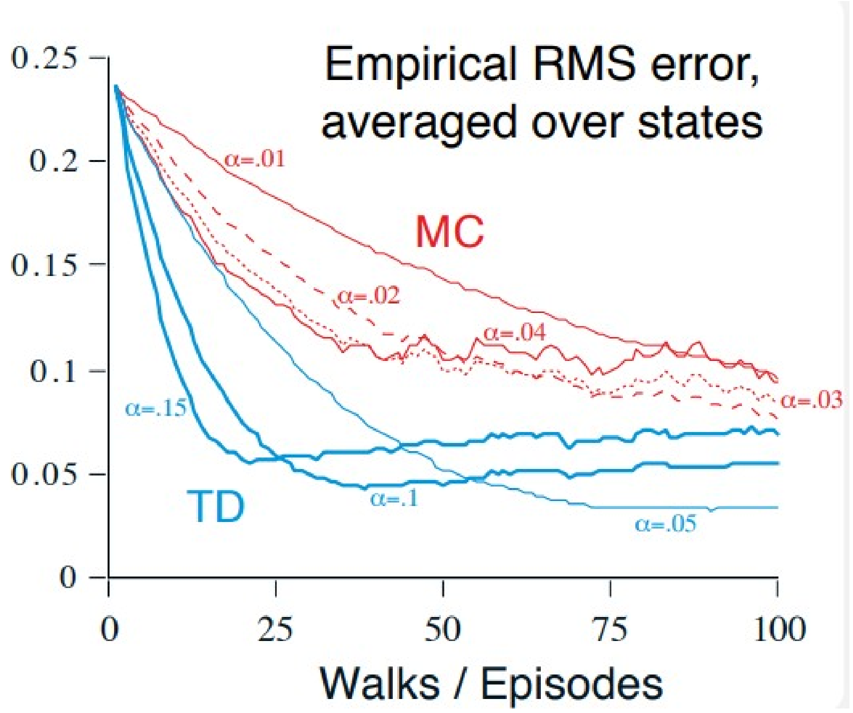
- TD Advantage: easy and natural to implement online.
  - This also works well when the value function is only approximated (e.g. with a neural network, next lesson).
- Convergence guarantees are possible (notably, requires to choose the learning rate well)
  - ...but best of all, it works well in practice!
- Usually faster convergence than MC methods.

TD versus MC

- Example (Sutton/Barto) for a *Markov Reward process* (no actions, just learn environment).



- Start at position C, go left or right with equal probability, two terminal states
- Only reward: +1 in right terminal state
- No discounting: value of state is probability of ending up in the right terminal state
- TD faster and usually better than MC
- TD fluctuates less



TD versus MC

- Consider the paths for which state values are estimated.
- The TD algorithm *averages* values over possible paths, thus giving improved estimates.
- Right panel: The two light blue paths (up to the crossing) are identical in their outcome, but MC (second panel from left) will not detect this.

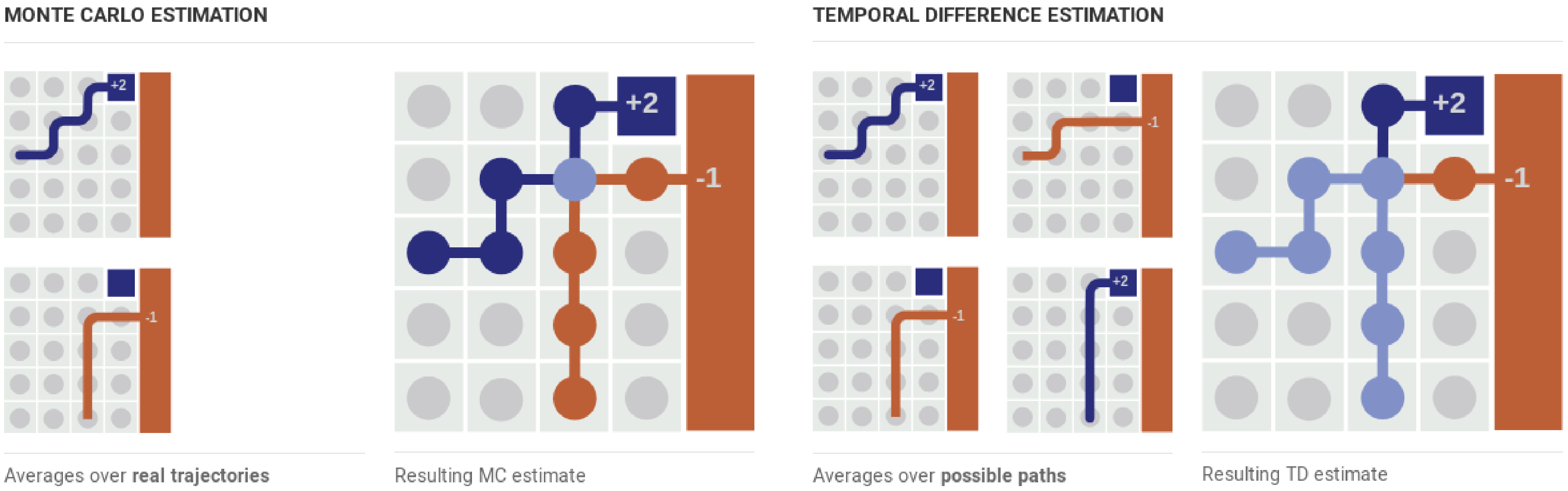

MONTE CARLO ESTIMATION
TEMPORAL DIFFERENCE ESTIMATION
+2
-1
Averages over **real trajectories**
Resulting MC estimate
Averages over **possible paths**
Resulting TD estimate

Image source: https://distill.pub/2019/paths-perspective-on-value-learning

Conclusion

- In this lecture, you have learned how to learn from experience (which is really what RL is all about!).
  - Works in the most common case of an unknown or intractable model.
- Specifically, we have covered Monte-Carlo algorithms and TD learning (SARSA and Q-Learning algorithms).
- We have always assumed a finite and observable set of states.
- Very important concepts: on-policy and off-policy learning.