Reinforcement Learning Lecture 2

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Variants of TD-learning methods and eligibility traces

Part 1: Review and Introduction of BackUp Diagrams

Objectives for today:

- TD learning refers to a whole class of algorithms
- There are many Variations of SARSA
- All are designed to iteratively solve the Bellman equation
- Eligibility traces and n-step methods extend learning over time

Sutton and Barto, Reinforcement Learning (MIT Press, 2nd ed. 2018),
Chapters 5.1-5.4 and 6.1-6.3 and 6.5-6.6, and 7.1-7.2

Reading for this week:

Sutton and Barto, Reinforcement Learning
(MIT Press, 2nd edition 2018, also online)

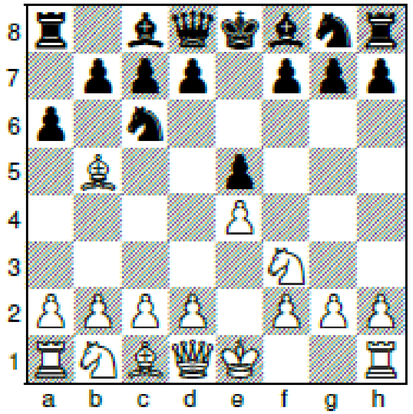
Chapter: 5.1-5.4 and 6.1-6.3 and 6.5-6.6, and 7.1-7.2

Background reading:

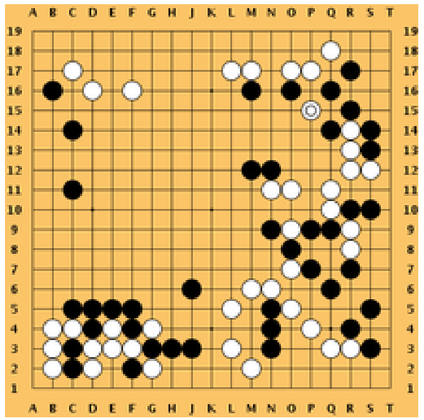
Temporal Difference Learning and TD-Gammon
by Gerald Tesauro (1995) pdf online

Review: Deep reinforcement learning

Chess



Go

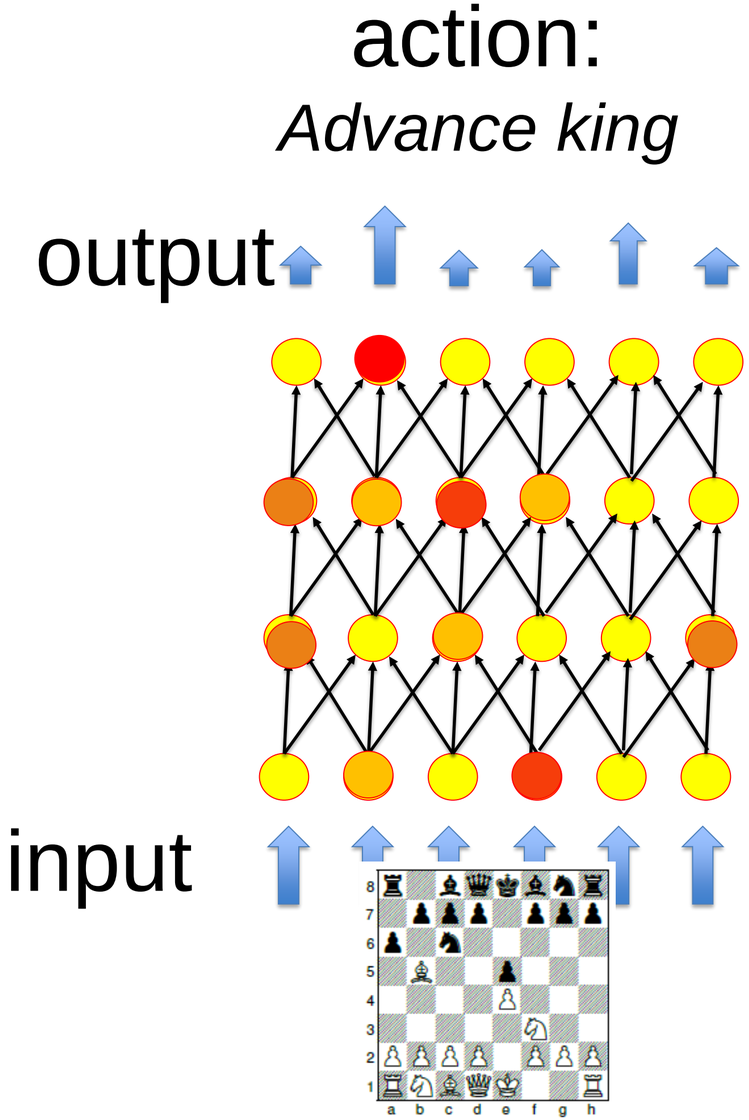


Artificial neural network (*AlphaZero*) discovers different strategies by playing against itself.

In Go, it beats Lee Sedol

Review: Deep reinforcement learning

Network for choosing action



Today

TD Learning as a generalization of SARSA

- How fast is learning?

→ Temporal Difference Learning
→ Variations of SARSA
→ Bootstrap versus Monte-Carlo
→ Continuous/Large State space

Next week

Neural Networks for TD Learning

- How can we learn network weights?

(previous slide)
The basic idea of Reinforcement Learning (RL) was introduced in a previous lecture. Today we make a first step to link RL to artificial neural networks.

Training in neural networks is typical done with a loss function (also called error-function) – but if so, what is the error function for RL? And how can we optimize the weights?

And finally how can we deal with a large state space, potentially even continuous?

Review: Branching probabilities and policy

Policy $\pi(s,a)$

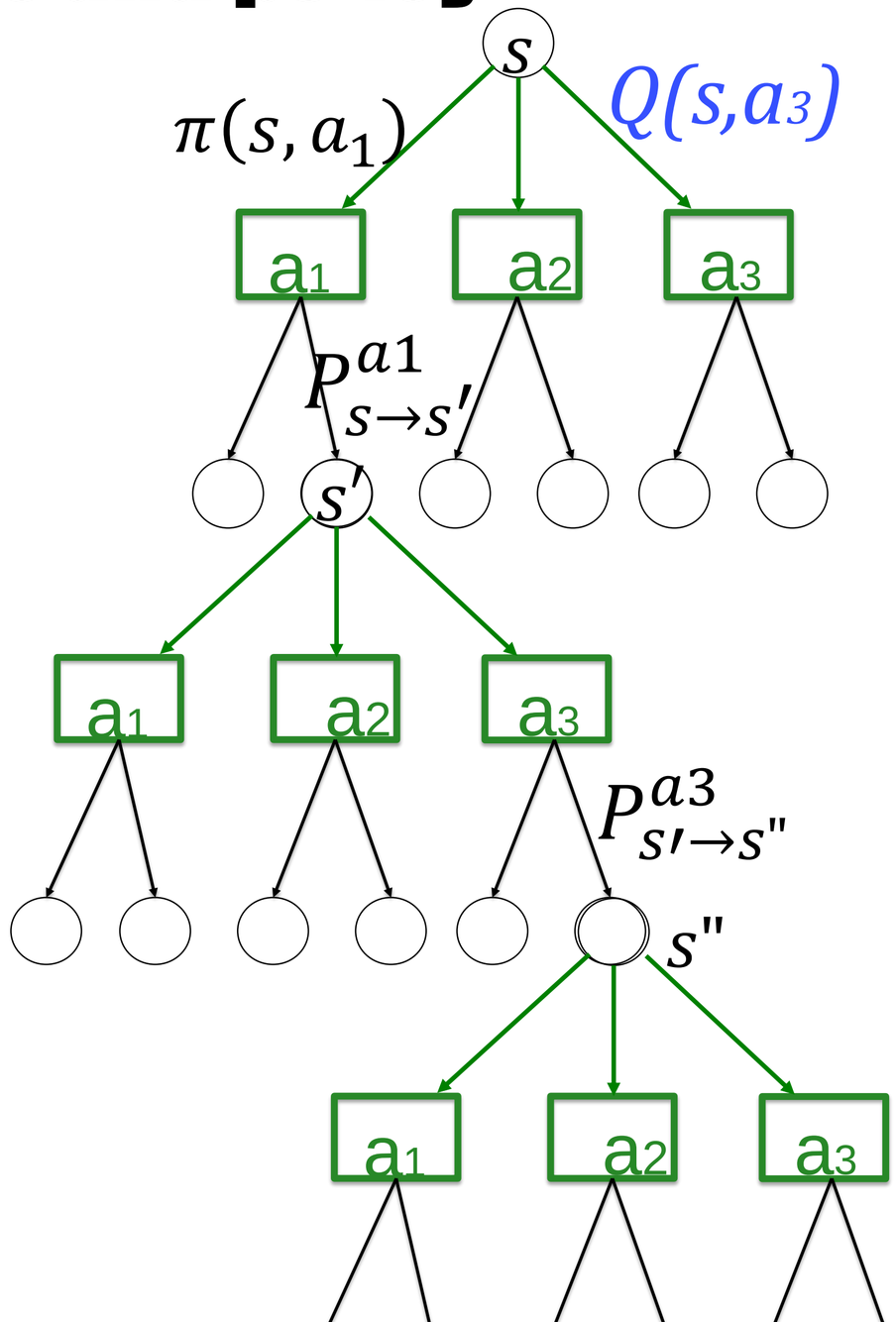
probability to choose action a in state s

$1=\sum_{a'}\pi(s,a')$

Examples of policy:
-epsilon-greedy
-softmax

Stochasticity $P^{a1}_{s\to s'}$

probability to end in state s' taking action a in state s

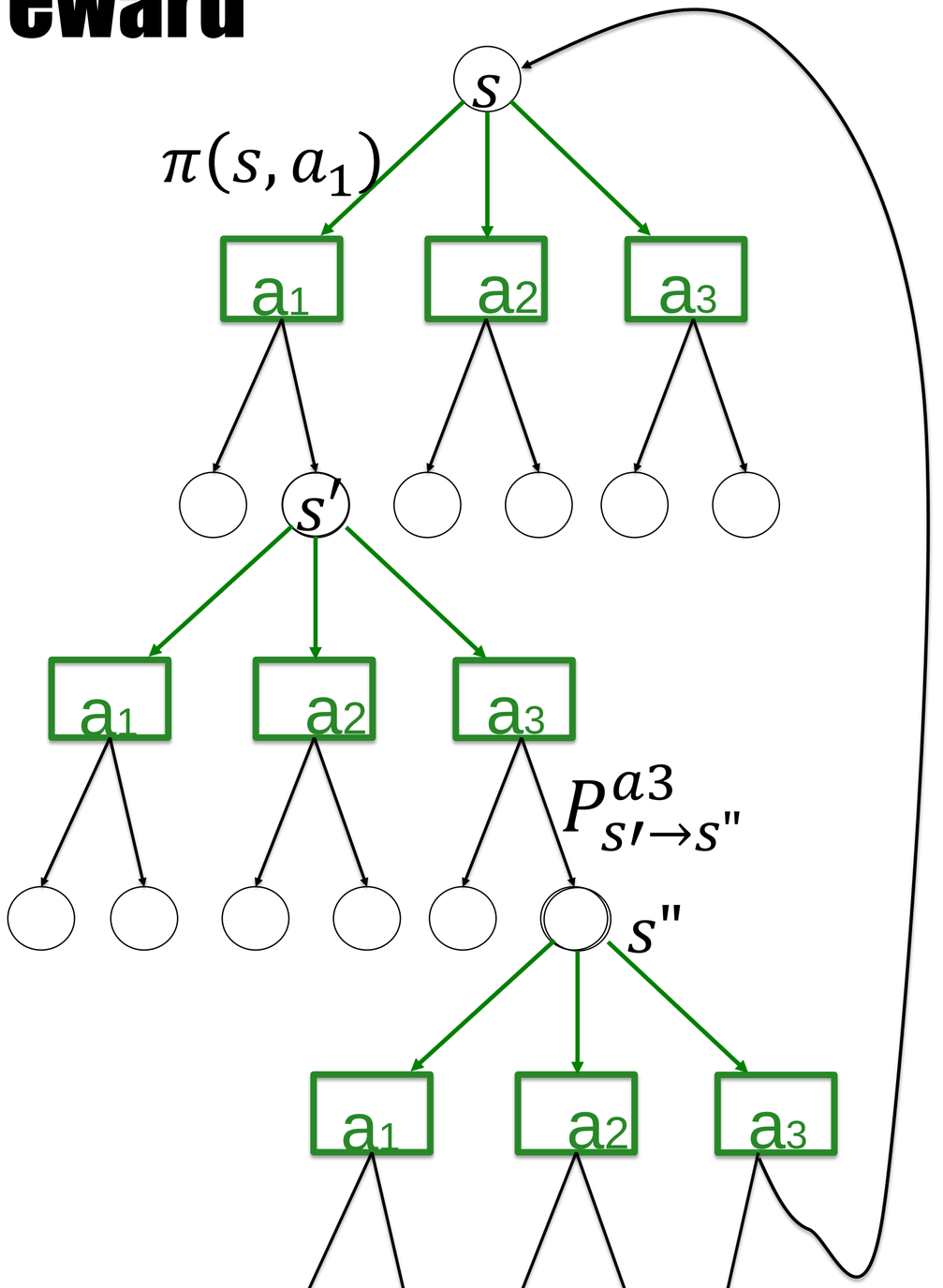


Review Total expected (discounted) reward

Starting in state s with action a

$Q(s,a)$ =

$$\langle r_t + \gamma\, r_{t+1} + \gamma^2 r_{t+2} + \gamma^3\, r_{t+3} + \cdots \rangle$$



Discount factor: γ <1

-important if graph of states is recurrent !

-avoids blow-up of summation

-gives less weight to reward in **far** future

(previous slides)

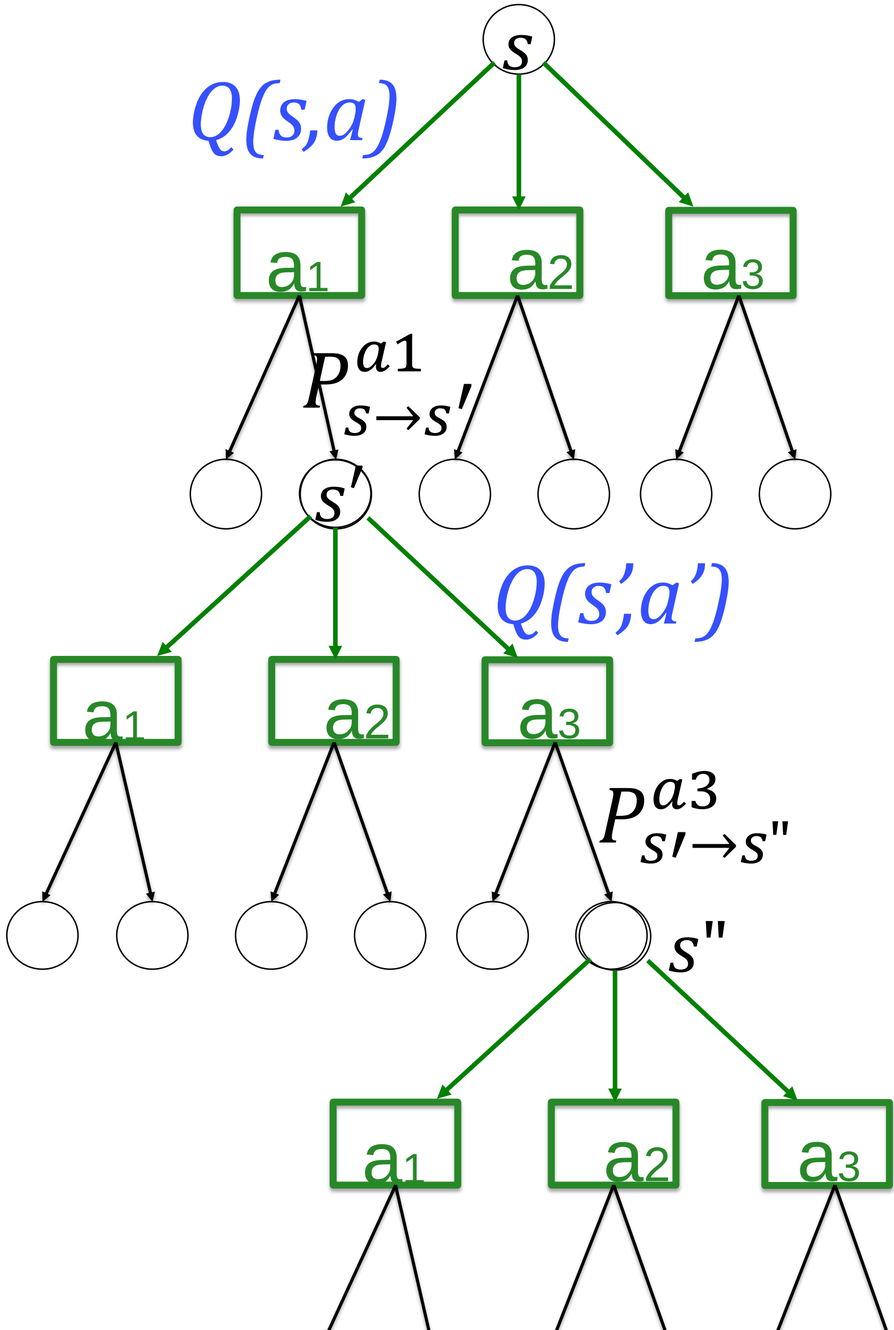
We know from previous lectures that RL works with states and actions that allow probabilistic transitions between the states. The two probabilistic factors are the policy π and the transition probabilities given state and action choice.

An important quantity is the Q-value which represents the expectation of the accumulated reward (discounted with a factor gamma smaller than one).

Review: Bellman equation

$$Q(s,a) = \sum_{s'} P^{a}_{s \to s'} \left[R^{a}_{s \to s'} + \gamma \sum_{a'} \pi(s',a') Q(s',a') \right]$$

Bellman equation = value consistency of neighboring states



(previous slide)
The Q-value Q(s,a) further up in the graph is the expected total discounted reward – summed over all possible future actions and states.
It can be decomposed in an average over the immediate rewards, actions, and states, and the Q-values Q(s’,a’) of all possible next states.
The Bellman equation can therefore be interpreted as summarizing the consistency between the Q-values in state s, and the Q-values in neighboring states s’.
The difference between Q(s,a) and Q(s’,a’) must be explained by the immediate reward.

We will exploit and extend the notion of consistency several times in the lecture today.

Review: SARSA algorithm

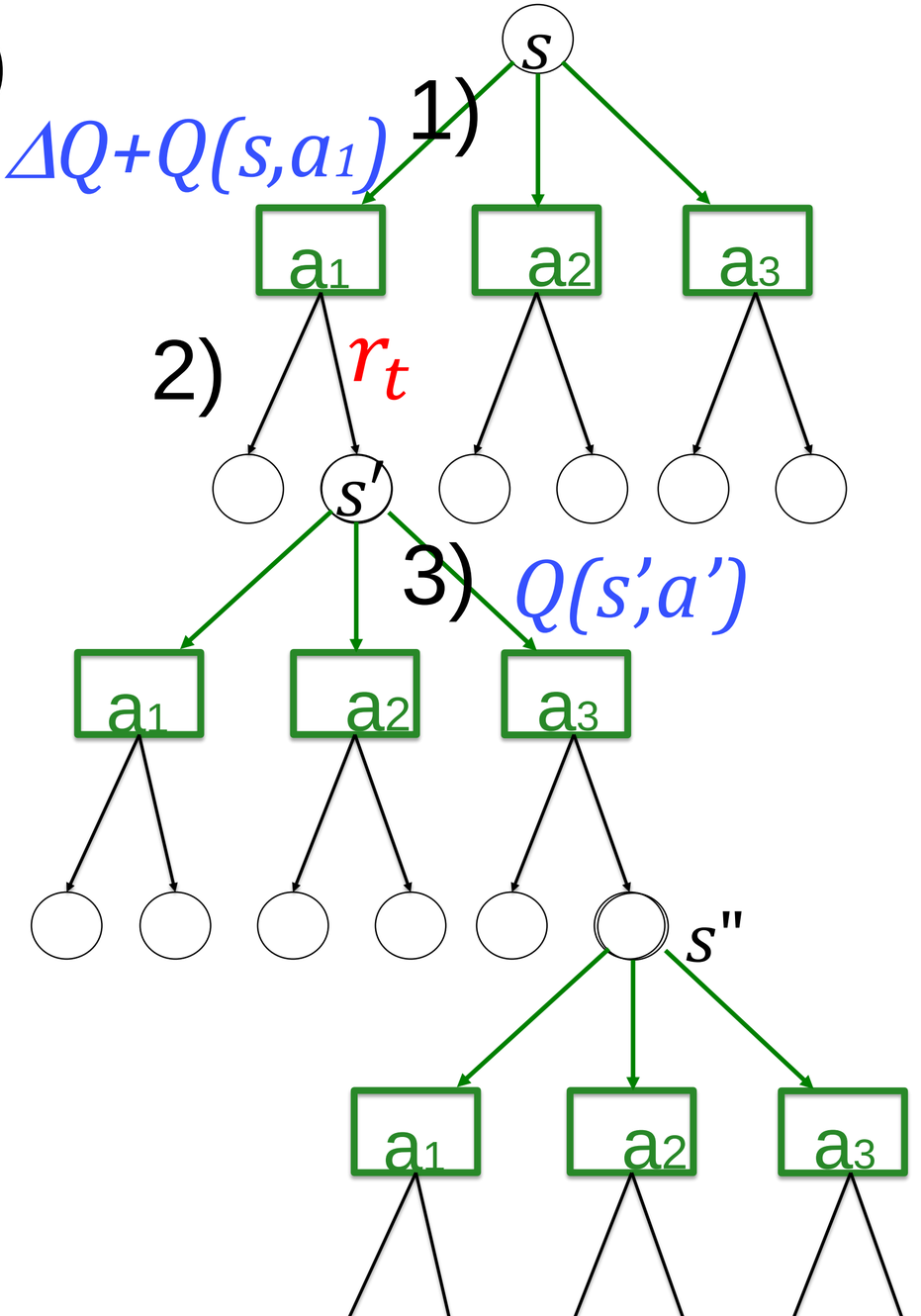
Initialise Q values
Start from initial state **s**

1) being in state **s**
   choose action **a**
   [according to policy $\pi(s,a)$]
2) Observe reward **r**
   and next state **s'**
3) Choose action **a'** in state s'
   [according to policy $\pi(s',a')$]
4) Update with SARSA update rule

$$\Delta Q(s,a) = \eta\,[r_t + \gamma Q(s',a') - Q(s,a)]$$

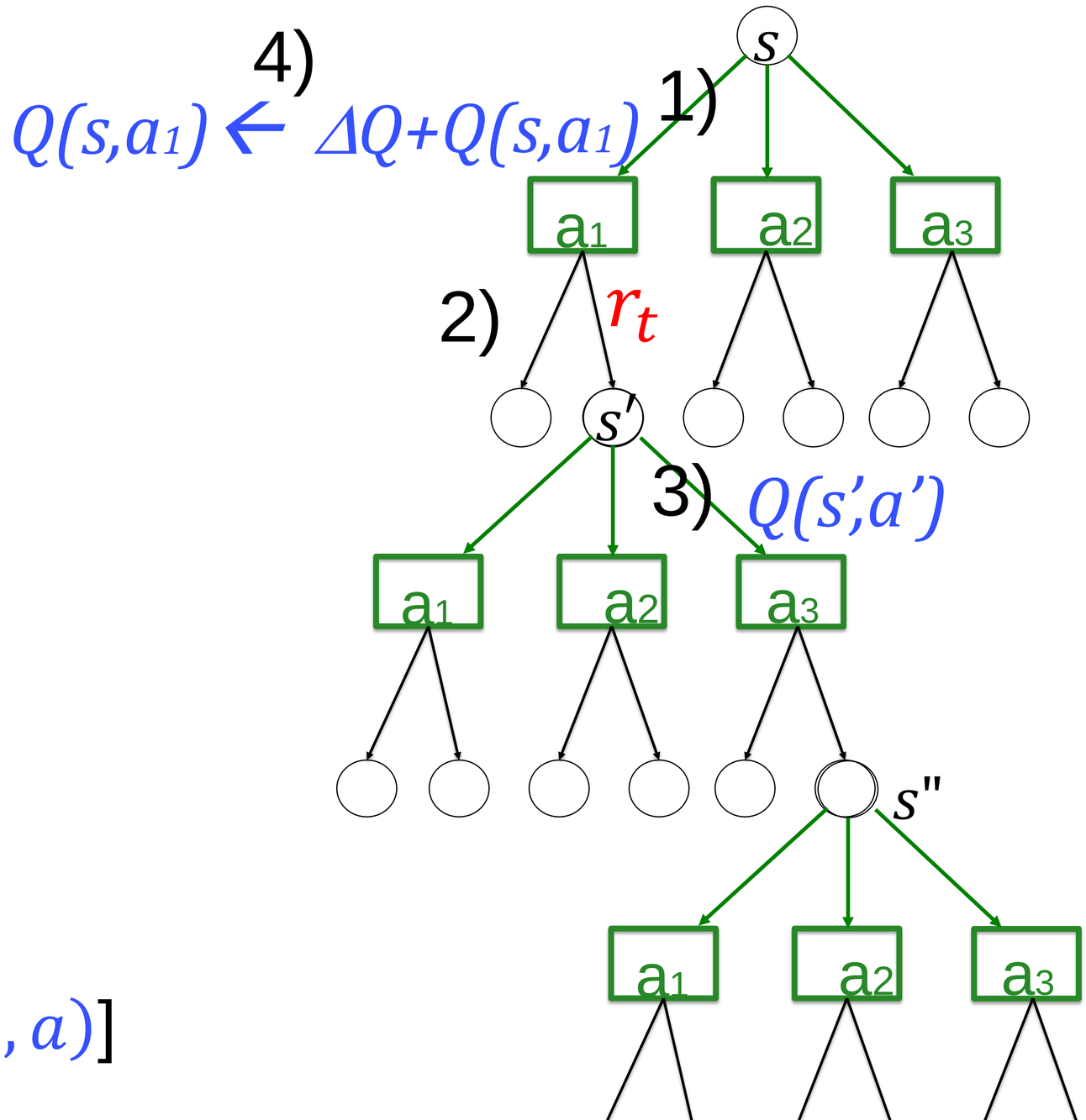
5) set: s ← s'; a ← a'
6) Goto 2)

Stop if all Q-values have converged (some criterion)



(previous slide)
The SARSA update in step 4 implements the idea that the immediate reward must account for the difference in Q-values between neighboring states.

Blackboard 1: Backup diagram



SARSA update step

$$\Delta Q(s,a) = \eta\left[r_t + \gamma Q(s',a') - Q(s,a)\right]$$

(previous slide)
The backup diagram describes how many states and actions the algorithm has to keep in memory so as to enable the next update step.

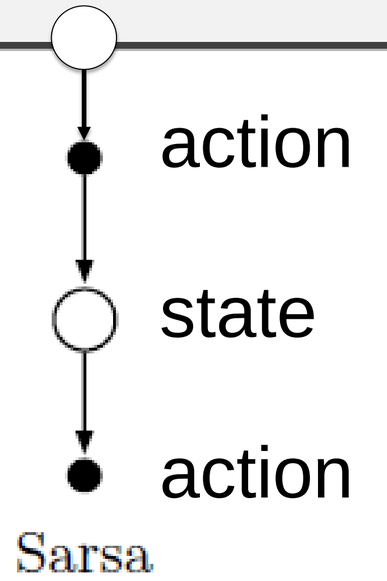
In SARSA, when you are in (s’,a’) you need to go back to the branch (s,a) so that you can do the SARSA update.

Summary: SARSA algorithm and Backup Diagram

Sarsa (on-policy) for estimating Q

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, 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., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-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

In algo: r_t is called R



Sarsa

pick next action a' before you update

Sutton and Barto, Ch. 6.4

(previous slide)
In SARSA, we can update Q(s,a), once we have seen the next state s' and the next action a'. In other words, the current action is a' and we had to keep the most recent state s' and the earlier 'branch' characterized by action a in memory.

Note: I would argue that we also need to keep the earlier state s in memory because you update Q(s,a) and not Q(a); therefore you need to know the full state action pair (s,a)! -- But Sutton and Barto use a slightly different convention and that is the one we follow here.

The backup diagrams play a role in the following for the analysis of other algorithms.

Notation in pseudo-algo (difference of the book of Sutton and Barto to lecture)
1.I simply write r_t for the actual reward at time t, and s, s' and a, a,' for the states and actions, respectively. In their book Sutton and Barto introduce in the Pseudocode dummy variables R,S,A, that take the role of place holders for the observed rewards, states, and actions.
2. I often call the learning rate η; Sutton and Barto call it α.

Reinforcement Learning Lecture 2

Variants of TD-learning methods and eligibility traces

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Part 2: Variations of SARSA

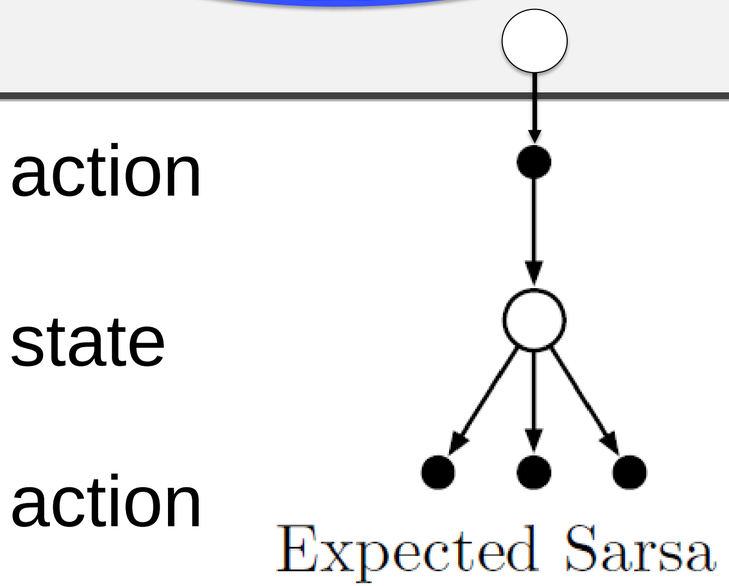
1. Review and introduction of BackUp diagrams
2. **Variations of SARSA**

(previous slide)
SARSA is one example of a whole family of algorithms that all look very similar.

Expected SARSA

Expected SARSA for estimating Q

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, 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., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\{R + \gamma[\sum_{\tilde{a}} \pi(S', \tilde{a})Q(S', \tilde{a})] - Q(S,A)\}$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal



Sutton and Barto, Ch. 6.6

(previous slide)
The first variant is ‘Expected SARSA’.
In standard SARSA, we pick the next action a’ and actually take it, before the updata of Q(s,a) is done.
In expected SARSA, the update rule averages over all possible next action with a weight given by the policy π.
The actual next action is chosen according to the policy.

Bellman equation for greedy policy

$$Q(s,a) = \sum_{s'} P^{a}_{s\to s'}\left[R^{a}_{s\to s'} + \gamma \sum_{a'} \pi(s',a')Q(s',a') \right]$$

Bellman equation = value consistency of neighboring states

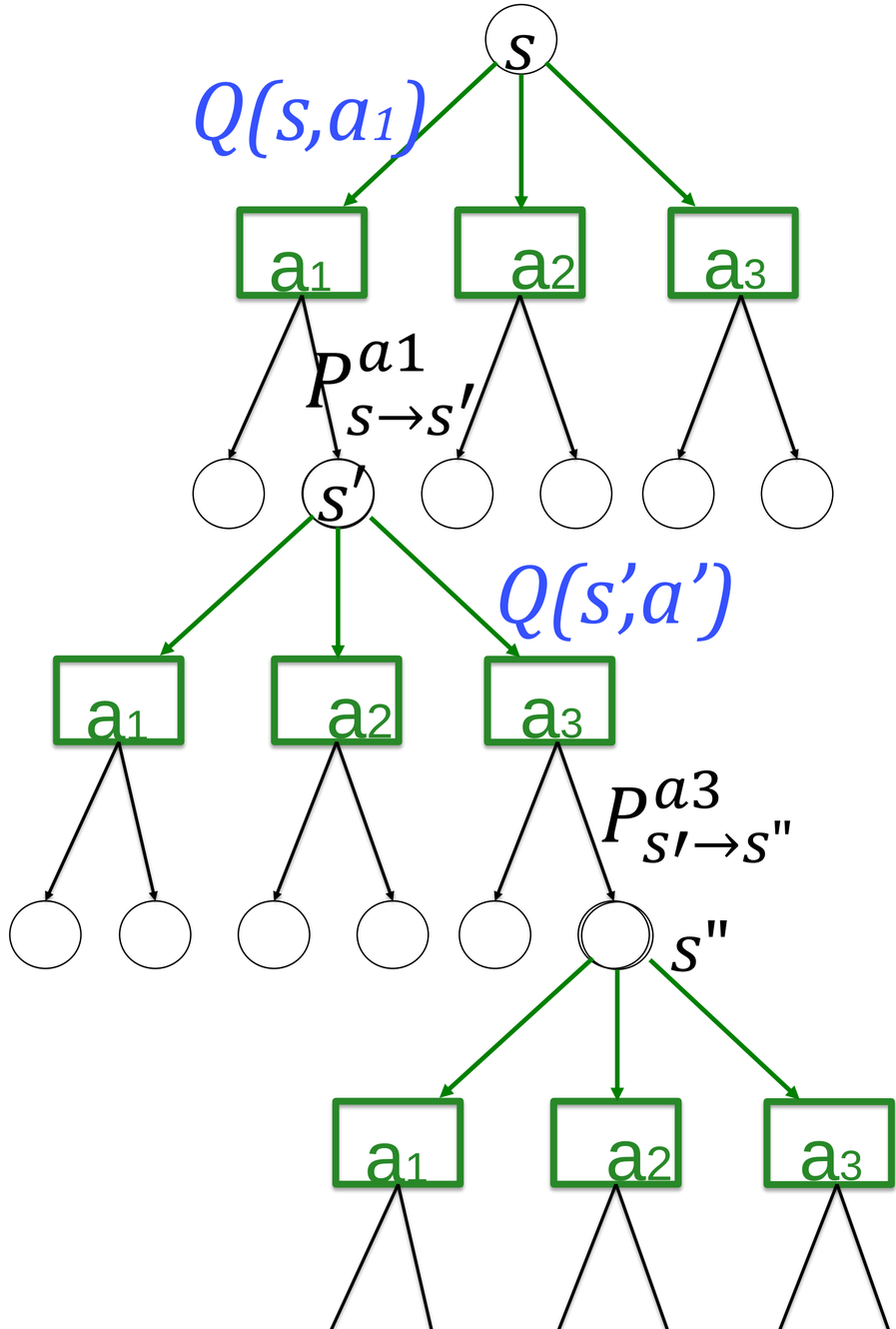
Remark:

Sometimes Bellman equation is written for greedy policy:

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

with action

$$a^* = \max_{a'} Q(s,a')$$



(previous slide)
The next variant is Q-learning.
Q-learning uses not an average with the current policy, but performs the averaging with the best policy, i.e., the greedy policy.

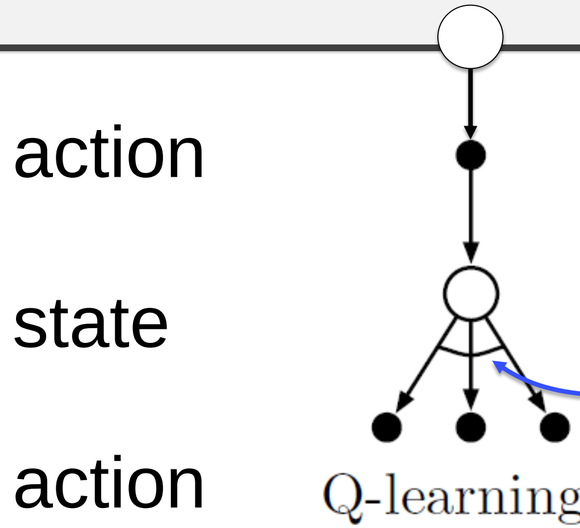
The idea is that you **run a stochastic policy that includes exploration** and visits all state-action pairs. However, since you plan to use **after learning the greedy policy** so as to maximize your returns, you already update the Q-values according the greedy policy.

Since the current policy and the update scheme differ, Q-learning is called ‘off-policy’.

Q-Learning algorithm

Q-learning (off-policy TD control)

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



max operation

Sutton and Barto, Ch. 6.5

(previous slide)
Q-learning is called ‘off-policy’ because you update as if you used a greedy policy whereas during learning you are really running a different policy (such as epsilon-greedy): it is as if you turn-off the current policy during the update.

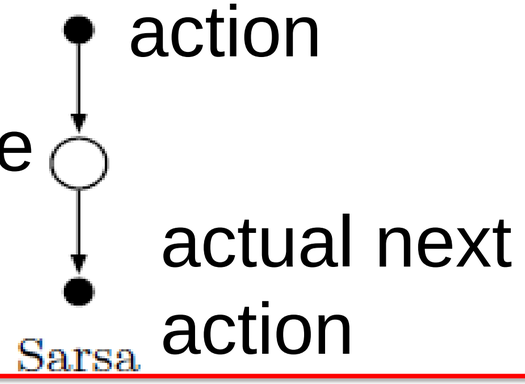
In Q-learning the update step is such that the current reward should explain the difference between Q(s,a) and the **maximum** Q(s’,a’) running over all possible actions a’. It is a TD algorithm (Temporal Difference), because neighboring states are visited one after the other. Hence neighbors are one time step away.

It does not play a role which action a’ you actually choose (according the your current policy). The max-operation is indicated in the back-up diagram by the little arc.

On-policy versus Off-policy algorithm:

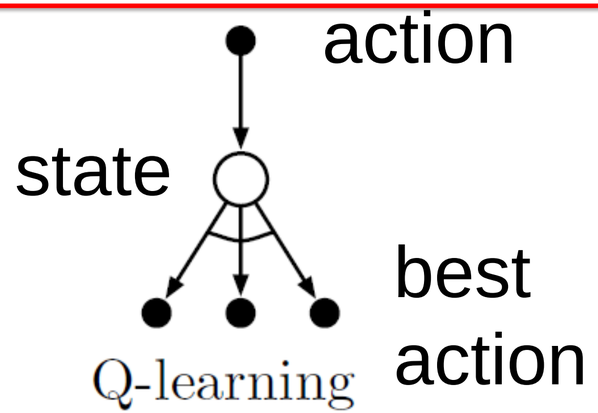
action

ON-POLICY: action a' in update rule is the REAL action



SARSA: you actually perform **next** action,
according to the policy,
and then you update Q(s,a)

OFF-POLICY: action a' in update rule is DIFFERENT from real one

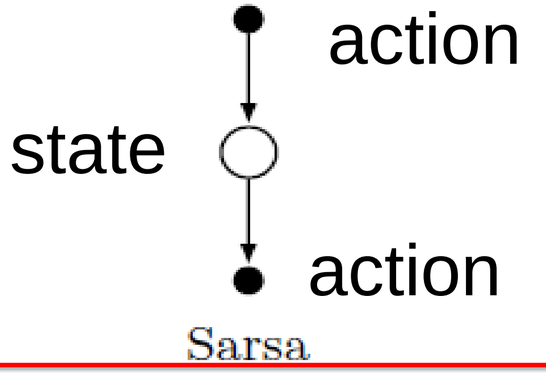


Q-learning: you look ahead and **imagine**
a greedy next action to update Q(s,a)
(but you then perform the actual next action
based on your current policy)

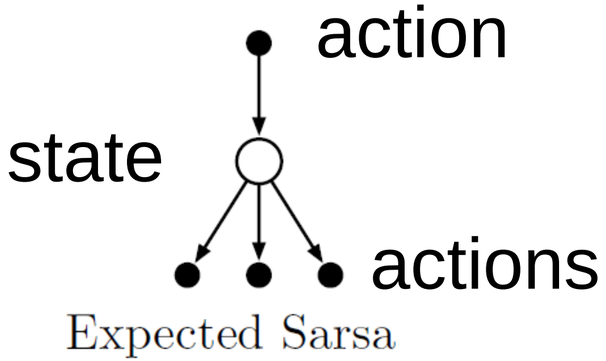
(previous slide)
On-policy versus OFF-policy.

Your real actions are chosen according to the policy (in both cases!).
But in OFF-policy algos this policy is NOT the one used for the update rule.

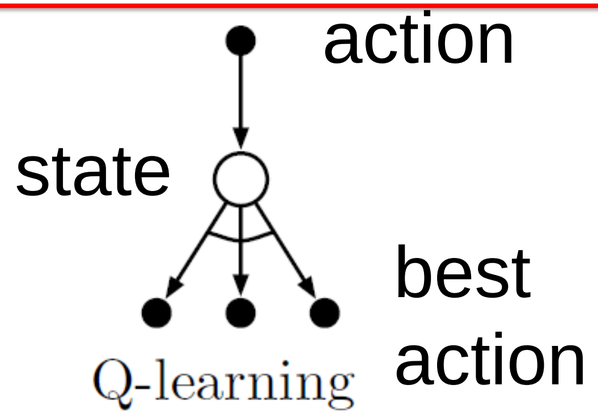
Summary: SARSA and related algorithms



SARSA: you actual perform **next** action,
according to the policy,
and then you update Q(s,a)



Exp. SARSA: you look ahead and average
over **potential next** actions
and then you update Q(s,a)



Q-learning: you look ahead and **imagine**
greedy next action to update Q(s,a)
(but you then perform the actual next action
based on your current policy)

(previous slide)
Summary of the three variations of SARSA and their back-up diagrams.

Reinforcement Learning Lecture 2

Variants of TD-learning methods and eligibility traces

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Part 3: Temporal Difference Learning

1. Review and introduction of BackUp diagrams
2. Variations of SARSA
3. **TD Learning (Temporal Difference)**

(previous slide)
The 3 algorithms in the previous section are examples of TD-algorithms (Temporal Difference). We now explain the term and also explore other Temporal Difference algorithms (TD-algos).
One special case of TD-algos is TD-learning (in the narrow sense).

TD-learning as bootstrap estimation

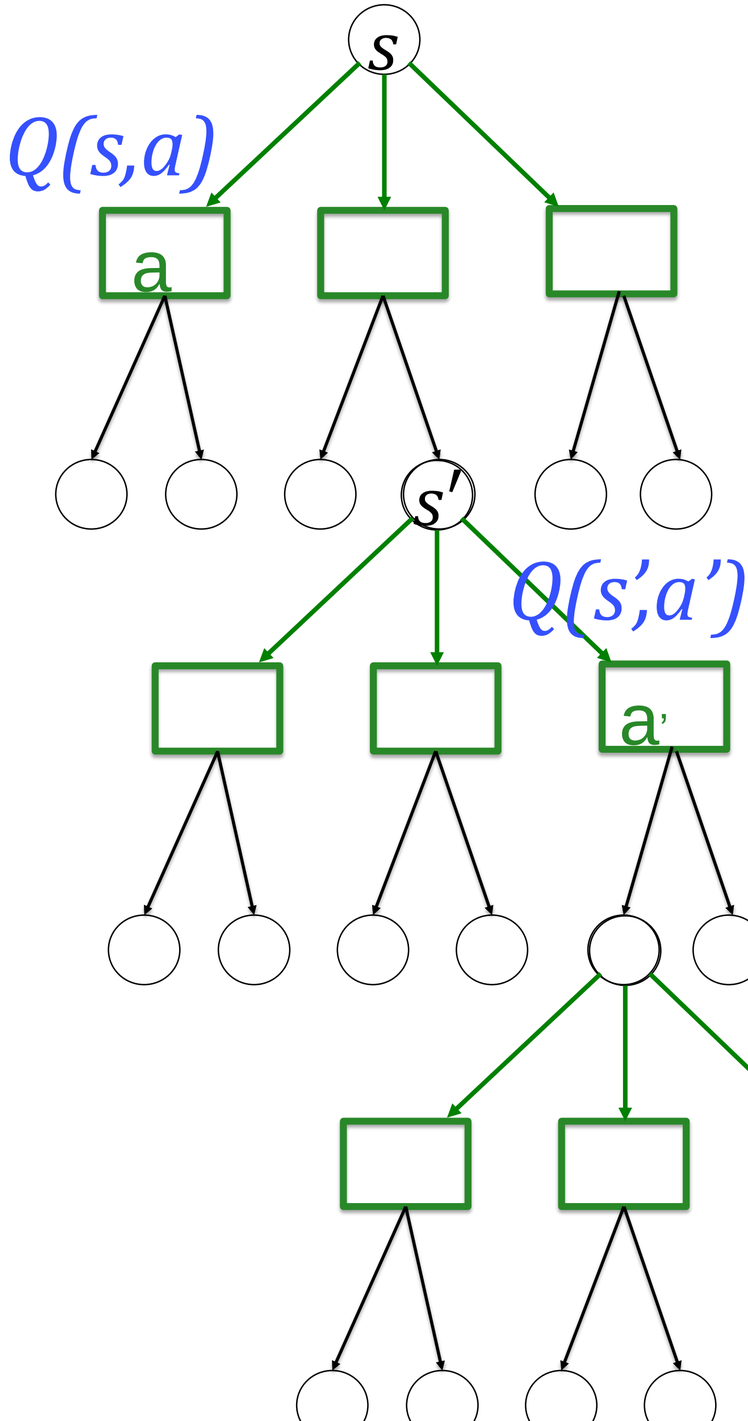
'bootstrap': summary of previous information

Temporal Difference

$$Q(s,a) = \sum_{s'} P^{a}_{s \to s'} \left[R^{a}_{s \to s'} + \gamma \sum_{a'} \pi(s',a') Q(s',a') \right]$$

Bellman equation = value consistency of neighboring states

Neighboring states → neighboring time steps

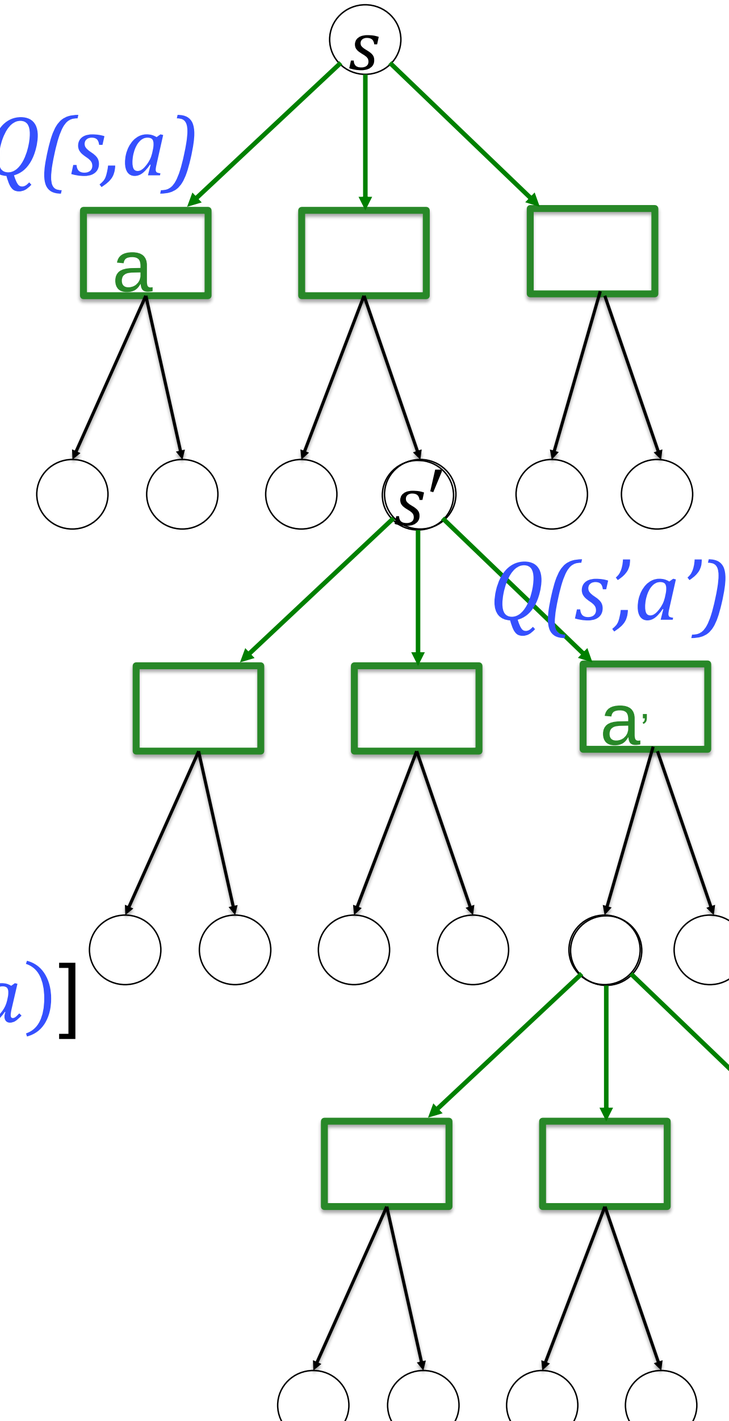


(previous slide)
1) If the agent runs through the state-action graph, neighboring states are one **time step** away from each other. This explains the term ‘Temporal Difference (TD)’

2) As mentioned before:
The Q-value Q(s,a) further up in the graph is the expected total discounted reward – summed over all possible future actions and states.
It can be decomposed in an average over the **immediate** rewards, actions, and states, and the Q-values Q(s’,a’) of all possible next states. Since calculation of Q(s,a) relies on (earlier) calculation of Q(s’,a’), Sutton and Barto call this a ‘bootstrap’ algorithm.

TD-learning in the general sense

Whenever an algorithm makes update steps based on a comparison of Q-values across neighboring states, it is a TD-method.



SARSA

$$\Delta Q(s,a) = \eta[r_t + \gamma Q(s',a') - Q(s,a)]$$

Expected SARSA

$$\Delta Q(s,a) = \eta[r_t + \gamma\{\sum_{a'^a}\pi(s',a')Q(s',a')\} - Q(s,a)]$$

Q-learning

$$\Delta Q(s,a) = \eta[r_t + \gamma \max Q(s',a') - Q(s,a)]$$

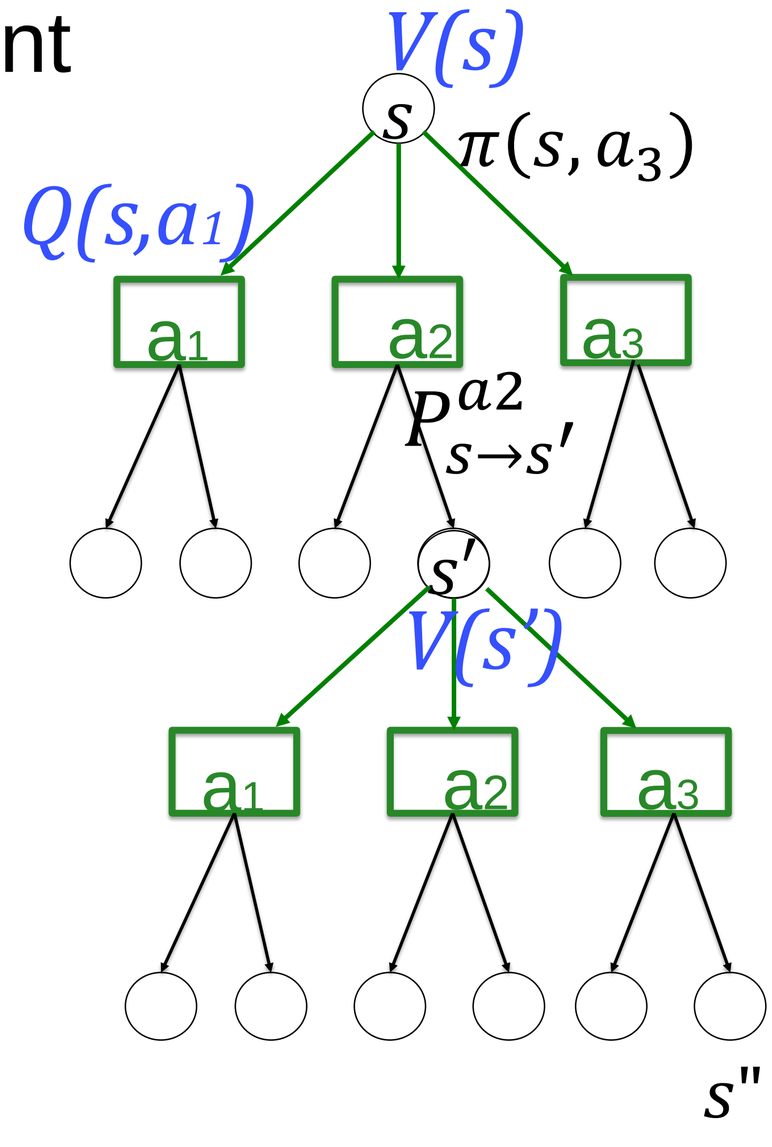
(previous slide)
Even more generally: Whenever an algorithm compares Q-values or V-values of neighboring states, it is a TD-method.

V-values will be discussed on the next slide!

State-values V

Value $V(s)$ of a state s
= total (discounted) expected reward the agent gets starting from state s

$$V(s) = \sum_a \pi(s,a) Q(s,a)$$



Bellman equation for *V(s)*

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

(previous slide)
Instead of working with Q-values, we can work with V-values that describe the value of a state (as opposed to the value of a state-action pair). The lecture of Dr. Brea on Markov Decision Problems also used V-values.

While each Q-value is associated with a state-action pair, V-values are the value of a state: V-values are defined as the expected total discounted reward that the agent will collect under policy π starting at that state.

The value of a state V(s) is the average over the Q-values Q(s,a) averaged over all possible actions that start from that state. The correct weighting factor for averaging is given by the policy π(s,a).

$$V(s) = \sum_{a} \pi(s,a) Q(s,a)$$

The resulting Bellman equation for V-values looks similar to that of Q-values, except that the location of the summation signs has been shifted.

Standard TD-learning

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0$, for all $s \in \mathcal{S}^+$)
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\left[R + \gamma V(S') - V(S)\right]$
    $S \leftarrow S'$
  until S is terminal

r_t is called R

TD-learning in the narrow sense

state

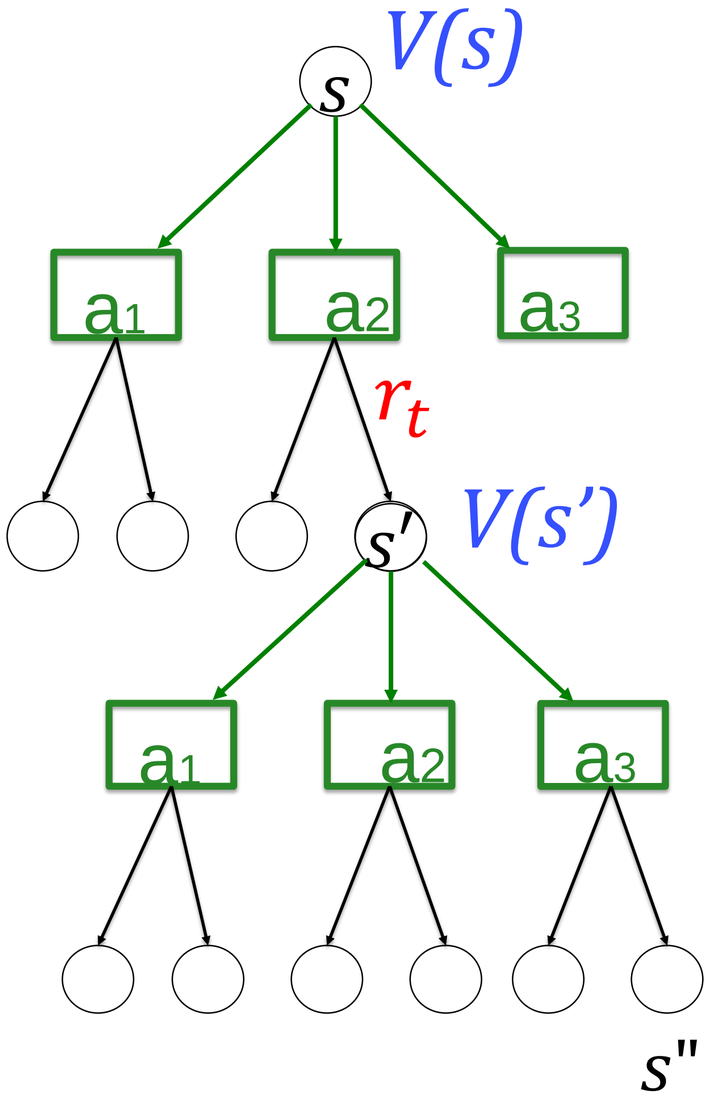
action

state

TD(0)

$$\Delta V(s) = \quad [r_t + \gamma V(s') - V(s)]$$

$$\Delta V(s) = \eta\,[r_t + \gamma V(s') - V(s)]$$



(previous slide)
The iterative update for V-values is analogous to that of Q-values, but the back-up diagram looks different. Once the agent is in the next state s', you can update the value V(s).
The resulting update rule is called TD learning (in the narrow sense).

In the broader sense, a large class of algorithms that exploits the Bellman equation for approximate iterative update rules is called Temporal Difference Learning (TD) or simply TD-methods:
Whenever an algorithm compares Q-values or V-values of neighboring states, it is a TD-method.

The zero in the argument of TD(0) becomes clear later.

Summary: TD-learning as bootstrap estimation

$$Q(s,a) = \sum_{s'} P^{a}_{s \to s'} \left[R^{a}_{s \to s'} + \gamma \sum_{a'} \pi(s',a') Q(s',a') \right]$$

Bellman equation = value consistency of
neighboring states

Neighboring states → neighboring time steps

Temporal Difference Methods (TD methods)

- explore graph over time
- compare values (Q-values or V-values)
  at **neighboring time steps**
- **'bootstrap'** estimation of values
- update after next time step, based on 'temporal difference'

(previous slide)
Summary – add your own comments. All terms should be clear by now.

Quiz: TD methods in Reinforcement Learning

[ ] SARSA is a TD method
[ ] expected SARSA is a TD method
[ ] Q-learning is a TD method
[ ] TD learning (in the narrow sense) is an on-policy TD method
[ ] Q-learning is an on-policy TD method
[ ] SARSA is an on-policy TD method

(previous slide)
This quiz applies a few definitions to a few algorithms.

Reinforcement Learning Lecture 2

Variants of TD-learning methods and eligibility traces

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Part 4: Monte-Carlo Methods

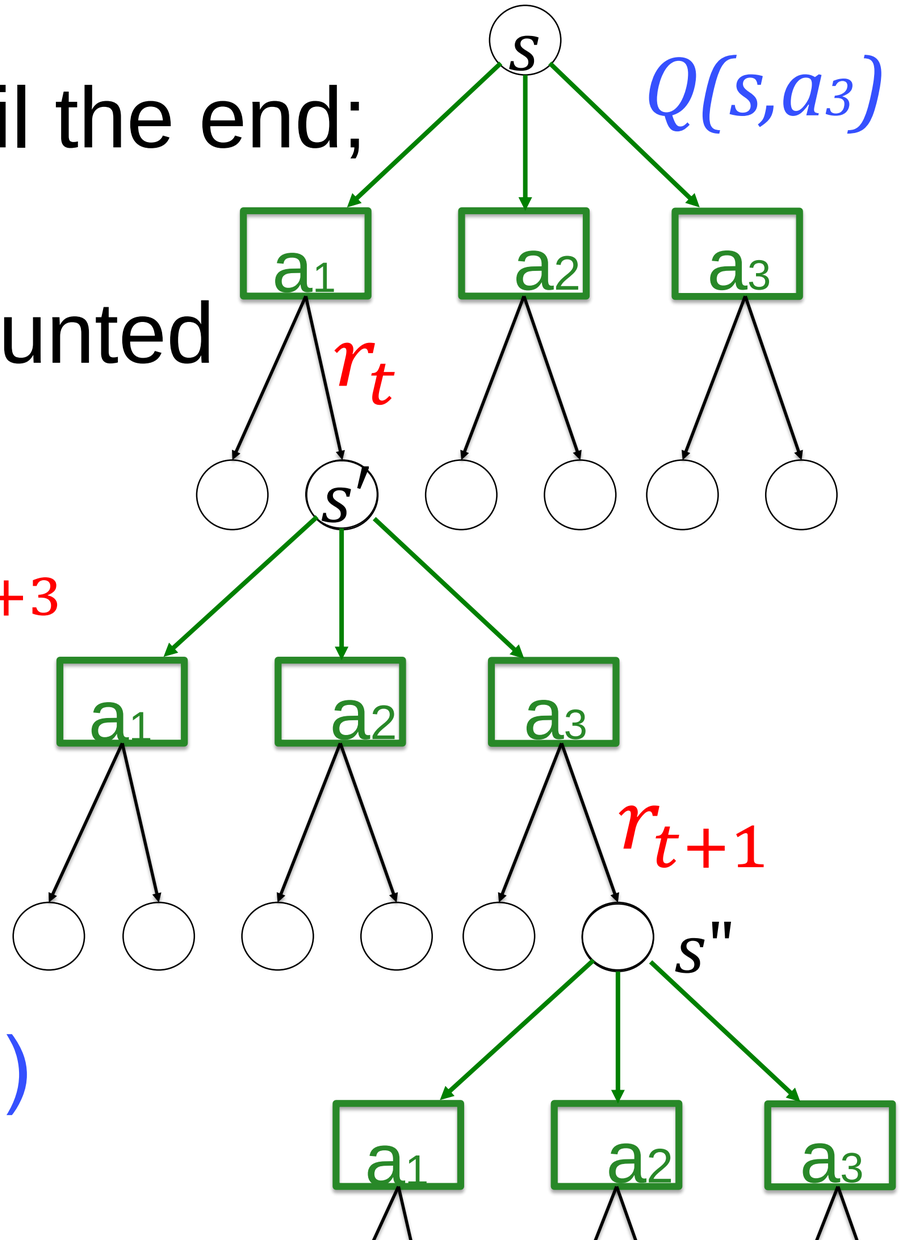
1. Review and introduction of BackUp diagrams
2. Variations of SARSA
3. TD Learning (Temporal Difference)
4. **Monte-Carlo Methods**

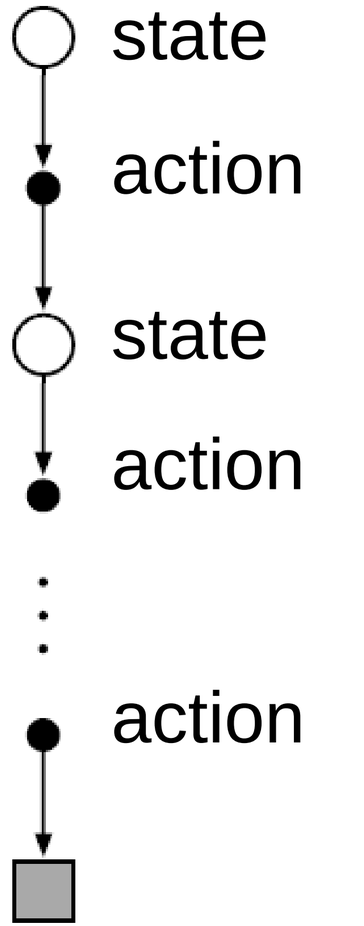
(previous slide)
Instead of using TD methods, the same state-action graph can also be explored with Monte-Carlo methods

Monte-Carlo Estimation (for Q-values)

play a trial (episode) until the end;
then update, using the
total accumulated discounted
reward **(='Return') =**

$r_t + \gamma\, r_{t+1} + \gamma^2 r_{t+2} + \gamma^3\, r_{t+3}$





end of trial

same episode is also
used to estimate Q(s',a')
of children state

(previous slide)
1) Suppose you want to estimate the value Q(s,a) of state-action pair (s,a).
Q(s,a) is the EXPECTED total discounted reward.

To estimate Q(s,a)) you start in state s with action a, run until the end and evaluate for this single episode the return defined as

$$Return\,(s,a) = r_t + \gamma\, r_{t+1} + \gamma^2 r_{t+2} + \gamma^3\, r_{t+3}$$

This is a single episode. If you start several times in (s,a), you get a Monte-Carlo estimate of Q(s,a).

2) You can be smart and you the SAME episode also to estimate the value $Q(s',a')$ of other states s'. Thus while you move along the graph, you open an estimation variable for each of the state-action pairs that you encounter.

Combining points 1) and 2) gives rise to the following algorithm.

Monte-Carlo Estimation of Q-values (batch)

Start at a random state-action pair (s,a) (exploring starts)

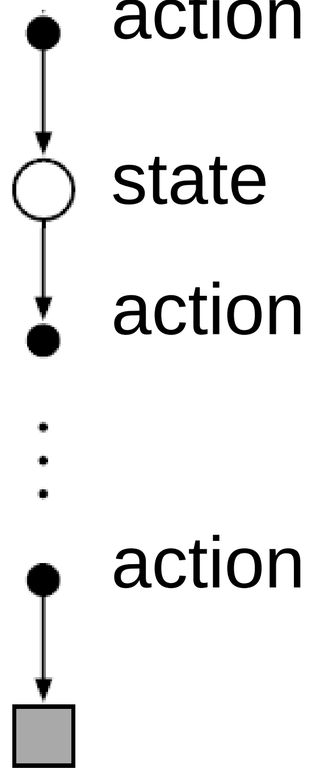
$$Return(s,a) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \ldots$$

Monte Carlo ES (Exploring Starts),

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
- $Q(s,a) \leftarrow$ arbitrary
- $\pi(s) \leftarrow$ arbitrary
- $Returns(s,a) \leftarrow$ empty list

ε-greedy is good policy

Repeat forever:
- Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ s.t. all pairs have probability > 0
- Generate an episode starting from S_0, A_0, following π
- For each pair s,a appearing in the episode:
  - $G \leftarrow$ the return that follows the first occurrence of s,a
  - Append G to $Returns(s,a)$
  - $Q(s,a) \leftarrow$ average$(Returns(s,a))$



$$Q(s,a) = average[Return(s,a)]$$

Note: single episode also allows to update Q(s'a') of children

(previous slide)
In this (version of the) algorithm you first initialize Q(s,a) and Return(s,a) for all state-action pairs.

For each state s that you encounter, you observe the (discounted) rewards that you accumulate until the end of the episode. The total accumulated discounted reward starting from (s,a) is the ‘Return(s,a)’

After many episode you estimate the Q-values Q(s,a) as the average over the Returns(s,a).

Note that
- stochasticity in the initial states assures that all pairs (s,a) are tested, even if the policy is not stochastic.
- In theory, this estimation method is hence compatible with a greedy policy.
- In practice, I always recommend epsilon greedy (and we can reduce epsilon as we have learned more and more).

Quiz: Monte Carlo methods

We have a network with 1000 states and 4 action choices in each state. There is a single terminal state.
We do Monte-Carlo estimates of total return to estimate **Q-values Q(s,a).**

Our episode starts with (s,a) that is 400 steps away from the terminal state. How many return $R(s,a)$ variables do I have to open in this episode?

[ ] one, i.e. the one for the starting configuration (s,a)
[ ] about 100 to 400
[ ] about 400 to 4000
[ ] potentially even more than 4000

(previous slide) your notes.

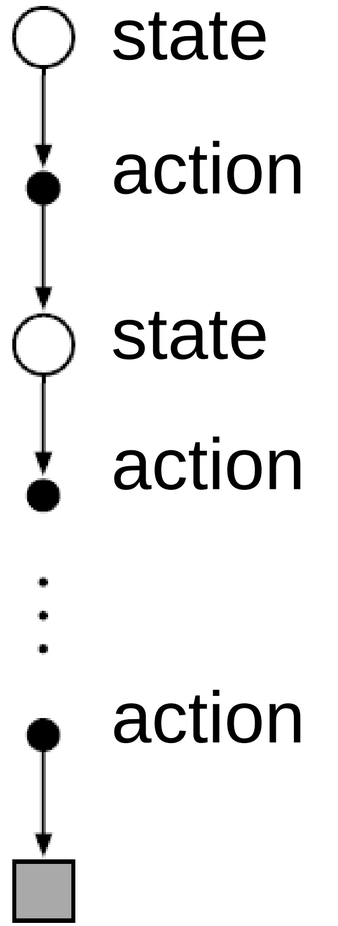
Monte-Carlo Estimation of V-values

$$Return(s) = r_t + \gamma\, r_{t+1} + \gamma^2 r_{t+2} + \gamma^3\, r_{t+3}$$

First-visit MC prediction, for estimating V

Initialize:
- $\pi \leftarrow$ policy to be evaluated
- $V \leftarrow$ an arbitrary state-value function
- $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:
- Generate an episode using π
- For each state s appearing in the episode:
  - $G \leftarrow$ the return that follows the first occurrence of s
  - Append G to $Returns(s)$
  - $V(s) \leftarrow$ average$(Returns(s))$



single episode starting in state s0 also allows to update V(s) of children states

(previous slide, not shown in class). Instead of Q-values, we can also use Monte-Carlo estimates for V-values
In this (version of the) algorithm you first open V-estimators for all states.

For each state s that you encounter, you observe the (discounted) rewards that you accumulate until the end of the episode. The total accumulated discounted reward starting from s is the ‘Return(s)’

After many episode you estimate the V-values V(s) as the average over the Returns(s).

Note that the above estimations are done in parallel for all states s that you encounter on your path.

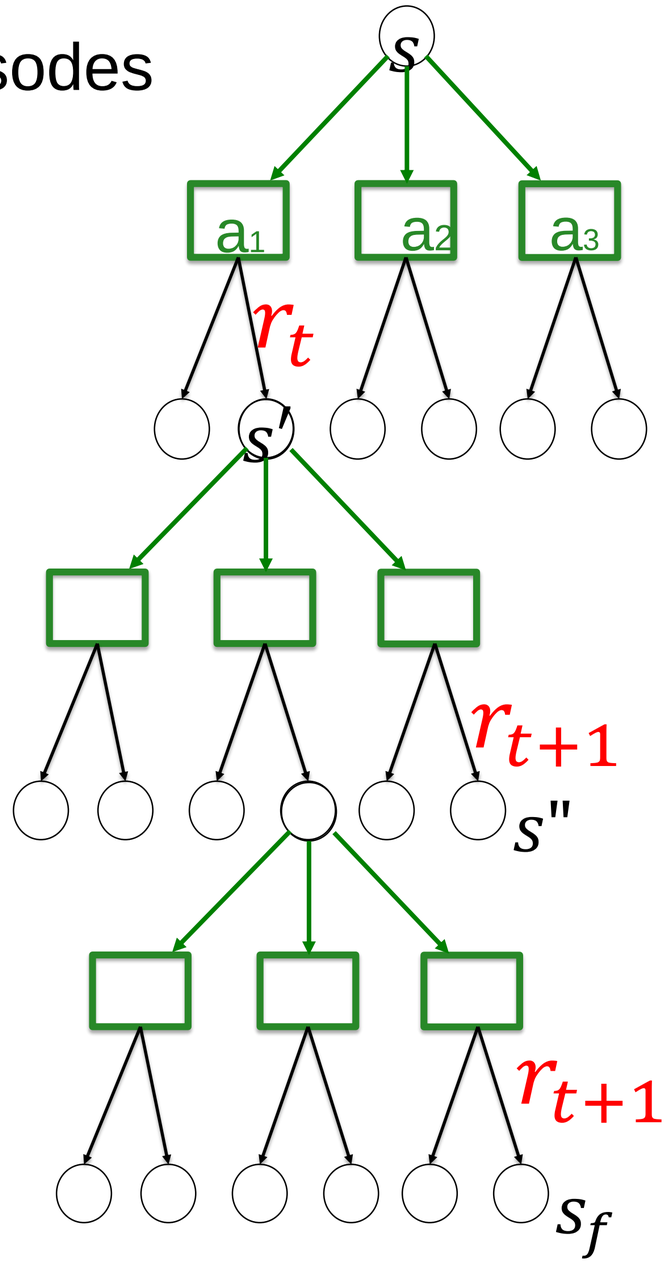
Also note that the Backup diagram is much deeper than that of Q-learning, since you always continue until the end of the trial before you can update Q-values of state-action pairs that have been encountered many steps before.

Batch-expected SARSA: solving Bellman step by step

Bellman: use all the available information after N episodes

$$Q(s,a) = \sum_{s'} P^{a}_{s \to s'} \left[R^{a}_{s \to s'} + \gamma \sum_{a'} \pi(s',a') Q(s',a') \right]$$

known



Conditions:

- directed graph,
- fixed policy
- N episodes played

(previous slide, not shown in class)

Alternatively, if you have a directed graph, the Bellman equation can also be used as in dynamic programming: starting from the bottom leaves of the graph (end of episodes, terminal state=set of final states s_f) you walk upward and find Q-values step by step. You know your policy, so it is similar to expected SARSA, except that you work in ‘batch’ mode. I call this batch-expected SARSA.

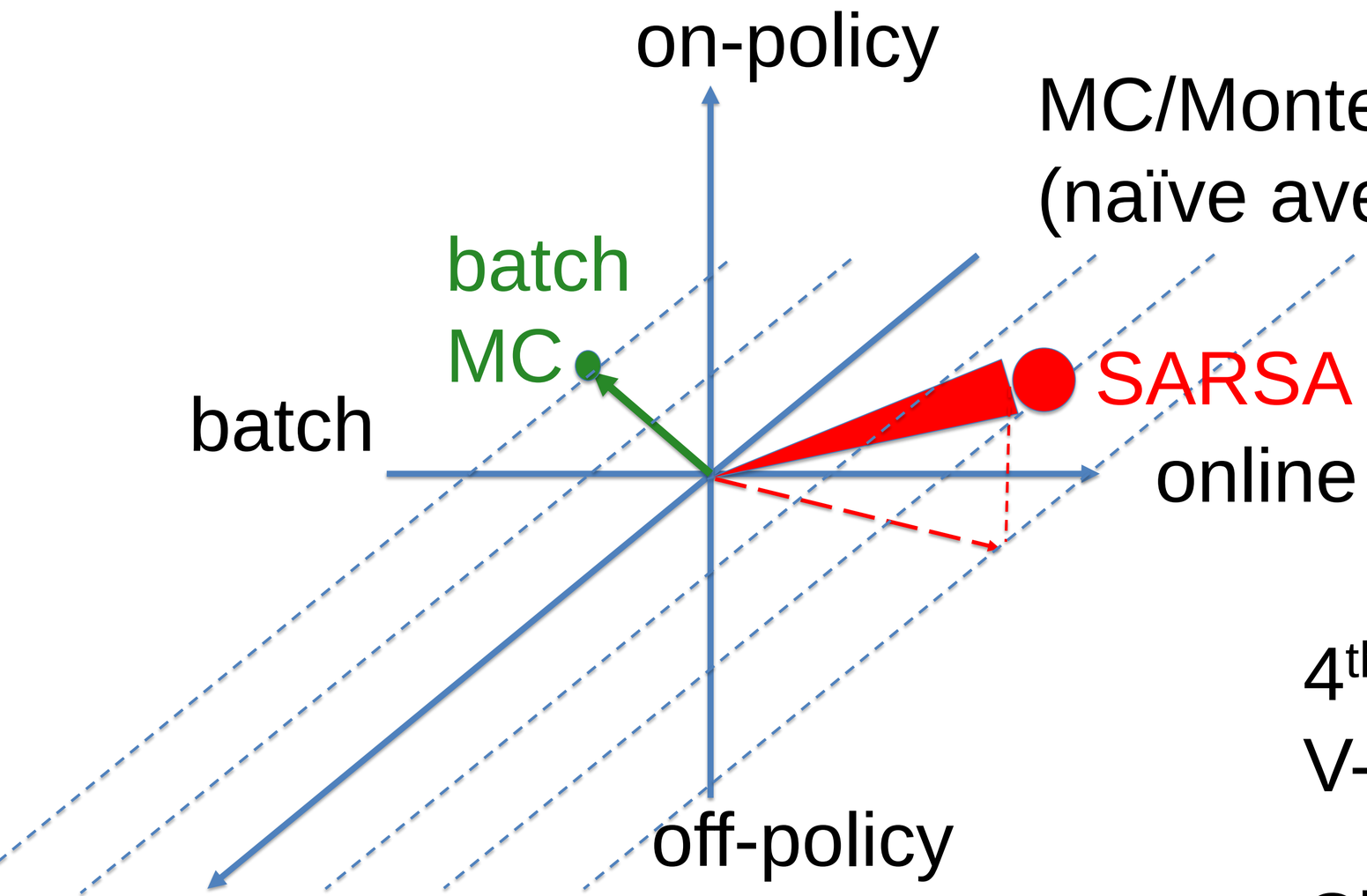
It is still an empirical estimation, since the rewards and the transitions need to be estimated from the episodes that have been played.

$$Q(s,a) = \{\langle r_t \rangle + \gamma \left\langle \sum_{a\prime} \pi(s',a')\, Q(s',a') \right\rangle\}$$

The first brackets: empirical estimate over immediate rewards.
The second brackets: empirical estimate over next states s’.

And now we ask: is this a good algorithm?? Else which of the previous ones is better?

“Oh, so many, many variants …”



4th axis:
V-value vs Q-value

Classify/locate algorithms

(previous slide)
There are many variants of algorithms.
We can organize these across three axes.
1) Batch versus online;
2) off-policy versus on-policy;
3) Monte-Carlo versus TD.

Q-learning or SARSA both use ‘bootstrapping’ since they update Q-values based on other Q-values. All TD methods have this bootstrapping feature.
Q-learning has the max-operation in the update (and hence off-policy), whereas SARSA is ‘on-policy’. Both Q-learning and SARSA are **Online** (as opposed to batch).
In **batch** algorithms you have to play several episodes before you do the update. We considered Batch Monte Carlo. But one can also construct a Batch-Expected SARSA that is closely related to solution of the Bellman equation (hidden slides) and uses the idea of ‘bootstrapping’ - whereas Monte-Carlo does not.

A fourth axis for the classification could be whether we use V-values or Q-values.

“Oh, so many, many variants …”

Question:

Three ways to estimate Q-values with policy π:

1) SARSA (online, on-policy, TD, bootstrap)
2) Expected SARSA (online, TD, bootstrap)
3) Monte-Carlo (batch over many episodes, not bootstrap, not TD)

We have played N trials (N full episodes to terminal state)
How do the three algorithms rank?
Which one is best? → commitment:

write down 1 or 2 or 3

(previous slide)
There are many variants of algorithms – but which one is the best?

To find out which one is best, consider the following example.

Monte-Carlo versus TD methods (Exercise *, preparation)

5 episodes, first action is always a1.

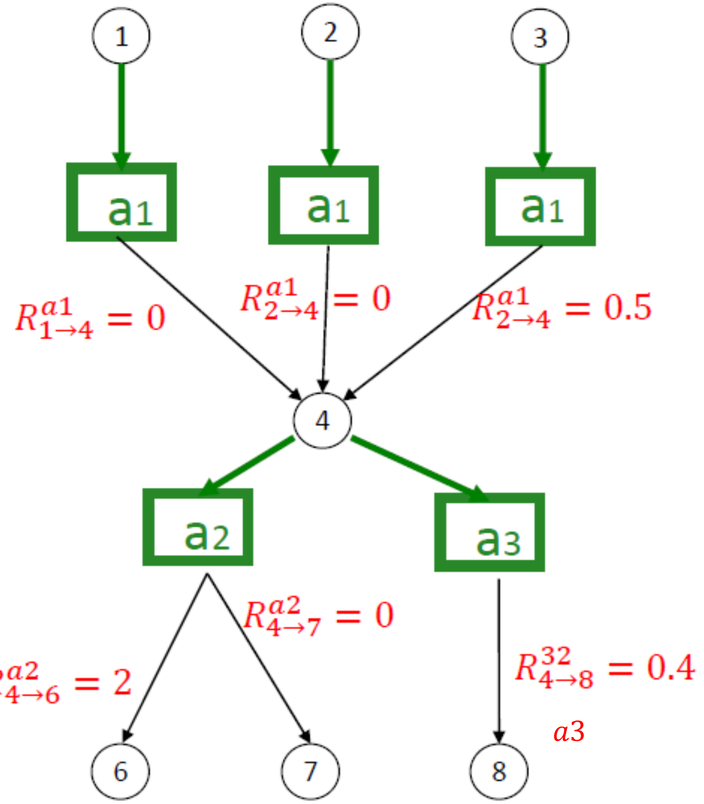
Episode 1: States 1-4-7 with action a2, Return=0

Episode 2: States 1-4-8 with action a3, Return=0.4

Episode 3: States 2-4-6 with action a2, Return=2

Episode 4: States 2-4-8 with action a3, Return=0.4

Episode 5: States 3-4-7 with action a2, Return=0.5



What is Q(s,a1) [with s=1,2,3] **after 5 trials, for two algorithms?**

(i) Monte-Carlo: average over total accumulated reward for given (a,s)

(ii) Expected SARSA –online updates after each step.
for each Q(s,a): first update step with rate $\eta_1=1$, second one with $\eta_2=1/3$

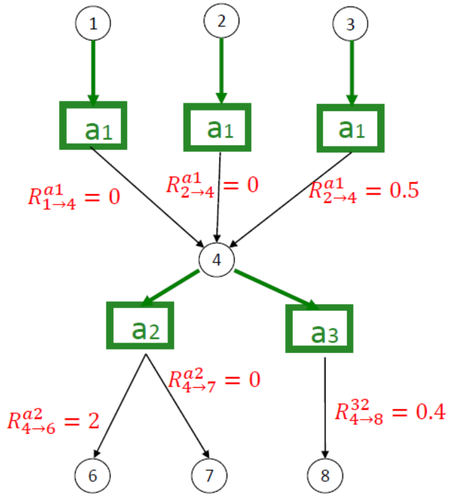
Episode 1: States 1-4-7 with action a2, Return=0

Episode 2: States 1-4-8 with action a3, Return=0.4

Episode 3: States 2-4-6 with action a2, Return=2

Episode 4: States 2-4-8 with action a3, Return=0.4

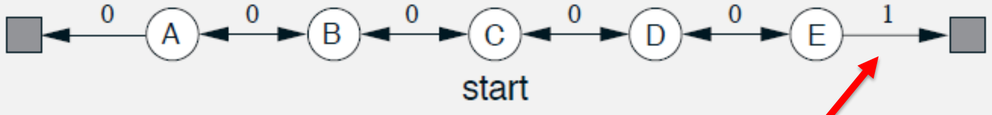
Episode 5: States 3-4-7 with action a2, Return=0.5



Monte-Carlo versus batch-TD methods/Bellman equation:

Comparison in **batch mode**: We have observed N episodes, and update (once) after these N episodes.

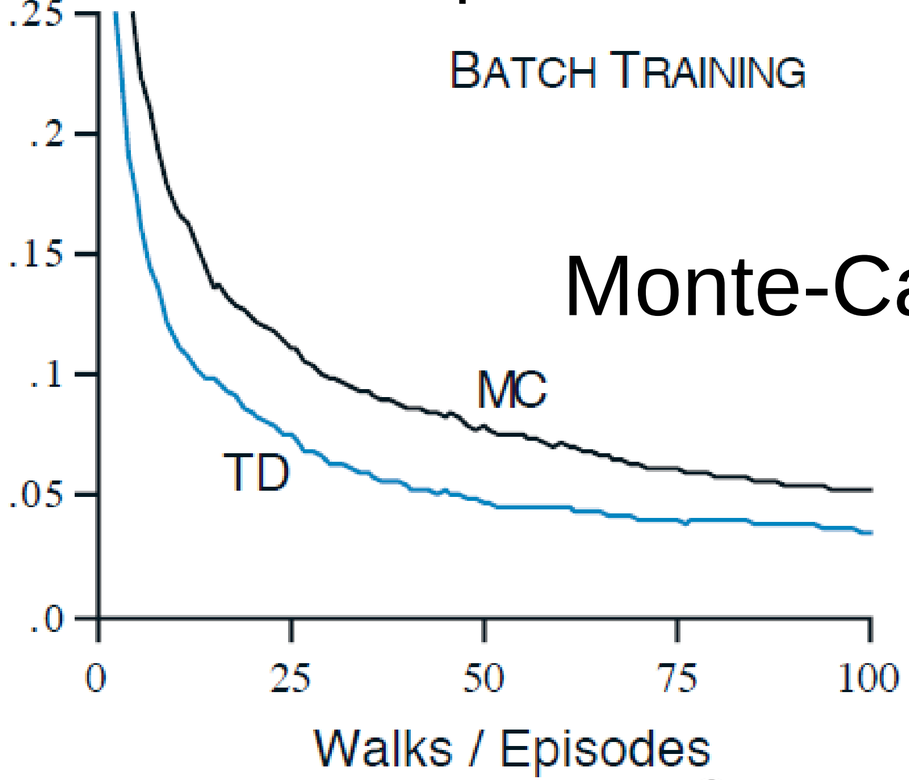
Example: 1d random walk



r=1

RMS error, averaged over states

error with respect to exact V-value



Monte-Carlo

Conclusion:
TD is better than
Monte Carlo

Sutton and Barto, 2018

Figure 6.2: Performance of TD(0) and constant-α MC under batch training on the random walk task.

(previous slide) All episodes start in the center state, C, then proceed either left or right by one state on each step, with equal probability (random walk). Episodes terminate either on the extreme left (reward zero) or the extreme right, (reward 1); all other rewards are zero.

Because we do not discount future rewards, the true value of each state V(s) can be calculated as, from A through E, 1/6; 2/6; 3/6; 4/6; 5/6.

The root-mean-square error (RMS) compares the estimated value with the above ‘true’ values V(s).
We see that TD performs better than MC in this case.

Sutton and Barto, 2018

Summary: Monte-Carlo versus TD methods

Exploiting Bellman: TD is better than Monte Carlo

The averaging step in TD methods ('bootstrap') is more efficient (compared to Monte Carlo methods) to propagate information back into the graph, since information from different starting states is combined and compressed in a Q-value or V-value.

(previous slide)
If we go back to the example: in Monte-Carlo methods you only exploit information of trials that go through the state-action pair (s,a) to evaluate Q(s,a); in TD methods (or with the Bellman equation) you compare Q(s,a) with Q(s’,a’) and all trials that pass through (s’,a’) contribute to estimate Q(s’,a’) even those that have started somewhere else and have never passed through (s,a). Hence in the latter case you exploit more information.

Note that in the explicit example above we compared a batch-expected-SARSA with Monte-Carlo. However, true online TD learning (such as SARSA or Q-learning) is also slow to converge, but for a different reason, as explained in the next section.

Monte-Carlo Estimation of Q-values (on-policy)

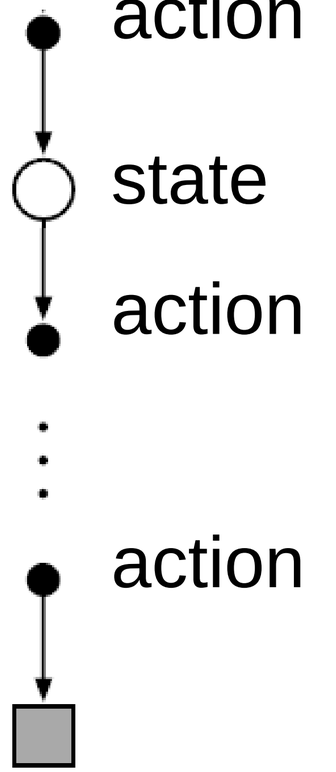
Combine epsilon-greedy policy with Monte-Carlo Q-estimates

On-policy first-visit MC control (for ε-soft policies), [redacted]

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
  $Q(s,a) \leftarrow$ arbitrary
  $Returns(s,a) \leftarrow$ empty list
  $\pi(a|s) \leftarrow$ an arbitrary ε-soft policy (e.g., epsilon-greedy)

Repeat forever:
  (a) Generate an episode using π
  (b) For each pair s,a appearing in the episode:
    $G \leftarrow$ the return that follows the first occurrence of s,a
    Append G to $Returns(s,a)$
    $Q(s,a) \leftarrow \text{average}(Returns(s,a))$
  (c) For each s in the episode:
    $A^* \leftarrow \arg\max_a Q(s,a)$ (with ties broken arbitrarily)
    For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq A^* \end{cases}$$



Q(s,a) = average[Return(s,a)]

Note: single episode also allows to update Q(s'a') of children

(previous slide/not shown in class/just as a reference)
This algorithm combines Monte-Carlo estimates with an epsilon-greedy policy.
Note for Monte-Carlo estimates, the agent waits until the end of the episode (end of trial), before it can update the Q-values.

Similar to the earlier Monte-Carlo algorithms, the Q-values of all those state-action pairs that have been visited in that trial are updated (as opposed to an algorithm where you would only update Q(s0,a0) of the initial state and action.)

Note that this is an on-policy algorithm because the epsilon-greedy policy is reflected in the final Q-values.

Quiz: Monte Carlo methods

We have a network with 1000 states and 4 action choices in each state. We use an epsilon-greedy policy.
There is a single terminal state.
We do Monte-Carlo estimates of total return to estimate **Q-values Q(s,a).**

Our episode starts with (s,a) that is 400 steps away from the terminal state. How many return $R(s,a)$ variables do I have to open in this episode?

[ ] one, i.e. the one for the starting configuration (s,a)
[ ] about 100 to 400
[ ] about 400 to 4000
[ ] potentially even more than 4000

Teaching monitoring – monitoring of understanding

[ ] today, up to here, at least 60% of material was new to me.

[ ] up to here, I have the feeling that I have been able to follow (at least) 80% of the lecture.

Reinforcement Learning Lecture 2

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Variants of TD-learning methods and eligibility traces

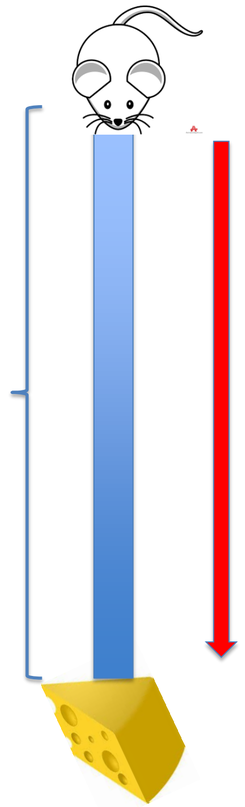
Part 5: Eligibility traces

1. Review and introduction of BackUp diagrams
2. Variations of SARSA
3. TD Learning (Temporal Difference)
4. Monte-Carlo Methods
5. **Eligibility traces**

(previous slide)
So far we have worked with discrete states.

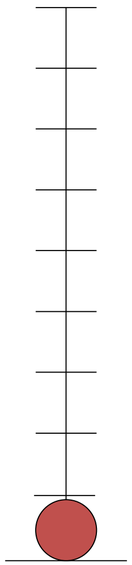
Exercise from last week: one-dimensional track

top view

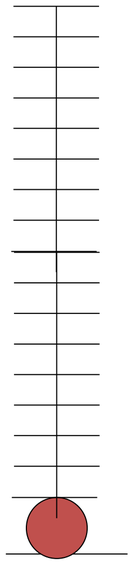


2m

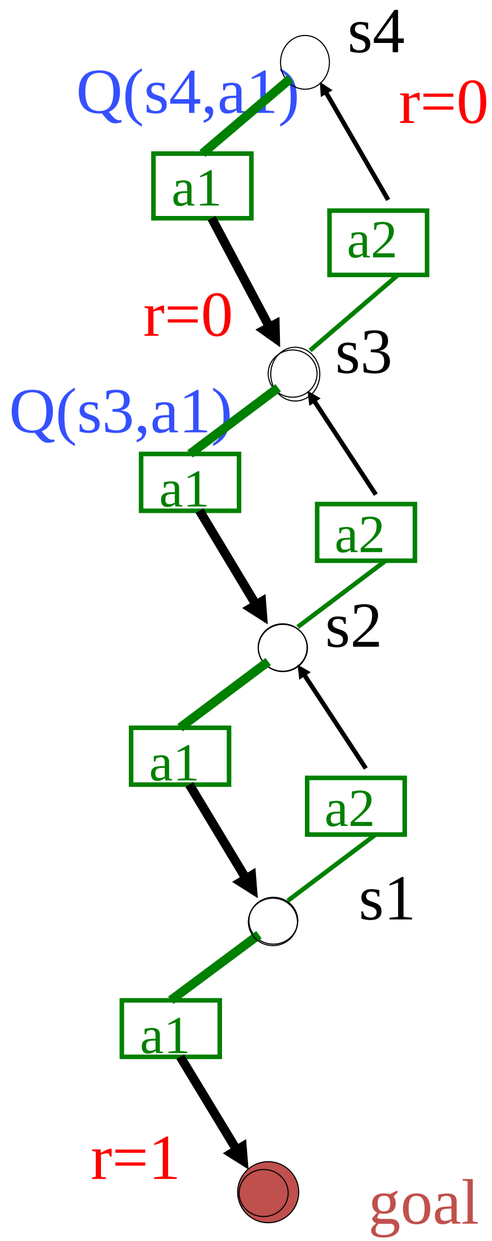
Discretize state



goal



goal



(previous slide)
However, if you think of an animal that walks along a corridor towards a piece of cheese (reward), then the natural space is continuous and any discretization is arbitrary. Why should we choose 10 states and not 20?

Once we are in the discrete space, the situation is similar to the random walk example considered earlier, except that here we are interested in an agent that adapts its policy so that it walks as quickly as possible to the reward.

Exercise from last week: one-dimensional track

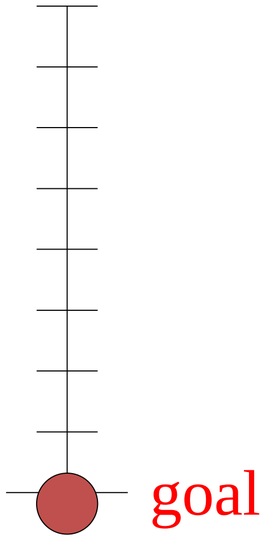
- **Initialise Q values at 0. Start trials at top (s4).**
- **Update of Q values with SARSA**

$$\Delta Q(s,a) = \eta \; [r + \gamma \, Q(s',a') - Q(s,a)]$$

- **Policy for action choice:**

  Pick most often action $a_t^* = \arg\max_a Q_a(s,a)$

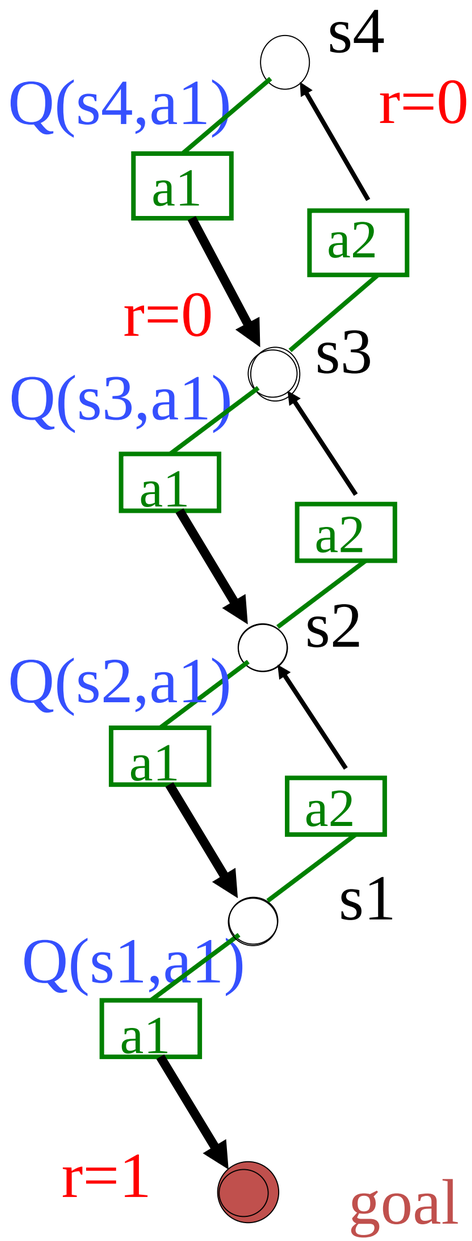
  To break ties: take action a1



Linear sequence of states.
Reward only at goal.
Actions are up or down.

[ ] After 2 trials the Q-value Q(s1,a1)>0

[ ] After 2 trials the Q-value Q(s3,a1)>0



(previous slide)
Your comments. See also the solution of exercise from last week.

Problem of online TD algorithms

Problem:

-‘Flow of information’ back from target is slow

- information flows 1 step per complete trial (‘episode’)

- 20 trials needed to get information 20 steps away from target

BUT:

- the discretization of states has been an arbitrary choice!!!

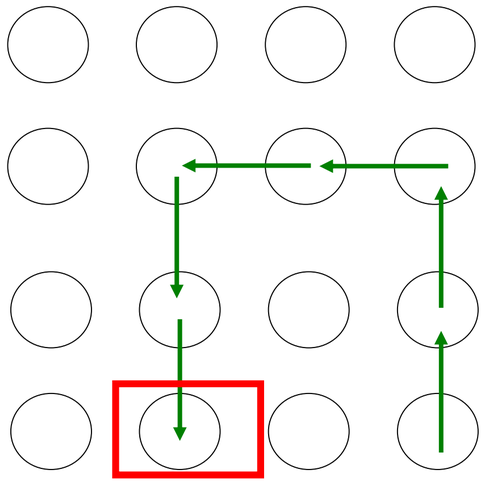
→ Something is wrong with the discrete-state SARSA algo

(previous slide)
In the SARSA algorithm and all other TD learning algorithms that we have seen so far, information about a reward at the target needs several trials before it shows up in the Q-values (or V-values) that are not close to the target.
In fact, if all Q-values are initialized at zero, it takes 10 trials before the Q-value of a state that is 10 steps away from the target is updated the first time.

So if we decide to discretize 1m of corridor into 20 states (instead of 10 states), then it will take 20 trials for the information to arrive at the start.

This is strange, because the performance of the agent (an animal!) should not depend on the discretization scheme that we have chosen.

Solution 1: Eligibility Traces, SARSA(λ)

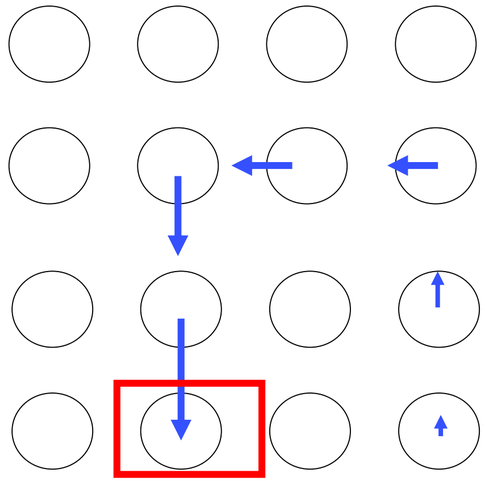


Idea:

- keep memory of previous state-action pairs
- memory decays over time
- update eligibility trace for **all** state-action pairs

$$e(s,a) \leftarrow \lambda\, e(s,a)$$ decay of **all** traces

$$e(s,a) \leftarrow e(s,a) + 1$$ if action a chosen in state s



- update **all** Q-values at **all time steps** t:

$$\Delta Q(s,a) = \eta\, [r_t + \gamma\, Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)]\, e(s,a)$$

TD error δ_t

Note: λ=0 gives standard SARSA

(previous slide)
Eligibility traces are a first solution to the above problem:

For each state-action pair we introduce a variable e(s,a), called eligibility trace. The eligibility trace is increased by one, if the corresponding state-action pair occurs. In each time step, all eligibility traces decrease by a factor λ <1.
(In fact, λ should be smaller than the discount factor gamma for reasons that become clear only later).

In each time step t, all Q-values Q(s,a) are update proportional to the TD error for the time step t.
The update is proportional to the corresponding eligibility trace e(s,a).

Note: in the original SARSA algorithm we have for each state-action pair a variable Q(s,a). In the new algorithm, we have for each state-action pair two variables: Q(s,a) and e(s,a). I will sometimes call e(s,a) the ‘shadow’ variables: each eligibility trace is the shadow of the corresponding Q-value.

Solution 1: Eligibility Traces

From: Reinforcement Learning,
Sutton and Barto 1998
First edition

7.5 *Sarsa(λ)*

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s, a and set e(s,a)=0 for all actions a and states s
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ϵ-greedy)
    $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
    $e(s, a) \leftarrow e(s, a) + 1$
    For all s, a:
      $Q(s, a) \leftarrow Q(s, a) + \alpha \delta e(s, a)$
      $e(s, a) \leftarrow \gamma \lambda e(s, a)$
    $s \leftarrow s'; a \leftarrow a'$
  until s is terminal

Figure 7.11 Tabular Sarsa(λ).

(previous slide)
Note: in some published versions of the algorithm the decay of the eligibility traces is the product of γ and λ, and not just λ.

The advantage is that you just have to impose λ <1 whereas on the preceding slide I should choose a decay rate $\lambda < \gamma$ (<1).

Quiz: Eligibility Traces

[ ] Eligibility traces keep information of past state-action pairs.

[ ] For each Q-value $Q(s,a)$, the algorithm keeps one eligibility trace e(s,a), i.e., if we have 200 Q-values we need 200 eligibility traces

[ ] Eligibility traces enable information to travel rapidly backwards into the graph

[ ] The update of $Q(s,a)$ is proportional to $[r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)]$

[ ] In each time step all Q-values are updated

(previous slide) your notes

Teaching monitoring – monitoring of understanding

[ ] The concept of eligibility traces was new to me.

[ ] I have the feeling that I have been able to follow (at least) 80% of the lecture on eligibility traces.

Exercise : Eligibility Traces

In week 2 in exercise 3, we applied the SARSA algorithm to the case of a linear track with actions 'up' and 'down'. We found that it takes a long time to propagate the reward information through state space. The eligibility trace is introduced as a solution to this problem.

Reconsider the linear maze from Fig 2. in exercise 3, but include an eligibility trace: for each state s and action a, a memory $e(s,a)$ is stored. At each time step, all the memories are reduced by a factor λ: $e(s,a) = \lambda e(s,a)$, except for the memory corresponding to the current state s^* and action a^*, which is incremented:

$$e(s^*,a^*) = \lambda e(s^*,a^*) + 1\,. \tag{1}$$

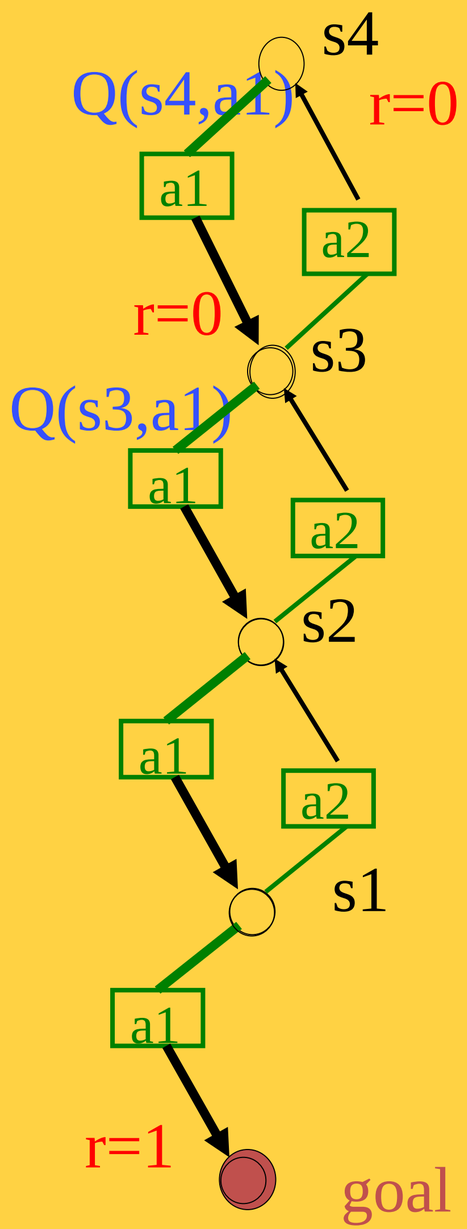
Now, unlike the case without eligibility trace, all Q-values are updated at each time step according to the rule

$$\forall(s,a) \quad \Delta Q(s,a) = \eta\,[r - (Q(s^*,a^*) - Q(s',a'))]\,e(s,a)\,. \tag{2}$$

where s^*, a^* are the current state and action, and s', a' are the immediately following state and action.

We want to check whether the information about the reward propagates more rapidly. To find out, assume that the agent goes straight down in the first trial. In the second trial it uses a greedy policy. Calculate the Q-values after two complete trials and report the result.

Hint: Reset the eligibility trace to zero at the beginning of each trial.



Reinforcement Learning Lecture 2

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Variants of TD-learning methods and eligibility traces

Part 6: n-step TD methods

1. Review and introduction of BackUp diagrams
2. Variations of SARSA
3. TD Learning (Temporal Difference)
4. Monte-Carlo Methods
5. Eligibility traces
6. **n-step TD methods**

(previous slide)
Let us now focus on n-step TD methods such n-step SARSA.

Problem of TD algorithms

Problem:

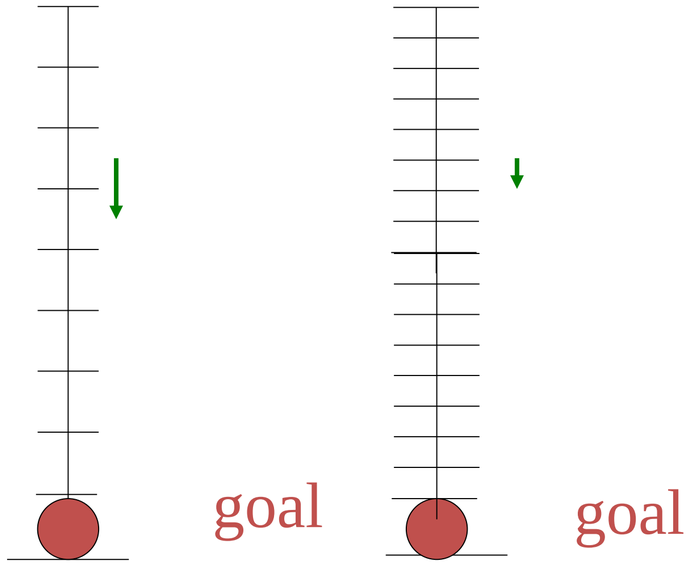
-‘Flow of information’ back from goal is slow

- information flows 1 step per complete trial

- 20 trials needed to get information 20 steps away from target

→ First solution: eligibility traces.

→ second solution: n-step TD methods.



(previous slide)
Eligibility traces make the flow of information from the target back into the graph more rapid. The speed of flow is now controlled by the decay constant λ of the eligibility trace – therefore we can keep the flow constant even if the discretization changes by readjusting λ.

However, there is also a second solution, called n-step SARSA.

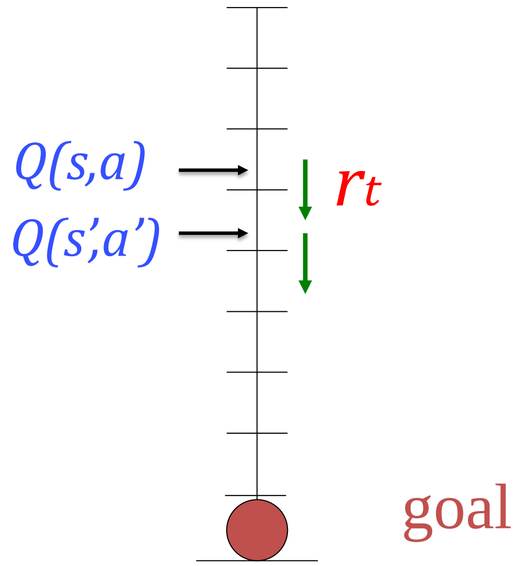
Solution 2: n=step SARSA

Standard SARSA

$\Delta Q(s,a) = \eta\,[r + \gamma Q(s',a') - Q(s,a)]$

$\Delta Q(s_t,a_t) = \eta\,[r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)]$

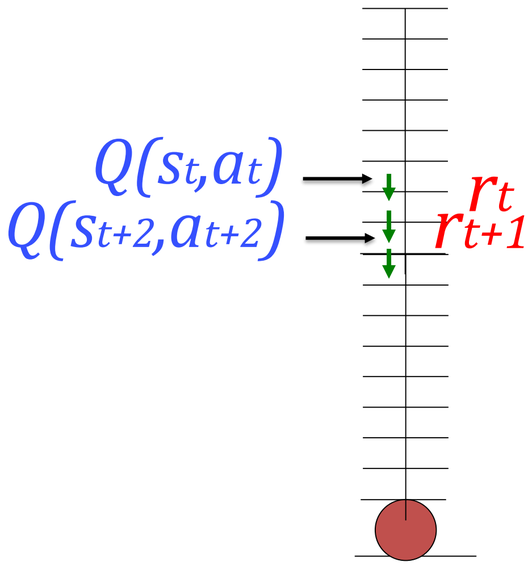
Temporal Difference (TD)



2-step SARSA

$\Delta Q(s_t,a_t) = \eta\,[r_t + \gamma r_{t+1} + \gamma\gamma Q(s_{t+2},a_{t+2}) - Q(s_t,a_t)]$

2-step TD



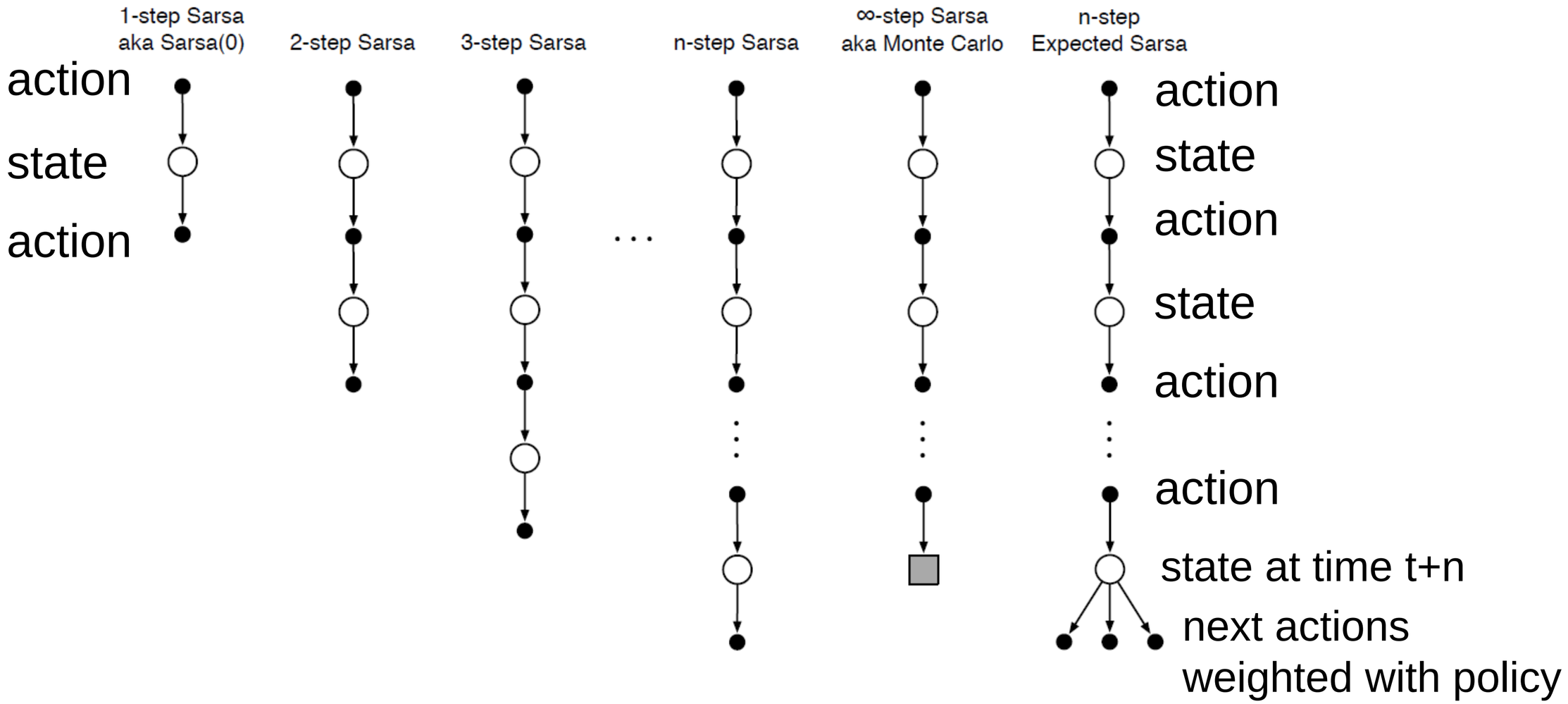
(previous slide)
Reminder:
SARSA and other standard TD methods compare the reward with **neighboring** Q-values.

In two step SARSA, we compare the two-step reward with the difference in Q-values of next-nearest neighbors.

In other words, the sum of the two rewards between s_t and s_{t+2} must be explained by the difference between the Q-values $Q(s_t,a_t)$ and (discounted) $Q(s_{t+2},a_{t+2})$.

The greek symbol γ denotes the discount factor, as before.

n-step SARSA and n-step expected SARSA



(previous slide)
The idea of 2-step SARSA can be extended to an arbitrary n-step SARSA.
Interestingly, if the number n of steps equals the total number of steps to the end of the trial, we are back to standard Monto-Carlo estimation.

Hence, n-step SARSA is in the middle between normal SARSA and Monte-Carlo estimation.

n-step SARSA algorithm

n-step Sarsa for estimating $Q \approx q_*$, or $Q \approx q_\pi$ for a given π

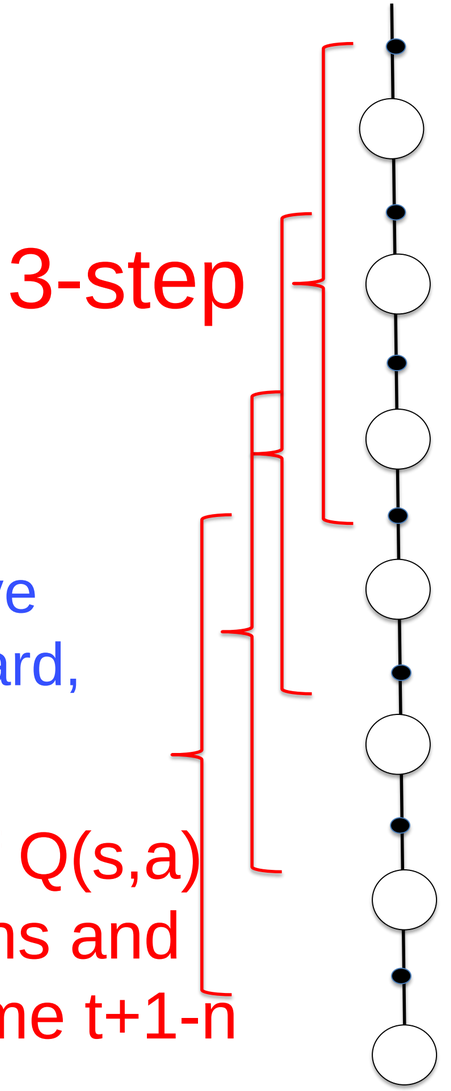
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be ε-greedy with respect to Q, or to another stochastic policy
Parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer n
All store and access operations (for S_t, A_t, and R_t)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

Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$
  $T \leftarrow \infty$
  For $t = 0, 1, 2, \ldots$:
    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 $A_{t+1} \sim \pi(\cdot|S_{t+1})$
    $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
    (1) $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
    (2) If $\tau + n < T$, then $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$ $(G_{\tau:\tau+n})$
    (3) $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\,[G - Q(S_\tau, A_\tau)]$
  Until $\tau = T - 1$

In algo: r_t is called R_{t+1}

Take action, observe next state and reward, choose next action

update of Q(s,a) with actions and state at time t+1-n



Sutton and Barto, Ch. 7.2

(previous slide)
The backup graph for three-step SARSA now contains 3 state-action pairs, because we need to keep more information in memory.

Note that we can update $Q(s_t, a_t)$ once we have chosen action a_{t+3} in state S_{t+3}

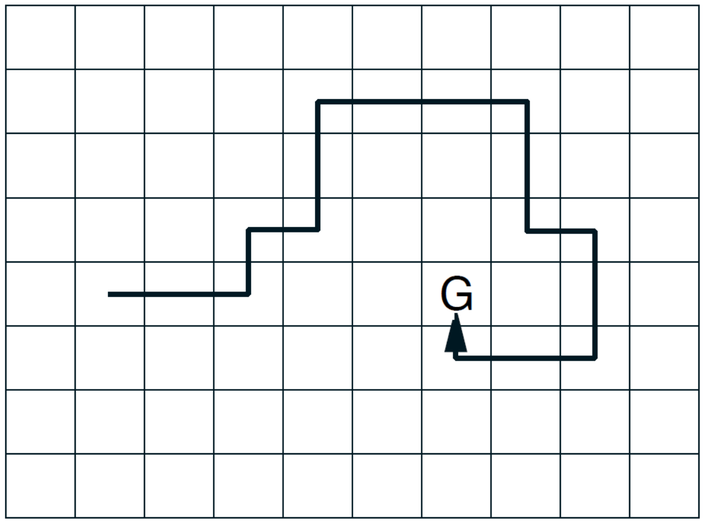
Lines marked (1), (2), (3).

(1) G is the reward summed over n steps (with discounts for steps >1)
(2) To this G the Q-value of the *n*th state is added (unless the episode terminates before)
(3) The update then happens with this new G as a target and learning rate alpha.

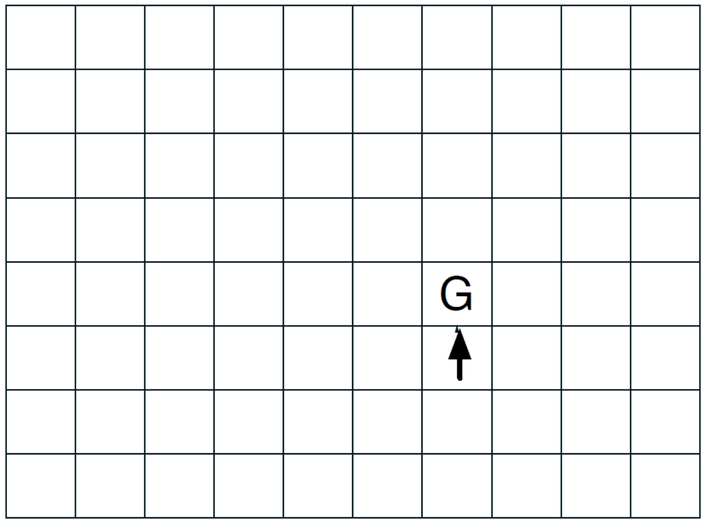
For some reason Sutton and Barto suggest epsilon-greedy in the pseudo-algo, but I changed this to arbitrary stochastic policies. Stochastic is important to make sure that all branches are explored; apart from this: SARSA is an on-policy algorithm and whatever you choose as a policy should work and yield a self-consistent solution.

Example: 10-step SARSA

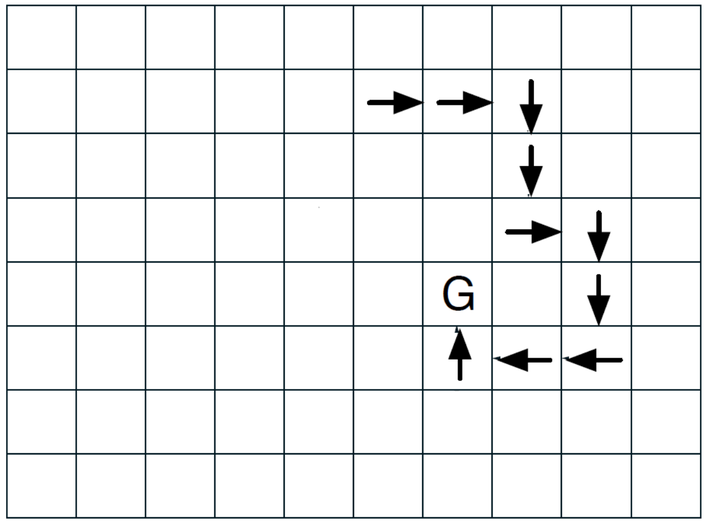
Path taken



Action values increased by one-step Sarsa



Action values increased by 10-step Sarsa

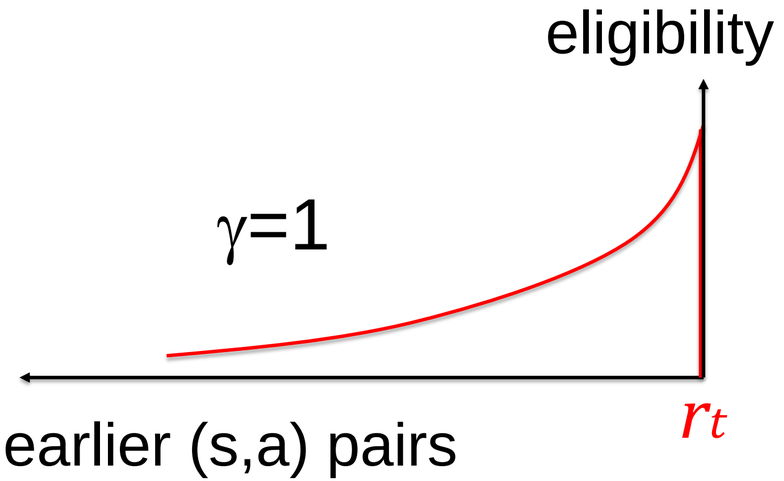


(previous slide).
The graphic suggests that the results of 10-step SARSA are very similar to an eligibility trace – which is indeed the case. therefore the two solutions (eligibility trace and n-step TD learning) are in fact closely related.

We will come back to this issue in lectures 11 and 12 on reinforcement learning.

Summary: Scaling Problem of TD algorithms

TD algorithms do not scale correctly if the discretization is changed

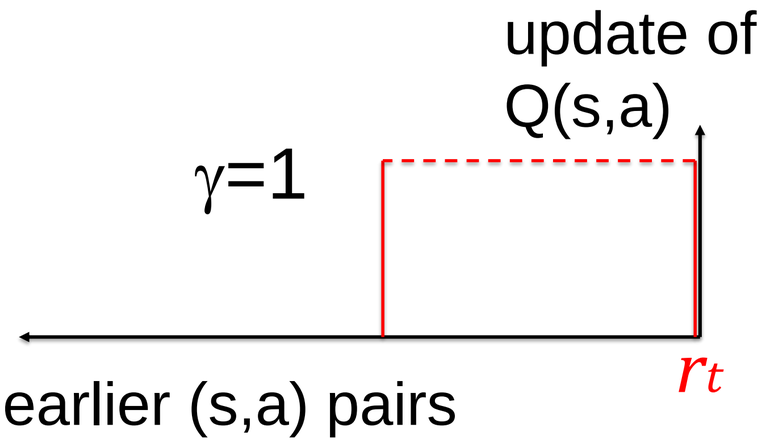


either

→ Introduce **eligibility traces** (temporal smoothing)

or

→ Switch from 1-step TD to **n-step TD**
(temporal coarse graining)



Remark: the two methods are mathematically closely related.

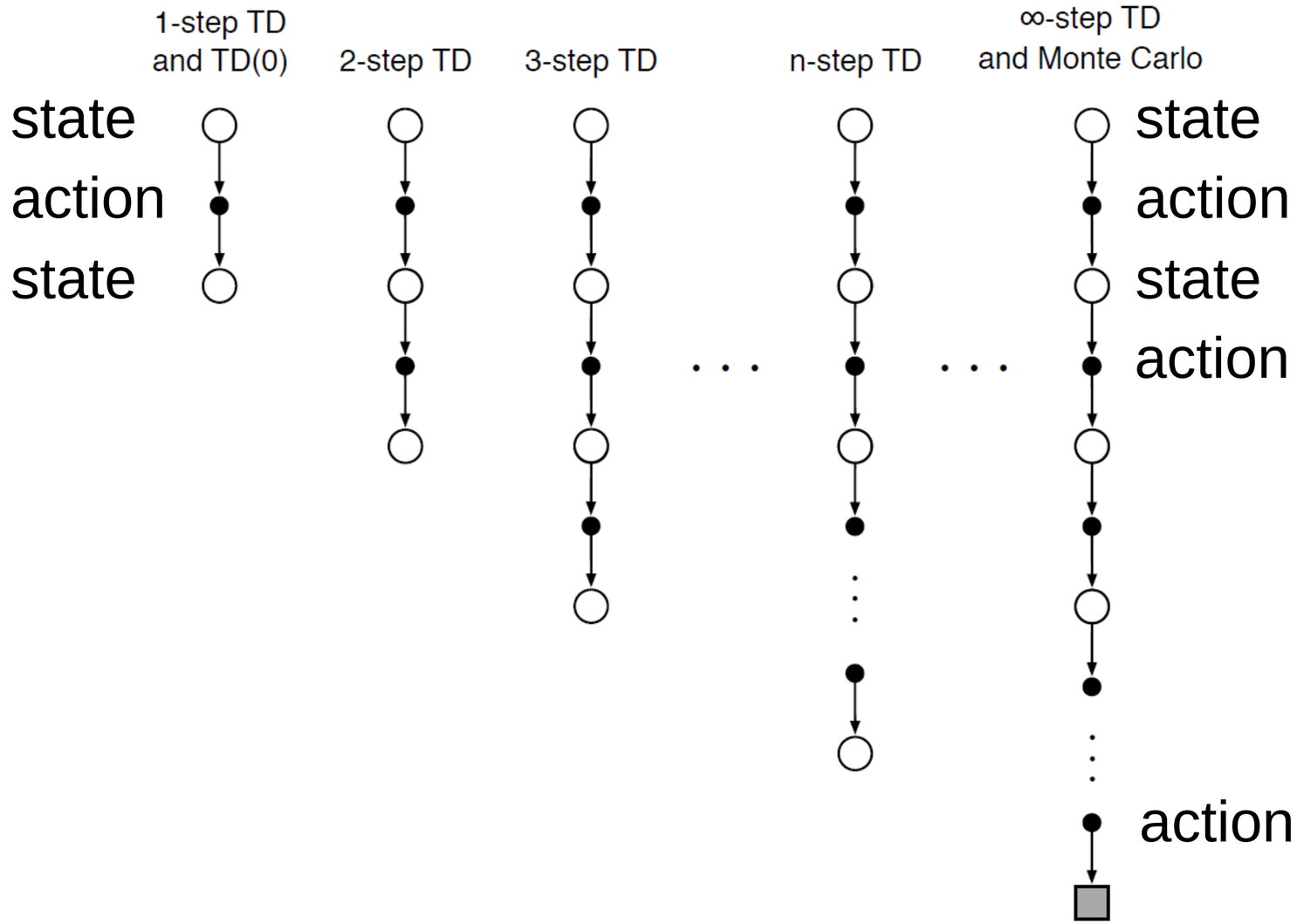
(previous slide)
One-step TD algorithms have problems as approximations to continuous states.
There are two closely related solutions, eligibility traces and n-step TD algorithms.

Eligibility traces can be interpreted as a temporal smoothing of state-action pairs with an exponential filter. On the horizontal axis, r_t denotes the moment of a reward.

'n-step algorithms' can be interpreted as temporal coarse graining.

- After the agent has passed a reward on its path and has continued for n-1 steps, the n Q-values corresponding to n state-action pairs before the reward have been updated. Thus you group Q-values as if you were using a larger discretization. (The specific picture with rectangular filter is valid for $\gamma=1$)
- Once you have reached the goal (a terminal state) you should formally continue the algorithm for n-1 steps with fictitious zero-reward. This avoids discretization effects and amounts to using the same rectangular filter as on the path. For example, with 4-step SARSA, you need to update the Q-values not only 4 steps before the goal, but also 3 steps, 2 steps and 1 steps. [Otherwise the Q-values 1 step before the goal would not be updated after the first episode!]

Detour: n=step TD methods for V-values



Sutton and Barto, Ch. 7.1

(previous slide)
Remarks regarding n-step V-value TD methods are completely analogous to those for Q-values.

Detour: n=step TD methods for V-values

n-step TD for estimating $V \approx v_\pi$

Initialize $V(s)$ arbitrarily, $s \in \mathcal{S}$
Parameters: step size $\alpha \in (0, 1]$, a positive integer n
All store and access operations (for S_t and R_t) can take their index mod n

Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  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 $T \leftarrow t + 1$
    $\tau \leftarrow t - n + 1$ (τ 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: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_{\tau:\tau+n})$
      $V(S_\tau) \leftarrow V(S_\tau) + \alpha\,[G - V(S_\tau)]$
  Until $\tau = T - 1$

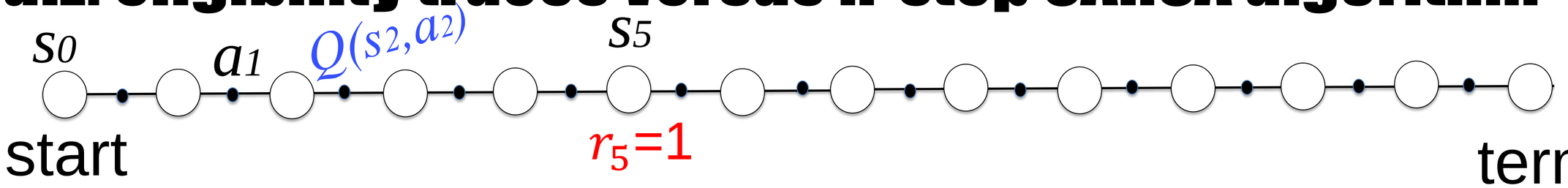
In algo: r_t is called R_{t+1}

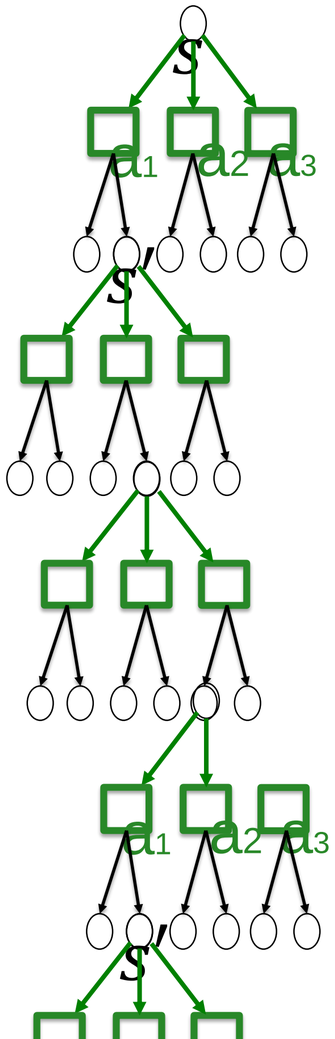
Sutton and Barto, Ch. 7.1

(previous slide/not shown in class/just as a reference)
The essential step of the algorithm is the update in the blue ellipse where G are the discounted accumulated rewards over n step.

The algorithm looks a bit more complicated because there is a clever way of dealing with the summation over the intermediate rewards while the agent moves along the graph.
See Book of Sutton and Barto 2018 for details.

Quiz: eligibility traces versus n-step SARSA algorithm





All Q-values have been initialized at zero. The first episode starts in 'start' and ends after 13 steps in the terminal state. In step 5, the reward is r_5=1. All other rewards are zero. The discount factor is γ=0.95.

Is the following true after the end of the first episode:

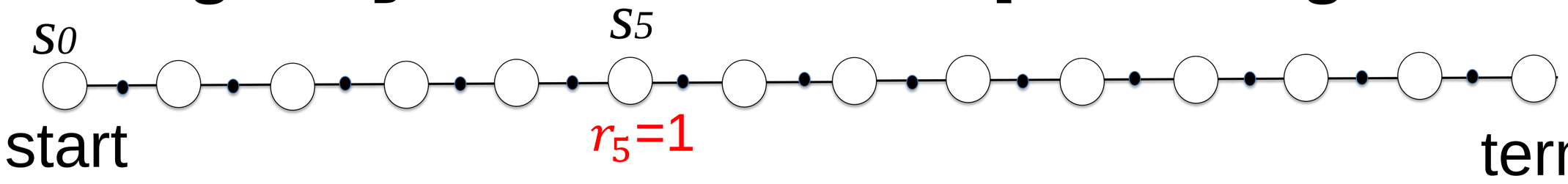
[ ] With 3 step SARSA, only one Q-value has increased, viz. the one 3 steps before the reward.

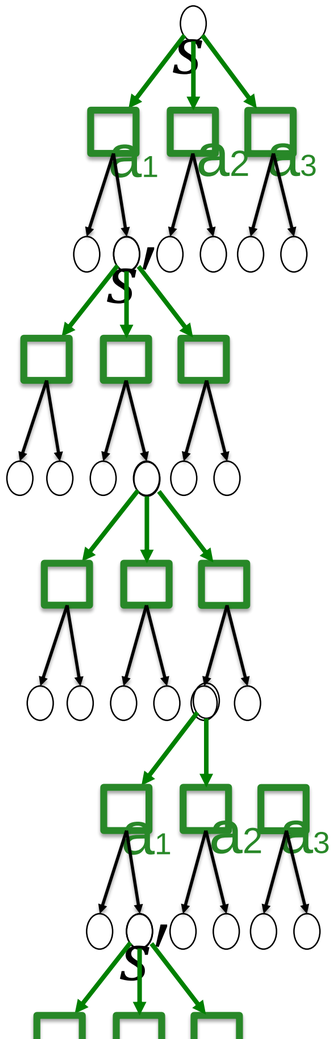
[ ] With 3 step SARSA, three Q-values have increased, viz. those 3 steps, 2 steps and 1 step before the reward.

[ ] None of the above

[ ] Increase is biggest for Q(s_4,a_4).

Quiz: eligibility traces versus n-step SARSA algorithm





All Q-values have been initialized at zero. The first episode starts in 'start' and ends after 13 steps in the terminal state. In step 5, the reward is r_5=1. All other rewards are zero. The decay factor is λ=0.95.

Is the following true after the end of the first episode:

[ ] Using SARSA with eligibility traces, only one Q-value has increased.

[ ] Using SARSA with eligibility traces, five Q-values have increased,

[ ] Using SARSA with eligibility traces, all Q-values have increased.

Teaching monitoring – monitoring of understanding

[ ] in the afternoon lecture, at least 60% of material
was new to me.

[ ] up to here, I have the feeling that I have been able to follow
(at least) 80% of the lecture.

Reinforcement Learning Lecture 2

Variants of TD-learning methods and eligibility traces

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Part 7: Consistency proofs for n-step methods

Variant A: 3-step SARSA is consistent w. Bellman equation

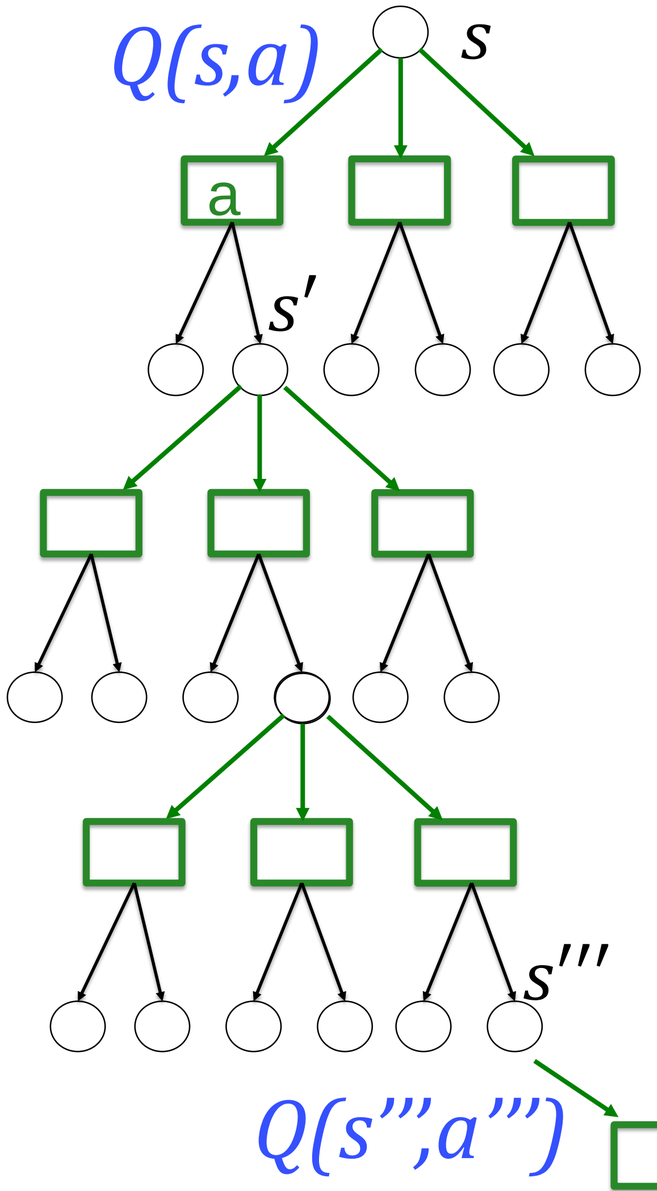
THEOREM-Variant A:

Suppose that we have found a set of Q-values.
We keep them frozen while evaluating expectation.
IFF $E[\Delta Q(s,a)] = 0$, then the Q-values solve the
3-step Bellman equation.

$$E[\Delta Q(s,a)|s,a,\theta^{old}] = \quad 0 =$$

$$= E[r_t + r_{t+1} + r_{t+2} + \gamma Q(s''',a''') - Q(s,a)|s,a,\theta^{old}]$$

→ See Exercise session today!



Previous slide and next slides:
The n-step Bellman equation is a direct generalization of the normal (1-step) Bellman equation. See exercise for the example of a 3-step Bellman function.

Note that the expectation is taken with the old parameters (frozen parameters). The parameters are all Q-values.

To proof this we use Variant A from the previous week – as reviewed now.

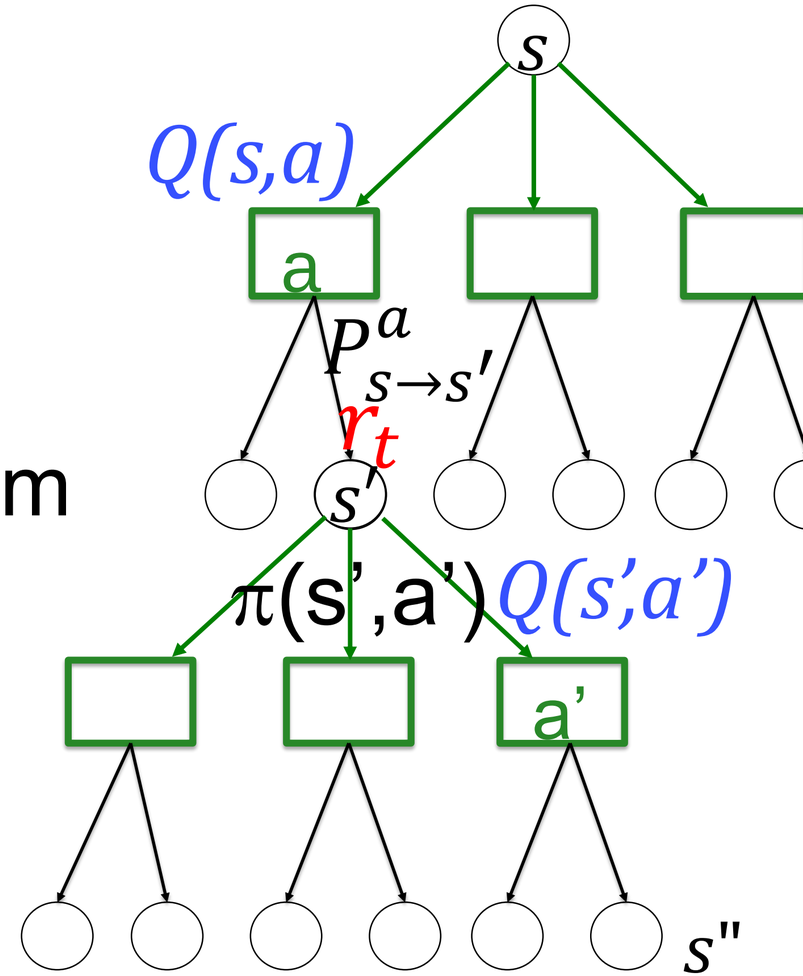
Recap from last week: normal 1-step SARSA

Variant A:

SARSA is consistent w. Bellman equation

$$E[\Delta Q(s,a)|\theta^{old}] = \eta\, E[r_t + \gamma Q(s',a') - Q(s,a)|\theta^{old}]=0$$

Expectations are ‘batch-like’ for frozen system



Variant B:

SARSA is consistent w. Bellman equation

$$\langle \Delta Q(s,a)|s,a\rangle = 0$$

Temporal average over many update step/over temporal jitter, assuming small learning rate ($\eta \to 0$).

Variant A: SARSA is consistent w. Bellman equation

We prove the following:

Suppose that we have found a set of Q-values.

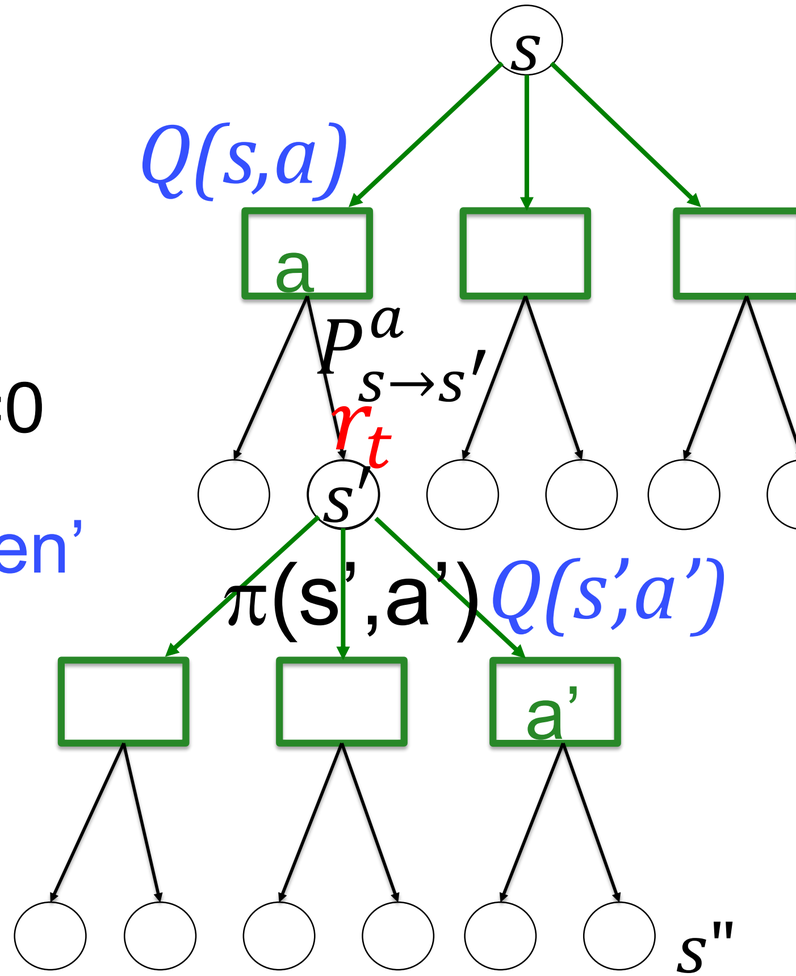
We keep them frozen while evaluating expectation.

IFF $E[\Delta Q(s,a)]|\theta^{old}] = 0$, then the Q-values solve the Bellman equation.

$$E[\Delta Q(s,a)]|\theta^{old}] = \eta\, E[r_t + \gamma Q(s',a') - Q(s,a)|\theta^{old}]=0$$

'frozen' 'frozen'

$$\sum_{s\prime} P^a_{s\to s\prime} R^a_{s\to s\prime} + \gamma \sum_{s\prime} P^a_{s\to s\prime} \sum_{a\prime} \pi(s',a')\, Q(s',a') = Q(s,a)$$



Look at graph to take expectations:

- if algo is on a branch (s,a), all remaining expectations are "given s and a"

Previous slide.
This is version A of the theorem. In the proof, we exploit the condition:

$$E[\Delta Q(s,a)|s,a] = 0$$

In order to take the expectations, we look at graph:
- if in the evaluation we are in state s', all remaining expectations are "given s'"
- if we are on a branch (s,a), all remaining exp. are "given s and a".

We exploit that all Q-values and the policy are frozen while we evaluate the expectation. Hence E[Q(s,a)] = Q(s,a).

Note that the proof works in both direction. If the Q-values are those of the Bellman equation, the expected SARSA update step vanishes. And if the expected SARSA update step is zero, then the Q-values correspond to the Bellman equation. Both direction work under the assumption of a fixed policy.

The stronger theorem (with fluctuations, version B) is sketched in the appendix (and video).

Additional Notes: This weaker theorem (variant A that corresponds to the one on the previous slide) takes expectations for FIXED Q-values. We can interpret these expectations as the following 'batch' computation

We assume a fixed policy (i.e., under the assumption of a fixed set of Q-values) and a 'batch version' of SARSA. Batch-SARSA means that in order to evaluate $E[\Delta Q(s,a)|s,a]$ we use a large number of starts from the same value (s,a) each time running one step up to (s',a') [note that this gives different (s',a'). Once the number of starts is large enough to get a full sample of the statistics we update Q(s,a). If the updates with the batch-SARSA do not lead to a change of Q values (for all state-action pairs), then this means that batch-SARSA has converged to the Bellman equation for this fixed policy. (That was the theorem in the main text).

Batch-SARSA is a computational implementation of the way many statistical convergence proofs work: you assume that you average over a full statistical sample of all possibilities given your current state or the current state-action pair. Expectation signs in the update step imply updating over a 'full batch of data'. In this approach Q-values no longer fluctuate, and hence do not need expectation signs; the policy no longer fluctuates and also does not need expectations signs.

Now: Variant B - SARSA and Bellman equation (proof for small η)

Setting: 'Temporal Averaging'

The SARSA algo with stochastic policy π has been applied for a long time with updates

$$\Delta\hat{Q}(s,a) = \eta\,[r_t + \gamma\hat{Q}(s',a') - \hat{Q}(s,a)]$$

IF (i) learning rate η is small; AND IF

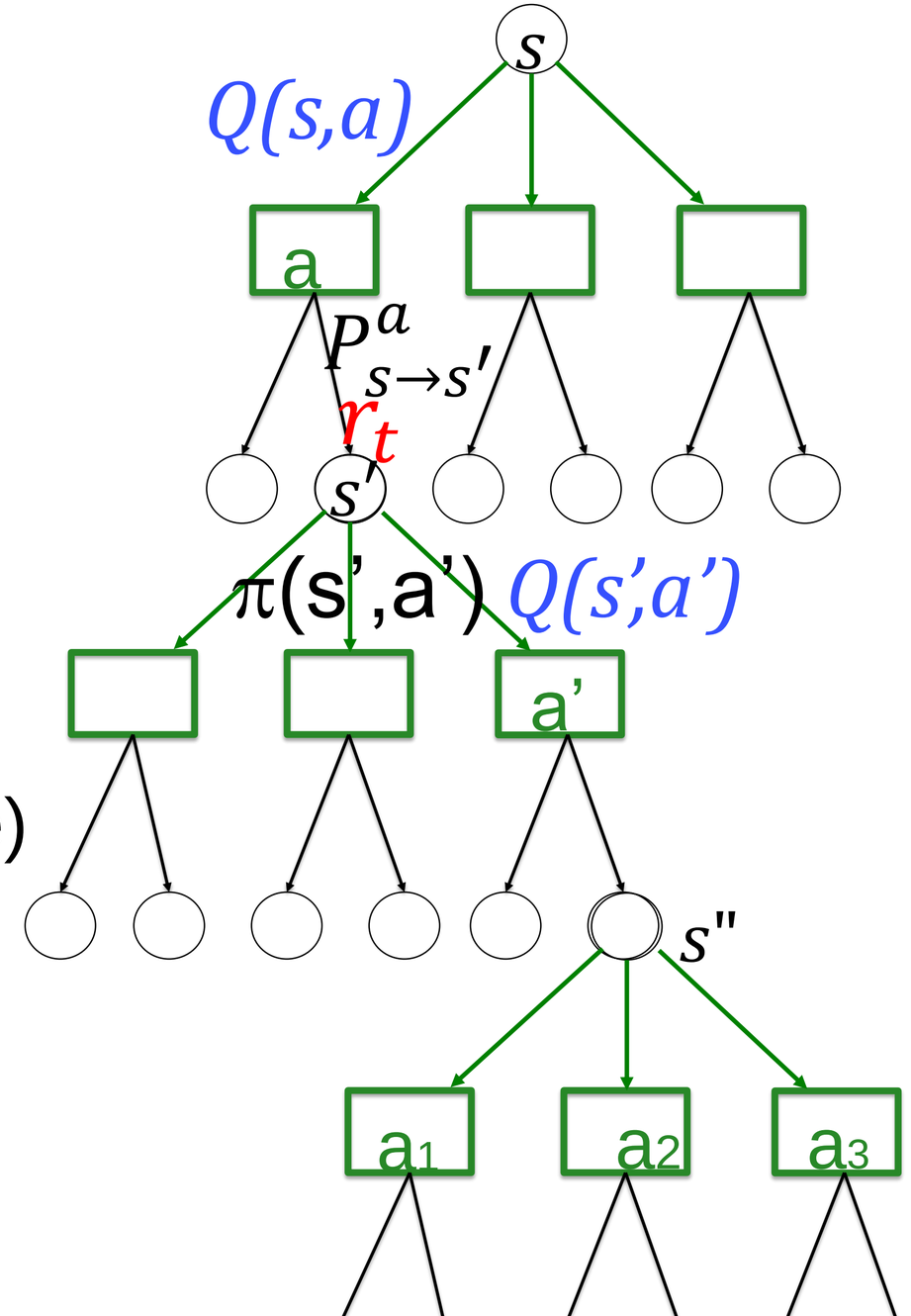
(ii) for all Q-values

$$\langle \Delta\hat{Q}(s,a) | s,a \rangle = 0$$

THEN the **expectation** values (temporal average) of the set of $\hat{Q}$-values solve the Bellman eq.

$$Q(s,a) = \sum_{s'} P^a_{s\to s'}\left[R^a_{s\to s'} + \gamma\sum_{a'}\pi(s',a')Q(s',a')\right]$$

with the current policy $\pi(s',a')$



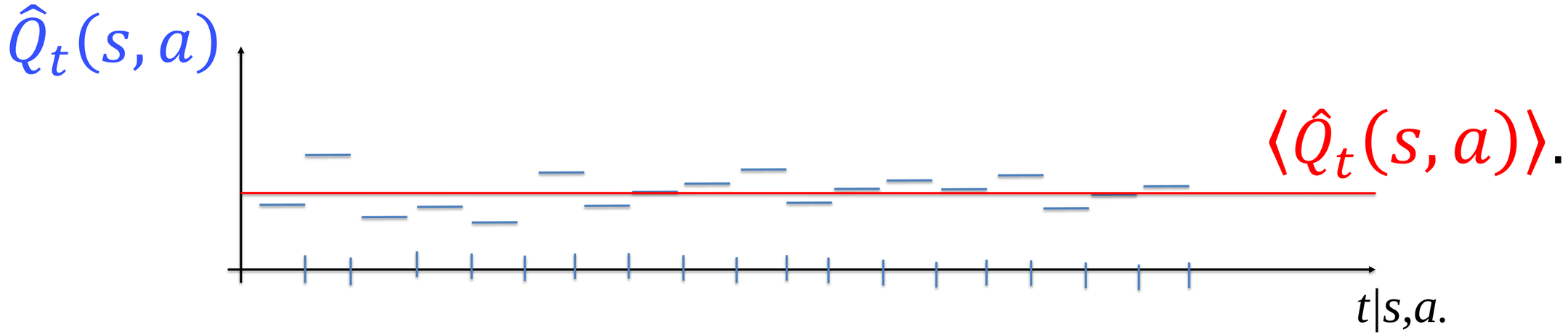
Previous slide. There are two different versions of the theorem. **This version B** is proven on the Blackboard 6B, in a fashion similar to the case of the 1-step horizon.
In class (earlier slides) we have shown for the multi-step horizon a weaker statement: **Expections over SARSA updates are consistent with the Bellman equation** if the expected update vanishes: **In version A, we assume that Q-values are fixed (frozen) when we take the expectation.**

Note that taking the expectation in version A is different from averaging over update steps in version B. In version A, taking the expectation means that we average over all possible outcomes in the **current** situation, with momentarily fixed Q-values and **fixed policy** (i.e., the one induced by the set of Q-values at time t). This distinction is important, because **for a fixed policy averaging is relatively easy.**

However, when averaging over time steps as in **variant B, the Q-values and policy are different in each time step**, and the proof therefore requires a limit $\eta \rightarrow 0$, so that changes can be neglected.
Hence there are therefore two versions of the theorem and two proof-sketches:
Blackboard 6A. On the earlier slides, we assume Q-values are fixed and do not fluctuate.
Blackboard 6B. Now, we assume that Q-values may fluctuate slightly round their stable values. This approach gives additional insights into the situation of the online SARSA, once it has converged in expectation.

Variant B of Theorem:

We work with the **online update** $\Delta\widehat{\boldsymbol{Q}}(\boldsymbol{s},\boldsymbol{a})$. With finite learning rate, the value of $\hat{Q}_t(s,a)$ fluctuates around a mean $\langle\hat{Q}_t(s,a)\rangle$.



Claim: Under the hypothesis $\langle\Delta\hat{Q}(s,a)=0\rangle$ **, the mean** $\langle\widehat{\boldsymbol{Q}}_t(\boldsymbol{s},\boldsymbol{a})\rangle$ **is equal to the 'correct' Q-value.**

Notes: A few points should be stressed:

1. This is not a convergence theorem. We just show consistency as follows:
   if SARSA has converged then it has converged to a solution of the Bellman equation.
2. In fact, for any finite η the SARSA Q-values (Q-hat) fluctuate a little bit. It
   is only the EXPECTATION value of the Q-hat which converges.
3. We should keep in mind that SARSA is an on-policy online algorithm for arbitrary
   state-transition graphs. Hence the value Q-hat at (s,a) and (s'a') will both fluctuate!
4. The policy depends on these Q-hat-values and hence fluctuates as well.
   To keep fluctuations of the policy small, we need small η .
   We imagine that all Q-hat values fluctuate around their expectation value
   with small standard deviation. As a result, π also fluctuates around a 'standard' policy.
5. The fluctuations of the policy can be smaller than that of the Q-values: for
   example in epsilon-greedy, you first order actions by the value of Q(s,a),
   and then only the rank of Q(s,a) matters, not their exact values. In the proof we assume
   that the fluctuations of the policy become negligible (→ shift policy outside expectation).
6. We show that the Q-values in the sense of Bellman are the expectation values of
   the Q-hat in the sense of SARSA.
7. Expectations are over many trials of the ONLINE SARSA.

The statement and proof is different to slide 127 and to the book of Sutton and Barto.

Variant B – temporal averaging: Preparation and Notation

$$\hat{Q}_t(s',a') = \langle \hat{Q}_t(s',a') \rangle + \delta\hat{Q}_t(s',a') \quad \text{Eq. (1).}$$

$$\bar{q} \qquad +\delta q$$

define notation:

$\bar{q} = \langle \hat{Q}_t(s',a') \rangle$

$\delta q = \delta\hat{Q}_t(s',a')$

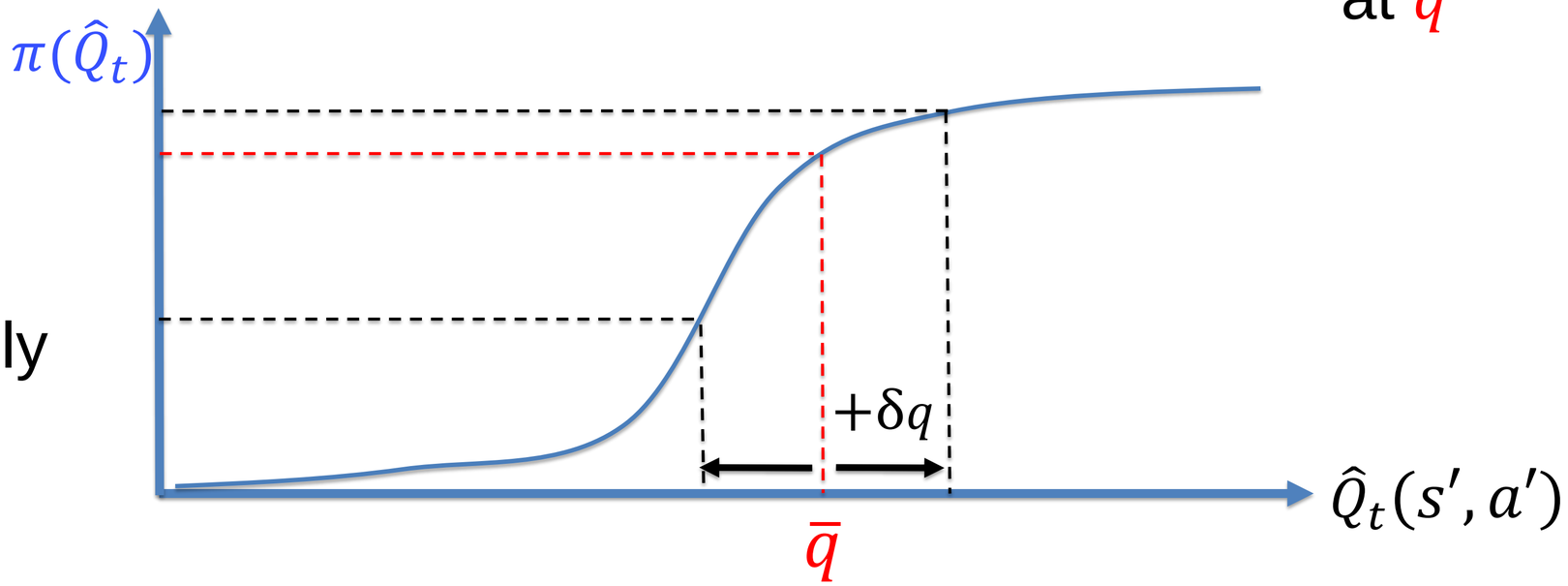
notation of policy:

Taylor expansion of policy:

$$\pi(s',a'|\hat{Q}_t(s',a') = \pi(\hat{Q}_t) = \pi(\bar{q}\,) + \frac{\partial\pi}{\partial\hat{Q}_t}\delta q + \frac{1}{2}\frac{\partial^2\pi}{\partial^2\hat{Q}_t}(\delta q)^2 + \cdots \quad \text{Eq. (2)}$$

derivatives evaluated at $\bar{q}$

policy is affected asymmetrically



Now: Variant B - SARSA and Bellman equation (proof for small η)

Setting: 'Temporal Averaging'

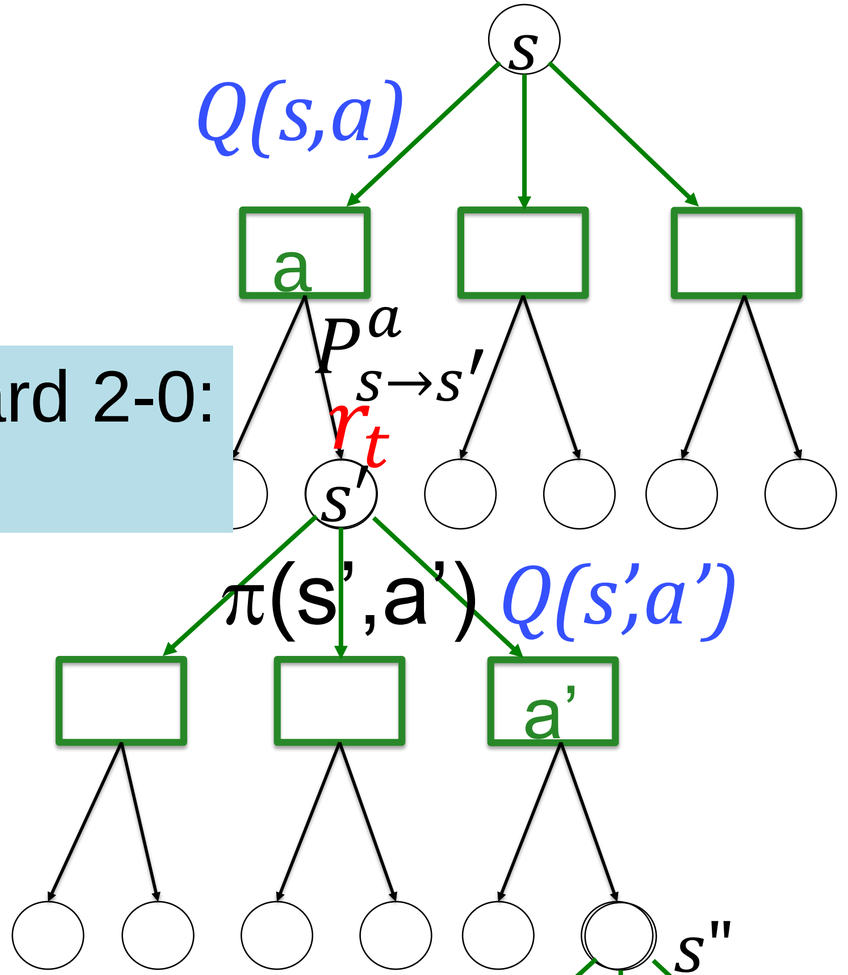
$$\Delta\hat{Q}(s,a) = \eta\,[r_t + \gamma\hat{Q}(s',a') - \hat{Q}(s,a)]$$

IF (i) learning rate η is small; AND IF

(ii) for all Q-values

$$\langle \Delta\hat{Q}(s,a) | s,a \rangle = 0$$

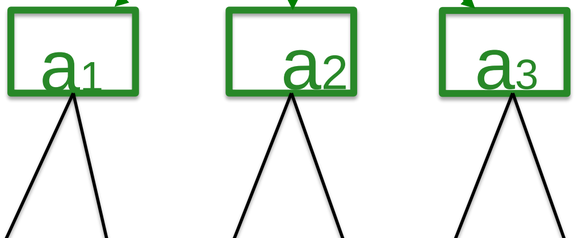
Blackboard 2-0: SARSA



Problem: we have to evaluate product

$$\langle \pi(\hat{Q}(s',a')) \cdot \hat{Q}(s',a') \rangle$$

Blackboard 2-2: SARSA



Now: Variant B - SARSA and Bellman equation (proof for small η)

Problem: we have to evaluate product

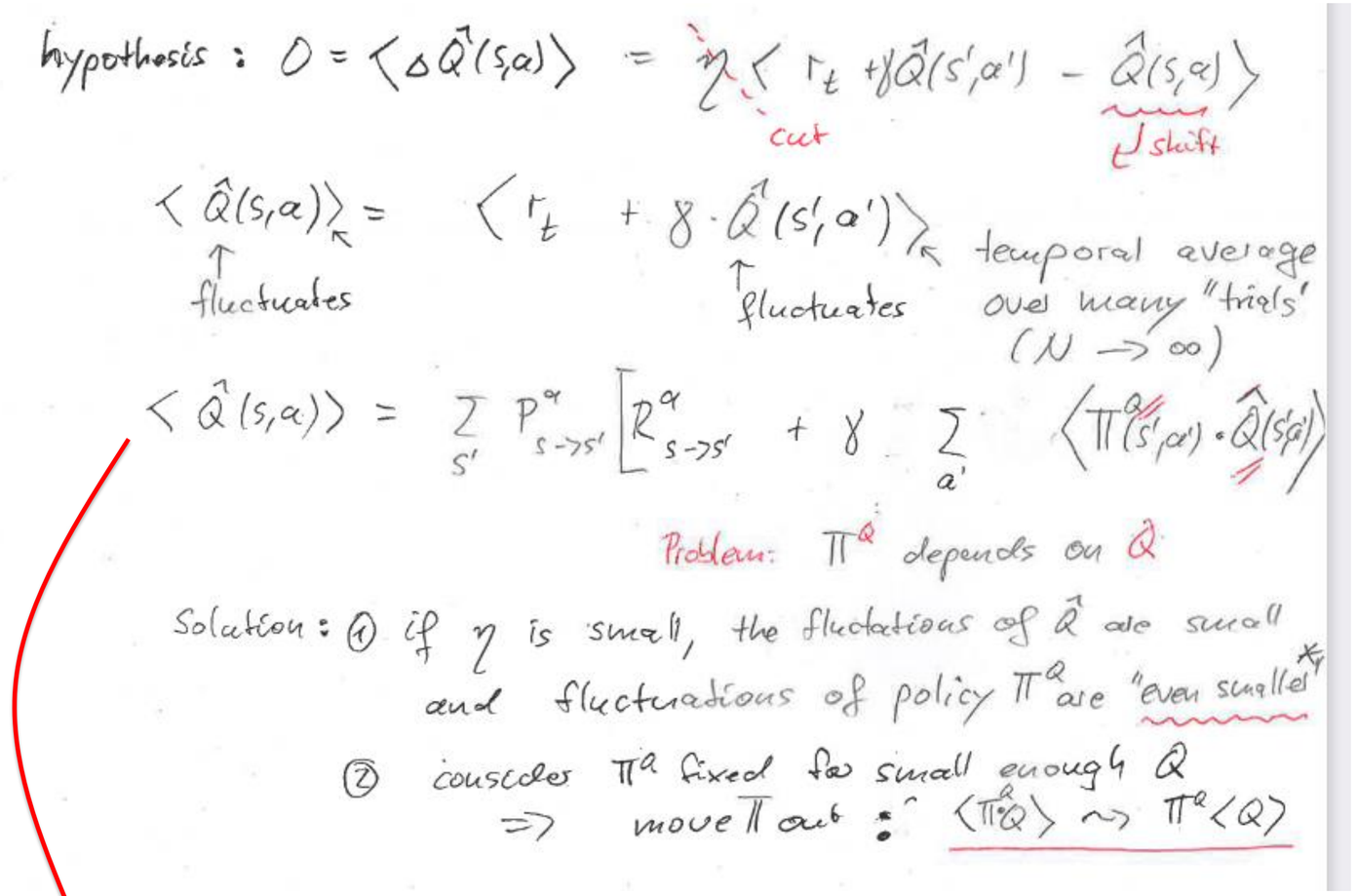
$$\left\langle \pi(\hat{Q}(s',a')) \cdot \hat{Q}(s',a') \right\rangle = \pi(\bar{q})\bar{q} + \text{'bias'}$$

Solution: bias vanishes if learning rate tends to zero.

Your Notes:

Variant B – temporal averaging: SARSA is consistent with Bellman (proof for small η)

Claim for Online SARSA: $\hat{Q}(s,a)$ jitters, but its mean $\langle \hat{Q}(s,a) \rangle$ solves Bellman



Look at graph to take expectations over update steps

- if algo is in state s, expectations "given s"
- if algo is on a branch (s,a), all remaining expectations are "given s and a"

*example of π^Q "even smaller": ε- greedy
only rank-order of Q-values matters: best / 2nd best / ...
if fluctuations $|\Delta \hat{Q}(s',a')| \ll Q(s',\text{best}) - Q(s',\text{2nd best})$
then π^Q remains stable!

$$\langle \hat{Q}(s,a) \rangle = \sum_{s'} P^a_{s \to s'} \left[R^a_{s \to s'} + \gamma \sum_{a'} \pi(s',a) \langle \hat{Q}(s',a') \rangle \right]$$

solves Bellman!

Notes: This is a proof sketch of the consistency of online SARSA **(Variant B).** We allow all Q-values to fluctuate around their expectation (visualized as ‘temporal averaging’) , but we still have to keep fluctuations of the policy negligibly small.

If we allow for small fluctuations of the policy, then we have to realize that these fluctuations are correlated with the fluctuations of Q-values. Thus the evaluation of the product $<\pi\ Q>$ is not trivial. Moreover, correlations can lead to a shift of the value and make the result inconsistent with the Bellman equation (‘bias’)..
This variant B therefore only works in the limit of vanishing learning rate.

The other theorem **(Variant A** that corresponds to the one on the slides in the main part of this lecture) takes expectations for FIXED Q-values (frozen). As discussed, such expectations for frozen parameter correspond in a broader sense to a ‘batch’ computation.

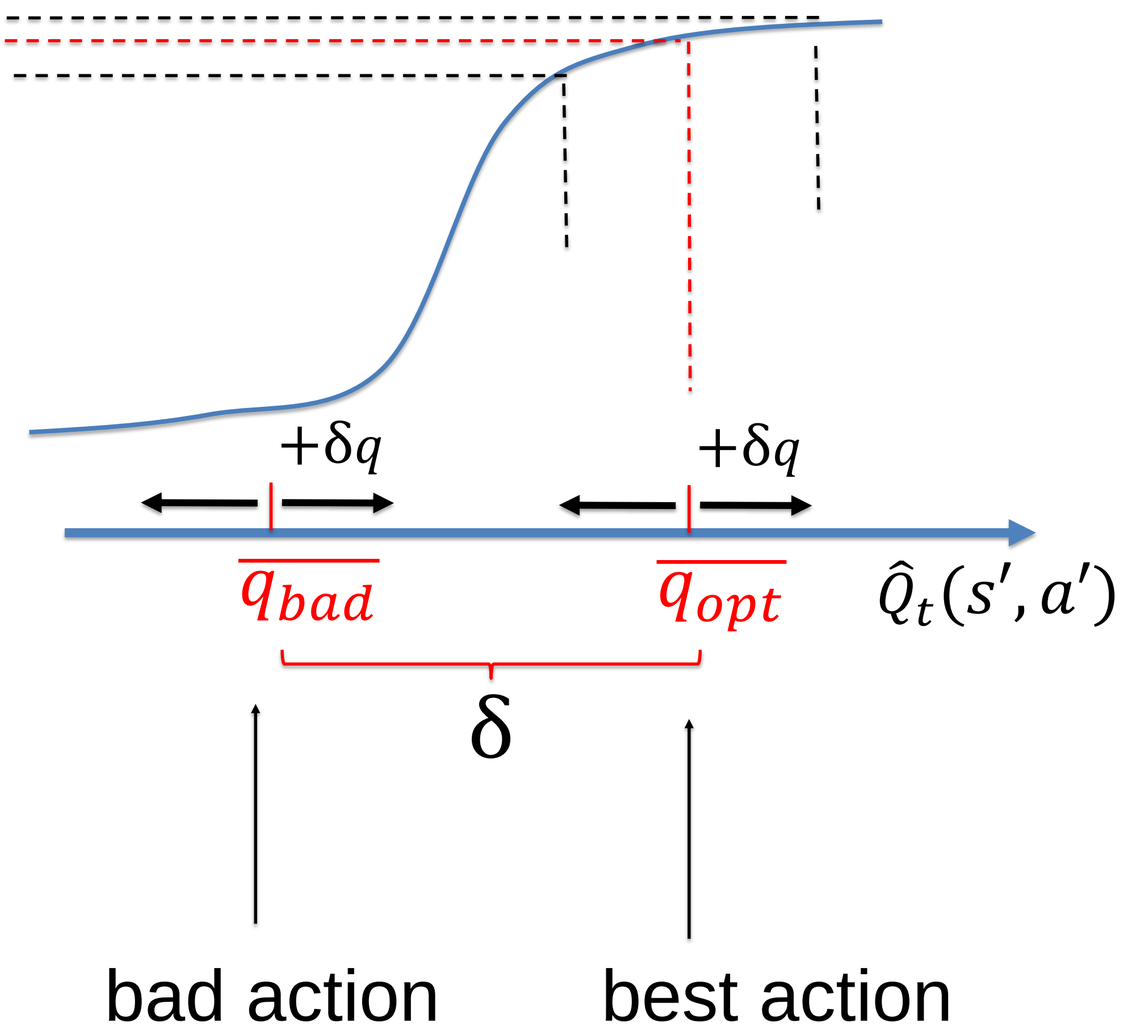
The advantage of Variant A is that it is easier to evaluate.
The advantage of **Variant B that it is much closer to the true online learning problem.**

Quiz: Exploration – Exploitation with Softmax policy

Softmax policy: take action a' with prob $P(a') = \frac{\exp[\beta Q(a')]}{\sum_a \exp[\beta Q(a)]}$

[ ] Suppose we have 3 possible actions a_1, a_2, a_3 and use the softmax policy. Is the following claim true?
For $Q(a_1) = 4, Q(a_2) = 1, Q(a_3) = 0$ the preference for action a_1 is more pronounced than for $Q(a_1) = 34, Q(a_2) = 31, Q(a_3) = 30$.

Notes: Hand-drawn sketch, relevant for the Quiz



Quiz: Effect of Fluctations of Q

softmax policy: take action a' with prob $\pi(a') = \frac{\exp[\beta Q(a')]}{\sum_a \exp[\beta Q(a)]}$

ε-greedy: take 'best action' with prob 1-ε

We have 3 possible actions a_1, a_2, a_3 with **ranking** (for exact Q-values)

$$\boldsymbol{Q(a_1) = 1 > Q(a_2) = Q(a_3) = Q(a_1) - \delta} \quad .$$

We have played SARSA for a long time. Suppose during online updating with SARSA the remaining fluctuations of $Q(a_k)$ are small:

$$\boldsymbol{\delta q} = |\Delta \boldsymbol{Q(a_k)}| < \boldsymbol{\delta/4}\textbf{; for all } \boldsymbol{k}\textbf{.}$$

[ ] Then with ε-greedy the actual action choices do not change during fluctuations.

Quiz: Softmax policy (not shown in class)

All Q values are initialized with the same value Q=0.1

Rewards in the system are r =0.5 for action 1 (always)
and r=1.0 for action 2 (always)

We use an iterative method and update Q-values with **eta=0.1**

1. [ ] if we use softmax with **beta = 10**, then, after 100 steps, action 2 is chosen almost always
2. [ ] if we use softmax with **beta = 0.1**, then, after 100 steps action 2 is taken about twice as often as action 1.

[yes], since beta[Q(a2)-Q(a1)]=5

[no], with beta=0.1, exp(beta*Q)=1+…
→both action chosen with about the same prob.

Softmax policy: take action a'
with prob $P(a') = \frac{\exp[\beta Q(a')]}{\sum_a \exp[\beta Q(a)]}$

Summary: Many Variations of a few ideas in TD learning

Learning outcomes and Conclusions

- TD – learning (Temporal Difference)

  → TD algo in narrow sens: V-values, rather than Q-values

- **Variations of SARSA**

  - off-policy Q-learning (greedy update)
  - Monte-Carlo
  - n-step Bellman equation/n-step SARSA

- **Eligibility traces**

  - allows rescaling of states, smoothens over time
  - similar to n-step SARSA

Basis of all:
iterative solution of Bellman equation

(previous slide)
Today we have seen a large variety of TD algorithms. All of these can be understood as iterative solutions of the Bellman equation.
The Bellman equation can be formulated with V-values or with Q-values. Bellman equations normally formulate a self-consistency condition over one step (nearest neighbors), but can be extended to n steps.
Monte Carlo methods do not exploit the ‘bootstrapping’ aspect of the Bellman equation since they do not rely on a self-consistency condition.
An n-step SARSA is somewhere intermediate between normal SARSA and Monte-Carlo.

Discretization of continuous spaces poses several problems.
The first problem is that a rescaling becomes necessary after a change of discretization scheme. This problem is solved by eligibility traces as well as by the n-step TD methods
The second problem is that a tabular scheme brakes down for fine discretizations. It is solved by a neural network where we learn the weights. Such a neural network enables generalization by forcing a ‘smooth’ V-value or Q-value,
But that is left for next week..

The END