Artificial Neural Networks: RL2

Wulfram Gerstner
EPFL, Lausanne, Switzerland

Variants of TD-learning methods and continuous space

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 designed to iteratively solve the Bellman equation
- Eligibility traces and n-step Q-learning to extend over time
- Continuous space and ANN models
- Models of actions and models of value

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 and 9.3

Reading for this week:

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

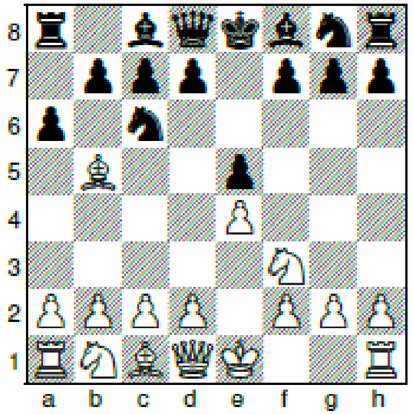
Chapter: 5.1-5.4 and 6.1-6.3 and 6.5-6.6, and 7.1-7.2 and 9.3

Background reading:

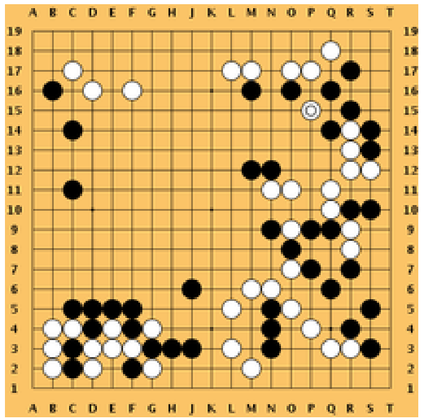
Temporal Difference Learning and TD-Gammon
by Gerald Tesauro (1995) pdf online

Review: Deep reinforcement learning

Chess



Go



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

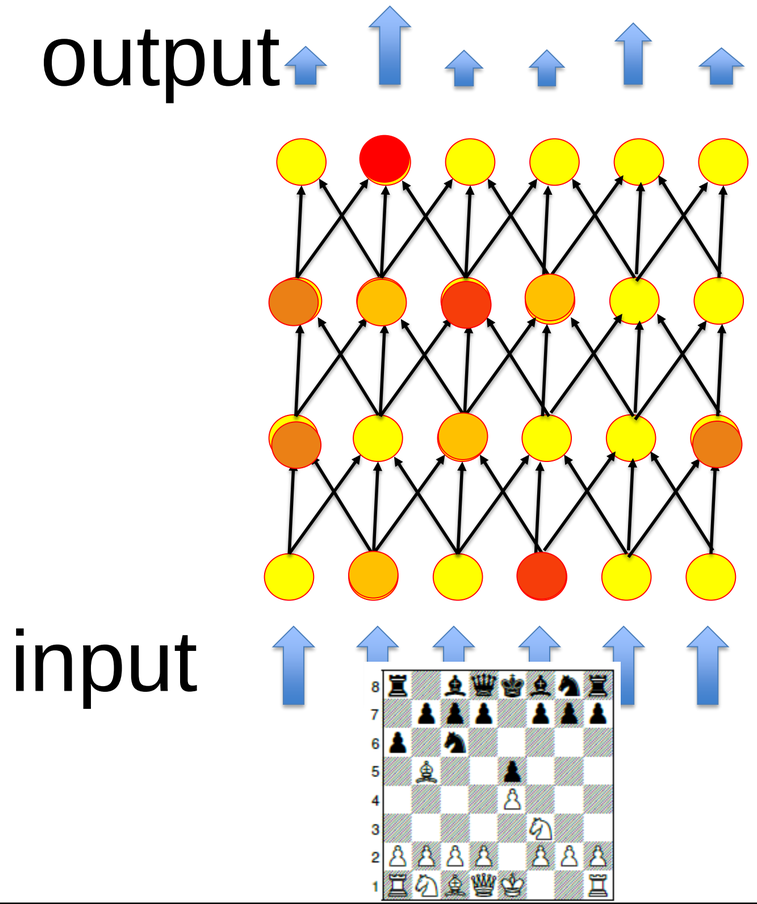
In Go, it beats Lee Sedol

Review: Deep reinforcement learning

Network for choosing action

action:

Advance king



Today

first steps toward learning

action choices in a small network:

- **How can we set-up such a network?**
- **What is the error function?**
- **How can we optimize weights?**

→ Temporal Difference Learning

→ Variations of SARSA

→ Continuous/Large State space

(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 networks is via an error-function – 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 $\pi(s,a)$

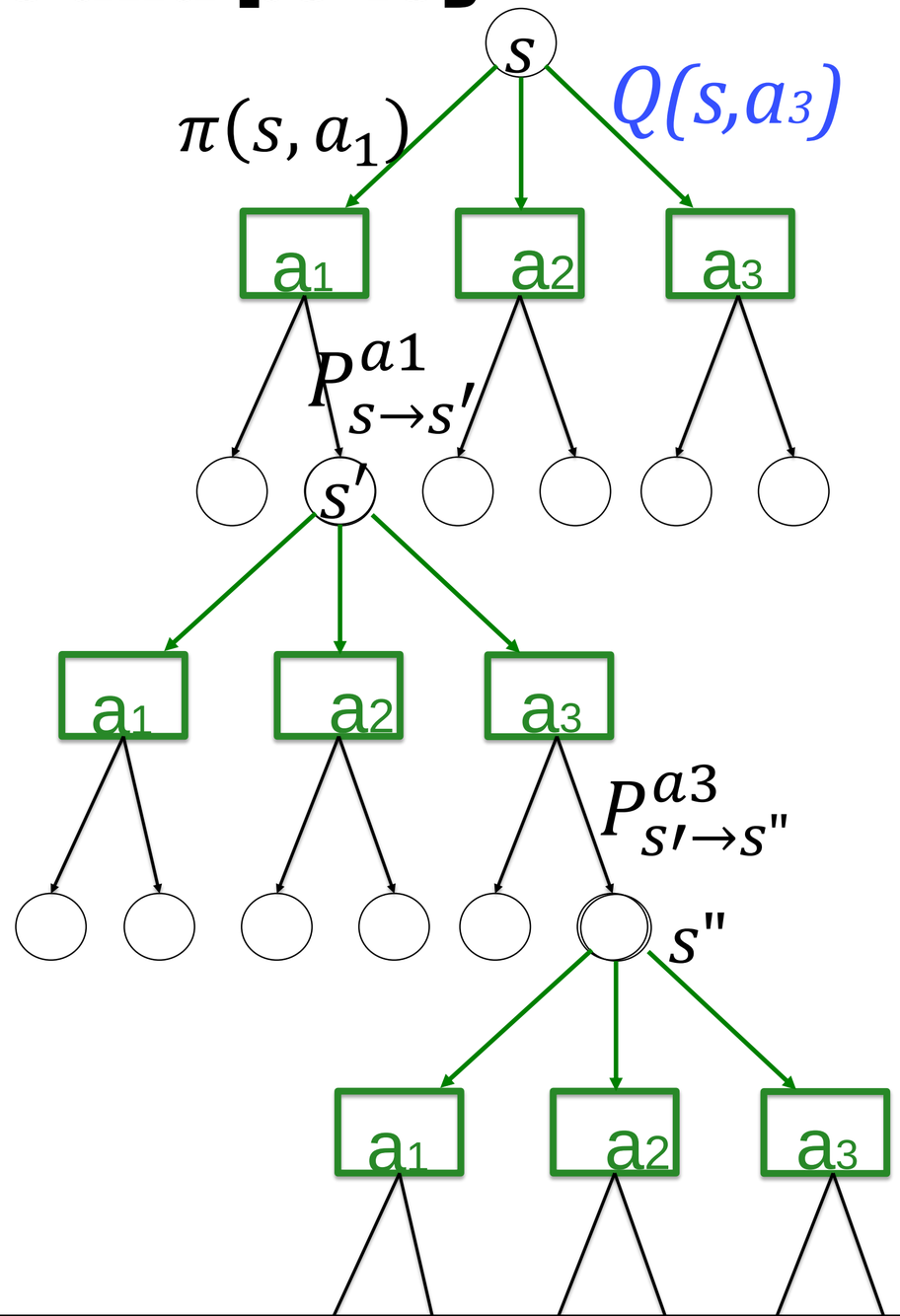
probability to choose action a in state s

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

Examples of policy:
-epsilon-greedy
-softmax

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

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

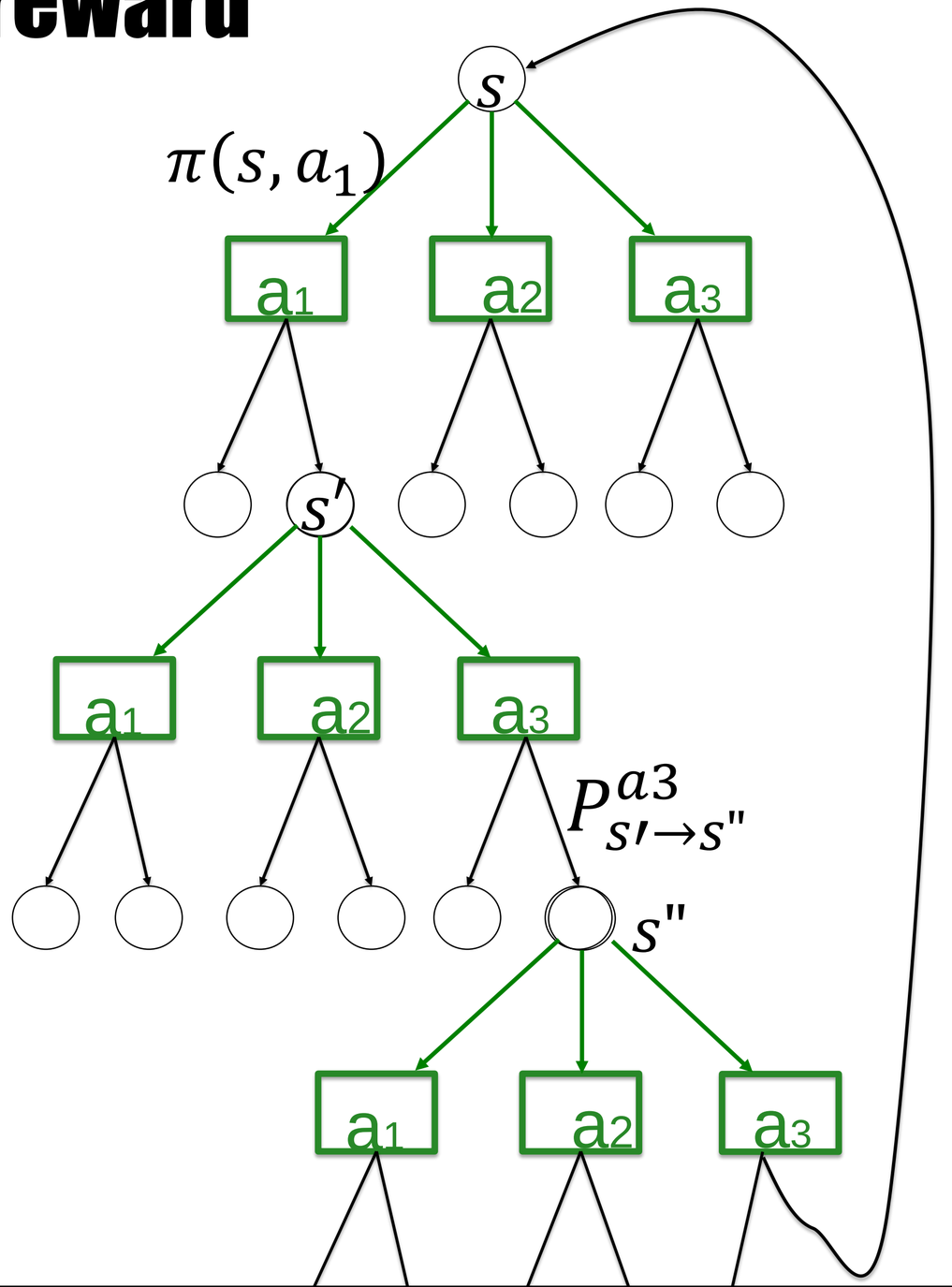


Review Total expected (discounted) reward

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)

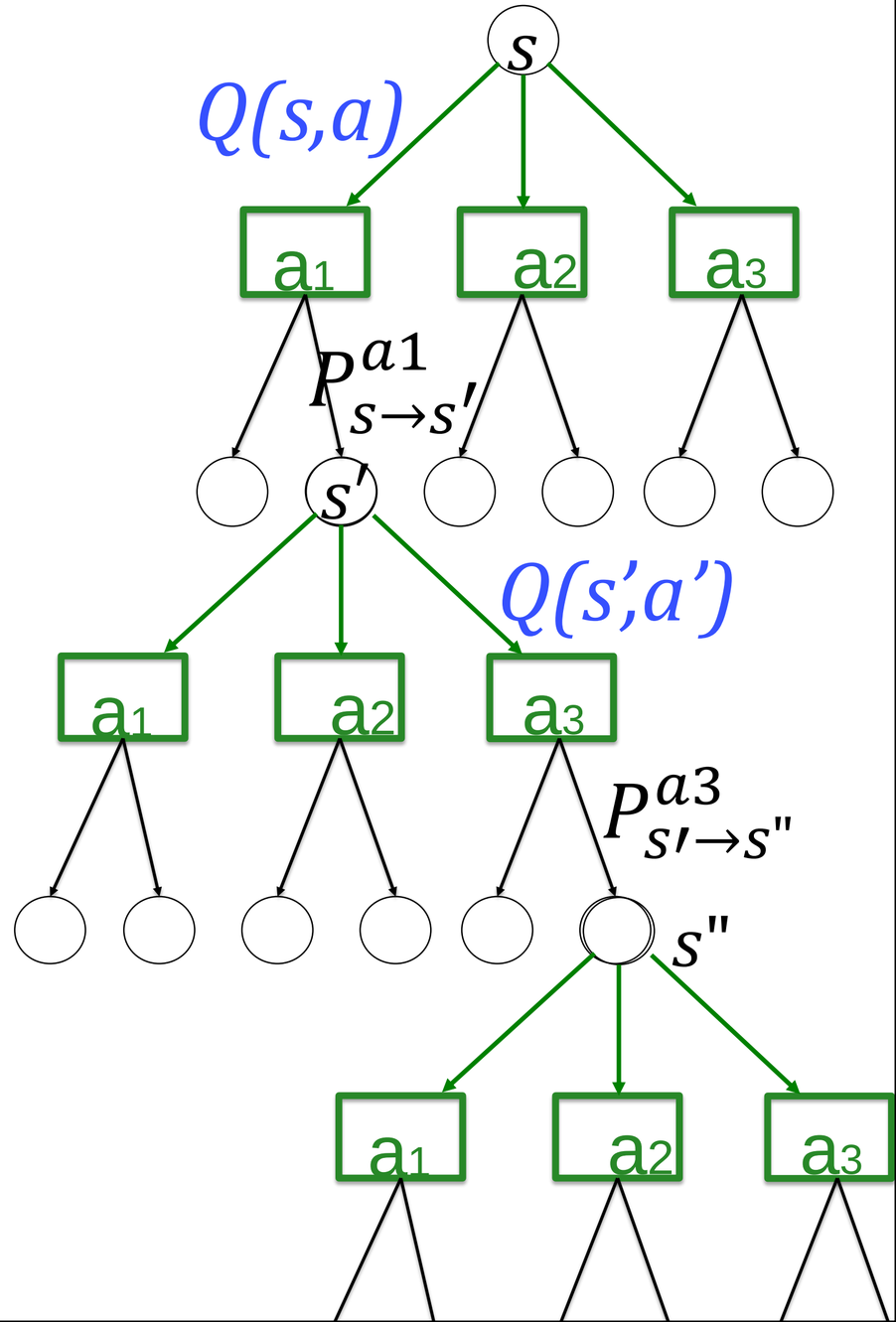
We know from previous lectures that RL works with states and actions that allow probabilistic transitions between the states.

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

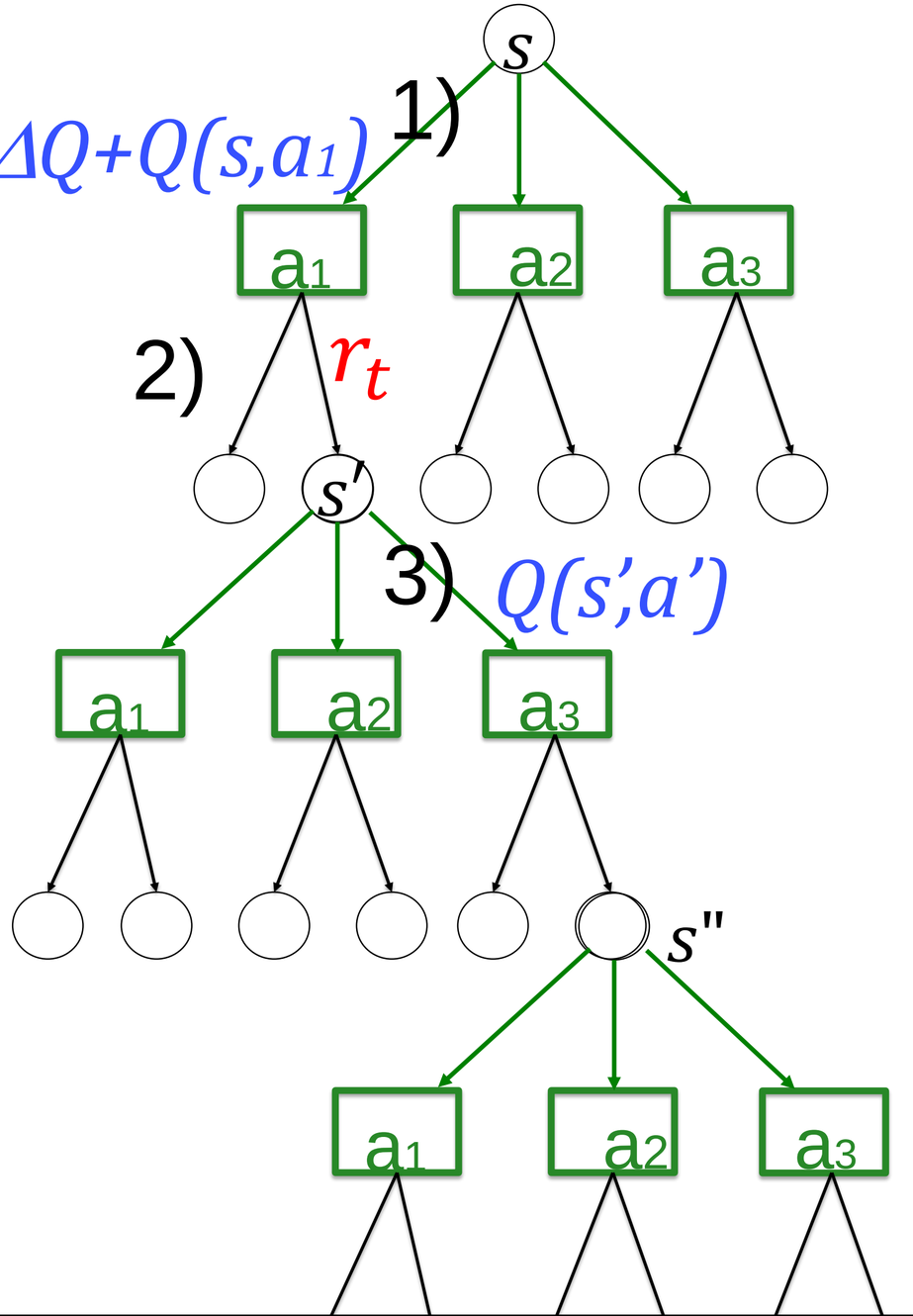
Initialise Q values
Start from initial state **s**

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

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

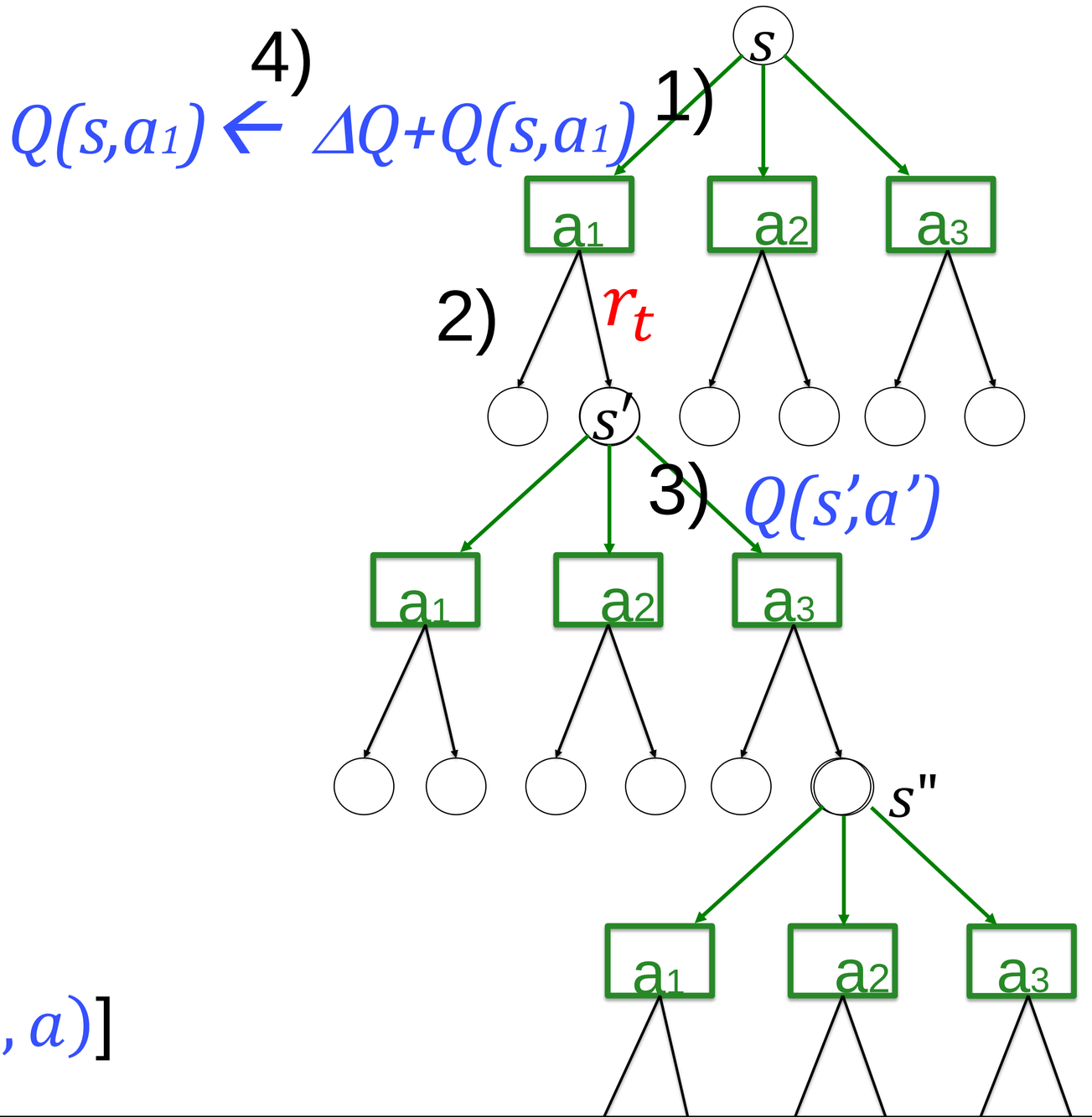
5) set: s ← s'; a ← a'
6) Goto 2)

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

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

(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.

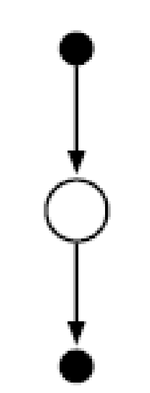
In SARSA, when you are in (s’,a’) you need to go back to the branch (s,a) so that you can do the SARSA update.

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.

Artificial Neural Networks: Lecture RL2

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Variants of TD-learning methods and continuous space

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

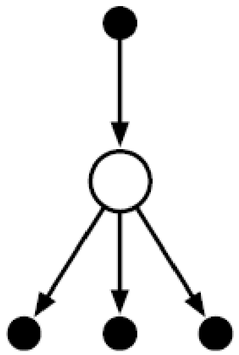
Expected SARSA for estimating Q

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

action

state

action



Expected Sarsa

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 we do not yet take the next action but average over all possible next action with a weight given by the policy π.

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

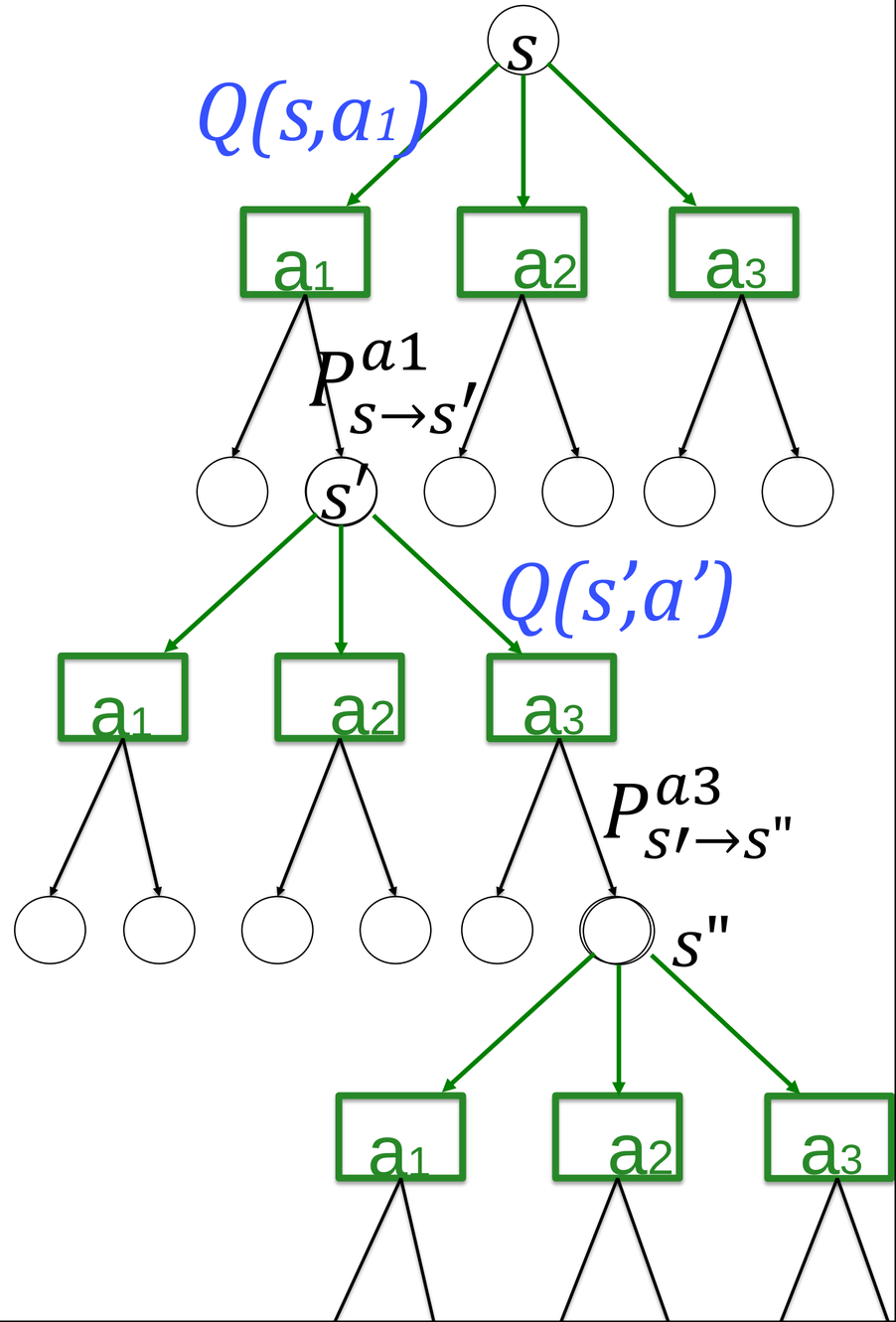
Remark:

Sometimes Bellman equation is written for greedy policy:

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

with action

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

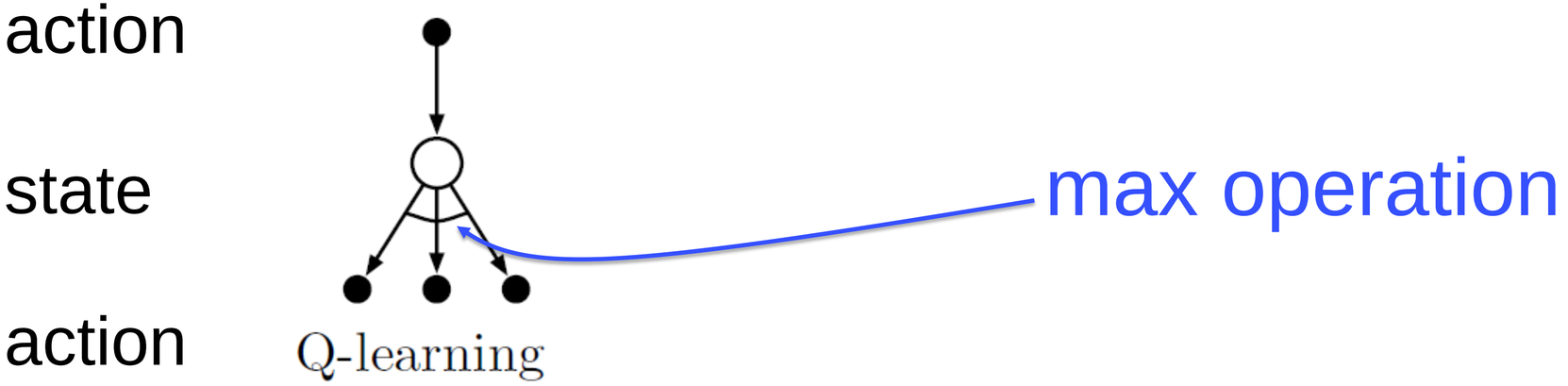


(previous slide)
The next variant is Q-learning.
Q-learning uses not an average with the current policy, but performs the averaging with the best policy, i.e., the greedy policy.
The idea is that you run a policy that includes exploration. However, since you know that after learning you will use the greedy policy so as to maximize your returns, you already update the Q-values according the greedy 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'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal



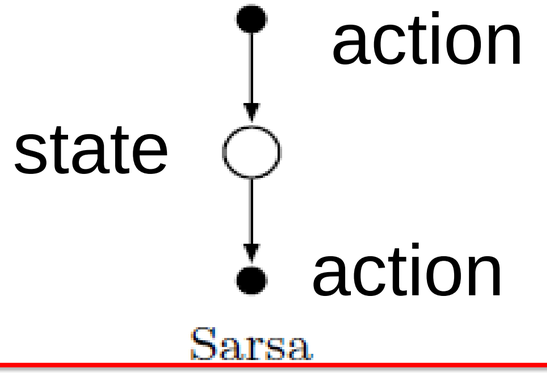
Sutton and Barto, Ch. 6.5

(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.

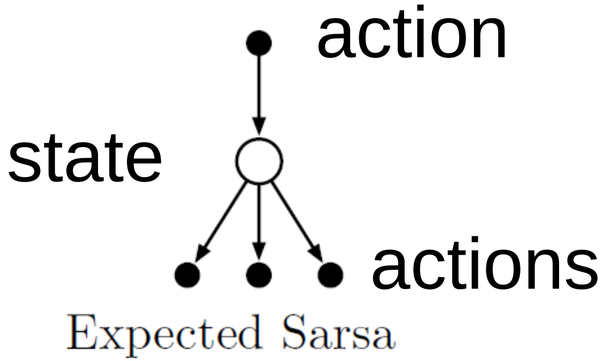
In Q-learning the update step is such that the current reward should explain the difference between Q(s,a) and the **maximum** Q(s’,a’) running over all possible actions a’. It is a TD algorithm (Temporal Difference), because neighboring states are visited one after the other. Hence neighbors are one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

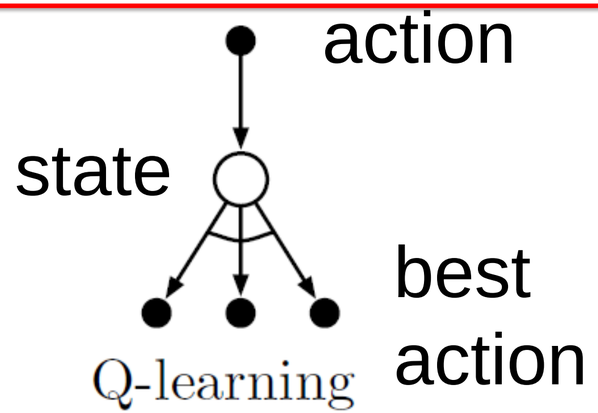
Summary: SARSA and related algorithms



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.

Artificial Neural Networks: Lecture RL2

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Variants of TD-learning methods and continuous space

Part 3: Temporal Difference Learning

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

(previous slide)
We now explore other Temporal Difference algorithms

TD-learning as bootstrap estimation

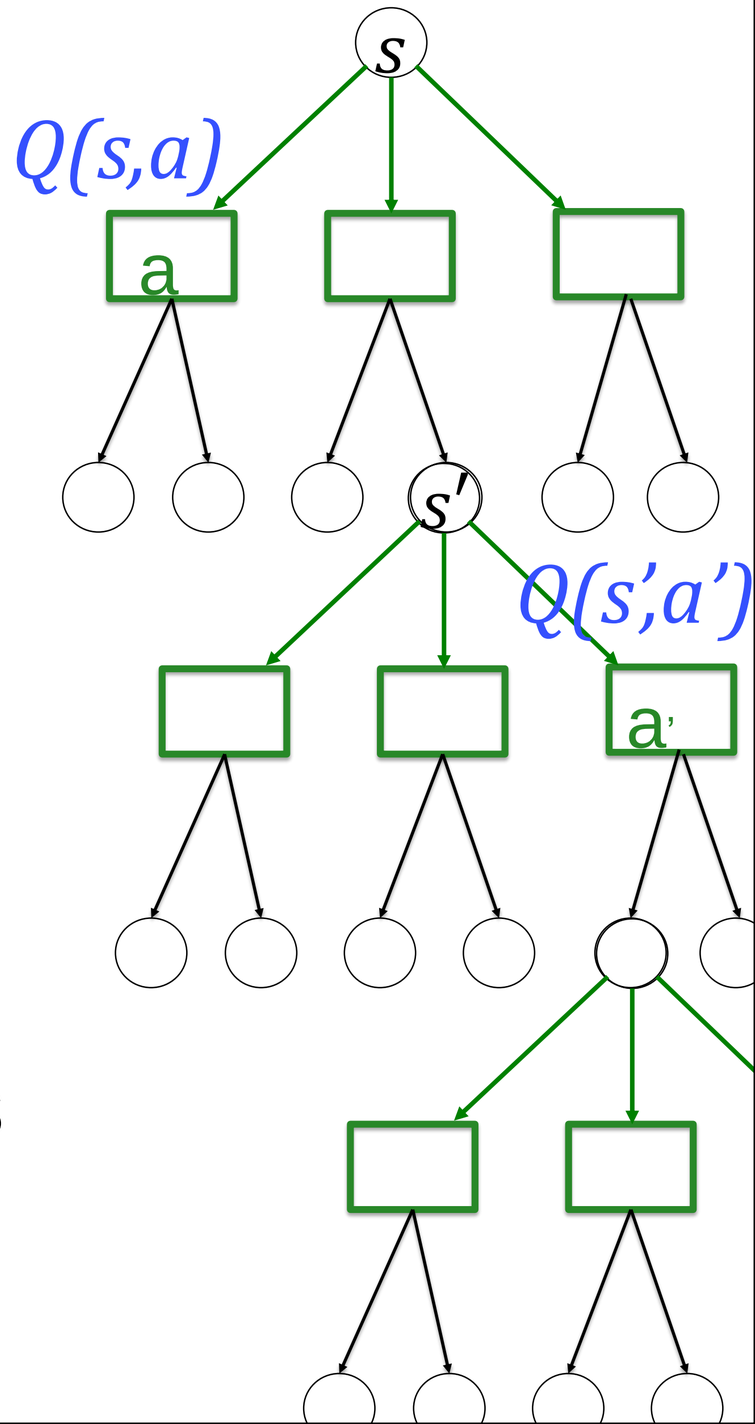
'bootstrap': summary of previous information

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



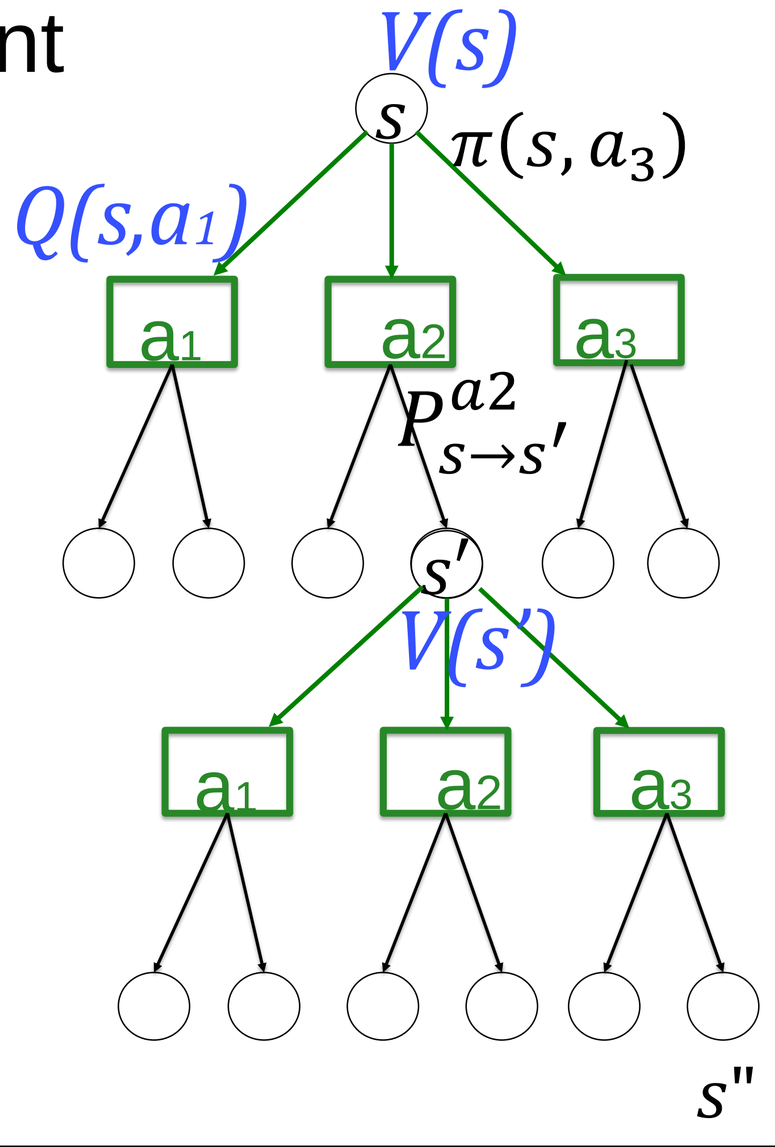
(previous slide)
1) If the agent runs through the state-action graph, neighboring states are one **time step** away from each other. This explains the term ‘Temporal Difference (TD)’

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.

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).

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).

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

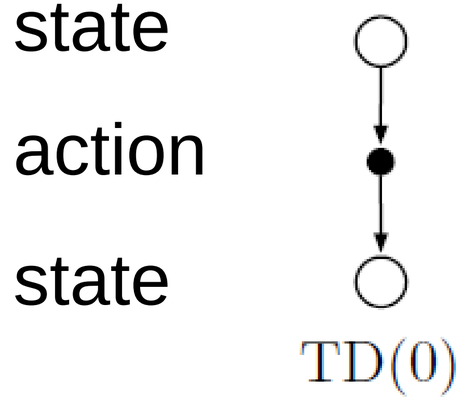
The resulting Bellman equation for V-values looks similar to that of Q-values, except that the location of the summation signs has been shifted.

Standard TD-learning

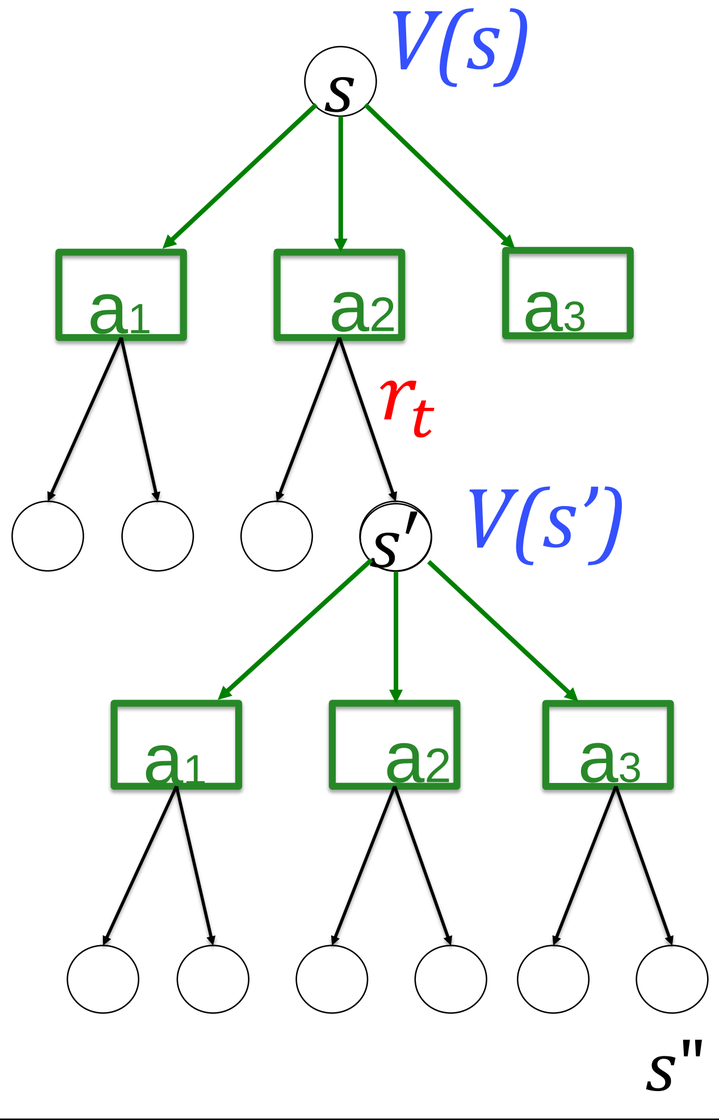
Tabular TD(0) for estimating v_π

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



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



(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 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.

Artificial Neural Networks: Lecture RL2

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Variants of TD-learning methods and continuous space

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

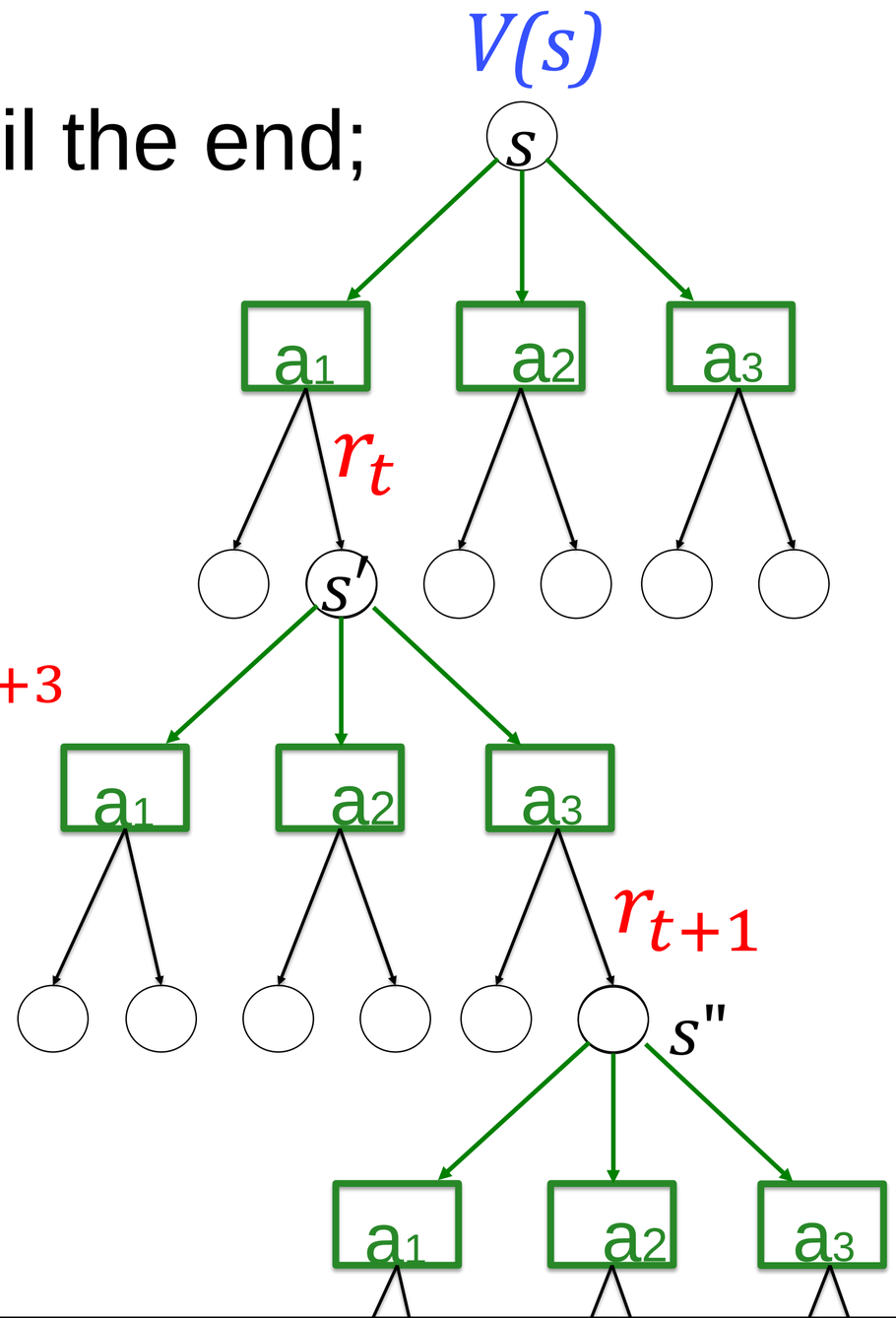
Monte-Carlo Estimation

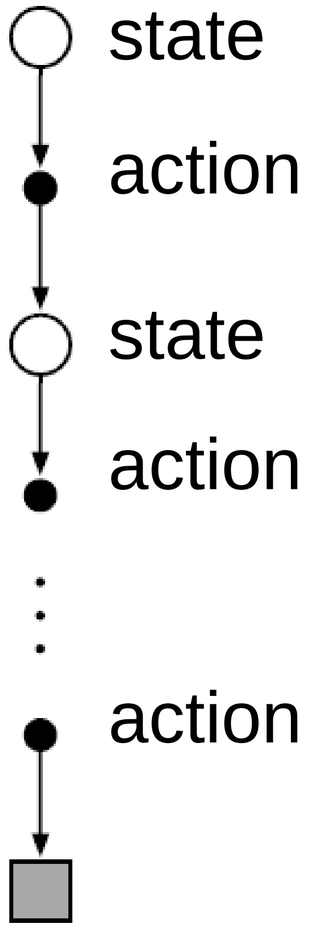
play a trial (episode) until the end;

then update, using
the total accumulated
reward (='Return') =

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

same episode is also
used to estimate V(s')





end of trial

(previous slide)
1) Suppose you want to estimate the value V(s) of state s.
V(s) is the EXPECTED total discounted reward.

To estimate V(s) you start in state s, run until the end and evaluate for this single episode the return

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

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

2) You can be smart and you the SAME episode also to estimate the value V(s') of other states s'. Thus while you move along the graph, you open an estimation variable for each of the states that you visit.

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

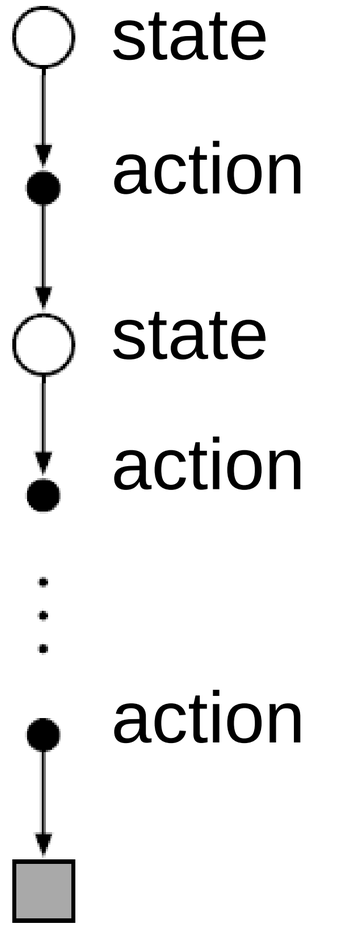
Monte-Carlo Estimation of V-values

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

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)
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.

Monte-Carlo Estimation of Q-values (batch)

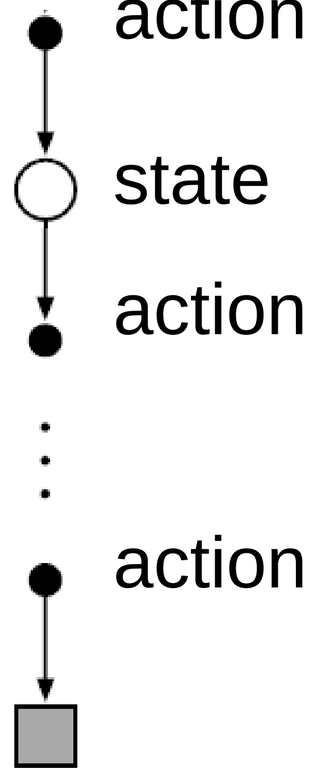
Start at a random state-action pair (s,a) (exploring starts)

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

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

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 open Q-estimators 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).

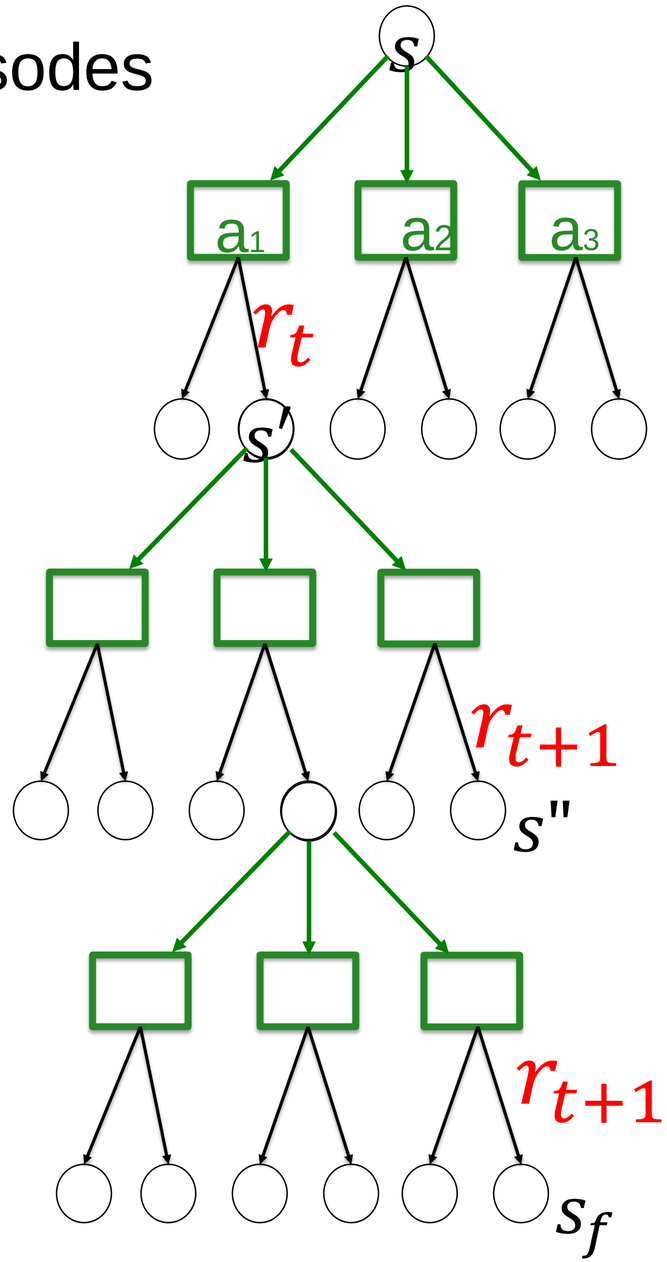
Note that
- stochasticity in the initial states assures that all pairs (s,a) are tested, even if the policy is not stochastic.
- In theory, this estimation method is hence compatible with a greedy policy.
- In practice, I always recommend epsilon greedy (and we can reduce epsilon as we have learned more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

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)

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 …"

Question:

Three ways to estimate Q-values with policy π:

1) SARSA/expected SARSA (online, TD, bootstrap)
2) Monte-Carlo (batch over many episodes)
3) Batch-expected-SARSA learning (batch over many episodes)

'work with empirical Bellman equation, bootstrap'

We have played N trials.
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?

In both **batch** algorithms you have to play several episodes before you do the update. The Bellman equation approach (Batch-Expected-SARSA) uses the idea of ‘bootstrapping’ whereas Monte-Carlo does not.

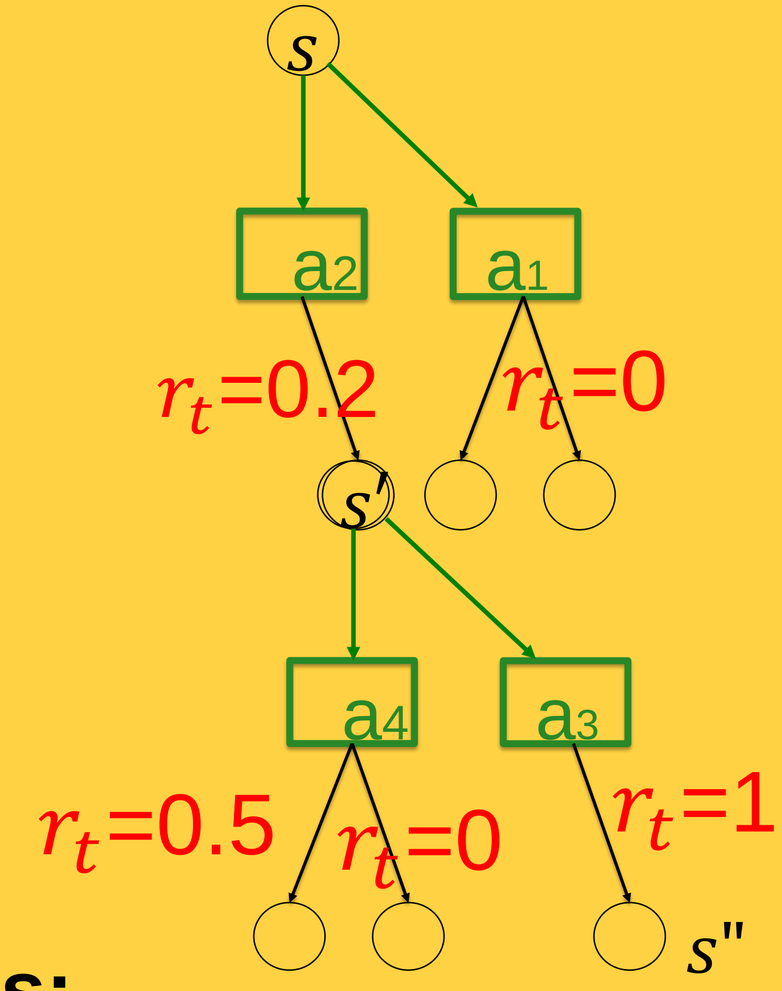
Q-learning or SARSA both use ‘bootstrapping’ since they update Q-values based on other Q-values. Q-learning has the max-operation, whereas SARSA is ‘on-policy’. Both Q-learning and SARSA are **Online** (as opposed to batch).

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

Monte-Carlo versus TD methods (Exercise 1, preparation)

Not discounted. 10 example episodes:

1: **s,a2**→ r=0.2,s',a4→ r=0
2: s',a3→ r=1
3: s',a4→ r=0
4: s',a3→ r=1
5: **s,a1** → r=0
6: s',a4→ r=0
7: s',a4→ r=0.5
8: s',a3→ r=1
9. **s,a2**→ r=0.2,s',a4→ r=0.5
10. **s,a1** → r=0



Batch update of Q(s,a) after all 10 trials:

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

(previous slide)
Batch mode means that we update after having played all 10 trials
(as opposed to normal SARSA where you update while you run through each trial).

Tip: For batch SARSA start from the bottom of the graph.
Notes: set the discount factor γ to one; if appropriate, initialize Q-values with zero.

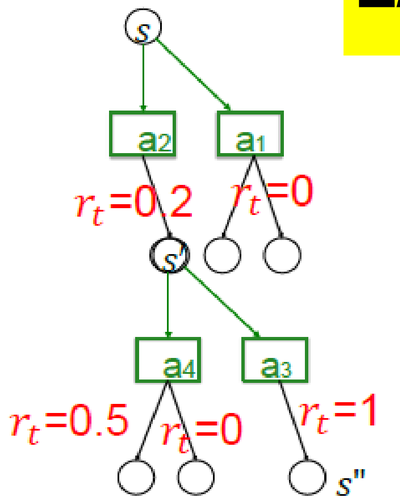
If you want to do online SARSA: use a learning rate that is inversely proportional to the number of time you have encountered a transition (as in 1-step horizon example last week).

Exercise 1a and 1b. Pause video

Exercise 1 a and b (c and d at home!)

example trials:

1: **s,a2**→ r=0.2,s',a4→ r=0
2: s',a3→ r=1
3: s',a4→ r=0
4: s',a3→ r=1
5: **s,a1** → r=0
6: s',a4→ r=0
7: s',a4→ r=0.5
8: s',a3→ r=1
9. **s,a2**→ r=0.2,s',a4→ r=0.5
10. **s,a1** → r=0



TASK: estimate Q-values with 2 methods:

- **Monte-Carlo batch mode**
- **Batch-expected-SARSA**

Monte-Carlo batch mode: update once after 10 trials

Batch-Expected-SARSA: update once after 10 trials, use Bellman equation

$$Q(s,a2) = \{\langle r_t \rangle + \gamma \left[\sum \pi(s',a')\, Q(s',a')\right]\}$$

γ=1

$\pi(s',a')$=0.5

1c: also compare with online version of expected SARSA

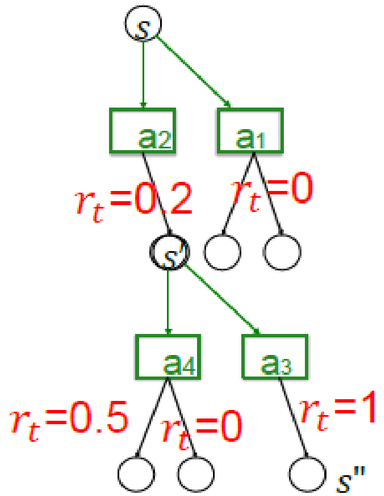
Note that the Bellman equation given on the slide has already been simplified by the fact that in the starting state s and action a2 there is not stochastic branching.

Space for your calculations.

Exercise 1 a and b (c and d at home!)

example trials:

1: **s,a2**→ r=0.2,s',a4→ r=0
2: s',a3→ r=1
3: s',a4→ r=0
4: s',a3→ r=1
5: **s,a1** → r=0
6: s',a4→ r=0
7: s',a4→ r=0.5
8: s',a3→ r=1
9. **s,a2**→ r=0.2,s',a4→ r=0.5
10. **s,a1** → r=0



TASK: estimate Q-values with 2 methods

	in trial	Q, Monte Carlo	Q, exp. SARSA

Blackboard 2 solution:

Monte-Carlo batch mode:
update once after 10 trials

Batch-Expected-SARSA:
update once after 10 trials

$$Q(s,a2) = \{\langle r_t \rangle + \gamma \left[\sum \pi(s',a')\, Q(s',a')\right]\}$$

γ=1

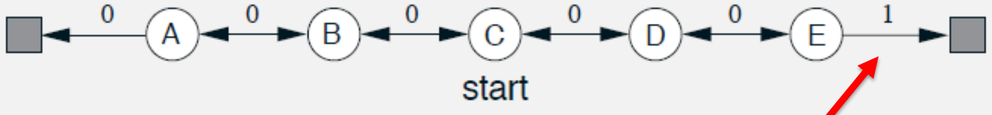
$\pi(s',a')$=0.5

Space for your calculations.

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

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

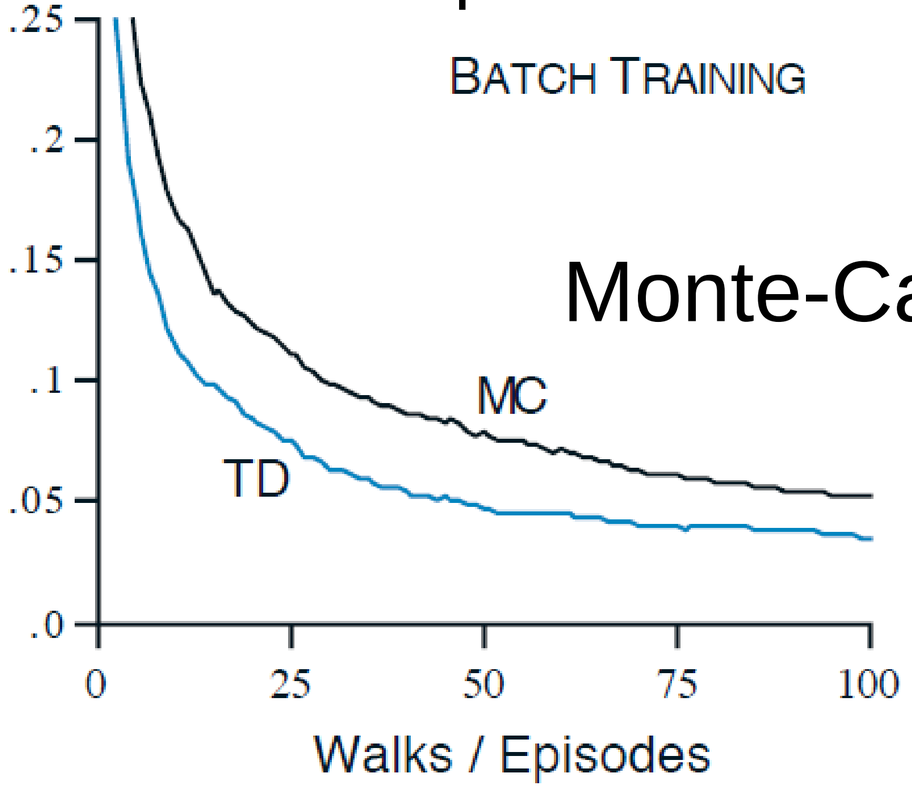
Example: 1d random walk



r=1

RMS error, averaged over states

error with respect to exact V-value



Monte-Carlo

Conclusion:
TD is better than Monte Carlo

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)

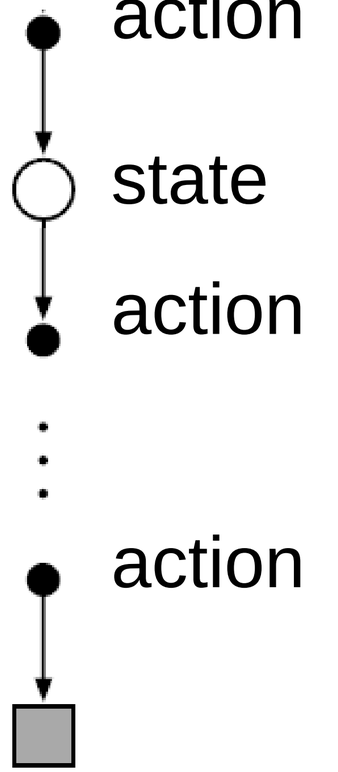
Combine epsilon-greedy policy with Monte-Carlo Q-estimates

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)
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. 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

Artificial Neural Networks: Lecture RL2

Wulfram Gerstner
EPFL, Lausanne, Switzerland

Variants of TD-learning methods and continuous space

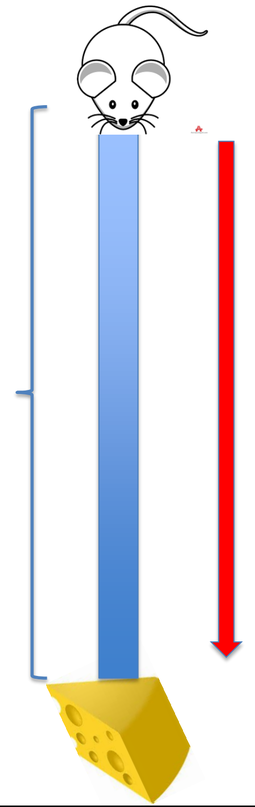
Part 5: Eligibility traces

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

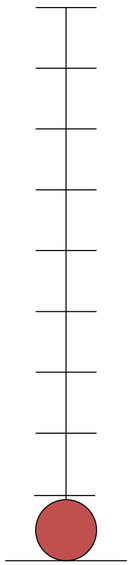
(previous slide)
So far we have worked with discrete states.

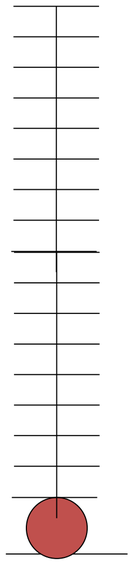
Exercise from last week: one-dimensional track

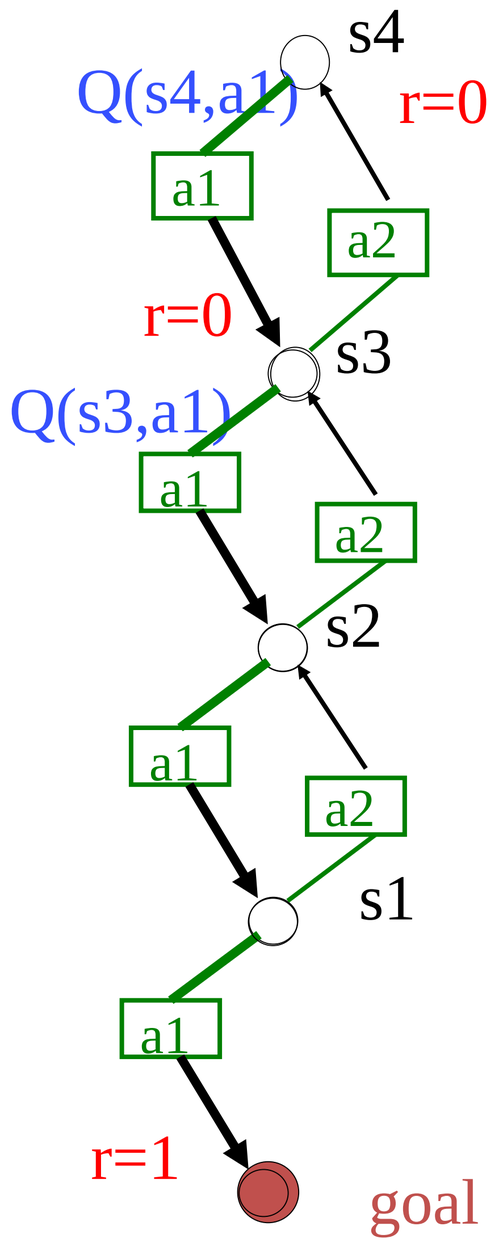
top view



Discretize state







(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

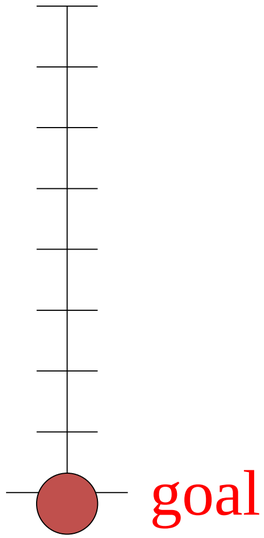
- **Initialise Q values at 0. Start trials at top (s4).**
- **Update of Q values with SARSA**

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

- **Policy for action choice:**

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

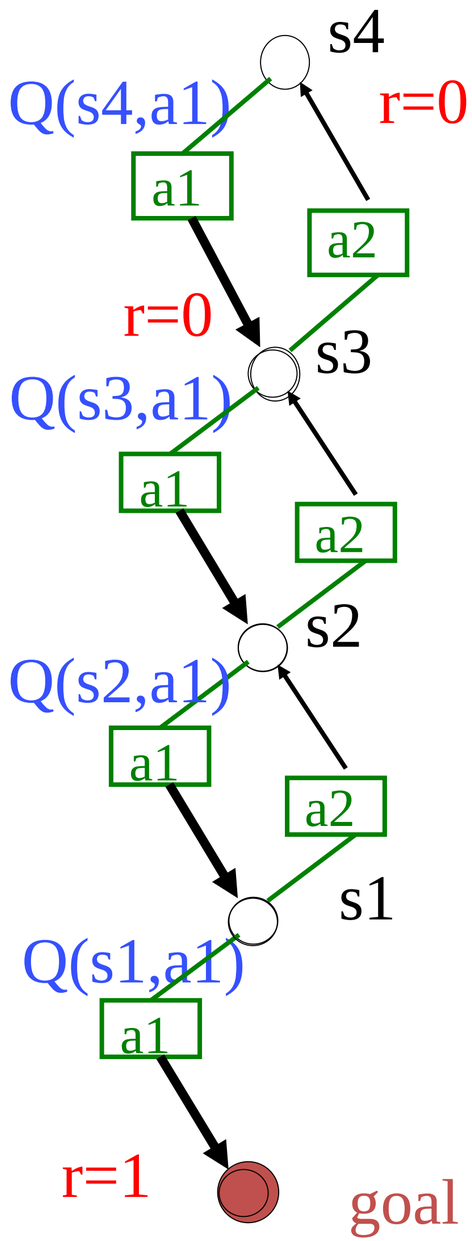
  To break ties: take action a1



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

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

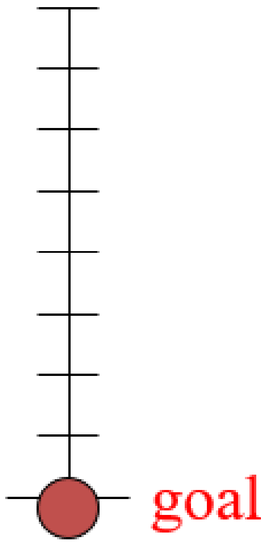
[ ] After 2 trials the Q-value Q(s3,a1)>0



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

Exercise from last week: one-dimensional track

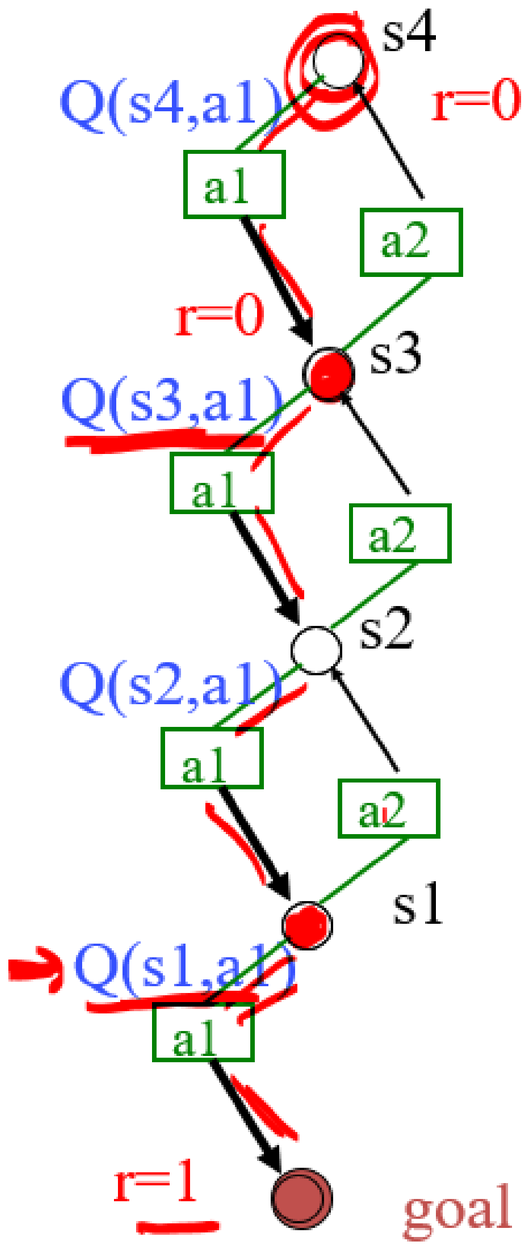
- **Initialise Q values at 0. Start trials at top (s4).**
- **Update of Q values with SARSA**
- $\Delta Q(s,a) = \eta \, [r + \gamma \, Q(s',a') - Q(s,a)]$
- **Policy for action choice:**
  - Pick most often action $a_t^* = \arg\max_a Q_a(s,a)$
  - To break ties: take action a1



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

[x] [ ] 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!!!

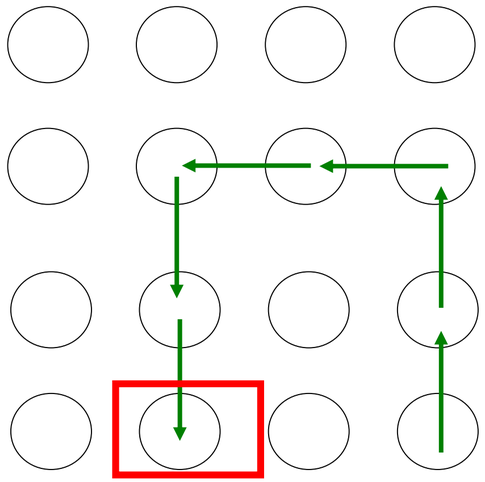
→ Something is wrong with the discrete-state SARSA algo

(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)

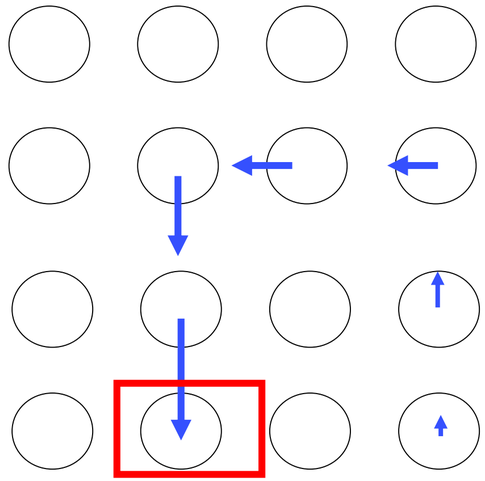


Idea:

- 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

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

Quiz: Eligibility Traces

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

[x] [ ] 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

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

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

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

Artificial Neural Networks: Lecture RL2

Wulfram Gerstner
EPFL, Lausanne, Switzerland

Variants of TD-learning methods and continuous space

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:

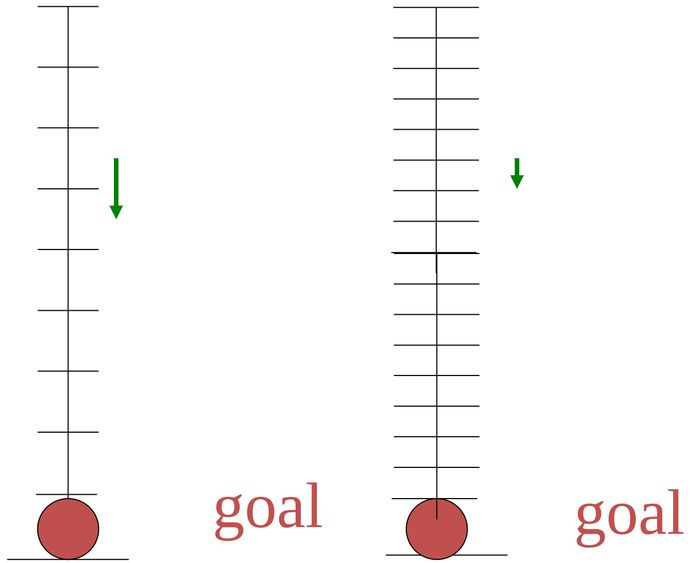
-‘Flow of information’ back from goal is slow

- 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.

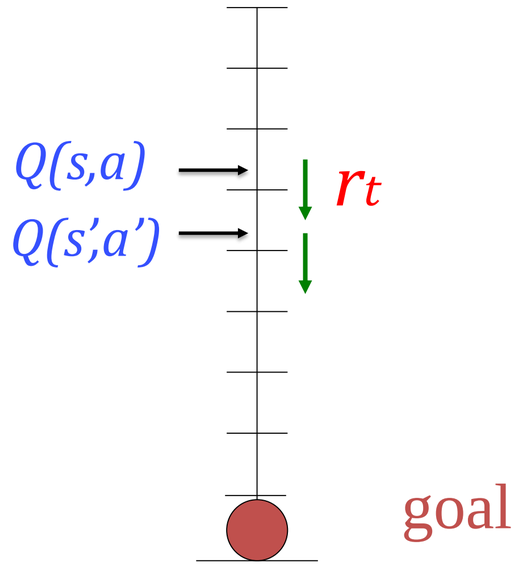
Solution 2: n=step SARSA

Standard SARSA

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

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

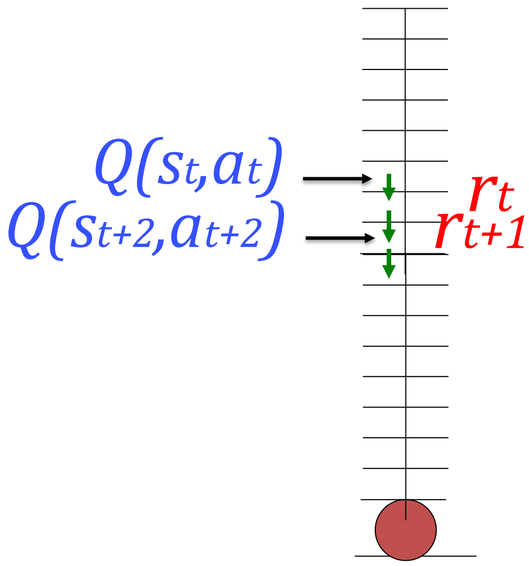
Temporal Difference (TD)



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



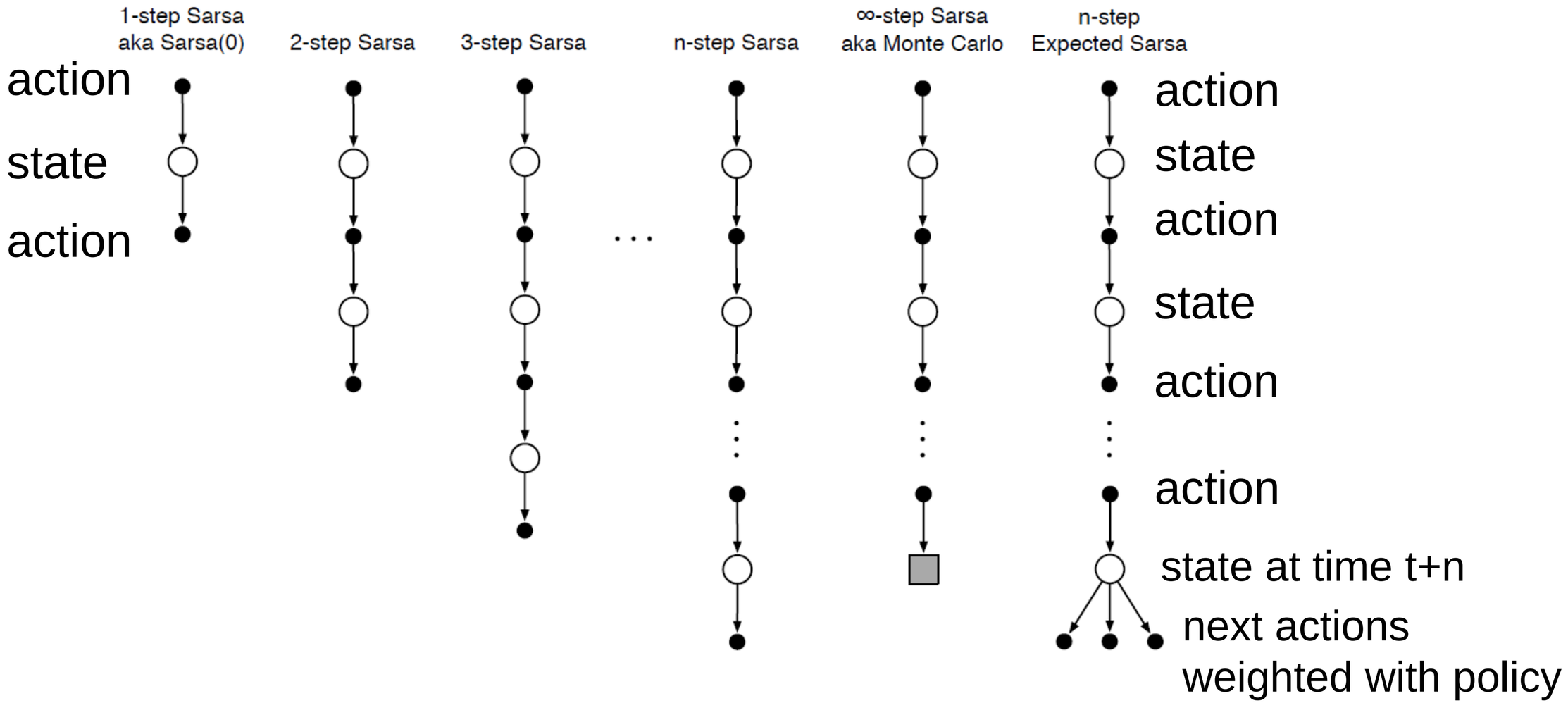
(previous slide)
Reminder:
SARSA and other standard TD methods compare the reward with **neighboring** Q-values.

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

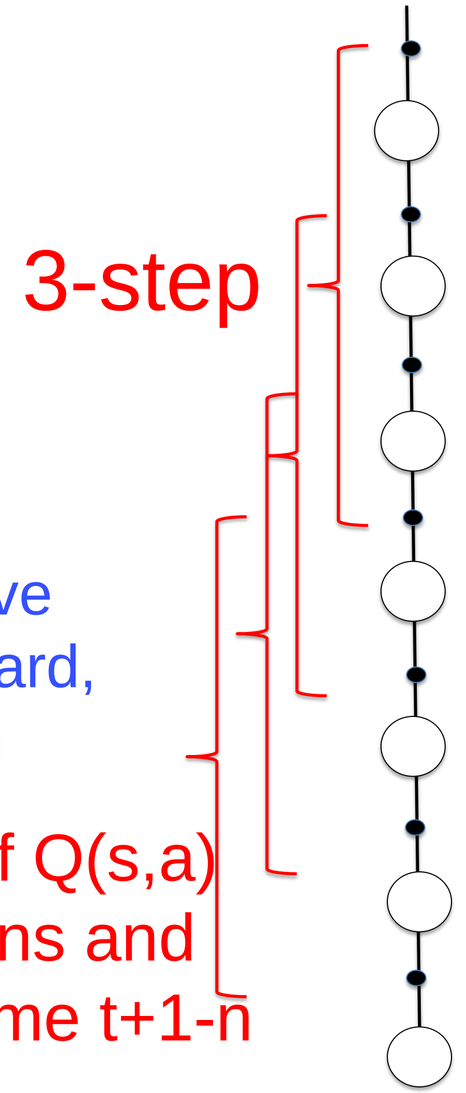
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be ε-greedy with respect to Q, or to another stochastic policy
Parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer n
All store and access operations (for S_t, A_t, and R_t)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

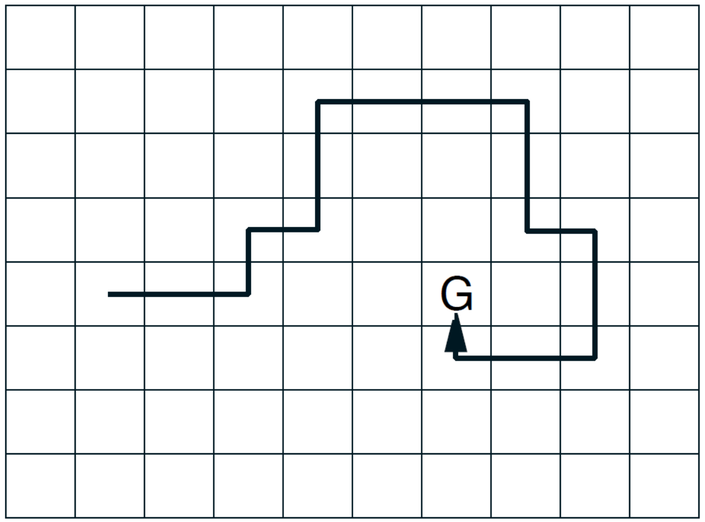
(2) To this G the Q-value of the *n*th state is added (unless the episode terminates before)

(3) The update then happens with this new G as a target and learning rate alpha.

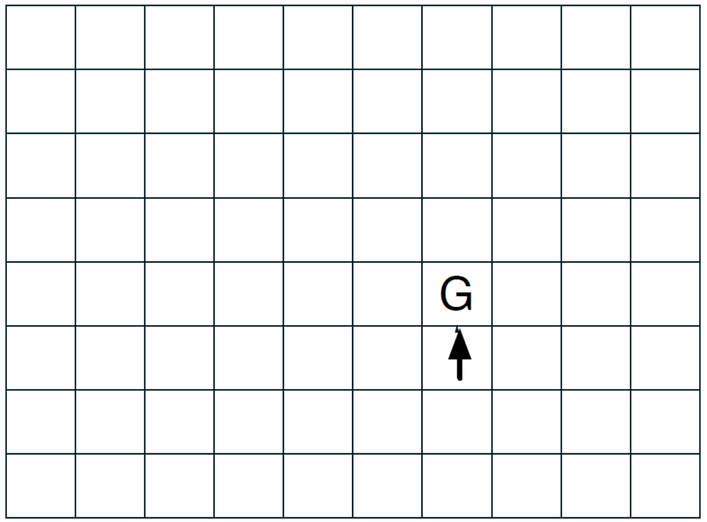
For some reason Sutton and Barto suggest epsilon-greedy in the pseudo-algo, but I changed this to arbitrary stochastic policies. Stochastic is important to make sure that all branches are explored; apart from this: SARSA is an on-policy algorithm and whatever you choose as a policy should work and yield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

Example: 10-step SARSA

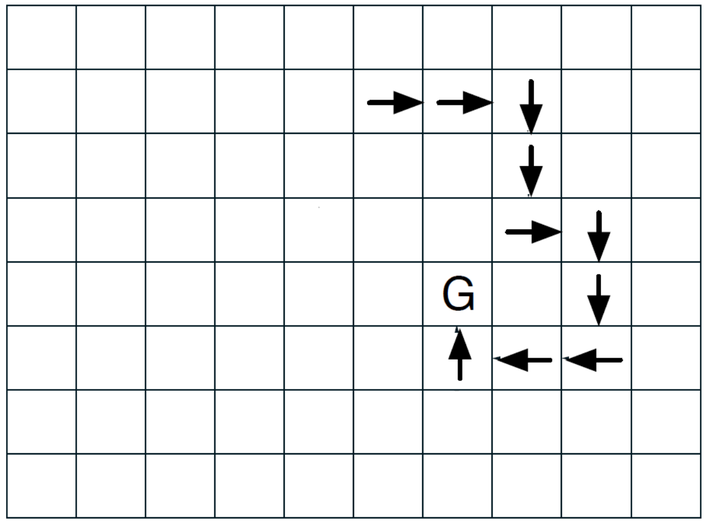
Path taken



Action values increased
by one-step Sarsa



Action values increased
by 10-step Sarsa

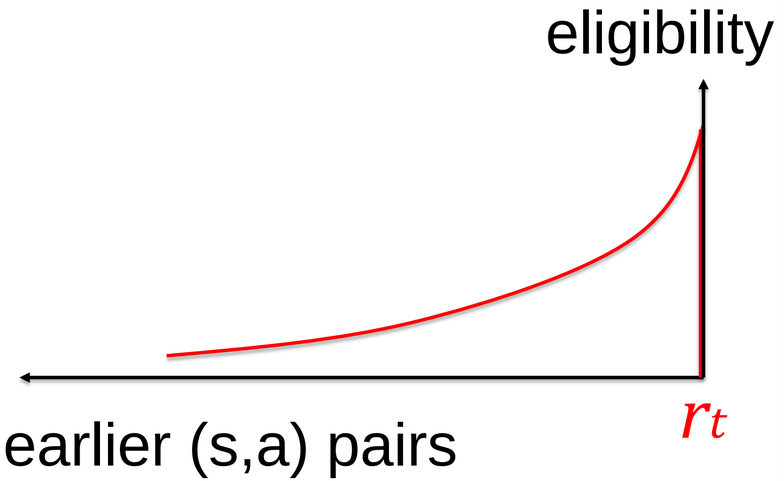


(previous slide).
The graphic suggests that the results of 10-step SARSA are very similar to an eligibility trace – which is indeed the case. therefore the two solutions (eligibility trace and n-step TD learning) are in fact closely related.

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

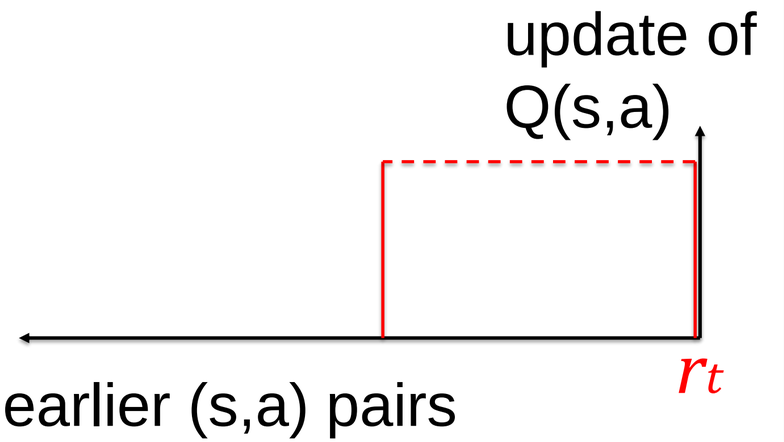


either

→ 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.

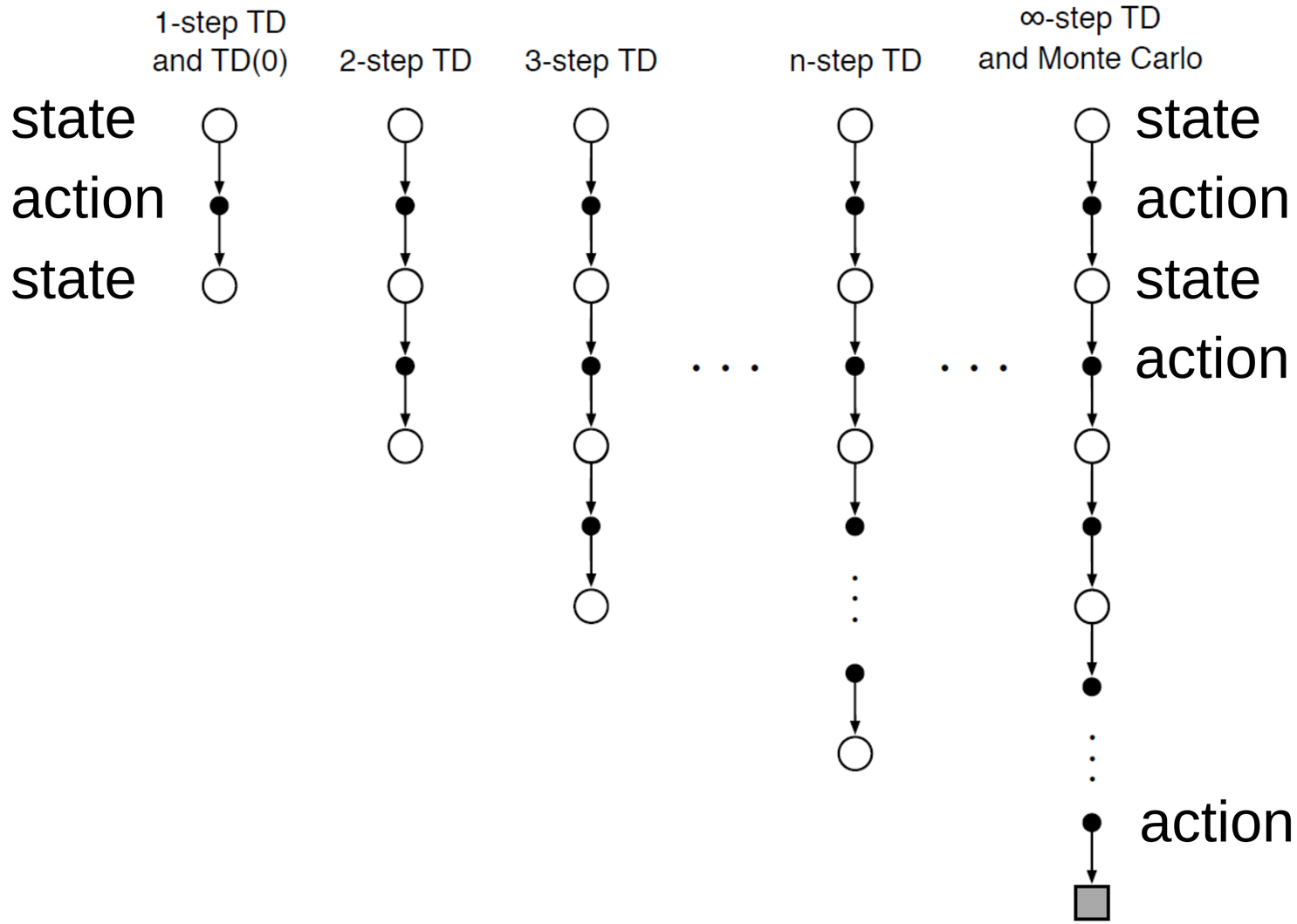
(previous slide)
One-step TD algorithms have problems as approximations to continuous states.
There are two closely related solutions, eligibility traces and n-step TD algorithms.

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 $\tau = T - 1$

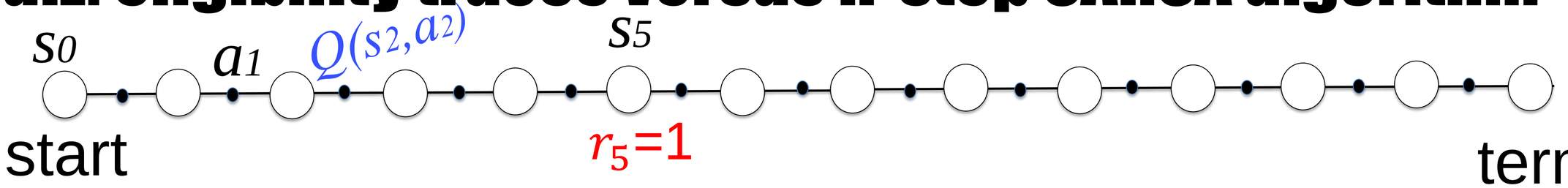
In algo: r_t is called R_{t+1}

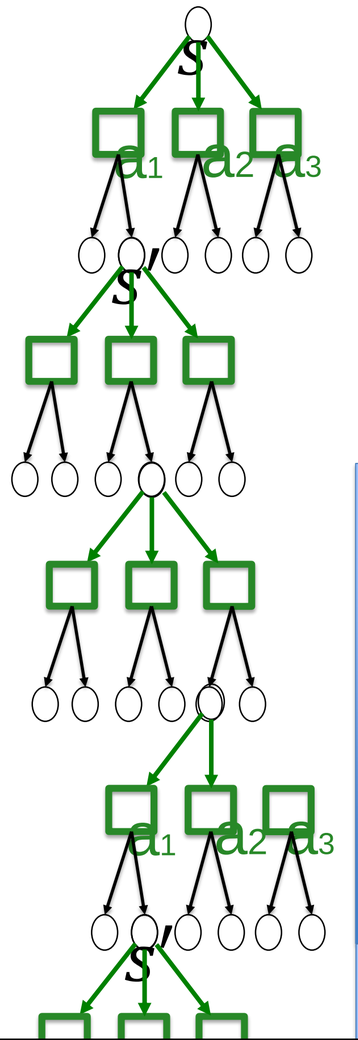
Sutton and Barto, Ch. 7.1

(previous slide)
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.

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:

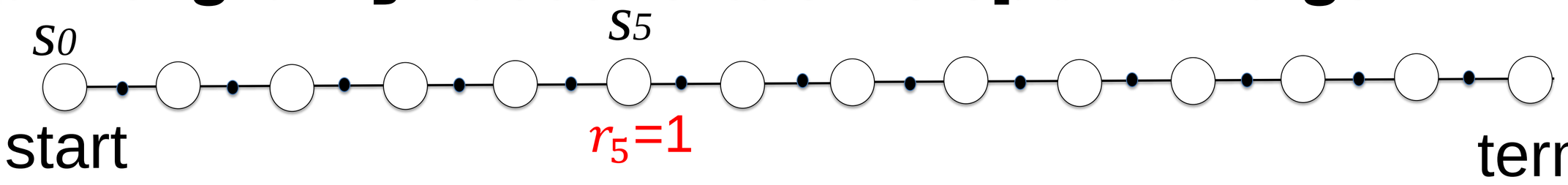
[ ] With 3 step SARSA, only one Q-value has increased, viz. the one 3 steps before the reward.

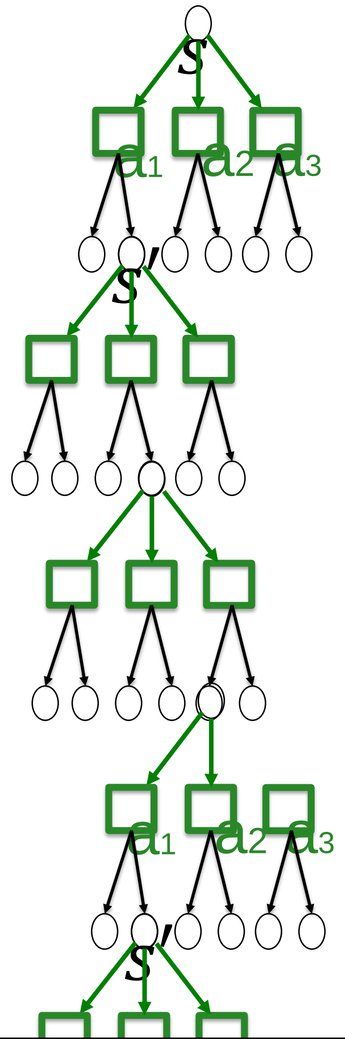
[ ] With 3 step SARSA, three Q-values have increased, viz. those 3 steps, 2 steps and 1 step before the reward.

[ ] 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.

Artificial Neural Networks: Lecture RL2

Wulfram Gerstner
EPFL, Lausanne, Switzerland

Variants of TD-learning methods and continuous space

Part 7: Modeling the Input Space

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
7. **Modeling the input space**

(previous slide)
Continuous input spaces have a second problem: there are many Q-values are V-values that you need to compute.

Problem of TD algorithms: representation of input

All algorithms so far are ‘tabular’:

Q-learning or SARSA:

→ build a table $Q(s,a)$ with entries for all states s and actions a

TD-learning of V-values:

→ build a table $V(s)$ for all states s

discrete states and actions

→ many entries,

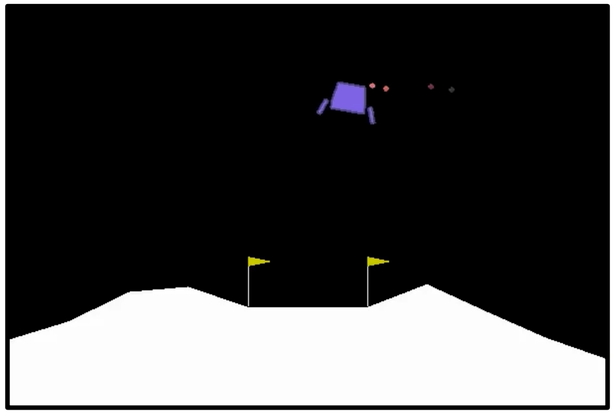
→ ‘independent’ (apart from self-consistency of Bellman)

(previous slide)
Two observations:
First, in a table all entries are independent – the only relation between Q-values or V-values arises from the self-consistency condition of the Bellman equation.
Second, there are many (!) entries.

Problem of TD algorithms: representation of input

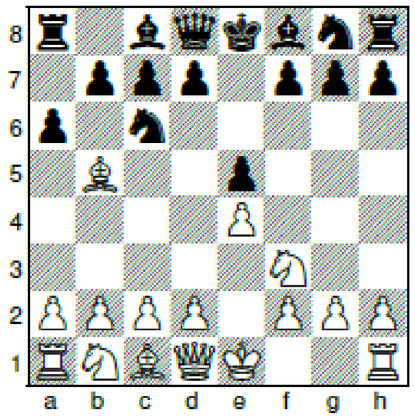
- for control problems, input space is naturally continuous

Moon lander
Aim: land between poles

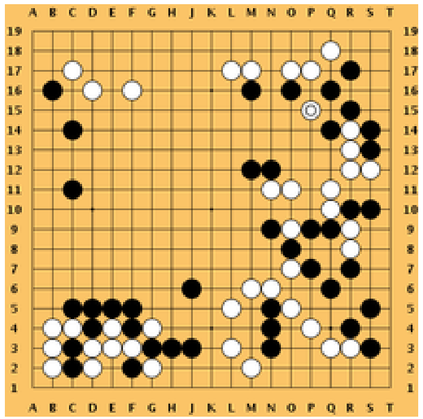


- for discrete games, the input space often too big

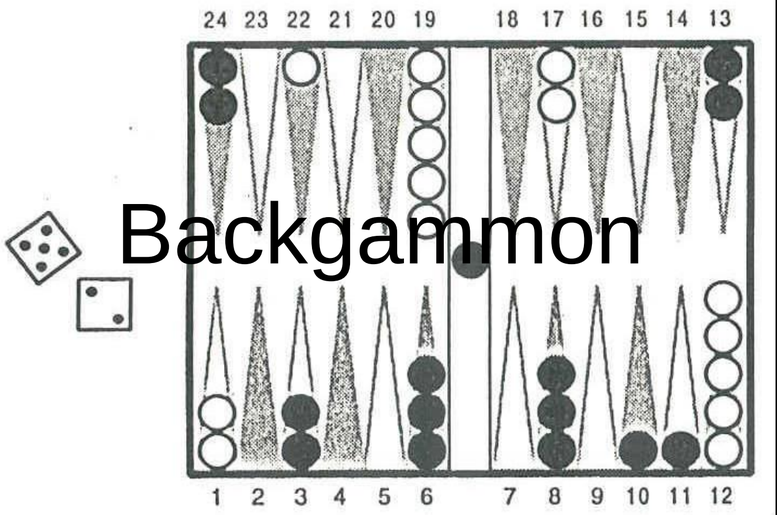
Chess



Go



Backgammon



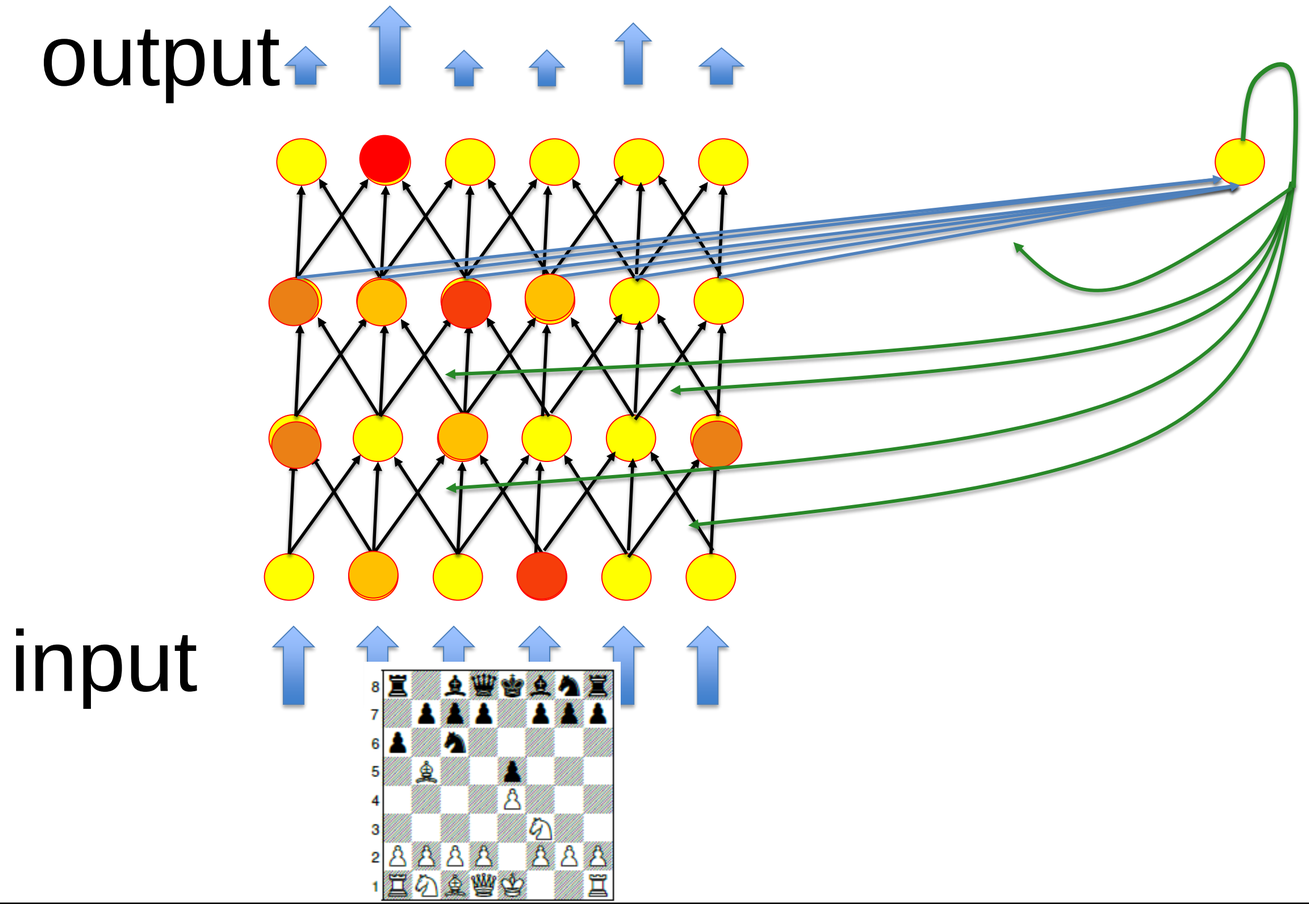
(previous slide)
Even in cases where the natural input space is descrete, such as in games, there might simply be too many states to keep fill tables with meaningful values.

Solution: Neural Network to represent input configuration

Schematically (theory will follow):

action:
Advance king

2e output for V-value
for current situation:



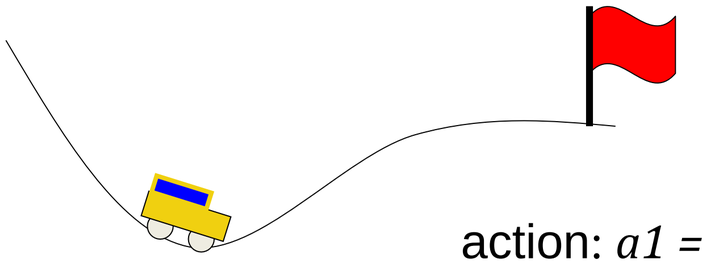
Note: alternatively,
action outputs could present
Q-values

learning:
- change connections

aim:
- Predict value of position
- Choose next action to win

(previous slide)
The basic idea that we will explore this week, next week, and also in the series on Deep Reinforcement Learning is that the mapping from the input states to actions; or from the input states to value functions can be represented by a model with parameters, typically a neural network with adjustable weights.

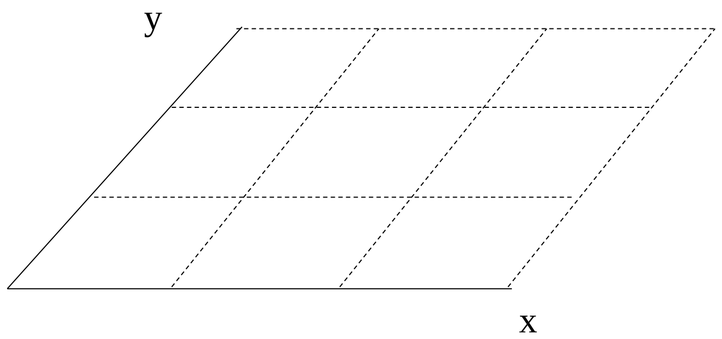
Solution: Continuous input representation



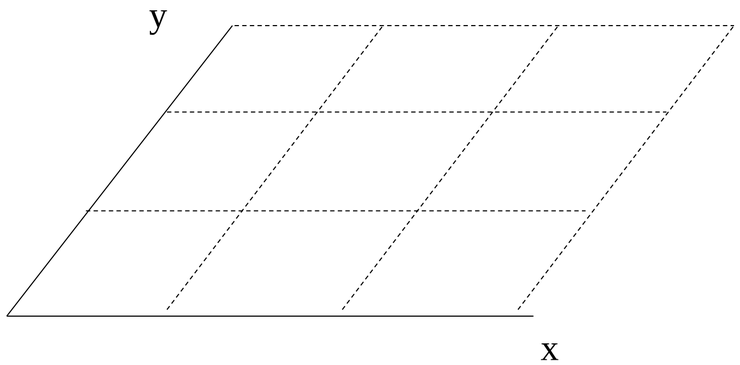
Example: Mountain Car

action: *a1 = right*
a2 = left

for action *a1*



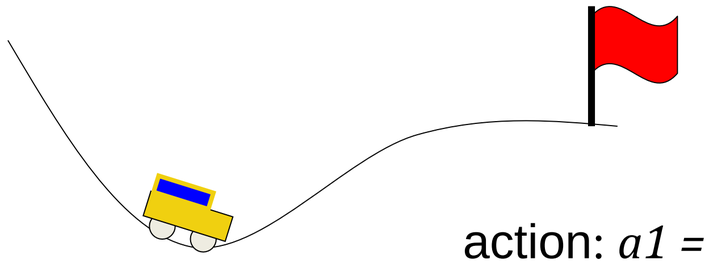
for action *a2*



(previous slide)
In the mountain car task, the input space is two dimensionals: the position x and the speed.
Suppose both dimensions are discretized into 3 values. The Q-values therefore have 9 entries for action a1 (force to the left) and 9 further entries for action a2 (force to the right).

Solution: Continuous input representation

Blackboard 3: Radial Basis functions

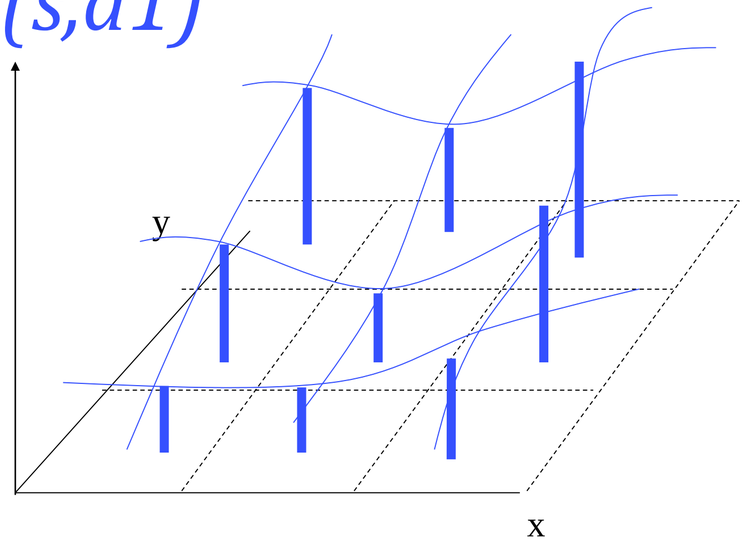


Example: Mountain Car

action: *a1 = right*
a2 = left

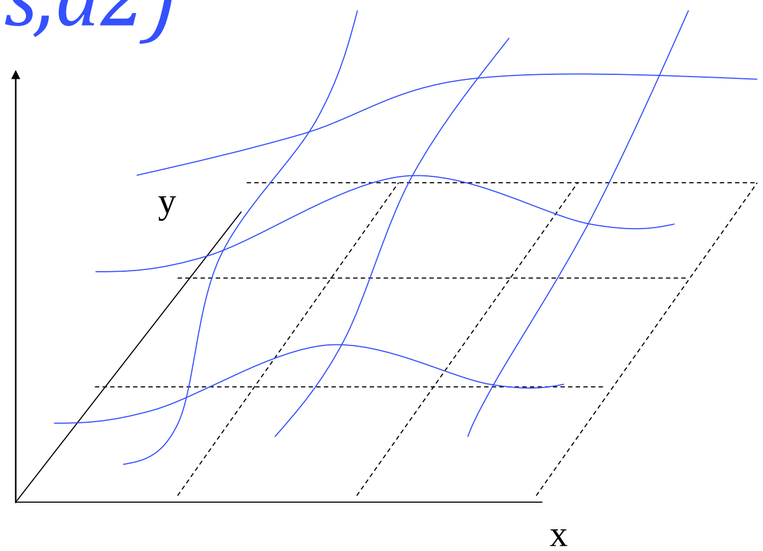
for action *a1*

$Q(s,a1)$



for action *a2*

$Q(s,a2)$



(previous slide)
Instead of considering 9 separate table entries of Q-values Q(s,a1) for action a1, we can also think of a smooth function on the two-dimensional input space that represents Q(s,a1) as a function of s.
Similarly, Q(s,a2) is a smooth function of s, but for action a2.

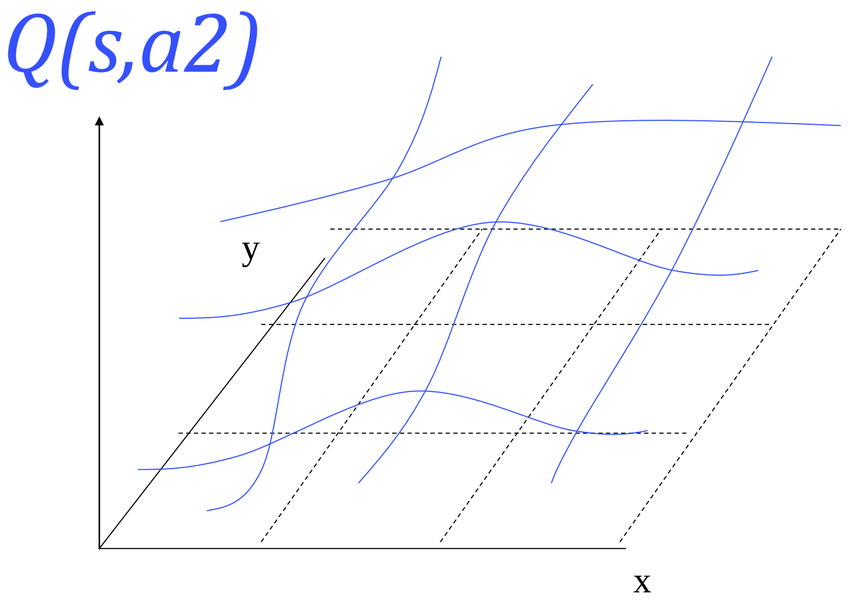
A first advantage is, that the question of discretization of the input space has now disappeared, since we can model the Q-values as a function of the continuous state variable s=(x,y).

The question arises how to model such Q-value functions.
One possibility is to use a combination of basis functions ϕ
so as to describe the Q-value

$$Q(s,a) = \sum_j w_{aj} \Phi(s - s_j)$$

where the weights between basis function j and action a are denoted by w_{aj}

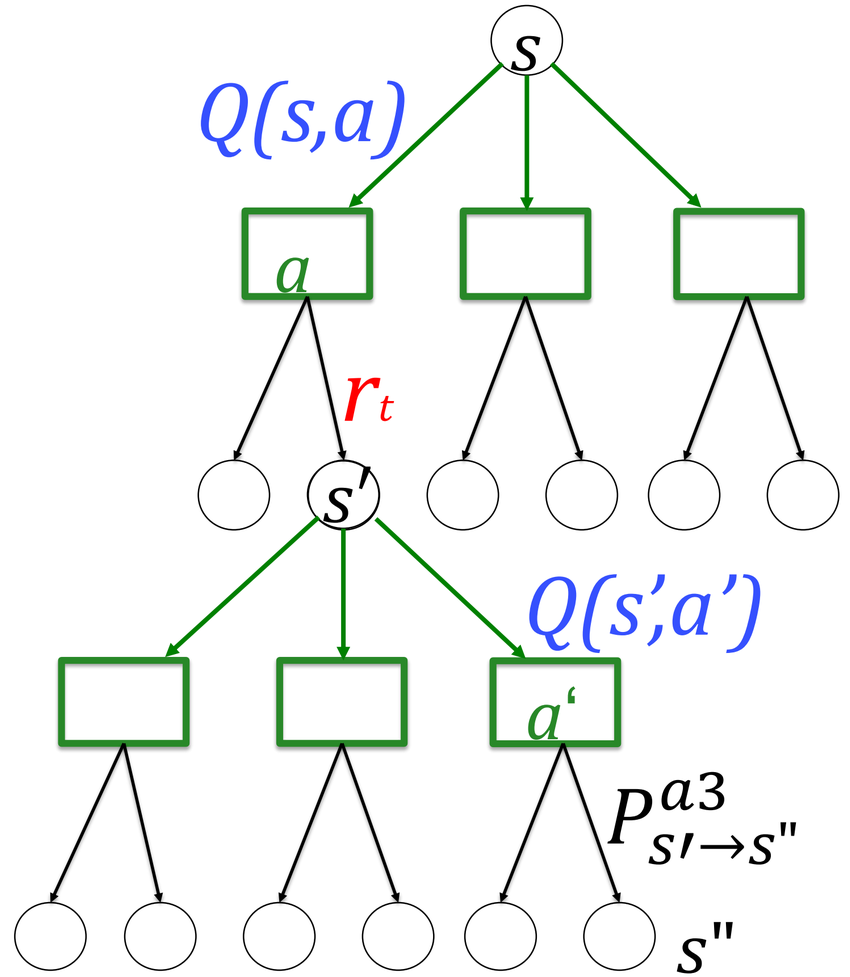
Blackboard 3: Radial Basis functions



From Bellman equation to Error function.

Consistency condition of Bellman Eq.

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



On-line consistency condition
(should hold on average)

target

$$Q(s,a) = r_t + \gamma\, Q(s',a')$$

yields online Error function (loss)

target

$$E(\boldsymbol{w}) = \frac{1}{2}\left[r_t + \gamma Q(s',a'|\boldsymbol{w}) - Q(s,a|\boldsymbol{w})\right]^2$$

‘semi-gradient’

ignore

take gradient w.r.t $\boldsymbol{w}$

(previous slide)
During the discussion of the Bellman equation and SARSA, we stated repeatedly that, if we neglect the discount factor, the difference between Q-values in neighboring time steps must be explained by the reward.
If we include the discount factor, the above statement reduces to

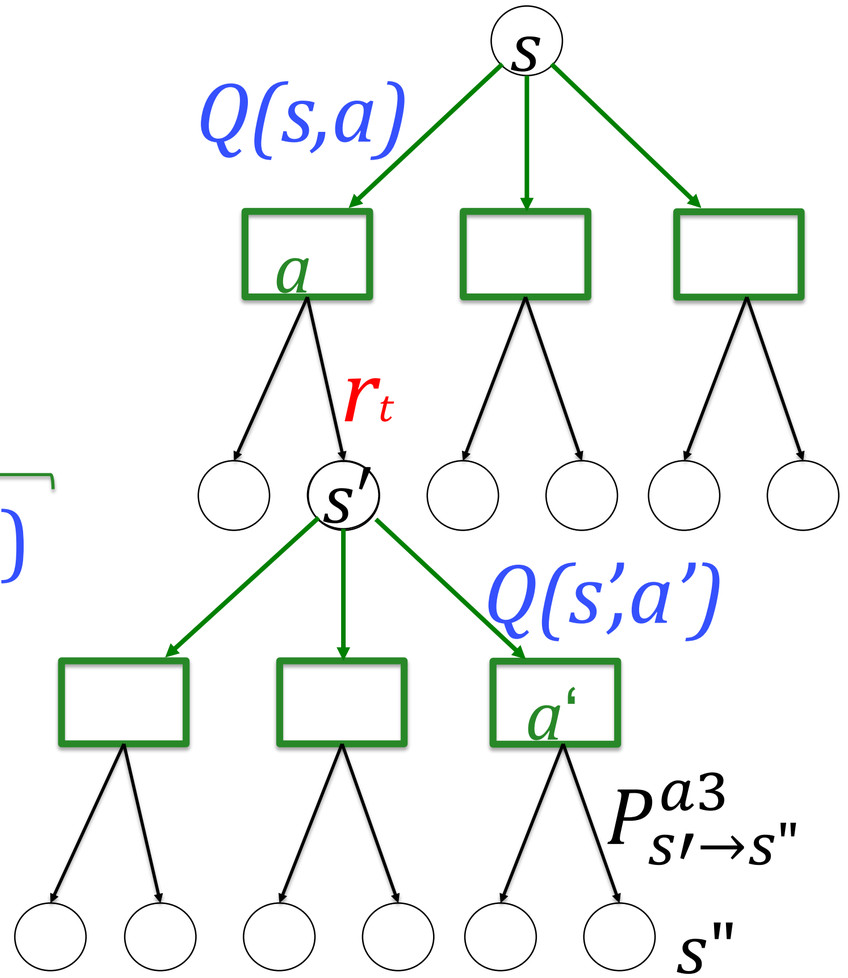
$$Q(s,a) = r + \gamma\, Q(s',a')$$

Where the equality sign has to be interpreted as ‘should ideally on average be close to’ and the right hand side is the ‘target of learning’

Therefore we can construct an error function *E* that measures how close we are to such an ideal case. The squared error function that implements this ideal is noted at the bottom of the slide.
Since the **‘target of learning**’ should be considered as momentarily fixed, we optimize the error function by taking the derivative of *E* with respect to w but ignore that the target also depends on w. We will explore this further in the next week and in the applications of Deep RL.

Blackboard 4: Error function

$$Q(s,a) = \underbrace{r + \gamma\, Q(s',a')}_{\text{target}}$$



In Class Exercise 5 now: 10 minutes

Exercise 5. Gradient-based learning of Q-values

Assume again that the Q-values are expressed as a weighted sum of 400 basis functions: $Q(s,a) = \sum_{k=1}^{400} w_{ak}\Phi(s - s_k)$. For the moment the function Φ is arbitrary, but you may think of it as a Gaussian function. Note that s and s_k are usually vectors in $\mathbb{R}^N$ in this case. There are 3 different actions so that the total number of weights is 1200. Now consider the error function $E_t = \frac{1}{2}\delta_t^2$, where

$$\delta_t = r_t + \gamma \cdot Q(s', a') - Q(s, a) \tag{3}$$

is the reward prediction error. Our aim is to optimize $Q(s,a)$ by changing the parameters w.

a. Find a learning rule that minimizes the error function E_t by gradient decent. Consider the case where the actions a and a' are different.

   Write down the learning rule. How many weights need to be updated in each time step?

b. Find a learning rule that minimizes the error function E_t by gradient decent. Consider the case where the actions a and a' are the same.

   Write down the learning rule.

   Is there any difference to the case considered in a?

Exercise 6. Consistency condition for 3-step SARSA

In class we have seen the arguments leading to the error function arising from the consistency condition of Q-values.

$$E = 0.5 \sum [\delta_t]^2$$

with $\delta_t = r_t + \gamma Q(s', a') - Q(s, a)$ This specific consistency condition corresponds to 1-step SARSA.

Write down an analogous consistency condition for 3-step SARSA.

(your calculations)

Error function: full gradient and semi-gradient

Discrete time steps: s,a → s',a'

target

$$E(\boldsymbol{w}) = \frac{1}{2}[r_t + \gamma Q(S', a'|\boldsymbol{w}) - \mathrm{Q}(S, a|\boldsymbol{w})]^2$$

take gradient w.r.t. this $\boldsymbol{w}$

Full gradient: you take the correct derivative with respect to $\boldsymbol{w}$

Semi-gradient: you take the derivative with respect to $\boldsymbol{w}$ in $\mathrm{Q}(S, a|\boldsymbol{w})$ but you ignore the $\boldsymbol{w}$-dependence of the target.

(This is a heuristic trick to stabilize learning)

(previous slide)

In the exercise, the difference between full gradient and semi-gradient becomes visible if a=a’.

However, the problem that the target needs to be considered as ‘fixed’ to make learning converge is a fundamental one that needs to kept in mind for all applications of deep reinforcement learning or reinforcement learning in continuous space.

Summary: Many Variations of a few ideas in TD learning

Learning outcomes and Conclusions

- TD – learning (Temporal Difference)

  → TD algo: works with 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, smooths over time
  - similar to n-step SARSA

- **Continuous space**

  → use a neural network to model Q-values and generalize

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.

The END