Learning in Neural Networks: RL1 (continued)

Reinforcement Learning and SARSA

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Parts 4-6: Examples of Reward-based Learning

Objectives for Lecture RL1:

- Reinforcement Learning (RL) is learning by rewards ✓
- Agents and actions, states and rewards ✓
- Convergence in expectation
- Exploration vs Exploitation
- Bellman equation
- SARSA algorithm

Reading:
Sutton and Barto, Reinforcement Learning (MIT Press, 2nd edition 2018)

Chapters: 1.1-1.4; 2.1-2.6; 3.1-3.5; 6.4

Reading for this week:

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

Chapters: 1.1-1.4; 2.1-2.6; 3.1-3.5; 6.4

Background reading:

Silver et al. 2017, Archive

Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm

Recall: Learning by reward

Learning by reward ‘’
BUT:
Reward is rare:
‘sparse feedback’ after
a long action sequence

Previous slide.

How does a human learn to play table tennis: How does a child learn to play the piano? How does a dog learn to perform tricks?

In all these cases there is no supervisor. No master guides the hand of the players during the learning phase. Rather the player ‘discovers’ good movements by rather coarse feedback. For example, the ball in table tennis does not land on the table as it should. That is bad (negative feedback). The ball has a great spin so that the opponent does not get. This is good (positive feedback).

Similarly, it is hard to tell a dog what to do. But if you reinforce the dog’s behavior by giving a ‘goodie’ at the moment when it spontaneously performs a nice action, then it can learn quite amazing things.

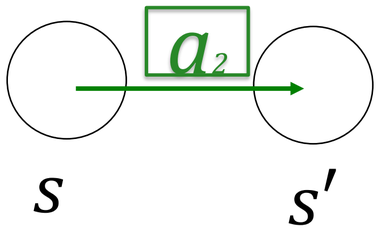
In all these cases it is the ‘reward’ that guides the learning. Rewards can be the goodie for the dog, or just the feeling ‘now I did well’ for humans.

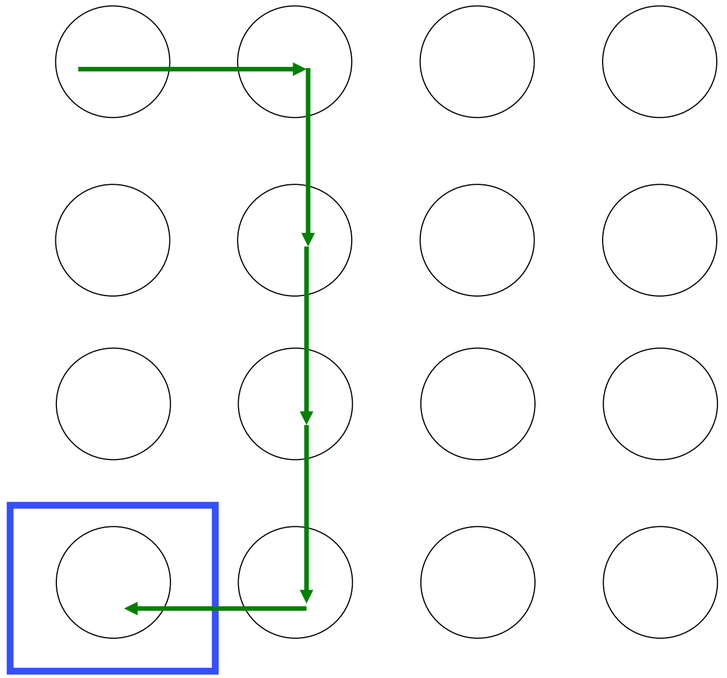
Recall: Elements of Reinforcement Learning:

- discrete states:
  - old state s
  - new state s'
- current state: s_t
- discrete actions: $a_1,\ a_2\ \ldots\ a_A$
- current action: a_t
- current reward: r_t
- Mean rewards for transitions:

$$R^{a}_{s \to s'}$$

often most transitions have zero reward





Previous slide.

The elementary step is:
The agent starts in state s.
It takes action a
It arrives in a new state s’
Potentially receiving reward r (during the transition or upon arrival at s’).

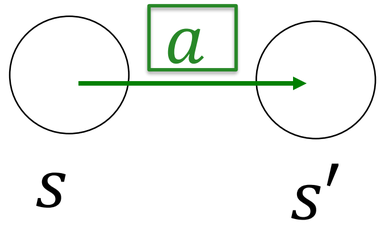
Since rewards are stochastic we have to distinguish the mean reward at the transition (capital R with indices identifying the transition) from the actual reward (lower-case r with index t) that is received at time t on a transition.

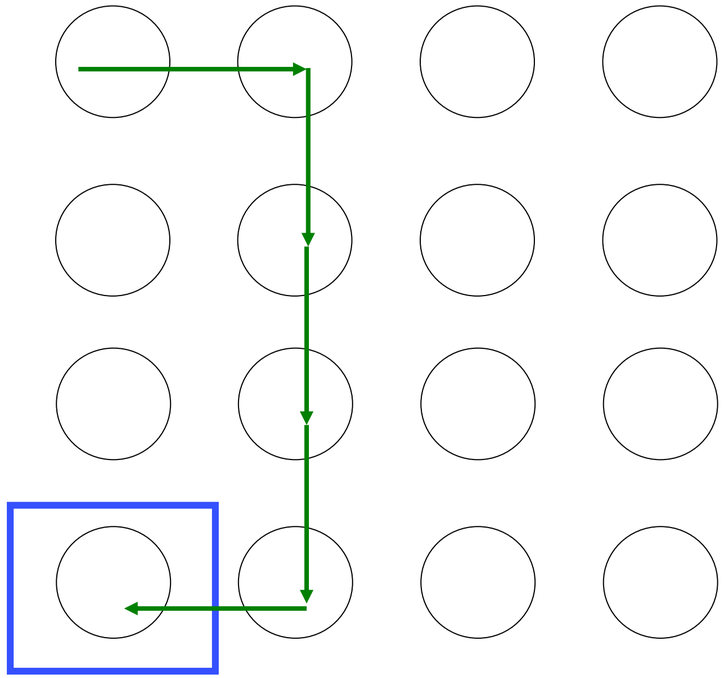
Note that in many practical situations most transitions or states have zero rewards, except a single ‘goal’ state at the end.

Recall: Elements of Reinforcement Learning

- Environment with discrete states
- Transitions (potentially stochastic) are driven by actions
- **Aim: choose good actions to optimize reward**

  → Markov Decision Problem (MDP)





Distinction between

- current actual reward: r_t
- Mean reward for transition:

$$R^{a}_{s \to s'} = E\left(r \middle| s, a, s'\right)$$

Previous slide.
Conclusion: In all practical situations, there is an enormous number of states.

In many situations we can think of the actions as discrete.
For the moment we also think of the states as discrete (but next week we will go to continuous state space).

Transitions between actions are influenced by action choices.

Transitions can be stochastic.

Actions should be chosen so as to maximize the reward. This setting is also known as Markov Decision Problem (MDP).

Recall: Q-value for one-step horizon games/bandit problem

Q-value $Q(s,a)$

Expected reward for action a starting from s

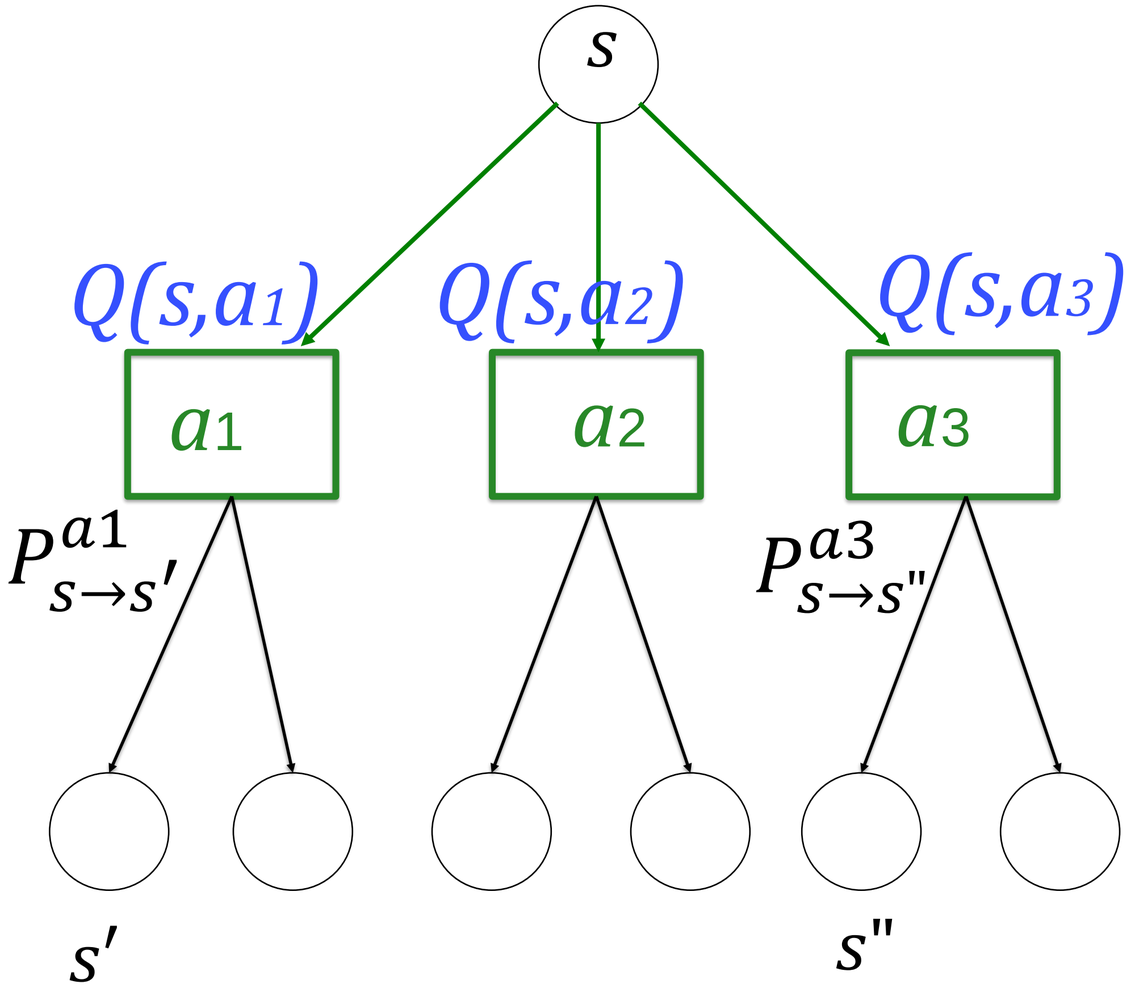
$$Q(s,a) = \sum_{s'} P^{a}_{s\to s'} R^{a}_{s\to s'}$$

Reminder:

$$R^{a}_{s\to s'} = E\left(r \middle| s', a, s\right)$$

Similarly:

$$Q(s,a) = E(r|s,a)$$



Now we know the Q-values: which action should you choose?

Previous slide.

$P_{s \to s'}^{a1}$ is the probability that you end up in a specific state s' if you take action a1 in state s.
We refer to this sometimes as the 'branching ratio' below the 'actions'.

Q(s,a) is attached to the branches linking the state s with the actions.

actions are indicated by green boxes; states are indicated by black circles.

The mean reward $R_{s \to s'}^{a}$ is defined as the expected reward given that you start in state s with action a and end up in state s' (see Blackboard 1).

Given the branching ratio and the mean rewards, it is easy to calculate the Q-values (Blackboard 1).

Recall: One-step horizon games (bandit problem)

Q-value = expected reward for state-action pair

If Q-value is known, choice of action is simple

→ take action with highest Q-value

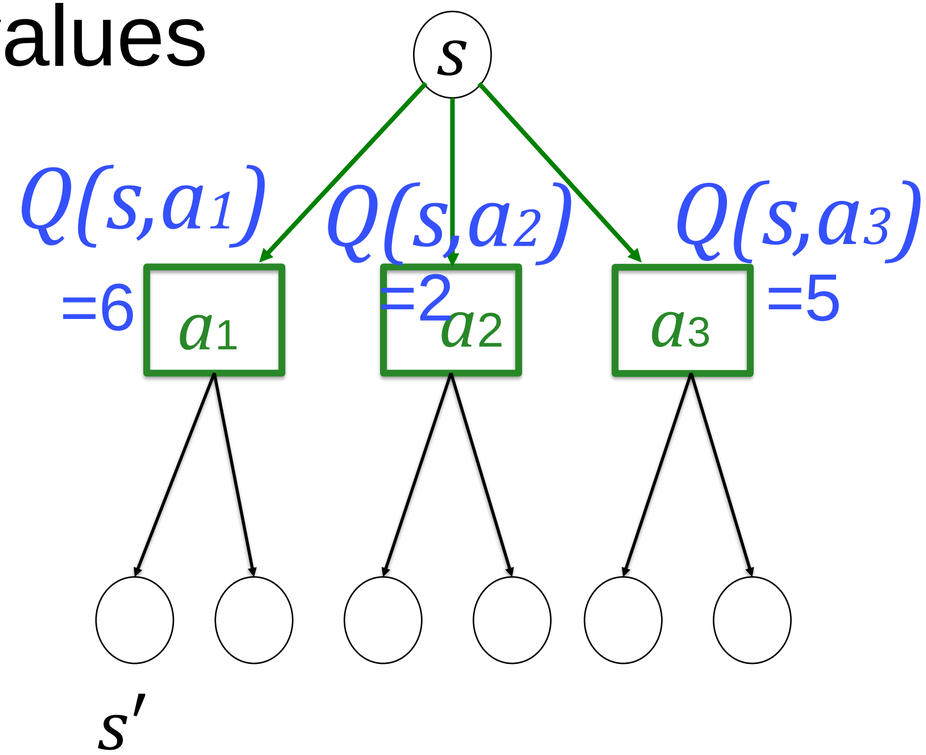
BUT: we normally do not know the Q-values

→ **estimate by online update**

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

Learning rate: often η , often α

$$\Delta Q(s,a) = \alpha\,[r_t - Q(s,a)]$$



Previous slide.
The only remaining problem is that we do not know the Q-values, because the casino gives you neither the branching ratio nor the reward scheme.

Hence the only way to find out is by trial and error (that is, by playing many times – the casino will love this!).

Recall: Update rule in Expectation (Theorem)

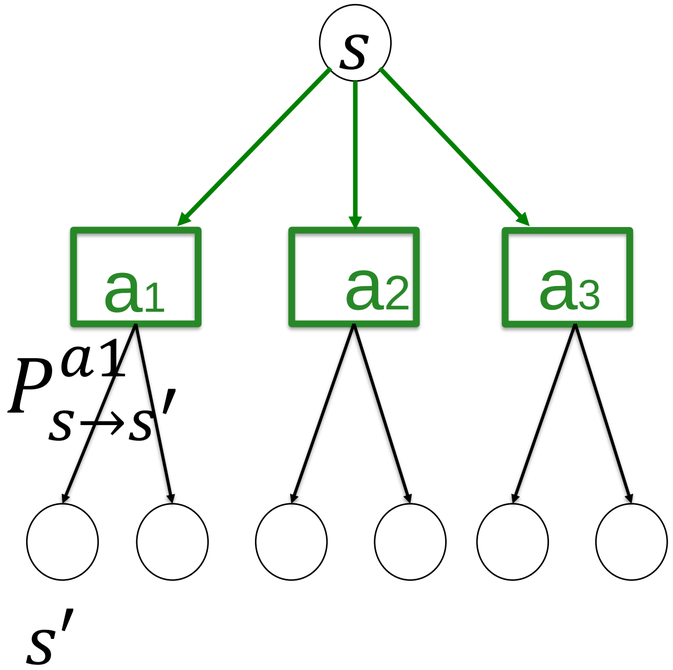
After taking action a in state s, we update with

$$\Delta\hat{Q}(s,a) = \eta\,[r_t - \hat{Q}(s,a)] \qquad (1)$$

(i) If (1) has **converged in expectation given (s,a),** then $\hat{Q}(s,a)$ has a value,

$$\widehat{Q}(s,a) = E\,[\widehat{Q}(s,a)|s,a] = Q(s,a) = \sum_{s'} P^{a}_{s\to s'}\, R^{a}_{s\to s'} \qquad (2)$$

(ii) If the learning rate η decreases, fluctuations around the **empirical mean** $\langle \widehat{\boldsymbol{Q}}(\boldsymbol{s},\boldsymbol{a})\rangle_{\boldsymbol{t}|\boldsymbol{s},\boldsymbol{a}}$ decrease. If $\langle \widehat{\boldsymbol{Q}}(\boldsymbol{s},\boldsymbol{a})\rangle_{\boldsymbol{t}|\boldsymbol{s},\boldsymbol{a}}$ converges for fixed η, then **the empirical mean approaches** $\boldsymbol{Q}(\boldsymbol{s},\boldsymbol{a})$.



Previous slide.

When evaluating the **expectation value given (s,a),** the learning rate drops out since we set the left-hand-side to zero. The exact value of η is not relevant, as discussed in the theorem. Part (i) of the theorem states that the expectation value of $\hat{Q}(s,a)$ is the correct Q-value. For a quick proof of $E\,[\hat{Q}(s,a)|s,a] = Q(s,a)$ see the video. On the blackboard a stronger statement was shown:

$$\hat{Q}(s,a) = Q(s,a).$$

Convergence in expectation is equivalent to imagining that you start millions of trials with the same value $\hat{Q}(s,a)$ without any intermediate update. So in that sense it is like an infinite 'batch' of examples. The stochastic variables are the **next** state s' and the received reward r_t. The value of $\hat{Q}(s,a)$ is not stochastic but 'frozen'. Therefore (trivially) $E\,[\hat{Q}(s,a)|s,a] = \hat{Q}(s,a)$.

In practice, we do not have expectations but online updates with fluctuations. It is important that η is small at the end of learning so as to limit the amount of fluctuations. Part (ii) states that **online mean** for small learning rate also goes to the correct Q-value.

Indeed, since the equations are linear (for the bandit problem = 1-step horizon), the calculation of part (i) apply analogously to the long-term empirical temporal average (denoted by angular brackets). The average is across all those time steps where action a was chosen in state s, denoted as $\langle \hat{\boldsymbol{Q}}(\boldsymbol{s},\boldsymbol{a})\rangle_{\boldsymbol{t}|\boldsymbol{s},\boldsymbol{a}}$. We assume convergence, hence our hypothesis reads

$$\langle \Delta\hat{\boldsymbol{Q}}(\boldsymbol{s},\boldsymbol{a})\rangle_{\boldsymbol{t}|\boldsymbol{s},\boldsymbol{a}} = \boldsymbol{\eta}\,\langle r_t - \hat{\boldsymbol{Q}}(\boldsymbol{s},\boldsymbol{a})\rangle_{\boldsymbol{t}|\boldsymbol{s},\boldsymbol{a}} = 0\ .$$

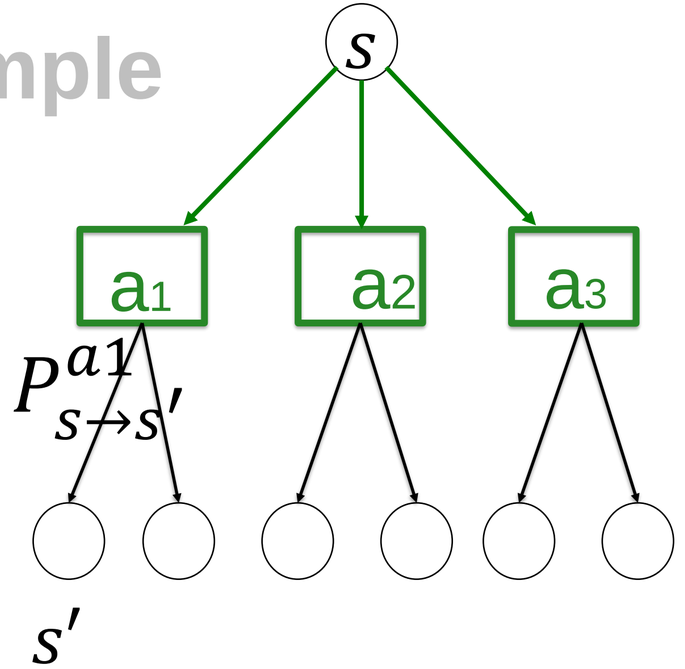
The specific result $\langle \hat{\boldsymbol{Q}}(\boldsymbol{s},\boldsymbol{a})\rangle_{\boldsymbol{t}|\boldsymbol{s},\boldsymbol{a}} = \boldsymbol{Q}(\boldsymbol{s},\boldsymbol{a})$ is based on linearity and is not true for the multi-step horizon that we discuss later.

Recall: One-step horizon: summary

Q-value = expected reward for state-action pair

If Q-value is known, choice of action is simple

→ take action with highest Q-value



If Q-value not known:

→ estimate $\hat{Q}$ by trial and error

→ update with rule

$$\Delta\hat{Q}(s,a) = \eta\,[r_t - \hat{Q}(s,a)\,] \qquad \textbf{(1)}$$

→Let learning rate η decrease over time

Iterative algorithm (1) fluctuates, but 'makes sense'

Previous slide.
Let us distinguish the ESTIMATE $\hat{Q}(s,a)$ from the real Q-value $Q(s,a)$

The update rule can be interpreted as follows:
if the actual reward is larger than (my estimate of) the expected reward, then I should increase (a little bit) my expectations.

The learning rate η :

In exercise 1, we found a rather specific scheme for how to reduce the learning rate over time. But many other schemes also work in practice. For example you keep η constant for a block of time, and then you decrease it for the next block.

Note: in later lectures I will often use the symbol α instead of η

Both symbols indicate what is called the ‘learning rate’ in Deep Learning.

Learning Neural Networks: RL1

Reinforcement Learning and SARSA

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Part 4: Exploration vs. Exploitation

- Examples of Reward-based Learning
- Elements of Reinforcement Learning
- One-step horizon (bandit problems)
- **Exploration vs. Exploitation**

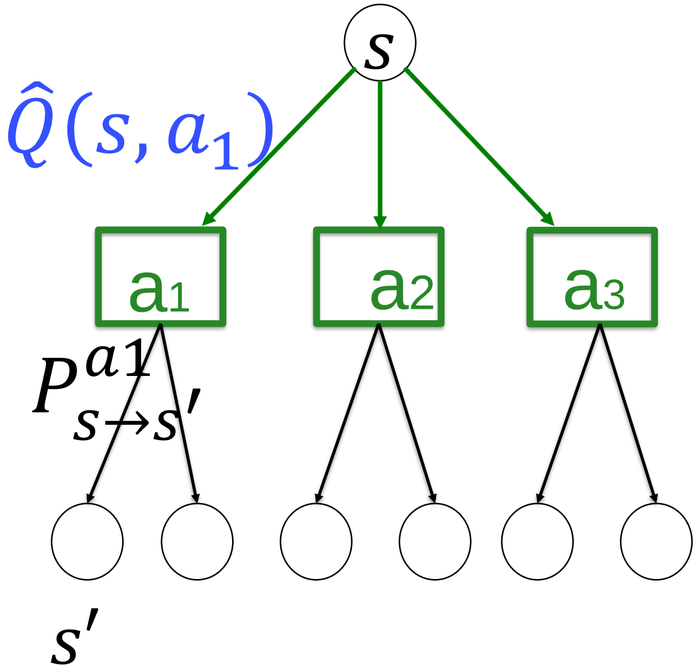
Previous slide.

To estimate the Q-values you have to play all the different actions several times. However, if you know the Q-values you should only play the best action.

Exploration – Exploitation dilemma

Ideal: take action with maximal $Q(s,a)$

Problem: correct Q values not known
(since reward probabilities and branching probabilities unknown)



Exploration versus exploitation

Explore so as to estimate reward probabilities

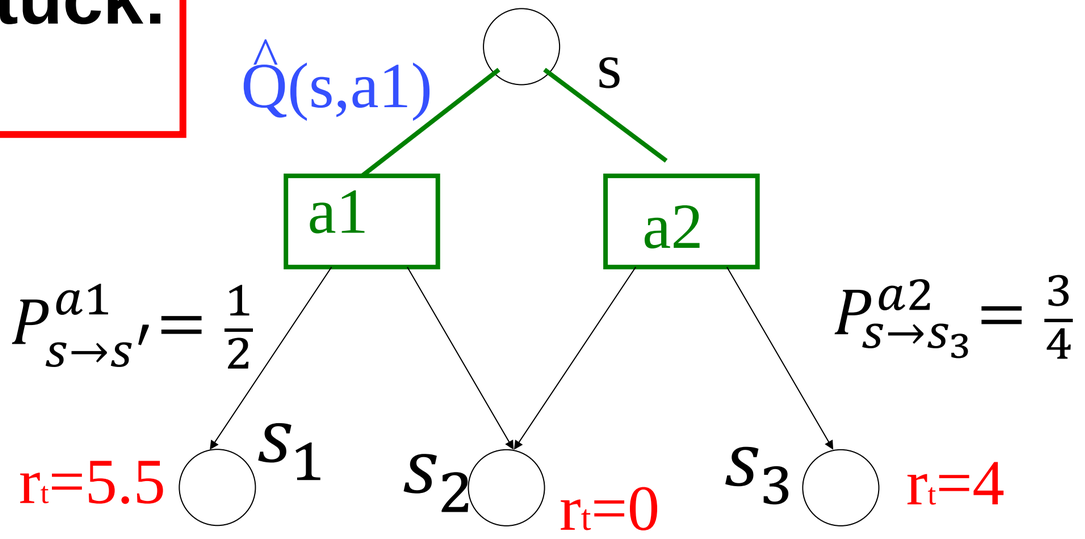
Take action which looks optimal, so as to maximize reward

Previous slide.

Since Q-values are not known, you are always in the situation of an exploration-exploitation dilemma.

Note: All estimates of Q will be empirical estimates. Here in this and the next slide I still write Q-hat for the empirical average. However, later, I simplify the notation and write for the empirical estimate Q(s,a) without the hat whenever the meaning is implicitly clear.

greedy makes you stuck: Example



s=state

a2 action

r_t=1 actual reward

s'=new state

Assume that you initialize all Q values with zero; set η =0.2 (constant)

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

Trial 1: you choose action a1, you get r_t=5.5

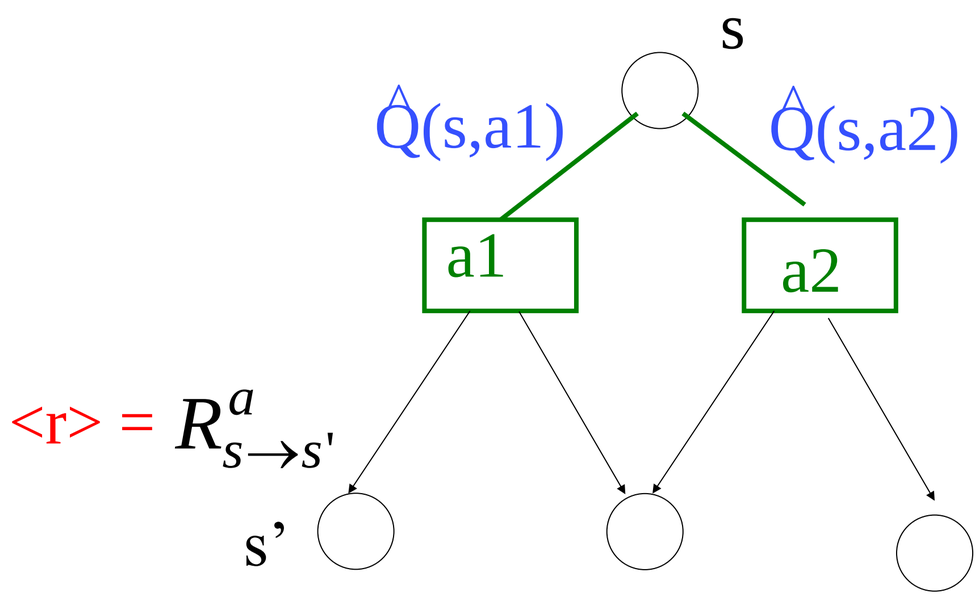
Trial 2: you choose action a2, you get r_t=4.0

Trial 3 – 4: continue **'greedy': → you continue with action 1**

BUT: the expected reward Q(s,a)= $\sum_{s'} P^{a}_{s\to s'} R^{a}_{s\to s'}$ is larger for action 2.

Given the outcomes of the first two trials, action a1 looks better.
You can check that whatever the outcome in trial 3 (even for reward=0!), the estimated Q-value of action a1 is still higher than that of action a2!

Exploration and Exploitation



Problem: correct Q values not known

greedy strategy:

- take action a* which looks best

$$\hat{Q}(s,a^*) \geq \hat{Q}(s,a_j) \quad \text{for all } j$$

ATTENTION:
with 'greedy' you may get stuck with a sub-optimal strategy (see Exercise!)

Previous slide.

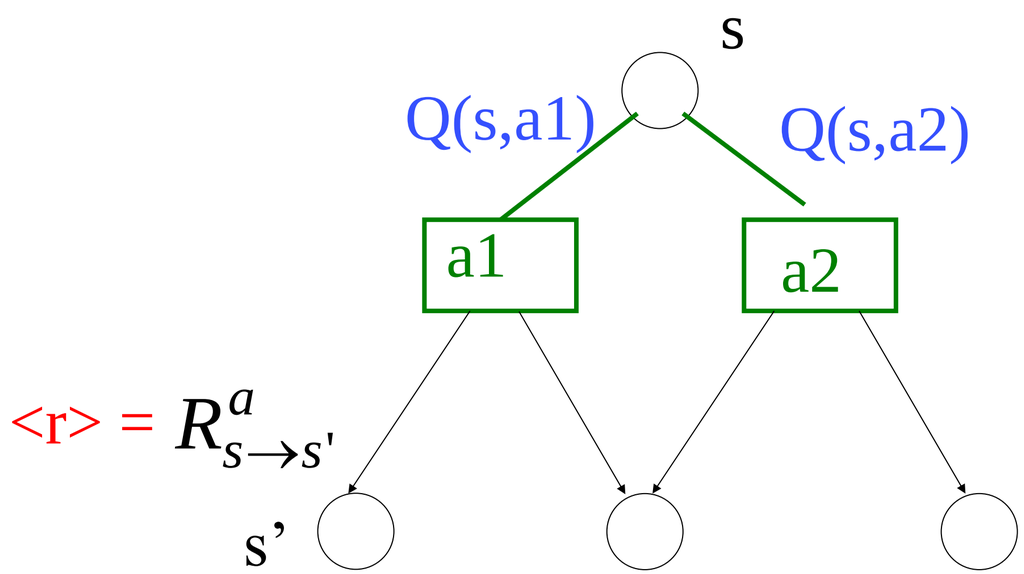
If you know the correct Q-values, the best choice would be to choose the action with maximal Q-value (called ‘greedy’ action). But since you don’t know the Q-values it is risky to choose the greedy action because you may get stuck with a suboptimal choice.

In (almost all) applications of reinforcement learning we work with estimated Q-values.

Previously we used a hat to distinguish the ESTIMATED $\hat{Q}(s,a)$ from the real Q-value $Q(s,a)$. However, in the following I will write the estimated Q-values without the hat. Nearly always Q means estimated Q.

Exploration and Exploitation: practical approach

hats have been dropped!



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

Problem: correct Q values not known

greedy strategy:
- take action a* which looks best

$$Q(s,a^*) \geq Q(s,a_j) \text{ for all } j$$

$\mathcal{E}$greedy strategy:
- take action a* which looks best with prob $P = 1-\varepsilon$

Softmax strategy: take action a' with prob

$$P(a') = \frac{\exp[\beta Q(a')]}{\sum_{a}\exp[\beta Q(a)]}$$

Optimistic greedy:
initialize with Q values that are too big

Previous slide.

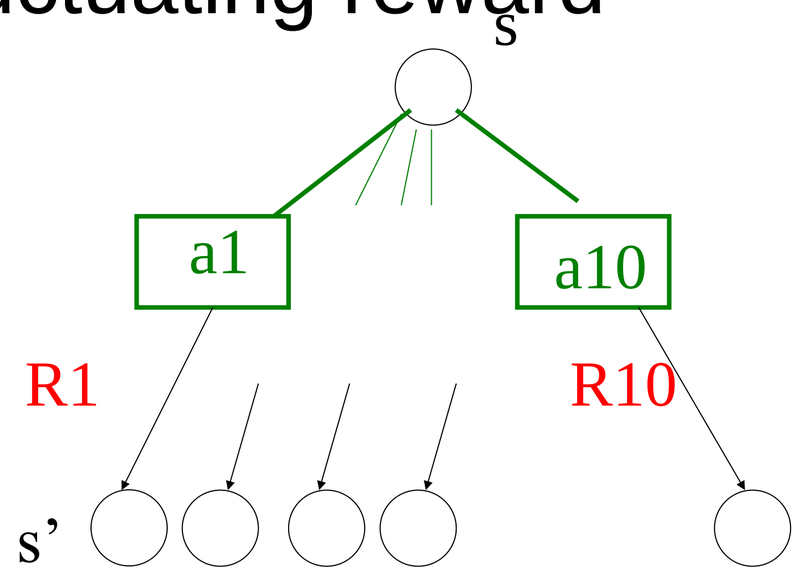
Softer versions of greedy allow you to choose occasionally an action which looks suboptimal, but which allows you to further explore the Q-values of other options.

Epsilon-greedy and softmax are examples following this idea.
Note that ‘softmax’ is a function that one also encounters in multiclass tasks with 1-hot coding (see course of ‘machine learning’;)

A radically different approach is optimistic greedy. If you initialize all Q-values at the same value, but clearly too high (compared to maximal reward that you can get in the scheme), then the Q-value of action a1 decreases initially each time you play a1, which in turn favors other actions that you have not yet played.

Exploration and Exploitation: practical approach

Example: 10-armed bandit with fluctuating reward



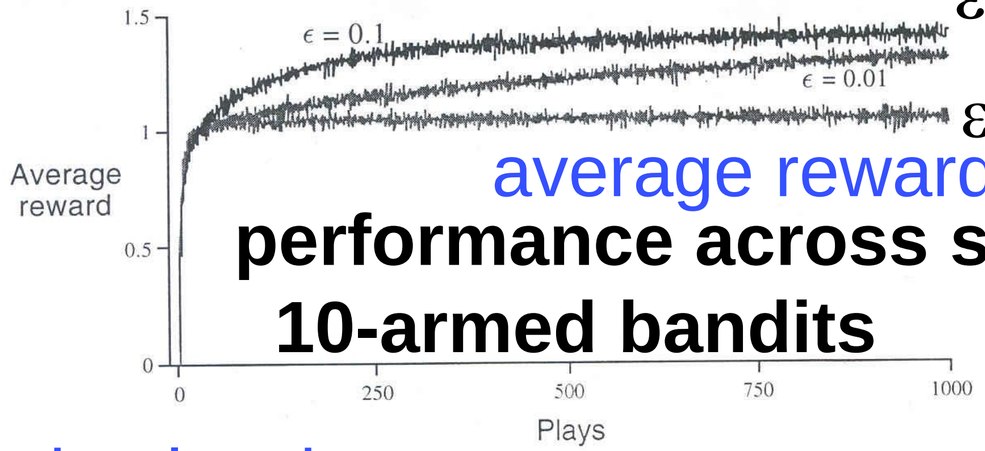
in each action, actual rewards fluctuate around a mean

Rk= $R_{s \to s'}^{a_k}$

Epsilon-greedy: simulation

ε=0.1

ε=0



average reward

performance across several 10-armed bandits

Optimal action

ε=0.1

ε=0.01

ε=0

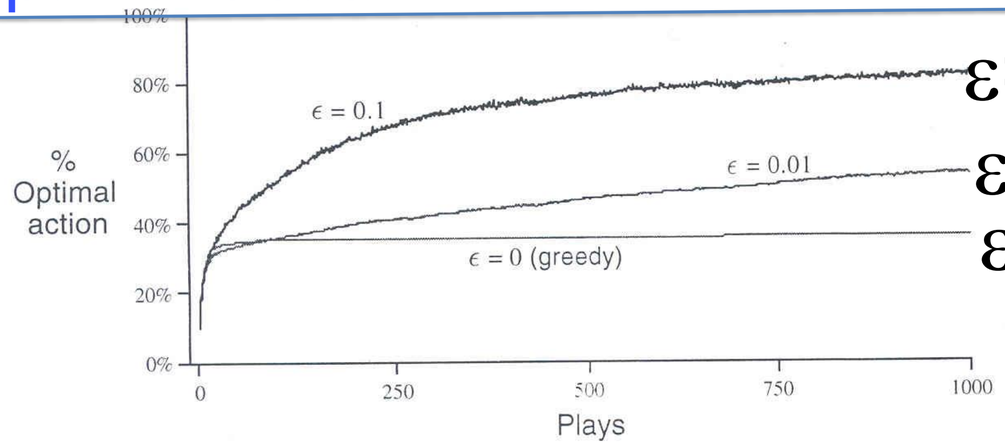


Figure 2.1 Average performance of ϵ-greedy action-value methods on the 10-armed testbed. These data are averages over 2000 tasks. All methods used sample averages as their action-value estimates.

book: Sutton and Barto

Previous slide.

Computer simulation of a situation where actual rewards r fluctuate around the mean reward R. There are 10 different actions a1, …, a10 each with a different mean reward R1, …, R10.

There exist two different ways to evaluate the performance.
Top: what is the average reward that you get by playing epsilon-greedy?
Bottom: what is the fraction of times that you play the optimal action, by playing epsilon-greedy.

Three different values of epsilon are used.

The curves are averages over many instantiations of different 10-armed bandits.

Exploration and Exploitation: practical approach

Epsilon-greedy, combined with iterative update of Q-values

A simple bandit algorithm

Initialize, for $a = 1$ to k:

$Q(a) \leftarrow 0$

$N(a) \leftarrow 0$

Repeat forever:

$$A \leftarrow \begin{cases} \arg\max_a Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$$

$R \leftarrow bandit(A)$

$N(A) \leftarrow N(A) + 1$

$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\big[R - Q(A)\big]$

Sutton and Barto call R what I call r_t

learning rate

Sutton and Barto, ch. 2

Previous slide.

This is the style of pseudo-code that we will see a lot over the next few weeks. It is taken from the book of Sutton and Barto (MIT Press, 2018); Sutton and Barto made a pdf online available.

Q(a) is the Q-value for action a. Since we have always the same starting state in which we have to make our choice of action, we can suppress the index of the state s. Q(a) = Q(s_{start},a).

N(a) is a counter of how many times the agent has taken action a.

In this specific example the learning rate eta is the inverse of the count N(a) (see earlier exercise); but in the more general setting we would remove the counter and just use some heuristic reduction scheme for eta.

Note that in class we define (1-epsilon) as the probability of taking the ‘best’ action corresponding to argmax Q and epsilon is then distributed over the OTHER actions. Sutton and Barto distribute epsilon over ALL actions, including the ‘best’.
Thus for a total choice of 3 actions, Sutton and Barto have a probability of epsilon/3 for the other actions (and with the definition in class it would be epsilon/2).

Quiz: Exploration – Exploitation dilemma

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

[ ] With a greedy policy the agent chooses the ‘best possible’ action

[ ] Using an epsilon-greedy method with epsilon = 0.1
means that, even after convergence of Q-values,
in about 10 percent of cases a suboptimal action is chosen.

[ ] If the rewards in the system are between 0 and 1 and Q-values are initialized with Q=2, then each action is played at least 5 times before exploitation starts.
(exploitation starts when you no longer choose the wrong action)

Previous slide.

Here we define as in class (1-epsilon) as the probability of taking the ‘best’ action corresponding to argmax Q and epsilon is then distributed over the OTHER actions.

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 $Q(a_1) = 34, Q(a_2) = 31, Q(a_3) = 30$.

Quiz: Exploration – Exploitation with Softmax policy

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.

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

Your notes (Quiz not given in class).

Use that exp(5) is a big number!

[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

$$P(a') = \frac{\exp[\beta Q(a')]}{\sum_{a} \exp[\beta Q(a)]}$$

Exploration and Exploitation: Summary

- If we know the Q-values we can exploit our knowledge
- Exploitation = action which is best = argmax Q(a)

- But we never know the Q-values for sure
- We need to estimate the Q-values by playing the game
- Explore possibilities, transitions, outcomes, reward

For complex problems, there is no perfect trade-off Between exploration and exploitation

Learning in Neural Networks: RL1

Reinforcement Learning and SARSA

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Part 5: Bellman Equation

- Examples of Reward-based Learning
- Elements of Reinforcement Learning
- One-step Horizon (Bandit Problems)
- Exploration vs. Exploitation
- **Bellman Equation**

Previous slide.
So far our Q-values were limited to situations with a 1-step horizon. Now we will get more general.

Multistep horizon

Policy $\pi(s,a)$

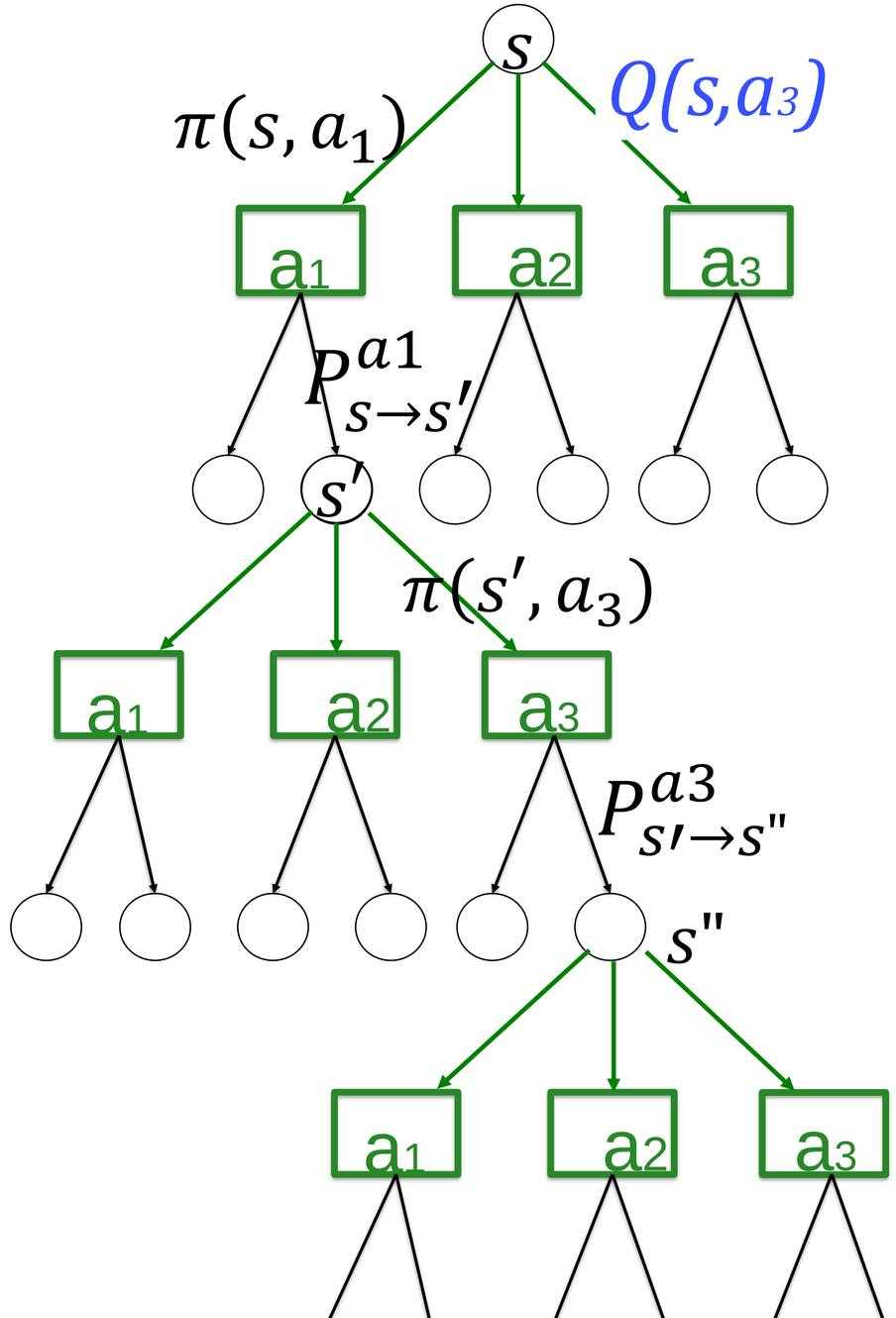
probability to choose action a in state s

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

Examples of policy:
-epsilon-greedy
-softmax

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

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



Previous slide.

After a first action that leads to state s' starting from state s , the agent can now take a second action starting from s'.

Note that there are two different types of branching ratio:

$\pi(s, a_1)$ describes the probability that the agent uses action a1 when it is in state s – based on the agent's policy (such as epsilon-greedy)

$P_{s \to s'}^{a1}$ describes as before the probability that the agent arrives in state s' given that it chooses action a1 in state s.

As before we are interested in the expected reward. The Q value Q(s,a) describes the total accumulated reward the agent can get starting in state s with action a.

Next slide: rewards that are n steps away are discounted with a factor γ^{n}

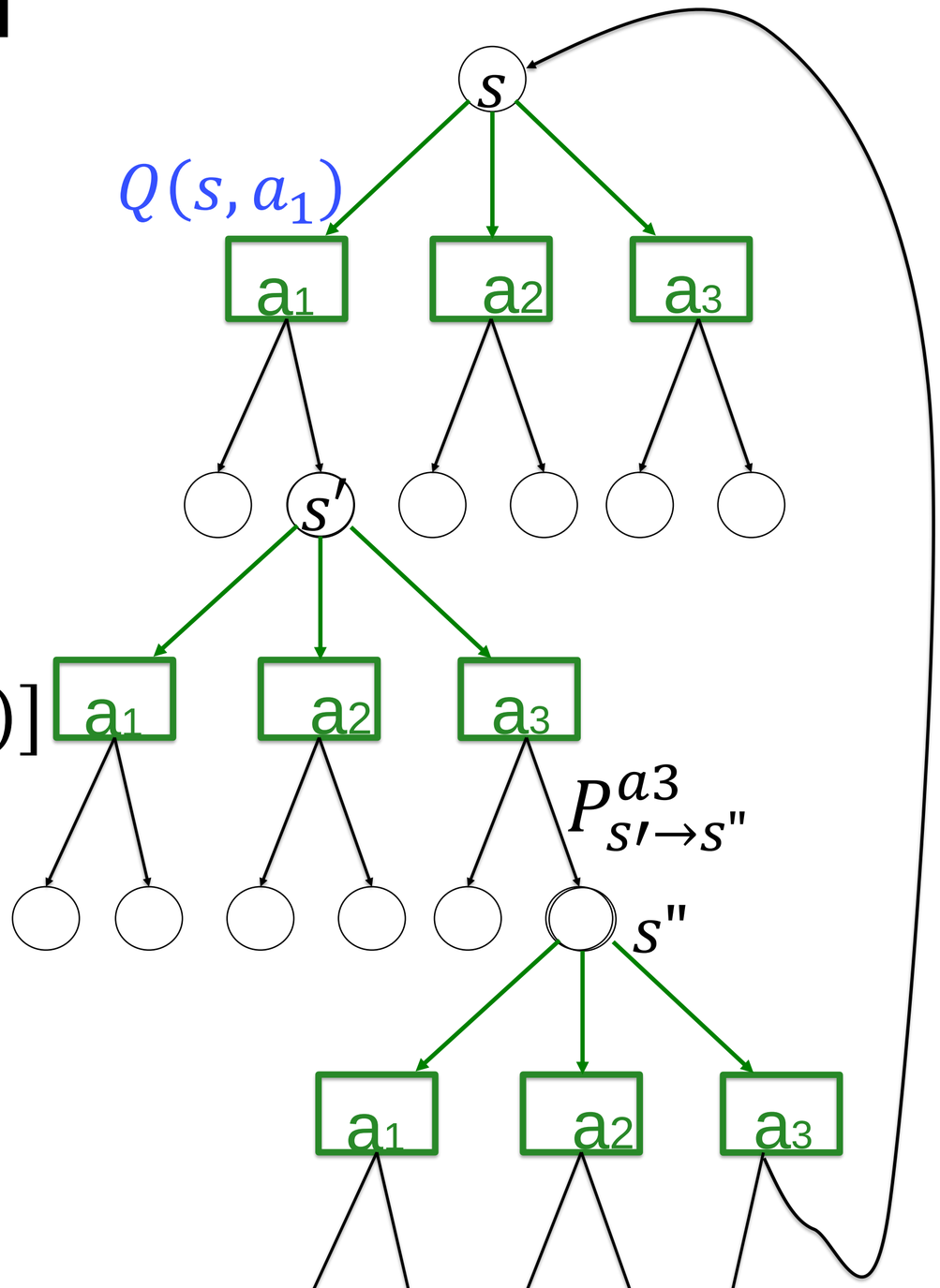
Total expected (discounted) reward

Starting in state s with action a

$Q(s,a)$ =

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

$$= E[r_t + \gamma\, r_{t+1} + \gamma^2 r_{t+2} + \gamma^3\, r_{t+3} + \ldots | s, a)]$$



Discount factor: γ <1

-important for recurrent state transition graphs!

-avoids blow-up of summation

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

Previous slide.

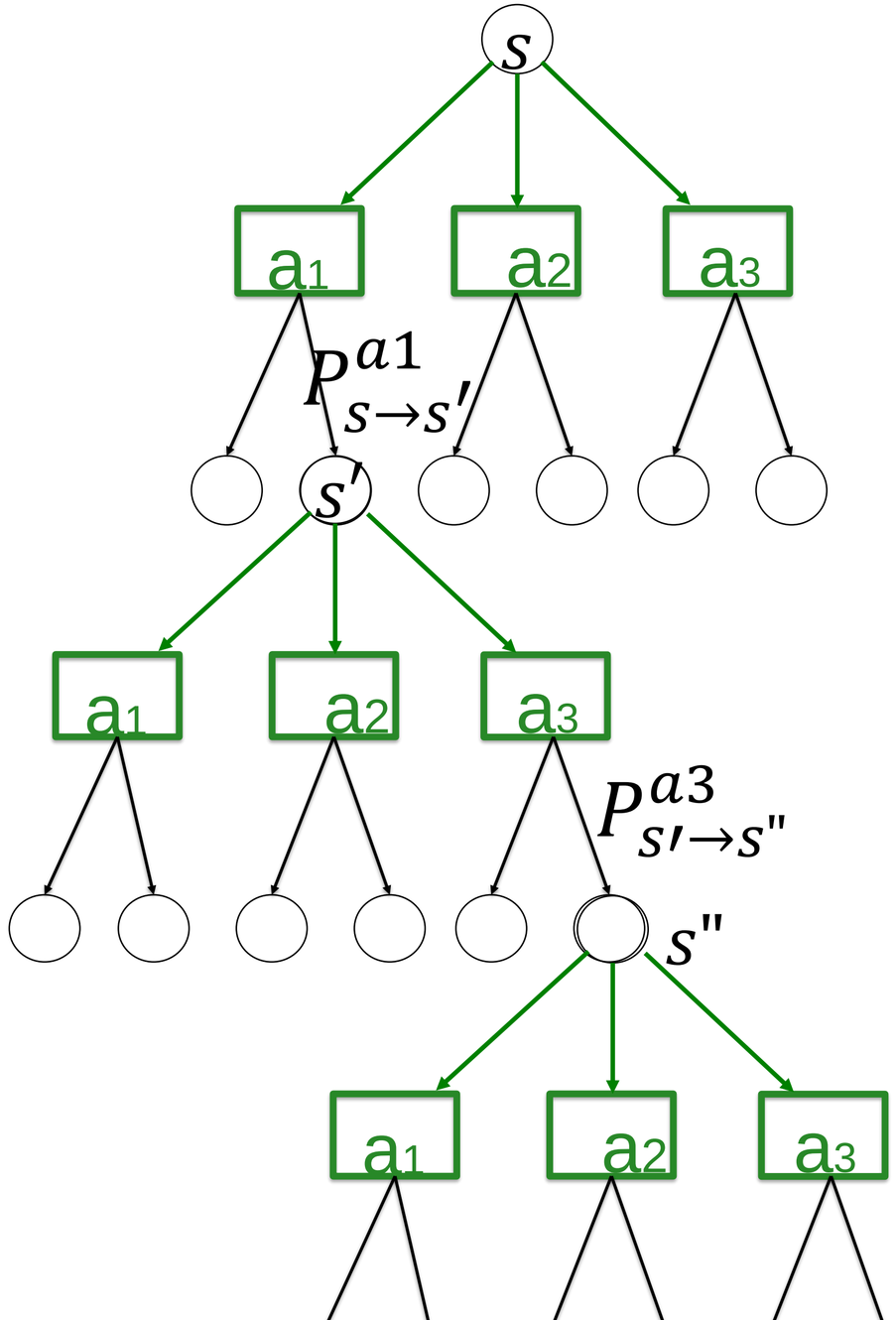
Angular brackets denote expectation (or averages over many trials, always with the same policy π(s,a) and all starting in (s,a).

Red-font lower-case r indicates the reward collected over multiple time steps in **one single episode**, starting in state s with action a.

Expectation means that we have to take the average over all possible future paths giving each path its correct probabilistic weight.
The probabilistic weight includes the fixed policy π(s,a)
as well as the branching ratio P(s,a)

Bellman equation

Blackboard4:
Bellman eq.

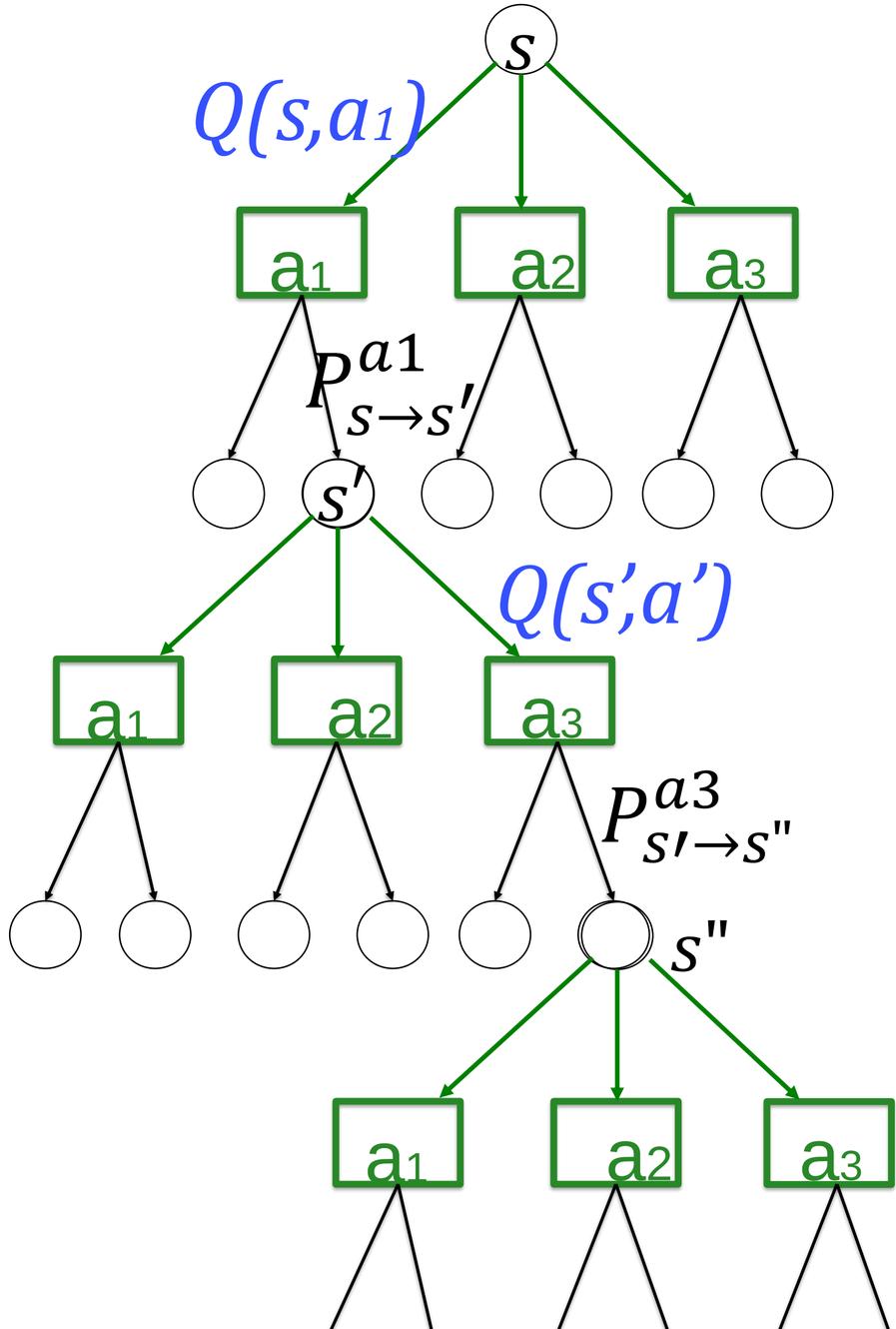


Space for calculations.

Bellman equation with policy π

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

Bellman equation = value consistency of neighboring states



Remark:

Sometimes Bellman equation is written for greedy policy: $\pi(s,a) = \delta_{a,a*}$

with action $a^* = \underset{a'}{\operatorname{argmax}}\, Q(s,a')$

Previous slide.

The Bellman equation relates the Q-value for state s and action a with the Q-values of the neighboring states.
Neighboring means reachable in a single step.
Note that the two different types of branching ratio both enter the equation.

Bottom: in the case of a greedy policy, the Bellman equation simplifies

Bellman equation (for optimal actions)

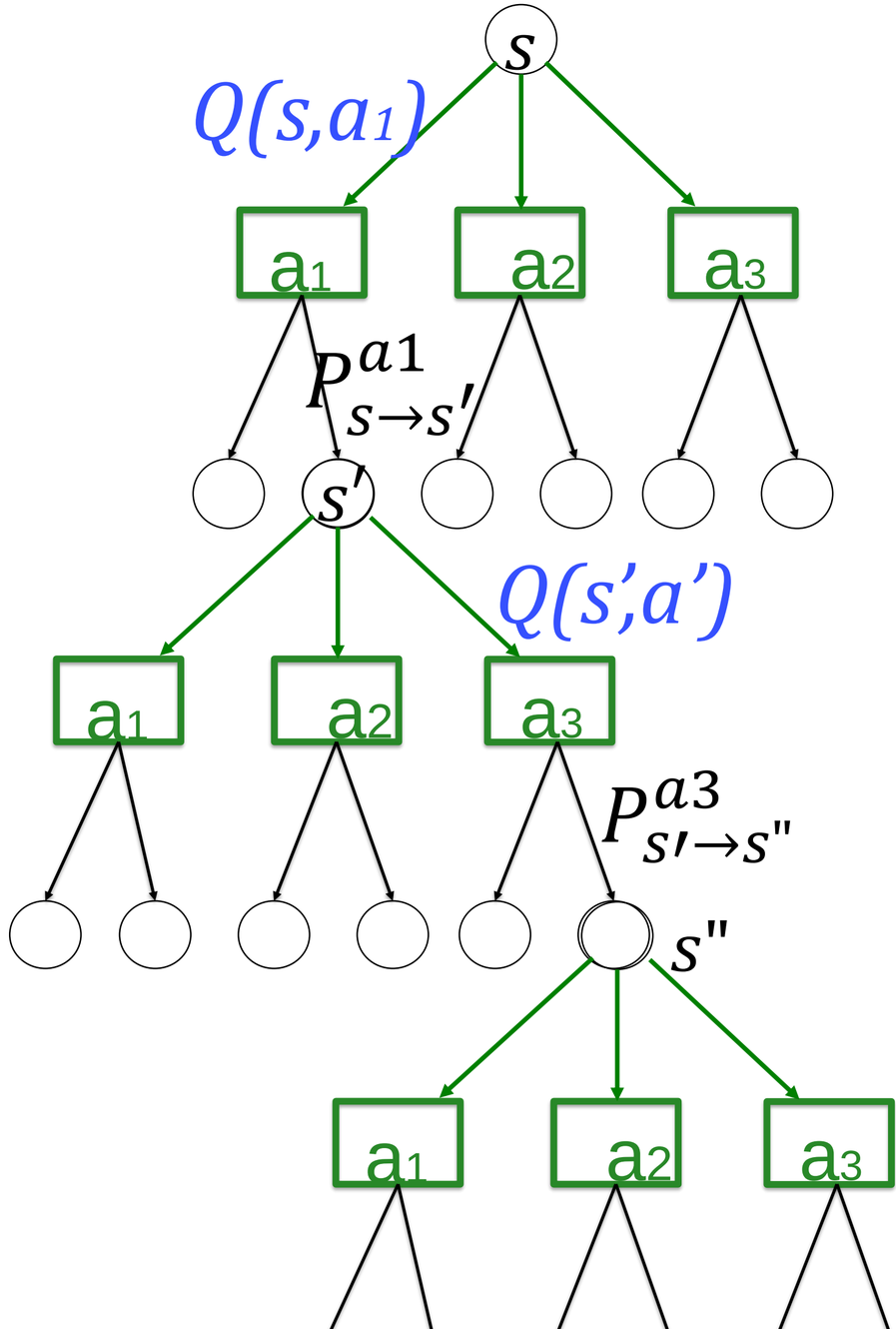
$$Q(s,a) = \sum_{s'} P^{a}_{s\to s'} \left[R^{a}_{s\to s'} + \gamma \sum_{a'} \pi(s',a') Q(s',a') \right]$$

for greedy policy:

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

with action $a^* = \underset{a'}{\operatorname{argmax}}\, Q(s,a')$

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



Previous slide.
For a greedy policy, the sum over actions disappears from the Bellman equation and is replaced by the max-sign.

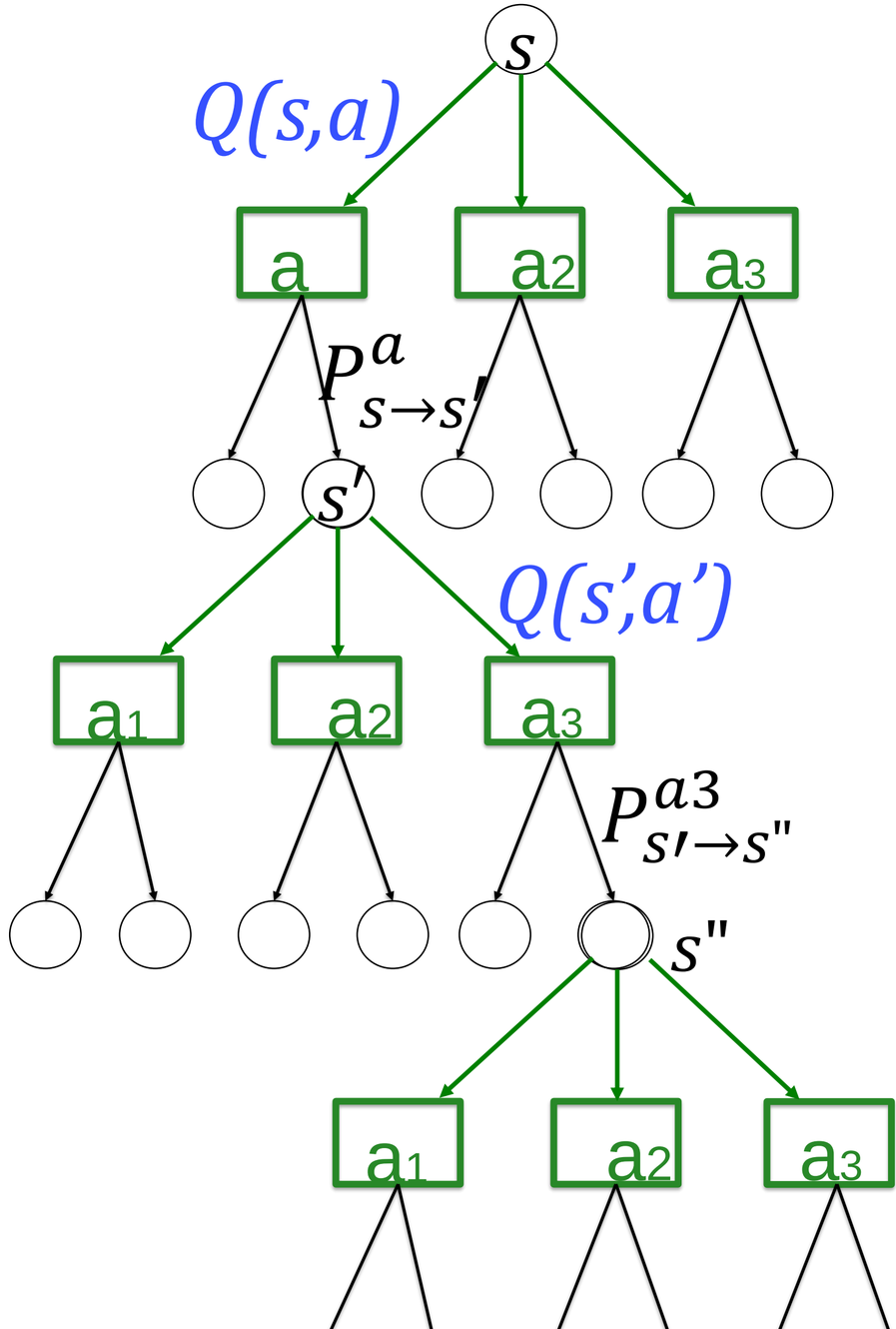
Quiz: Bellman equation with policy π

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

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

[ ] The Bellman equation is linear in the variables $Q(s'a')$

[ ] The set of variables $Q(s',a')$ that solve the Bellman equation is unique and does not depend on the policy



Your comments.

Learning Neural Networks: RL1

Reinforcement Learning and SARSA

Wulfram Gerstner

EPFL, Lausanne, Switzerland

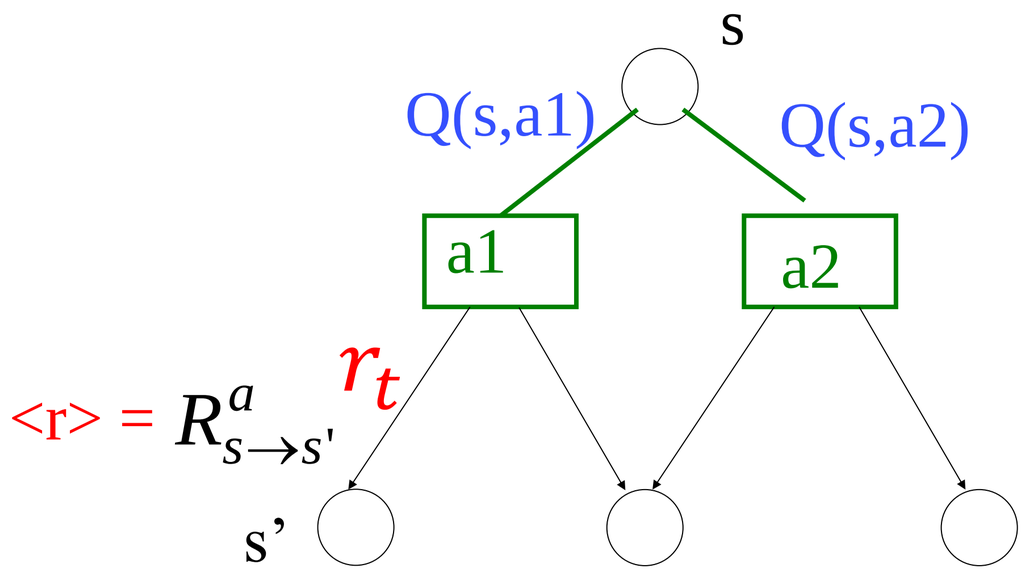
Part 6: SARSA Algorithm

- Examples of Reward-based Learning
- Elements of Reinforcement Learning
- One-step Horizon (Bandit Problems)
- Exploration vs. Exploitation
- Bellman Equation
- **SARSA Algorithm**

Previous slide.
We not turn to the first practical algorithm, called SARSA. This is an algorithm that is widely used in the field of reinforcement learning.

Review: Iterative update of Q-values

Problem: Q-values not given



Solution: iterative update

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

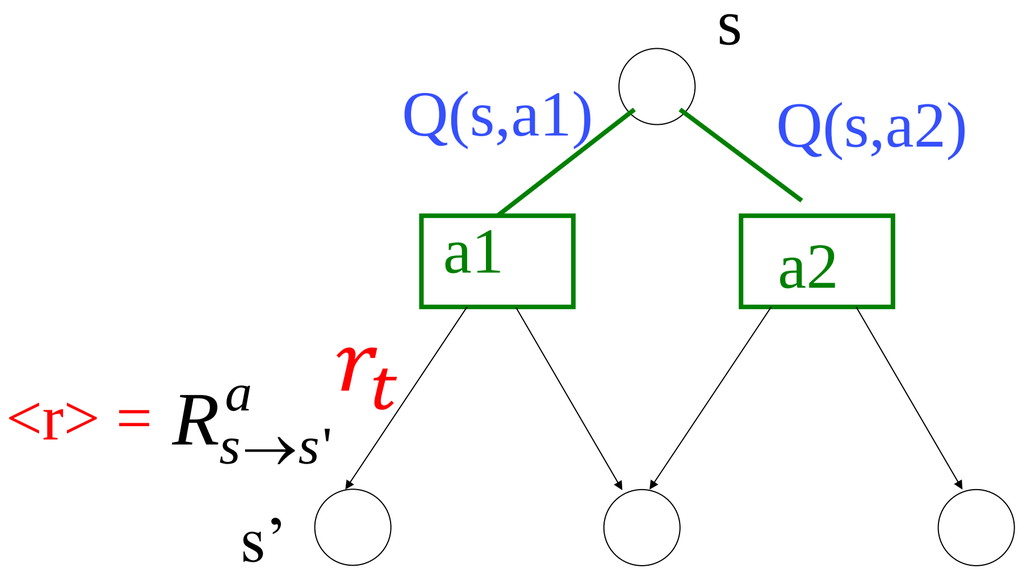
while playing with policy $\pi(s,a)$

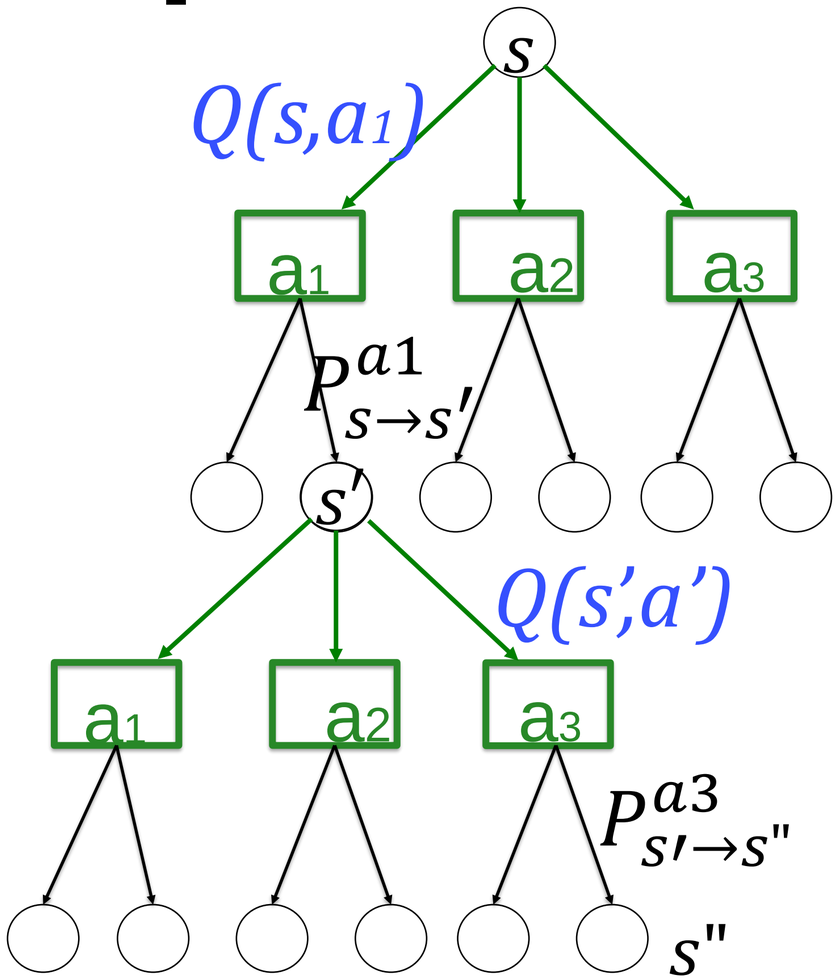
Previous slide.
Reminder: for the 1-step horizon scenario we found that we could calculate the Q-values iteratively.
We increase the Q-value by a small amount (with learning rate 0<eta<<1) if the reward observed at time t is larger than our current estimate of Q.
And we decrease the Q-value by a small amount if the reward observed at time t is smaller than our current estimate of Q.

Iterative updates with one data point at a time are also called ‘online algorithms’.
Thus our update rule is an online algorithm for the estimation of Q-values.

Iterative update of Q-values for multistep environments

Problem: Q-values not given





Solution: iterative update

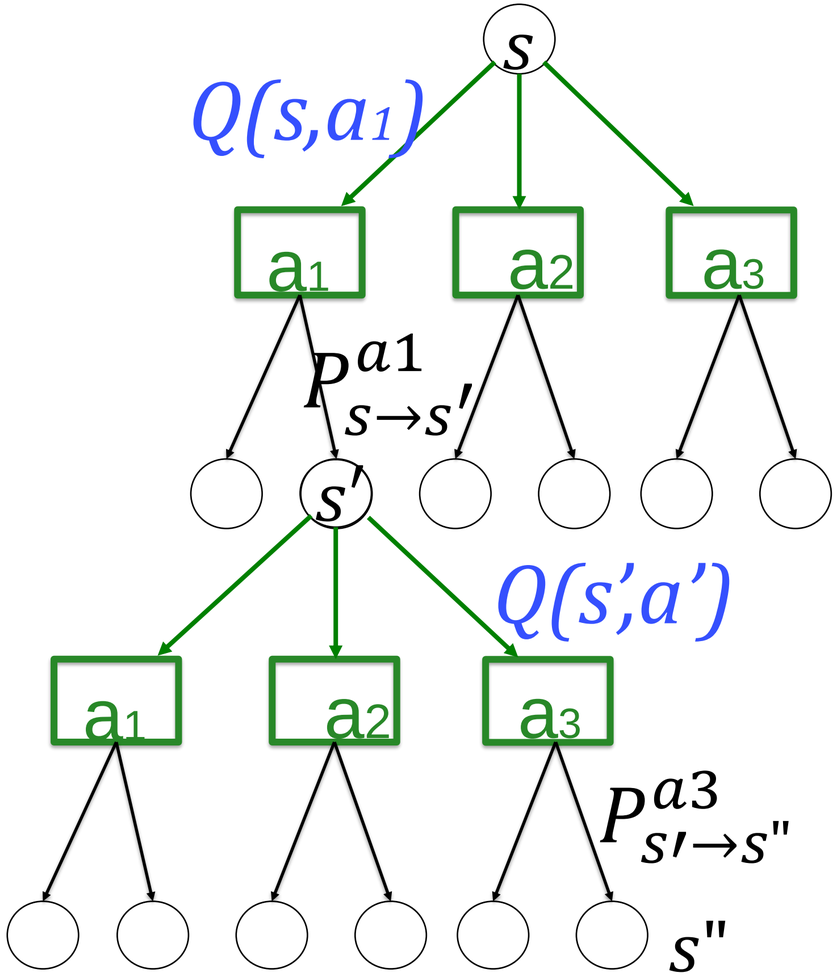
$$\Delta\hat{Q}(s,a) = \eta\,[r_t - \hat{Q}(s,a)]$$

$$\Delta\hat{Q}(s,a) = \;?$$

while playing with policy $\pi(s,a)$

Previous slide.
The question now is: can we have a similar iterative update scheme also for the multi-step horizon?

Blackboard5: SARSA update



Your notes.

Iterative update of Q-values for multistep environments

Bellman equation:

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

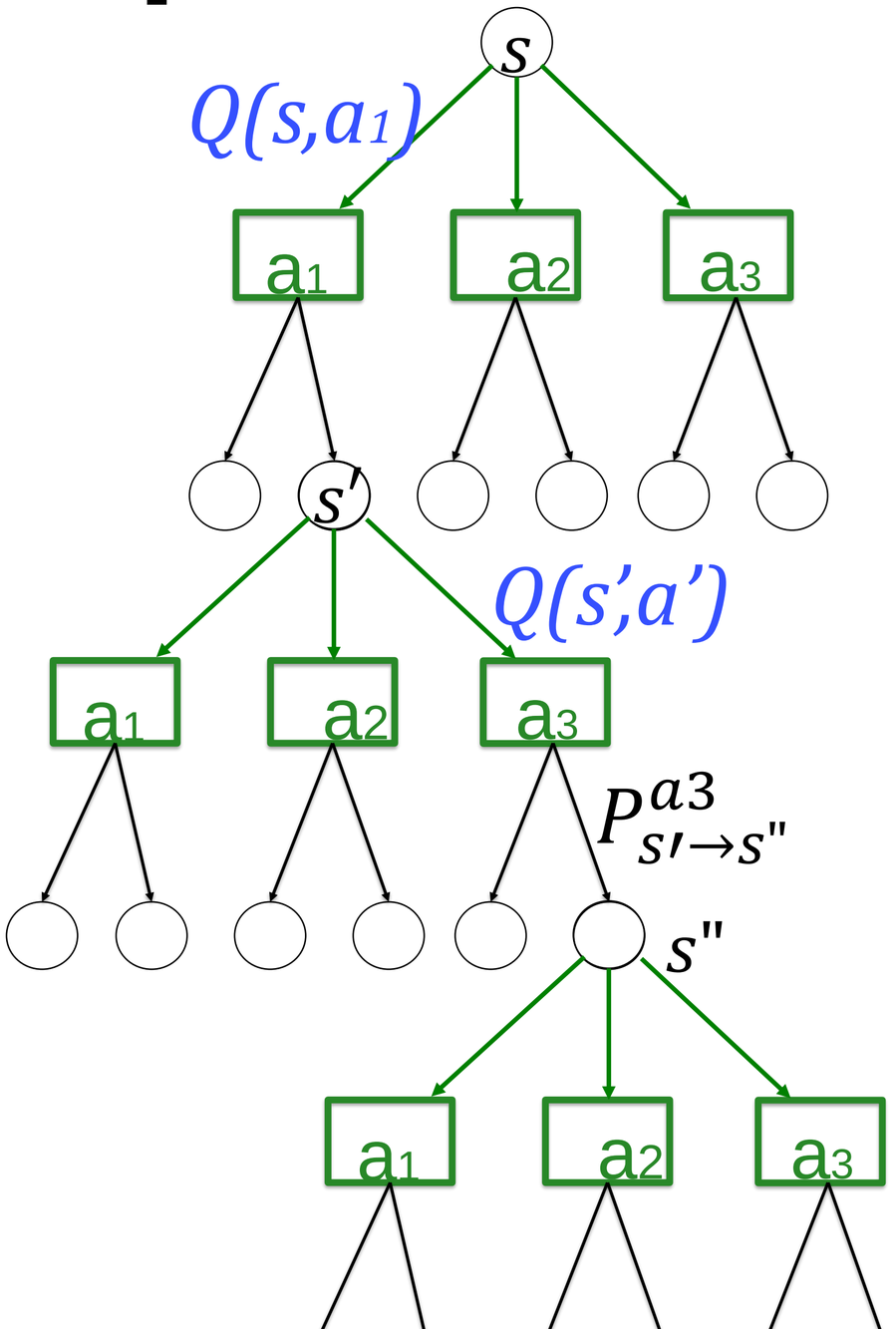
Problem:

- Q-values not given
- branching probabilities not given
- reward probabilities not given

Solution: iterative update

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

while playing with policy $\pi(s,a)$



Previous slide.
Even for the case of the multi-step horizon, we can estimate the Q-values by an interative update:

The Q-values Q(s,a) is increased by a small amount if the sum of (reward observed at time t plus discounted Q-value in the next step) is larger than our current estimate of Q(s,a).

This iterative update gives rise to an online algorithm.

NOTE: in the following we always work with empirical estimates, and drop the ‘hat’ of the variable Q.

SARSA vs. Bellman equation

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

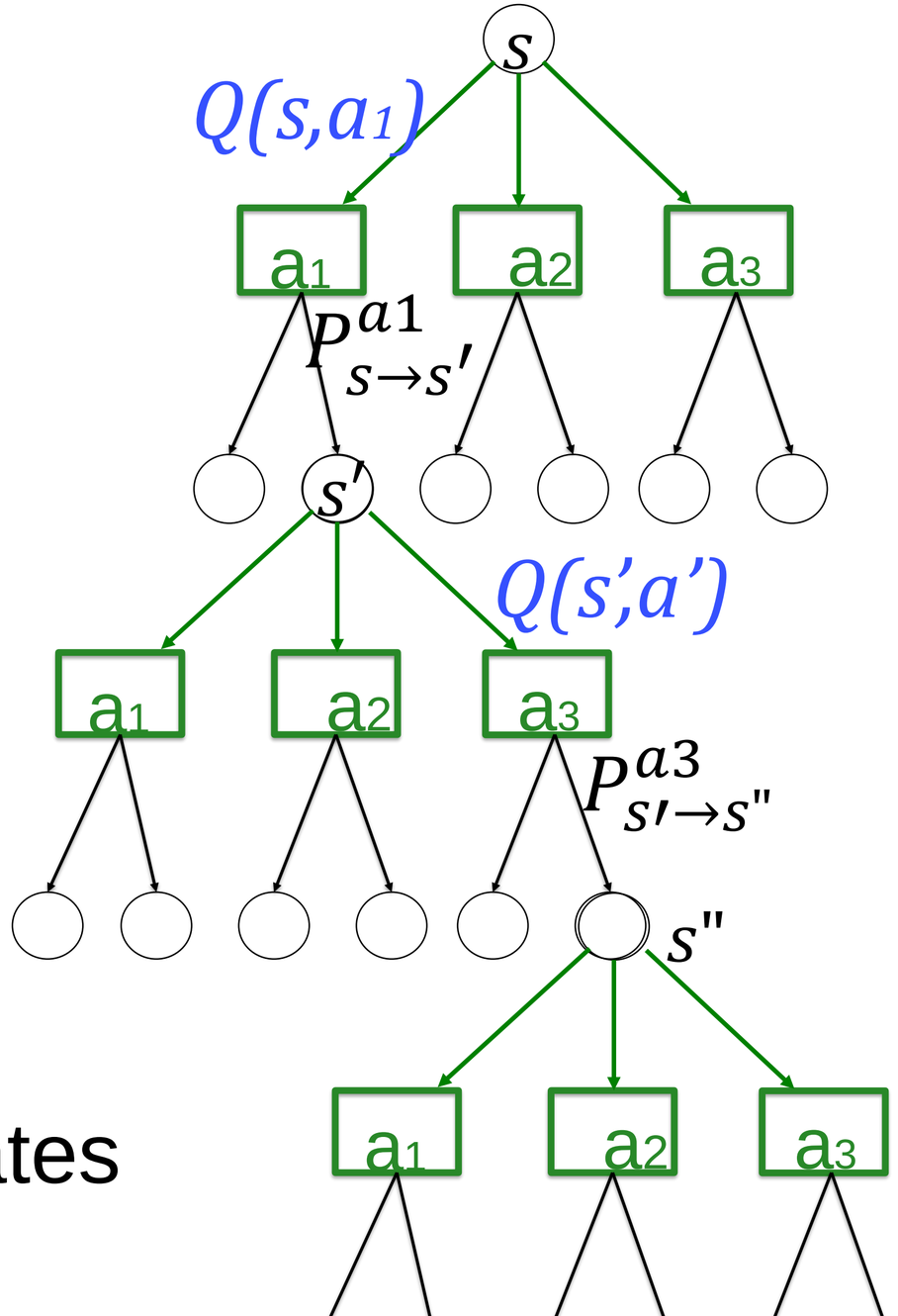
Bellman equation

= consistency of Q-values
across neighboring states

SARSA update rule

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

= make Q-values of neighboring states
more consistent



Previous slide.
The Bellman equation summarizes the consistency condition:
The (average) rewards must explain the difference between Q(s,a) and γ Q(s',a') averaged over all s' and a'.
Or equivalently:
Q(s,a) must be explained by the (average) reward in the next step and the discounted Q-value in the next state.

The iterative update formula implies that Q(s,a) needs to be adapted so the current reward explains the difference between Q(s,a) and Q(s',a').

An equivalent form of writing the update is:

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

This form highlights the similarity to the case of the update rule in the cae of the one-step horizon.

SARSA algorithm

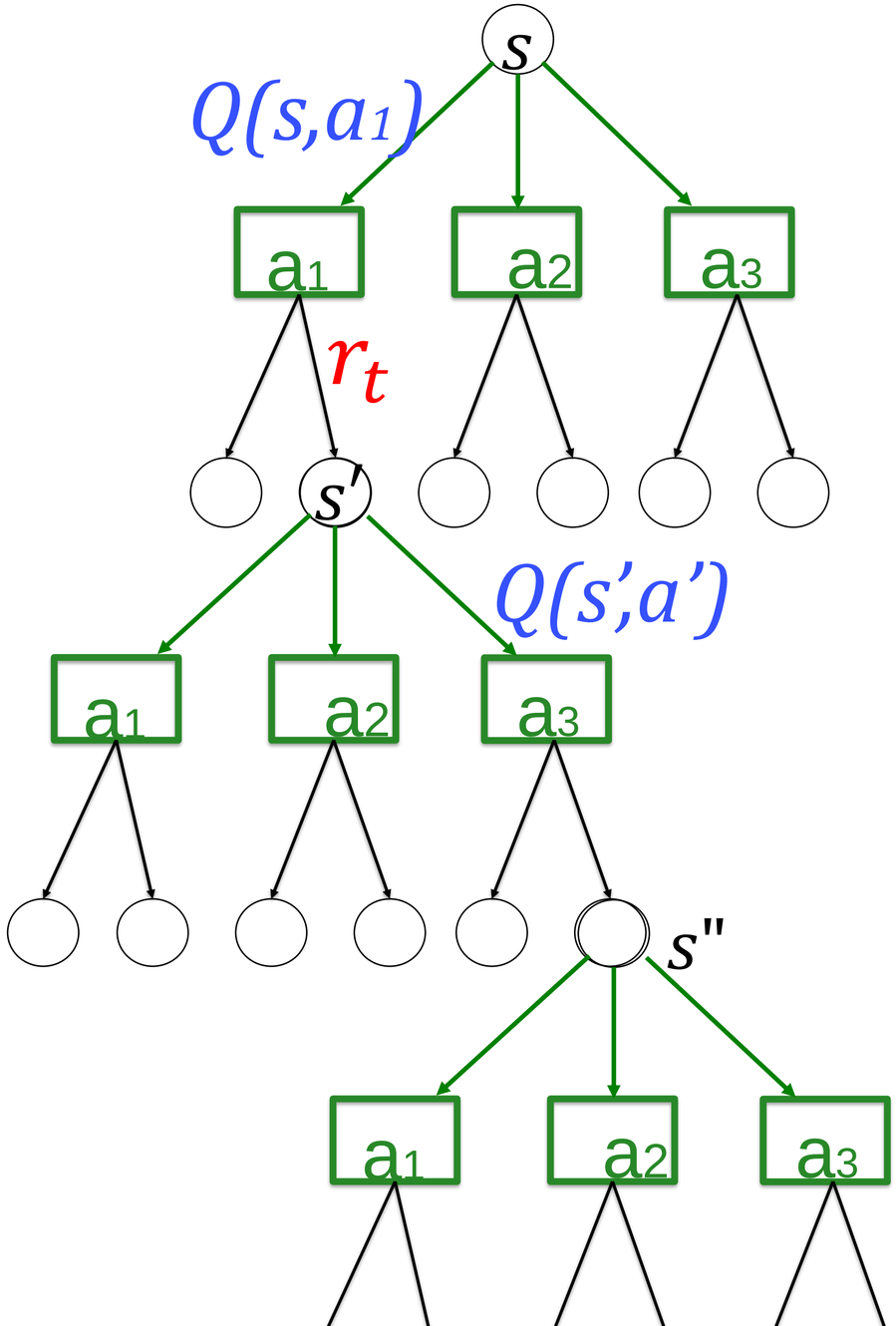
Initialise Q values
Start from initial state **s**

1) being in state **s**
   and having chosen 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 1)

Stop when all Q-values have converged



Previous slide.
The update rule gives immediately rise to an online algorithm. You play the game. While you run through one of the episodes you observe the state s, choose action a, observe reward r, observe next state s’ and choose next action a’. At this point in time (and not earlier) you have all the information to update the Q-value Q(s,a).

The name SARSA comes from this sequence state-action-reward-state-action.

SARSA algorithm.

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

We have initialized SARSA and played for n>2 steps.
Is the following true for the next steps?

[ ] in SARSA, updates are applied after each move.

[ ] in SARSA, the agent updates the Q-value $Q(s_{(t)},a_{(t)})$ related to the **current** state $s_{(t)}$

[ ] in SARSA, the agent updates the Q-value $Q(s_{(t-1)},a_{(t-1)})$ related to the **previous** state, once it has chosen $a_{(t)}$

[ ] in SARSA, the agent moves in the environment using the policy $\pi(s,a)$

[ ] SARSA is an online algorithm

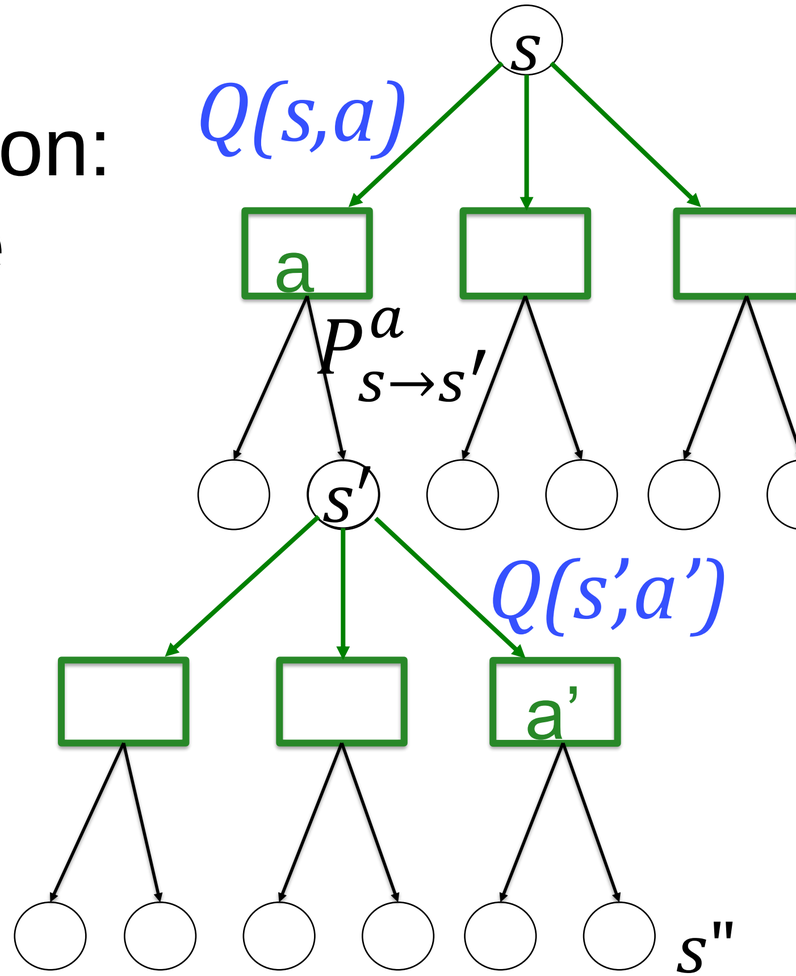
Previous slide.

Variant A: SARSA is consistent w. Bellman equation

Blackboard 6A: SARSA

We proof the following:

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



Notes:

- Expectation is taken for fixed Q-values and hence for fixed policy (consider Q-values as $\boldsymbol{\theta^{old}}$)
- Expectation $\mathrm{E}[\Delta Q(s,a)]$ is taken over all possible paths starting in (s,a). I call this ‘batch-like’.
- The length of the path is given by the needs of the update equation: Here from (s,a) to (r,s’,a’)
- Look at state-action-diagram to keep track of terms

Your comments.

Variant A: SARSA is consistent w. Bellman equation

We have proven the following:

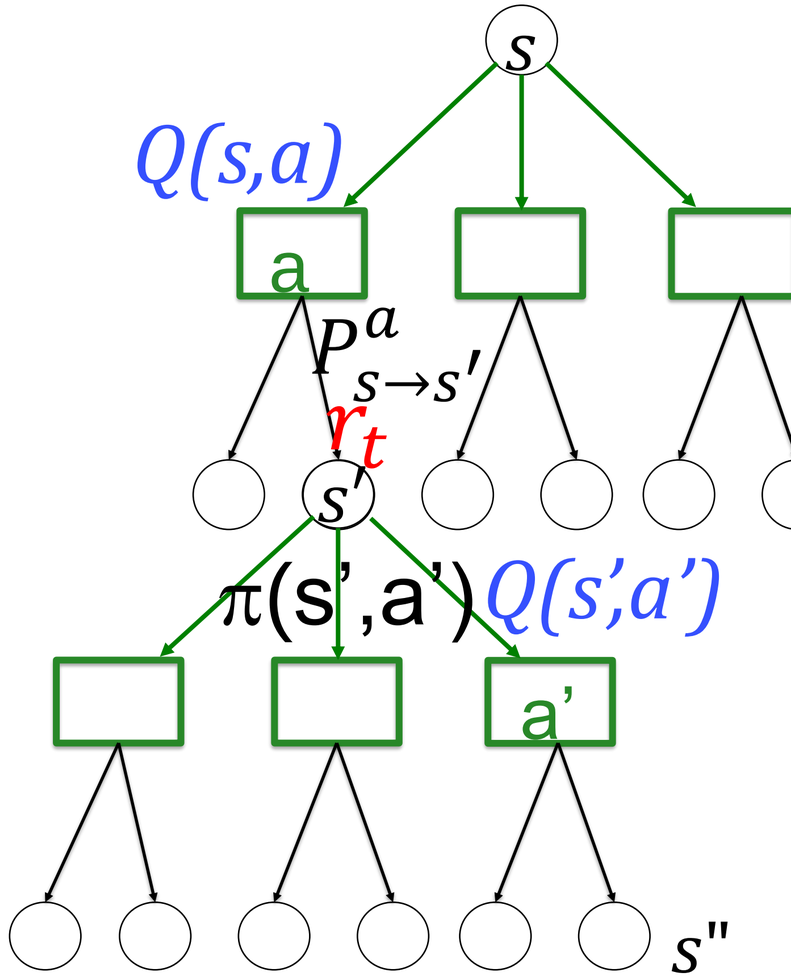
Suppose that we have found a set of Q-values.

We keep them frozen while evaluating expectation.

IFF $E[\Delta Q(s,a)] = 0$, then the Q-values solve the Bellman equation.

$$E[\Delta Q(s,a)] = \eta E[r_t + \gamma Q(s',a') - Q(s,a)]=0$$

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



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 fixed 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.

Your comments.

Variant B: Bellman equation and SARSA: theorem for small η

Setting: 'temporal averaging'

The SARSA algo has been applied for a very long time, using updates

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

IF (i) averaged over many update steps

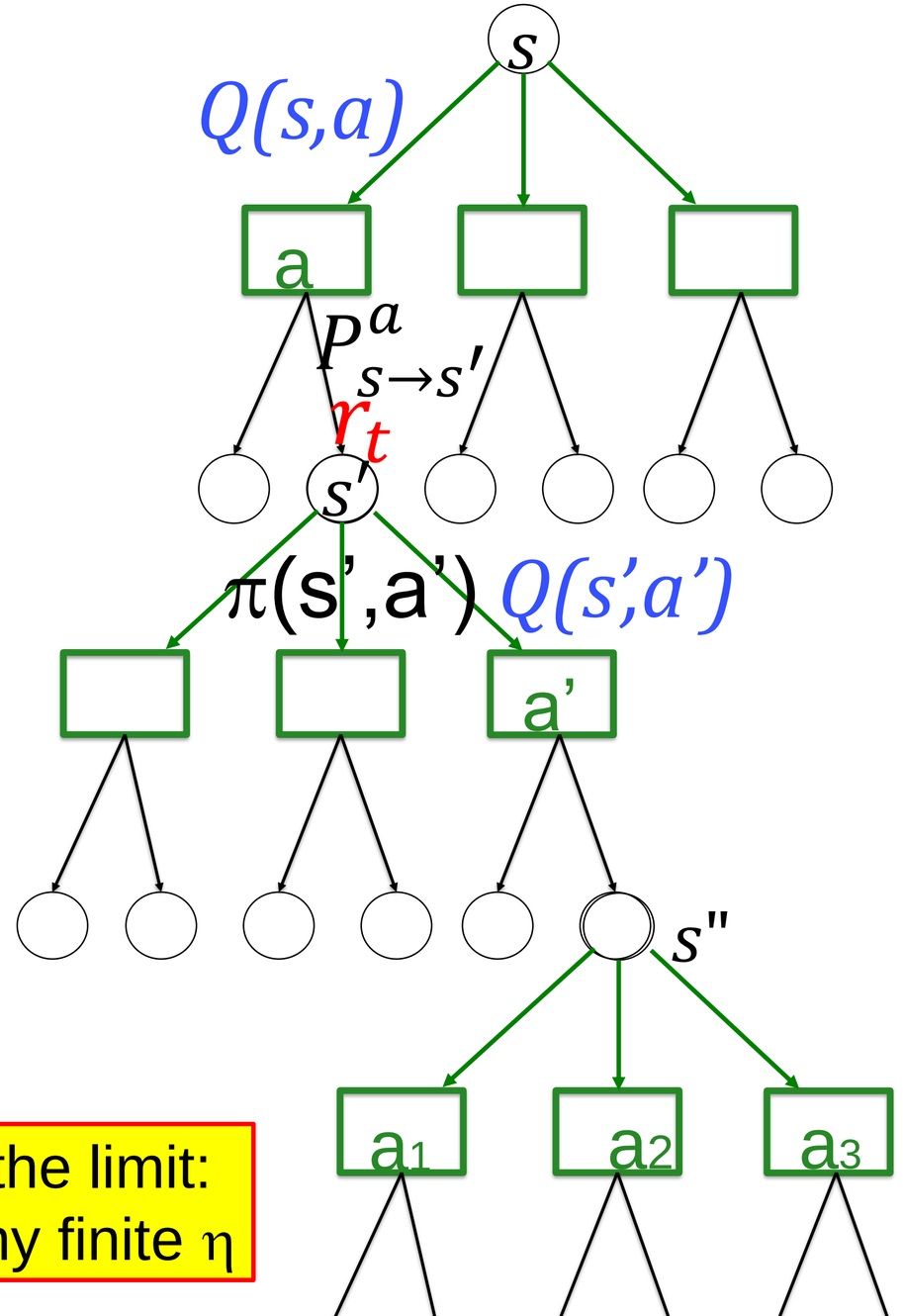
$$\langle \Delta Q(s,a) | s,a \rangle = 0$$

(ii) learning rate η is VERY small $(\eta \to 0)$

THEN: fluctuations of Q are negligible and the set of expected Q-values solves 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 $\pi(s',a')$



Only true in the limit:
Wrong for any finite η

Previous slide. There are two different versions of the theorem. This version B is proven in the annex, 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. In version B, we take the empirical mean over all moments when we played action a in state s.** Hence the next update uses Q-values that have changed in the previous one.

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 $\eta \to 0$, so that changes can be neglected.
Hence there are 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. In the Annex, we assume that Q-values may fluctuate slightly about their stable values. This approach gives additional insights into the situation of the online SARSA, once it has converged in expectation.

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.

Summary: Reinforcement Learning and SARSA

Before Next Week:
MUST DO:
Exercise SARSA

Learning outcome and conclusions:

- **Reinforcement Learning is learning by rewards**
  → world is full of rewards (but not full of labels)
- **Agents and actions**
  → agent learns by interacting with the environment
  → state s, action a, reward r
- **Exploration vs Exploitation**
  → optimal actions are easy if we know reward probabilities
  → since we don't know the probabilities we need to explore
- **Bellman equation**
  → self-consistency condition for Q-values
- **SARSA algorithm:** state-action-reward-state-action
  → update while exploring environment with current policy

Exercise: SARSA in 1-dim environment

- **Update of Q values in SARSA (with γ =1)**

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

- **policy for action choice:**

  Pick most often action

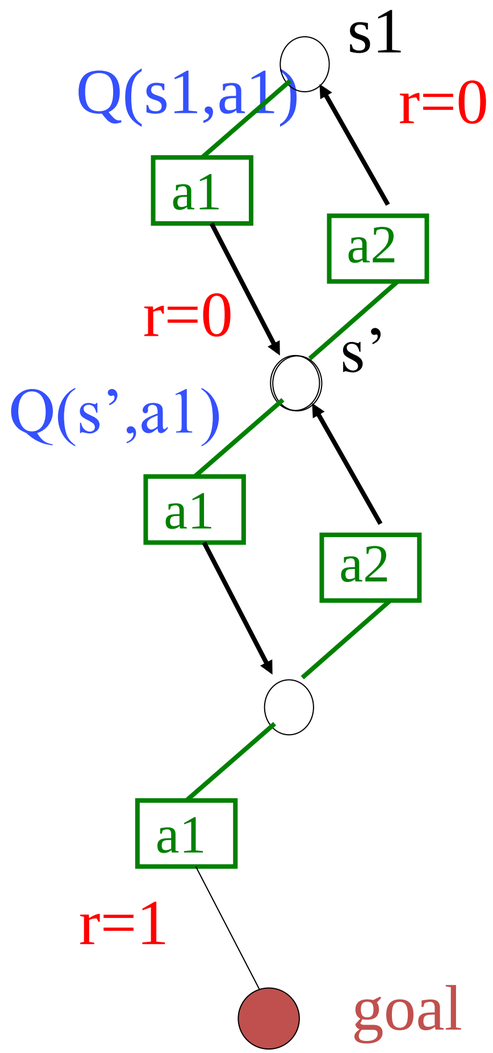
  $$a_t^* = \arg\max_a Q_a(s,a)$$

  Break ties stochastically

- States form linear sequence
- Reward only at goal.

- Initialise Q values at 0. Start at top state s1.

- Q values after 2 complete episodes?



Exercise 4: SARSA for Linear Track.

Exercise 4 (at 15h15)

Exercise 4. SARSA algorithm

In the lecture, we introduced the SARSA (state-action-reward-state-action) algorithm, which (for discount factor $\gamma = 1$) is defined by the update rule

$$\Delta Q(s,a) = \eta \left[r - \left(Q(s,a) - Q(s',a') \right) \right] , \tag{1}$$

where s' and a' are the state and action subsequent to s and a. In this exercise, we apply a greedy policy, i.e., at each time step, the action chosen is the one with maximal expected reward, i.e.,

$$a_t^* = \arg\max_a Q_a(s,a) . \tag{2}$$

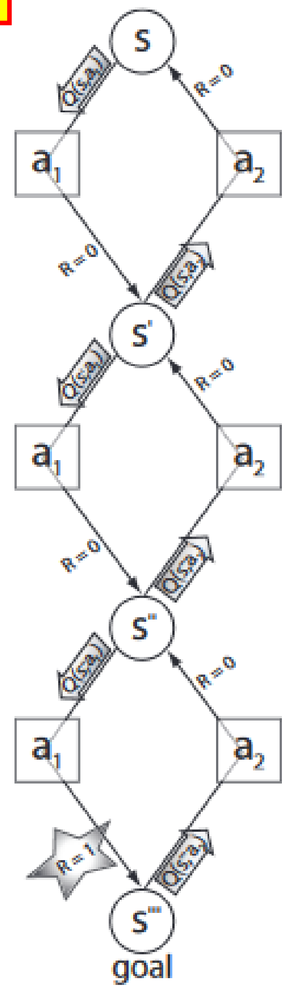
If the available actions have the same Q-value, we take both actions with probability 0.5.

Consider a rat navigating in a 1-armed maze (=linear track). The rat is initially placed at the upper end of the maze (state s), with a food reward at the other end. This can be modeled as a one-dimensional sequence of states with a unique reward ($r = 1$) as the goal is reached. For each state, the possible actions are going up or going down (Fig. 2). When the goal is reached, the trial is over, and the rat is picked up by the experimentalist and placed back in the initial position s and the exploration starts again.

a. Suppose we discretize the linear track by 6 states, $s_1, \ldots, s_6$. Initialize all the Q-values at zero. How do the Q-values develop as the rat walks down the maze in the first trial?

b. Calculate the Q-values after 3 complete trials. How many Q-value values are non-zero? How many trial do we need so that information about the reward has arrived in the state just 'below' the starting state?

c. What happens to the learning speed if the number of states increases from 6 to 12? How many Q-values are non-zero after 3 trials? How many trial do we need so that information about the reward has arrived in the state just 'below' the starting state?



ure 2: A linear maze.

Annex: 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

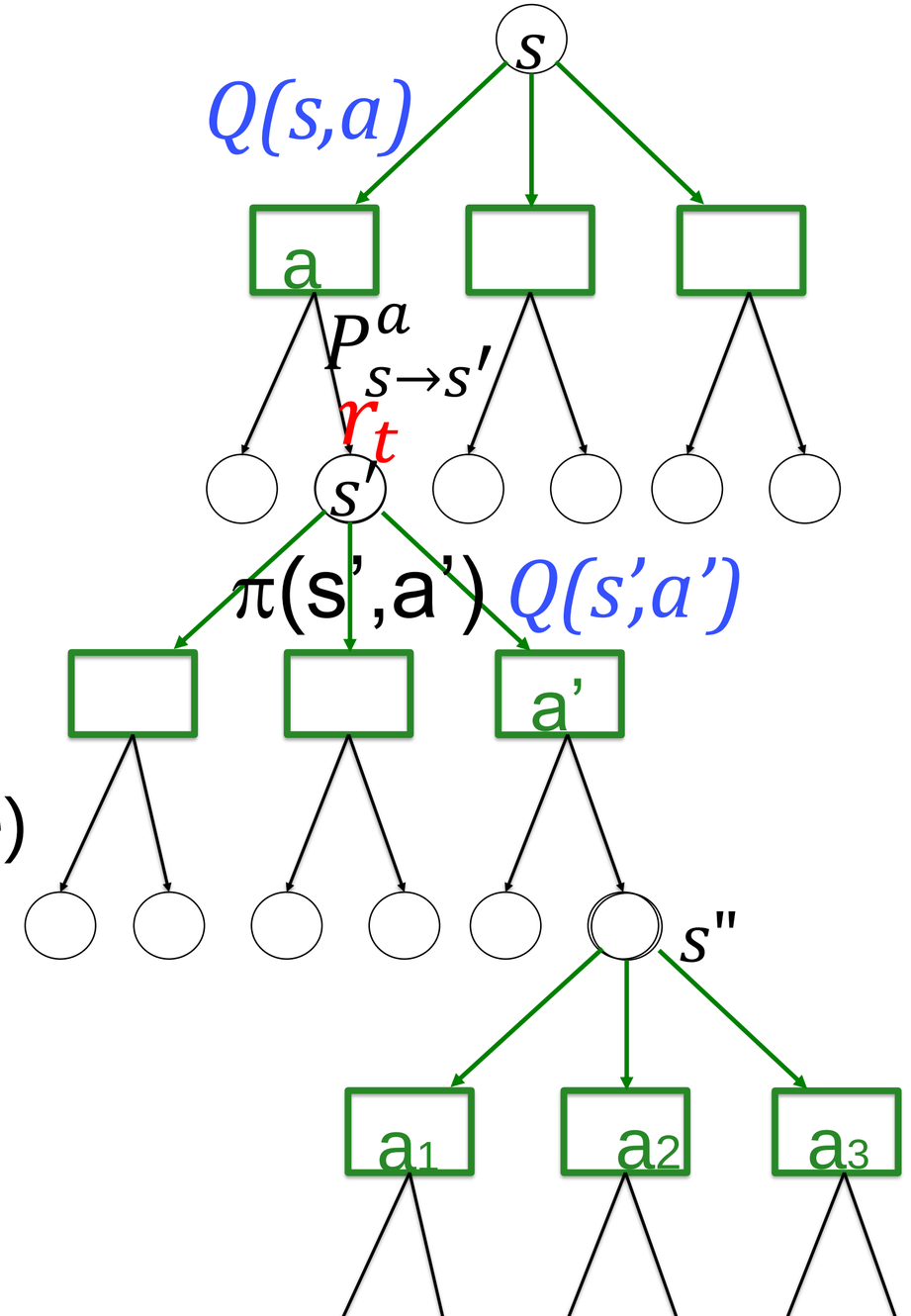
(ii) for all Q-values

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

THEN the **expectation** values (temporal average) of the set of $\hat{Q}$-values solves 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 $\pi(s',a')$



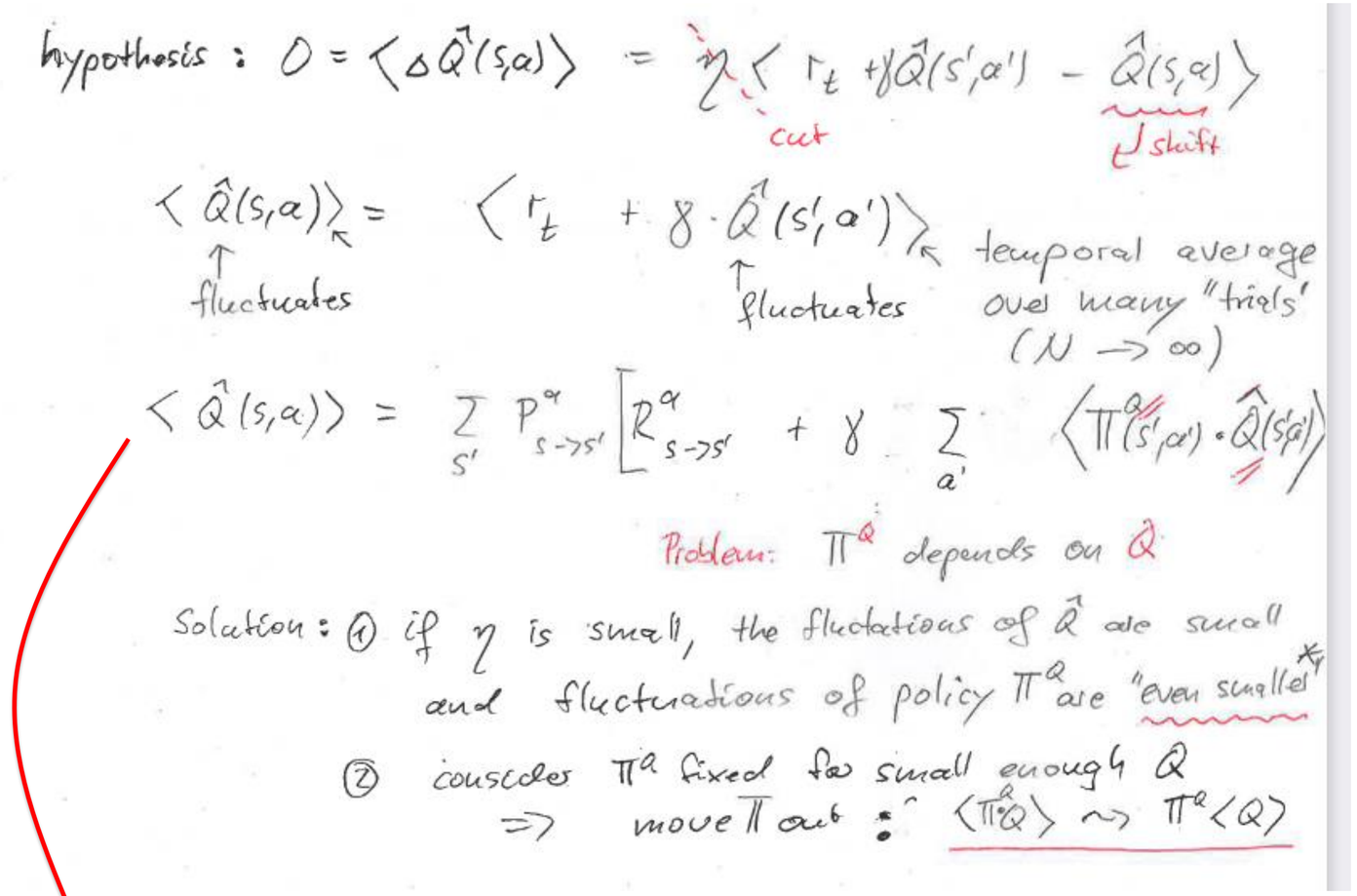
Notes: A few points should be stressed:

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: 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 $E(\pi\ Q)$ is not trivial. Moreover, correlations can lead to a shift of the value and make the result inconsistent with the Bellman equation.
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. We can interpret these expectations as corresponding to a ‘batch’ computation.
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.

SARSA is consistent with Bellman ($\eta \to 0$)

Version of Video (Variant B)

$$\frac{1}{\eta}\Delta\hat{Q}(s,a) = \eta\,[r_t + \gamma\hat{Q}(s',a') - \hat{Q}(s,a)]$$

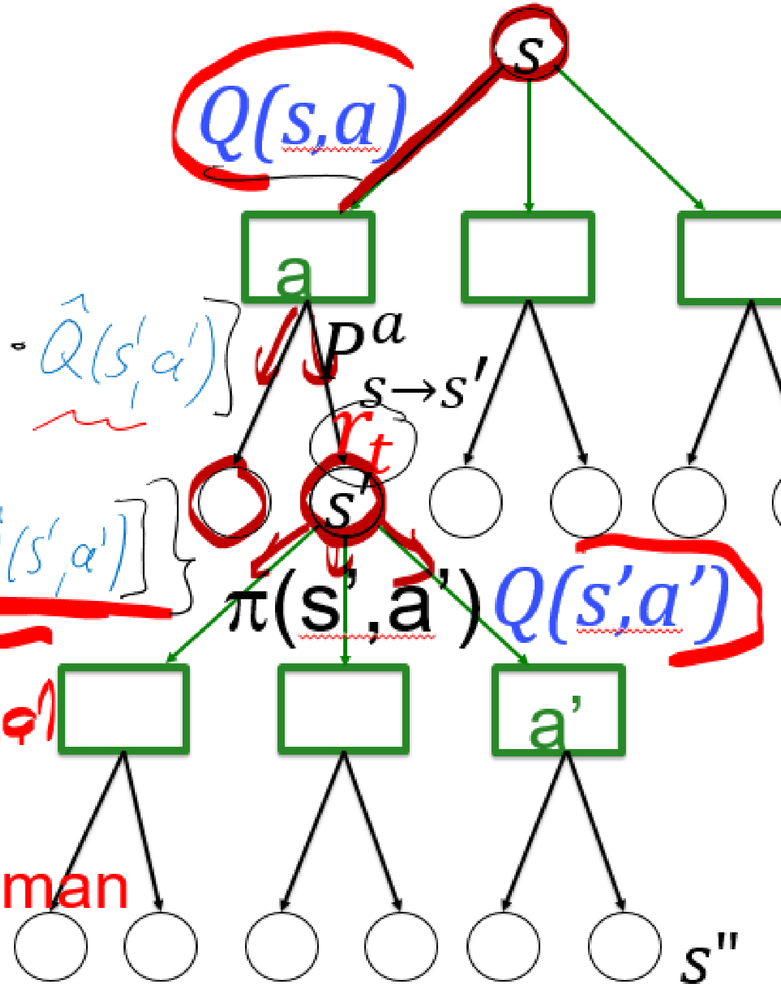
$$0 = \frac{1}{\eta}E\left(\Delta\hat{Q}(s,a)\right) = E[r_t] + \gamma E[\hat{Q}(s',a')] - E[\hat{Q}(s,a)]$$

$E[\hat{Q}(s,a)]$ given (s,a) $= \sum_{s'} P^a_{s\to s'} R^a_{s\to s'} + \gamma \sum_{s'} P^a_{s\to s'} \sum_{a'} E[\pi^Q(s',a') \cdot \hat{Q}(s',a')]$

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

$Q(s,a)$

Bellman



Online: Q(s,a) jitters, but its mean (expectation) solves Bellman

Look at graph to take expectations:

- if algo is in state s, all remaining expectations are “given s”
- if algo is on a branch (s,a), all remaining exp. are “given s and a”

The proof steps for version A were shown in class.
The proof steps for version B are not shown in class (available as video).

The end of RL1

Review: SARSA algorithm

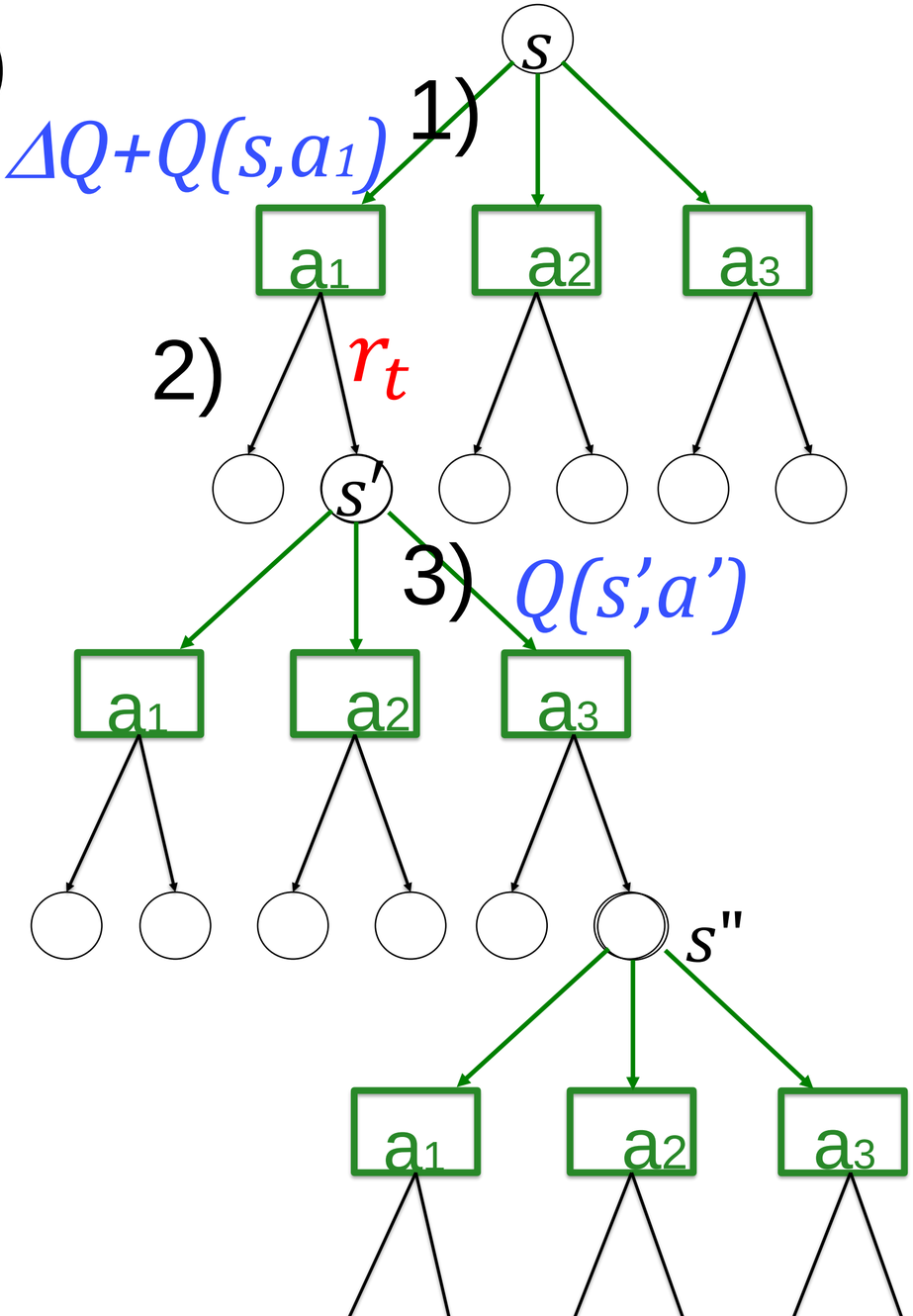
Initialise Q values
Start from initial state **s**

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

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

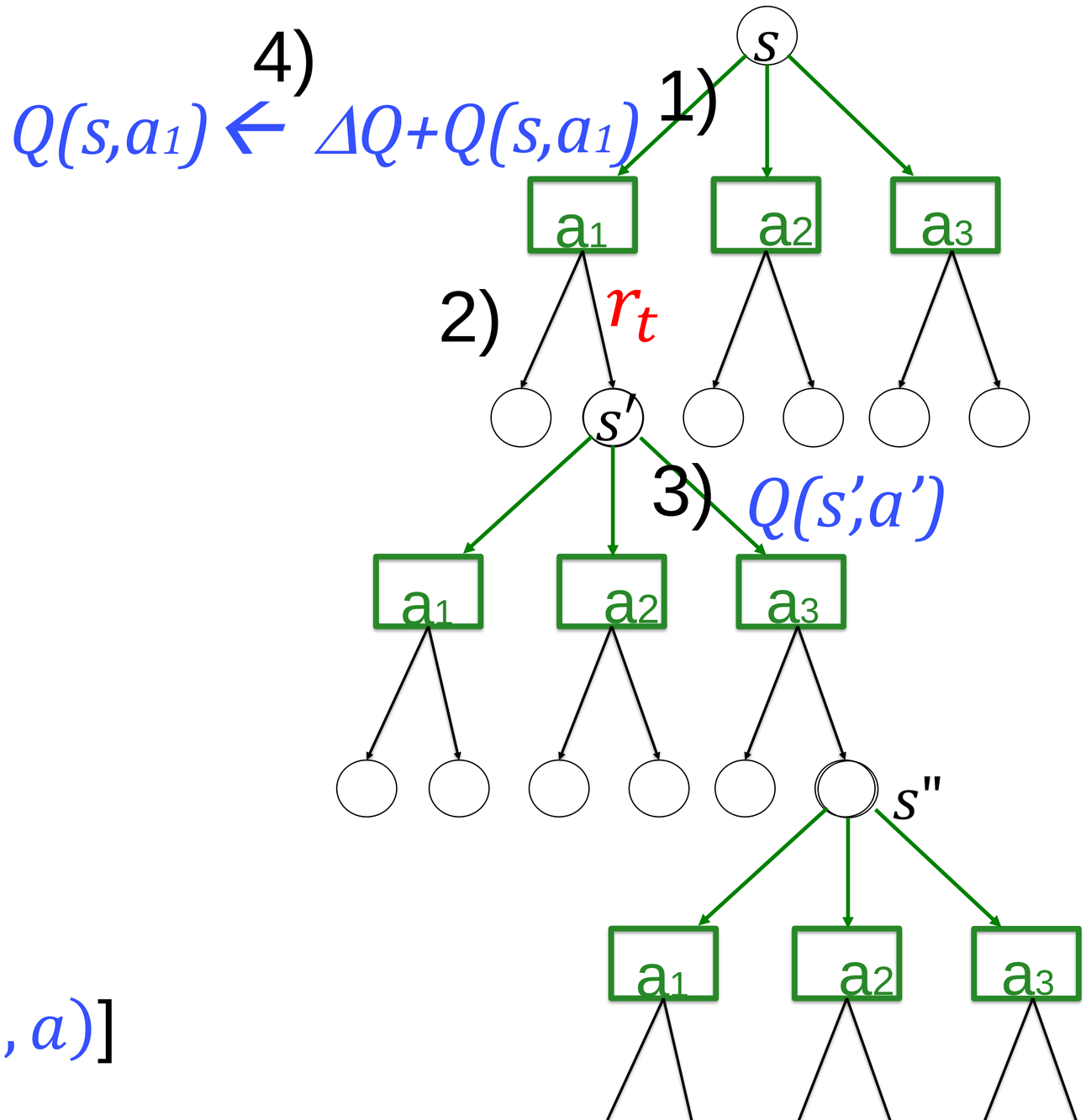
5) set: s ← s'; a ← a'
6) Goto 2)

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.

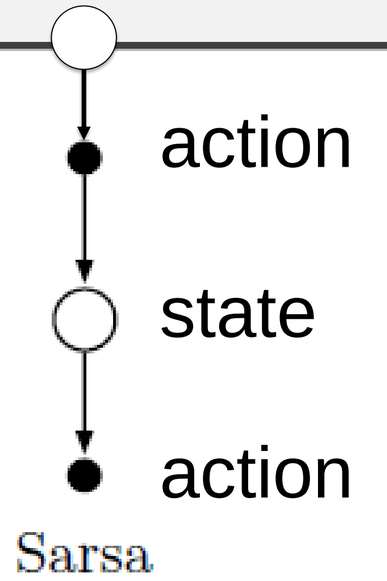
In SARSA, when you are in (s’,a’) you need to go back to the branch (s,a) so that you can do the SARSA update.

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

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Variants of TD-learning methods and eligibility traces

Part 2: Variations of SARSA

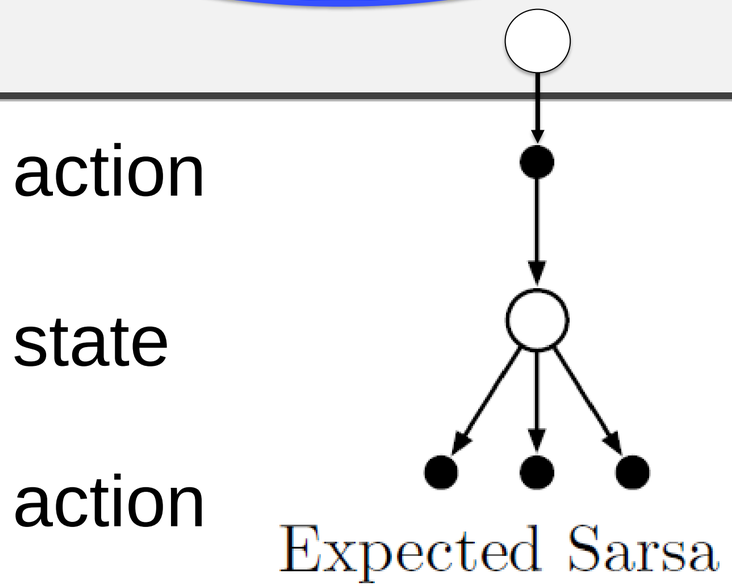
1. Review and introduction of BackUp diagrams
2. **Variations of SARSA**

(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

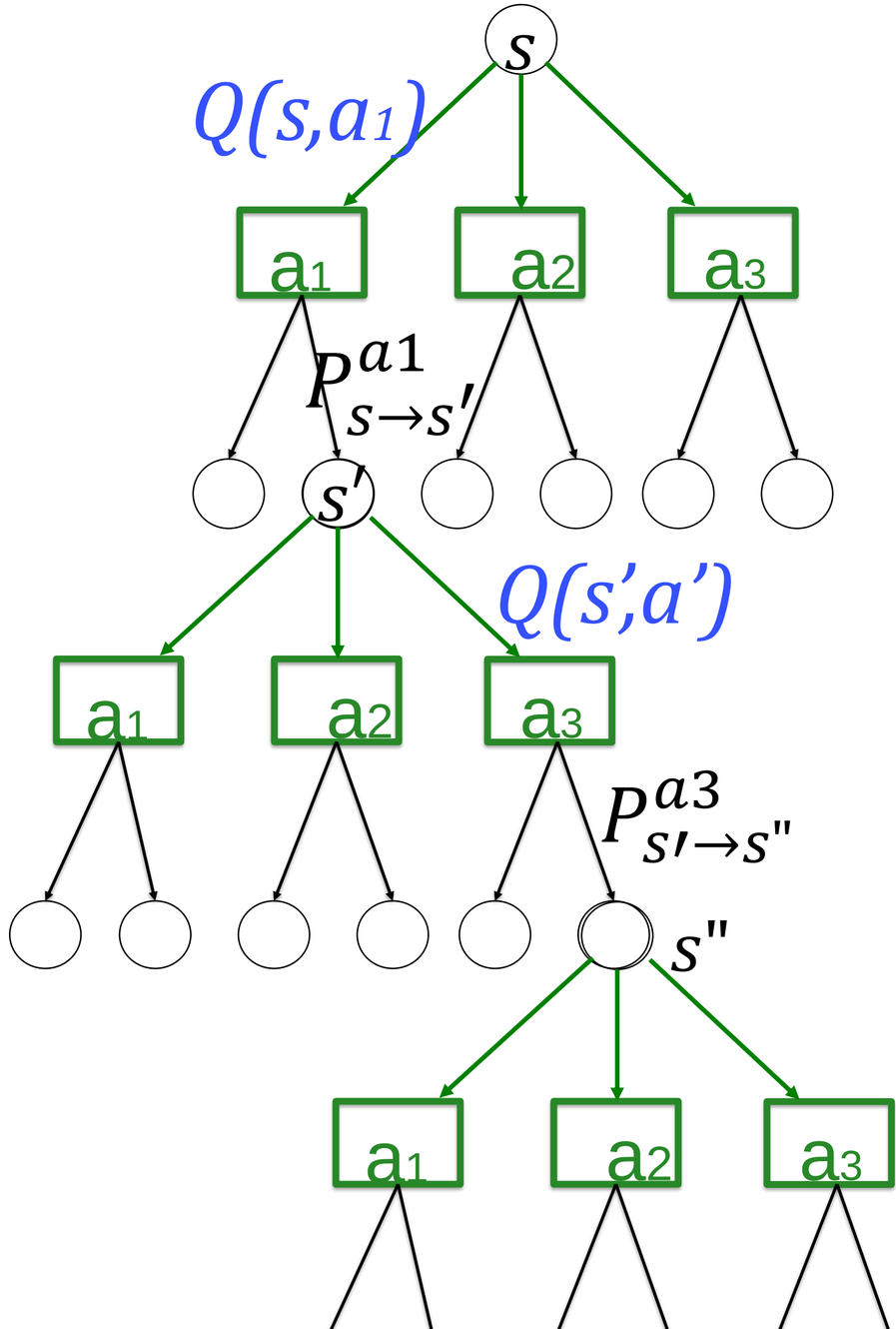
Remark:

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.

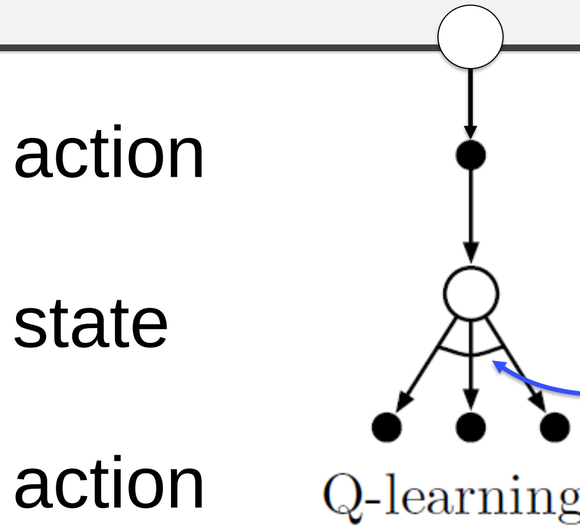
The idea is that you **run a stochastic policy that includes exploration** and visits all state-action pairs. However, since you plan to use **after learning the greedy policy** so as to maximize your returns, you already update the Q-values according the greedy policy.

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.

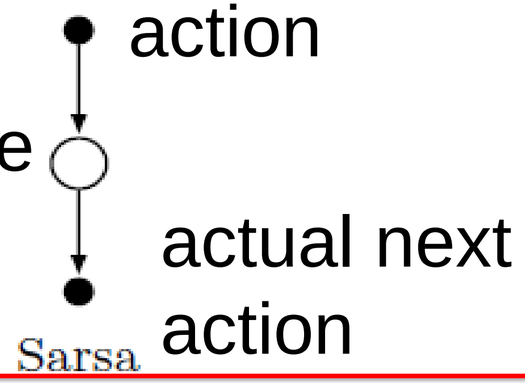
In Q-learning the update step is such that the current reward should explain the difference between Q(s,a) and the **maximum** Q(s’,a’) running over all possible actions a’. It is a TD algorithm (Temporal Difference), because neighboring states are visited one after the other. Hence neighbors are one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

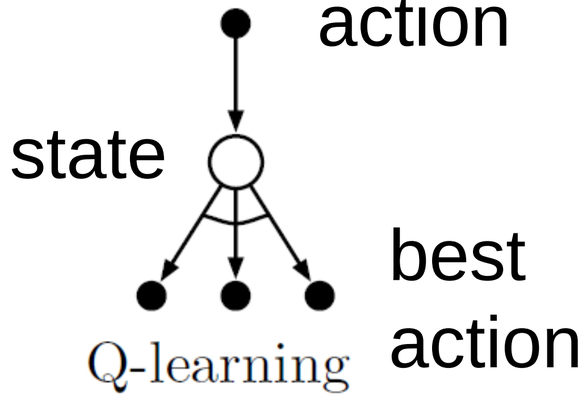
action

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

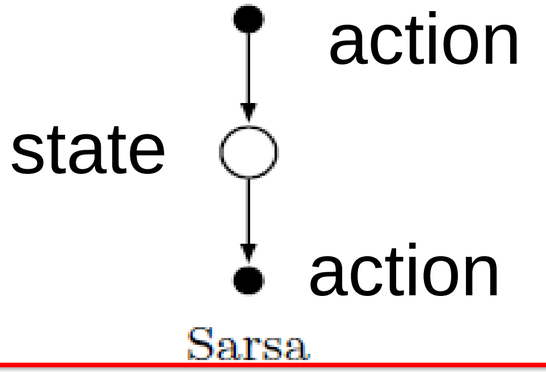


Q-learning: you look ahead and **imagine**
a greedy next action to update Q(s,a)
(but you then perform the actual next action
based on your current policy)

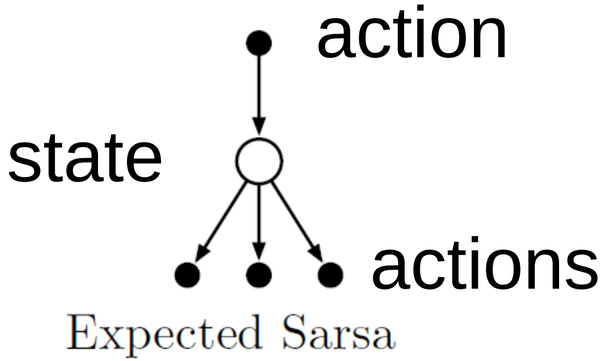
(previous slide)
On-policy versus OFF-policy.

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.

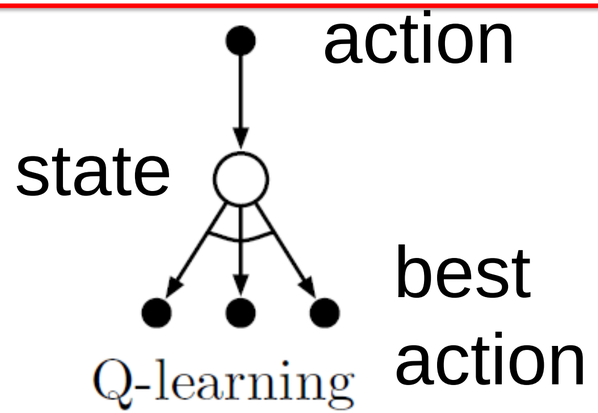
Summary: SARSA and related algorithms



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.

Teaching monitoring – monitoring of understanding

[ ] today, after the break, at least 60% of material was new to me.

[ ] after the break, I have the feeling that I have been able to follo
(at least) 80% of the lecture.

(previous slide)

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**

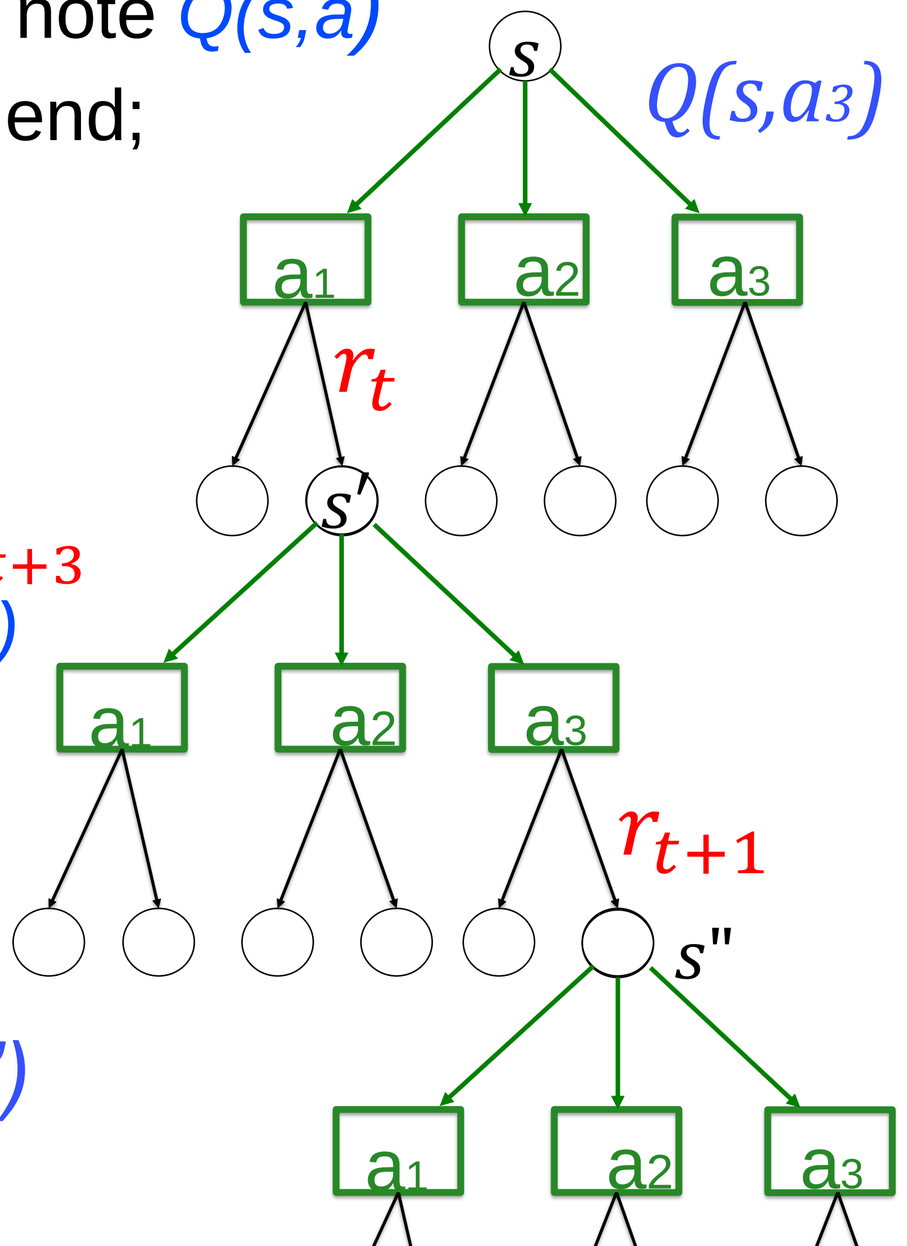
(previous slide)
Instead of using TD methods, the same state-action graph can also be explored with Monte-Carlo methods

Monte-Carlo Estimation (for Q-values)

- in state s, take action a and note *Q(s,a)*
- play trial (episode) until the end;
- then update, using the total accumulated discounted reward **(='Return') =**

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

- use Return to update *Q(s,a)*



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, also called expected *Return.*

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), by averaging over all episodes that started in (s,a).

2) You can be smart and use 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)

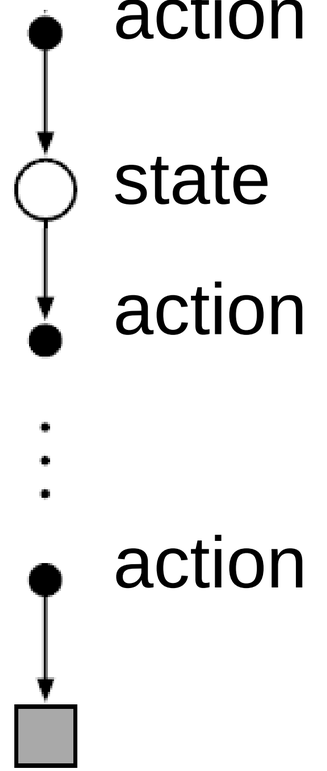
$$Return(s,a) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \ldots$$

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.

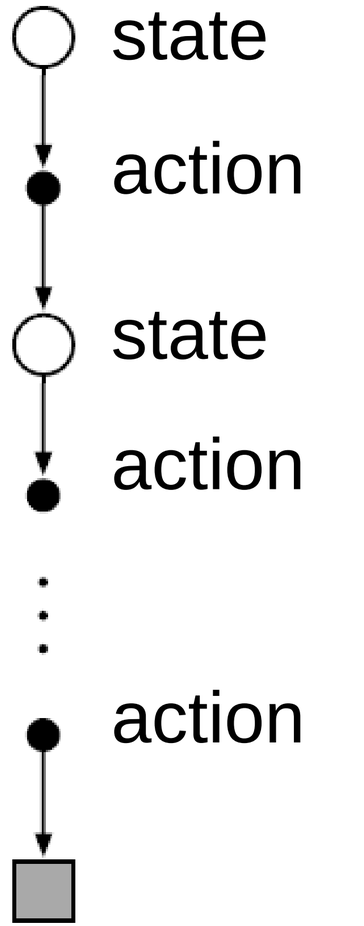
Monte-Carlo Estimation of V-values

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

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.

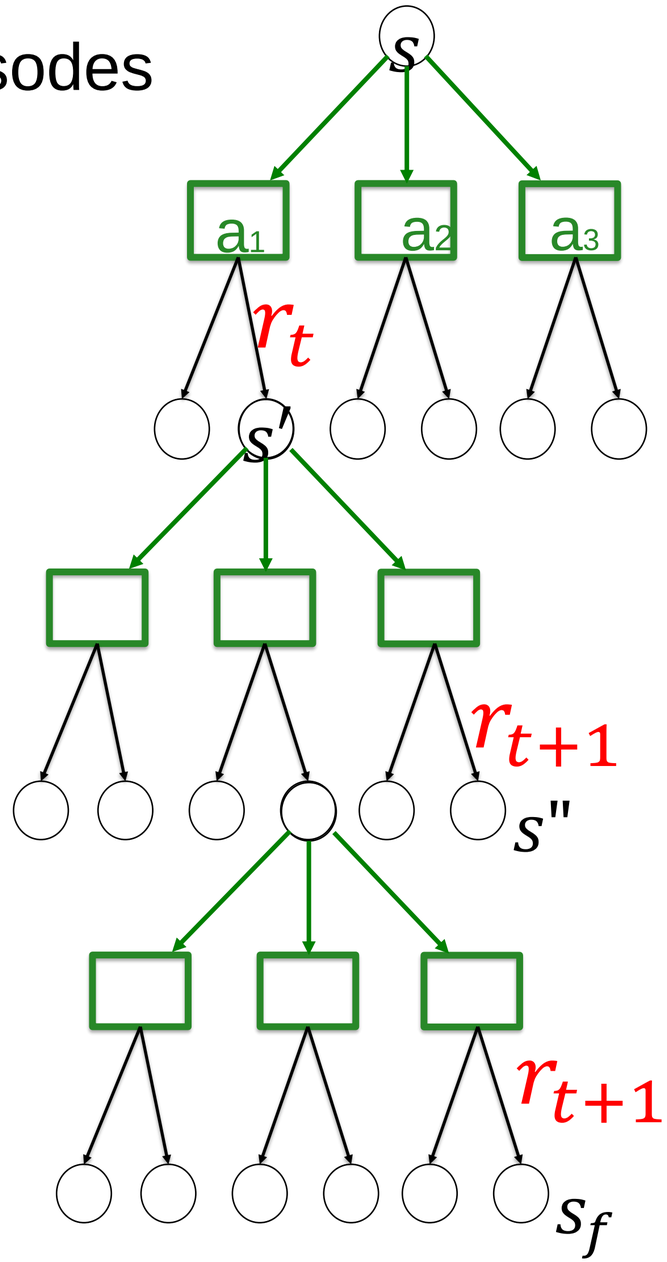
Also note that the Backup diagram is much deeper than that of Q-learning, since you always continue until the end of the trial before you can update Q-values of state-action pairs that have been encountered many steps before.

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.

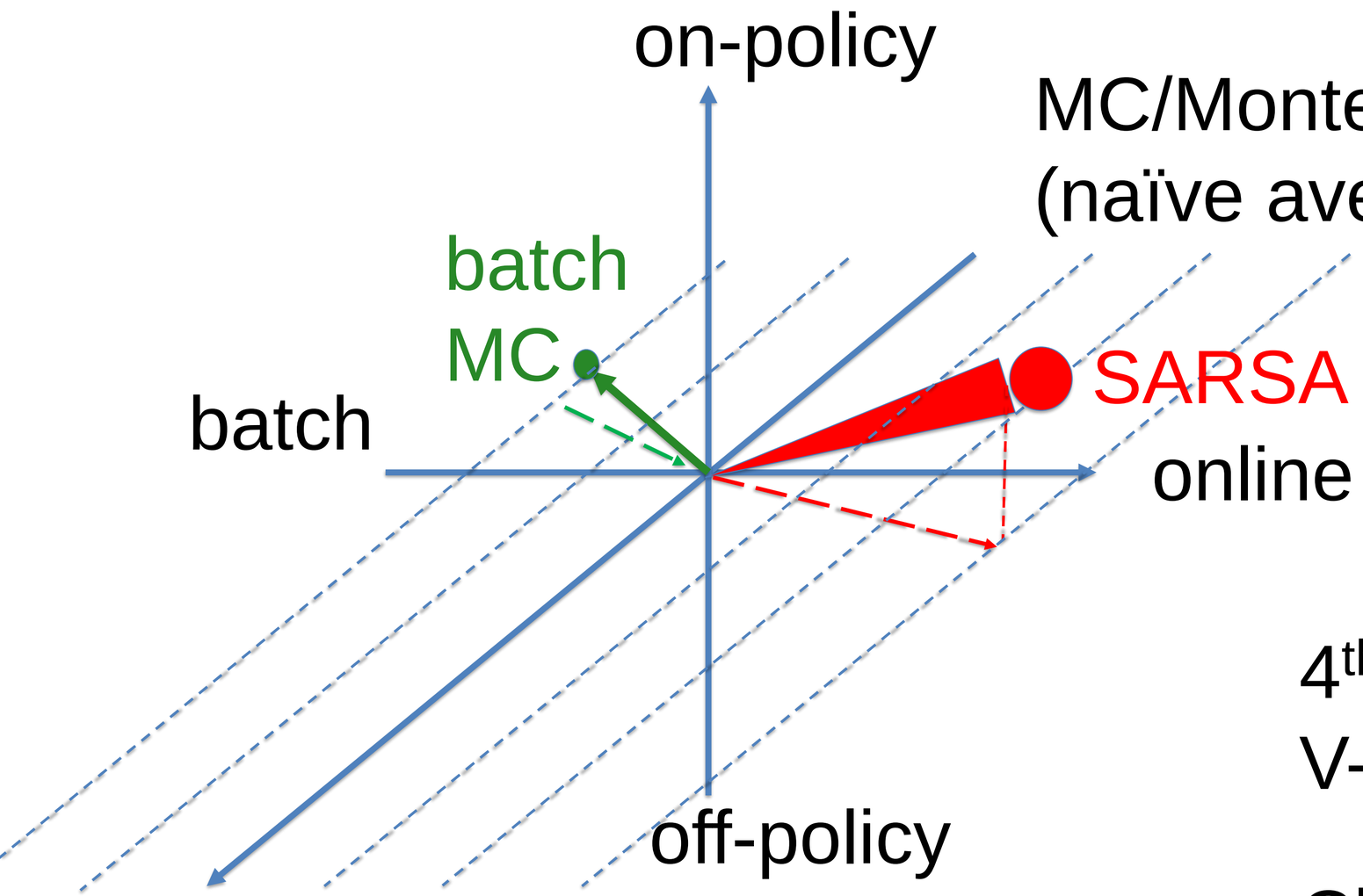
It is still an empirical estimation, since the rewards and the transitions need to be estimated from the episodes that have been played.

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

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) 'only looks back'
2) Expected SARSA (online, on-policy, 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

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)
The example on the next slide illustrates the following: 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.

The End

(previous slide, not shown in class)
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.

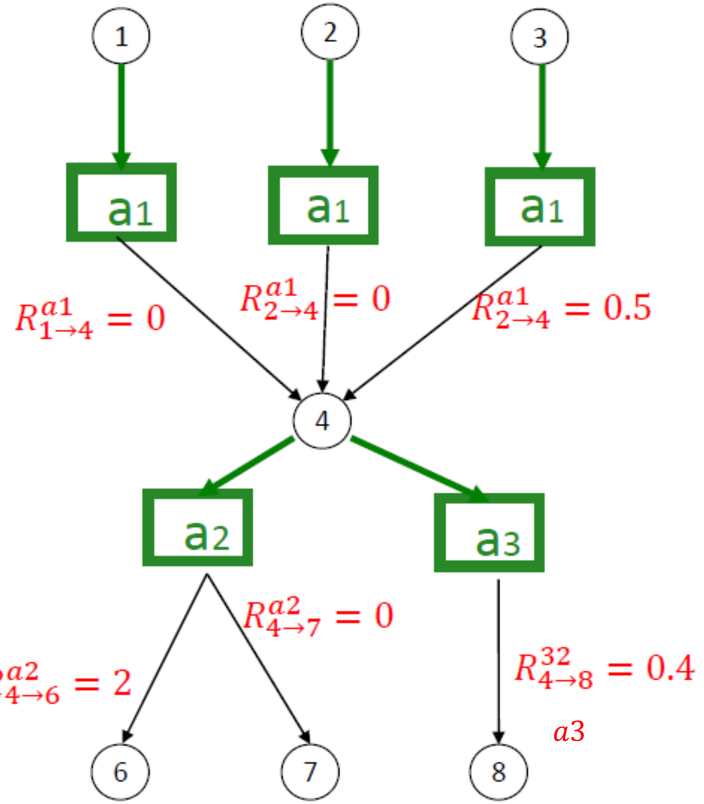
Episode 1: States 1-4-7 with action a2, Return=0

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?**

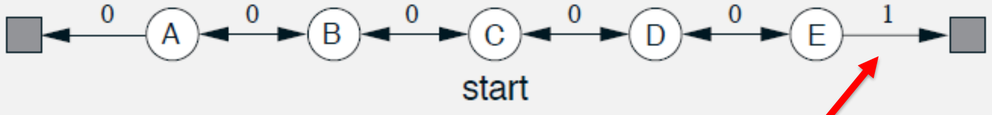
(i) Monte-Carlo: average over total accumulated reward for given (a,s)

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

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

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

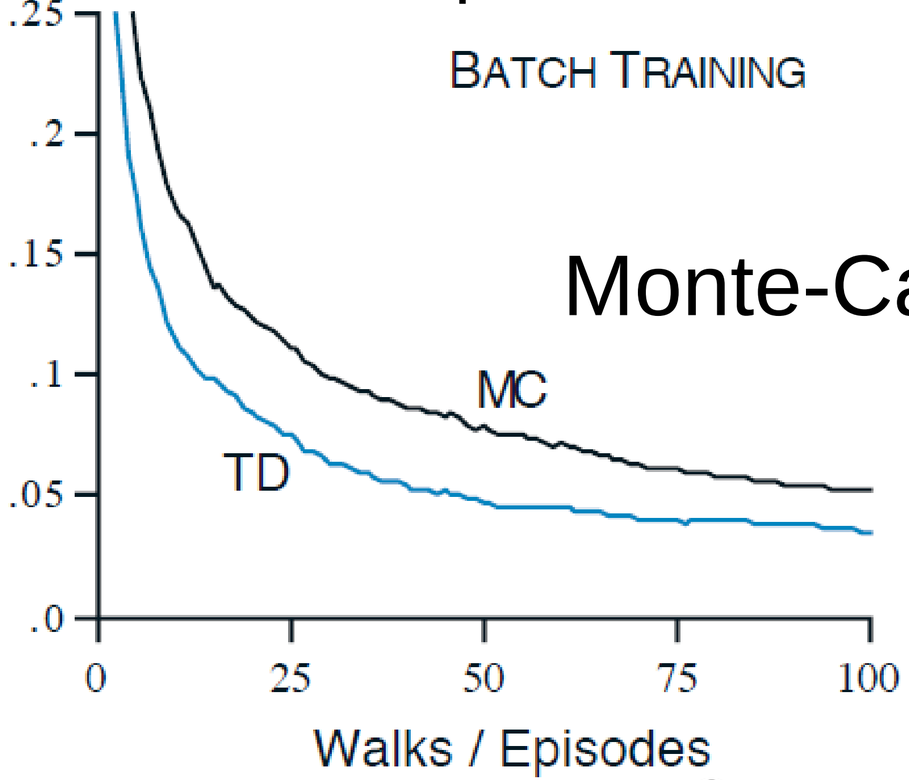
Example: 1d random walk



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

Teaching monitoring – monitoring of understanding

[ ] today, after the break, at least 60% of material was new to me.

[ ] after the break, I have the feeling that I have been able to follo
(at least) 80% of the lecture.

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

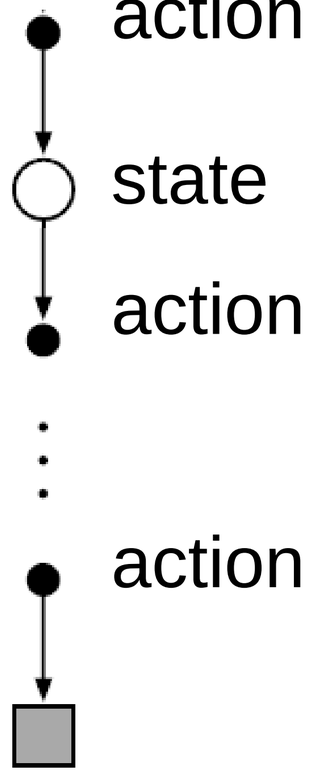
Combine epsilon-greedy policy with Monte-Carlo Q-estimates

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.

Now starts a Detour.

Artificial Neural Networks: RL1 Detour

Wulfram Gerstner
EPFL, Lausanne, Switzerland

Part 3b: **Detour to standard ML**
Expectation, Batch, and ONLINE rules

- Examples of Reward-based Learning
- Elements of Reinforcement Learning
- One-step horizon (bandit problems)
- **Expectation, batch, and online rules**

Video on https://lcnwww.epfl.ch/gerstner/VideoLecturesRL-Gerstner.html

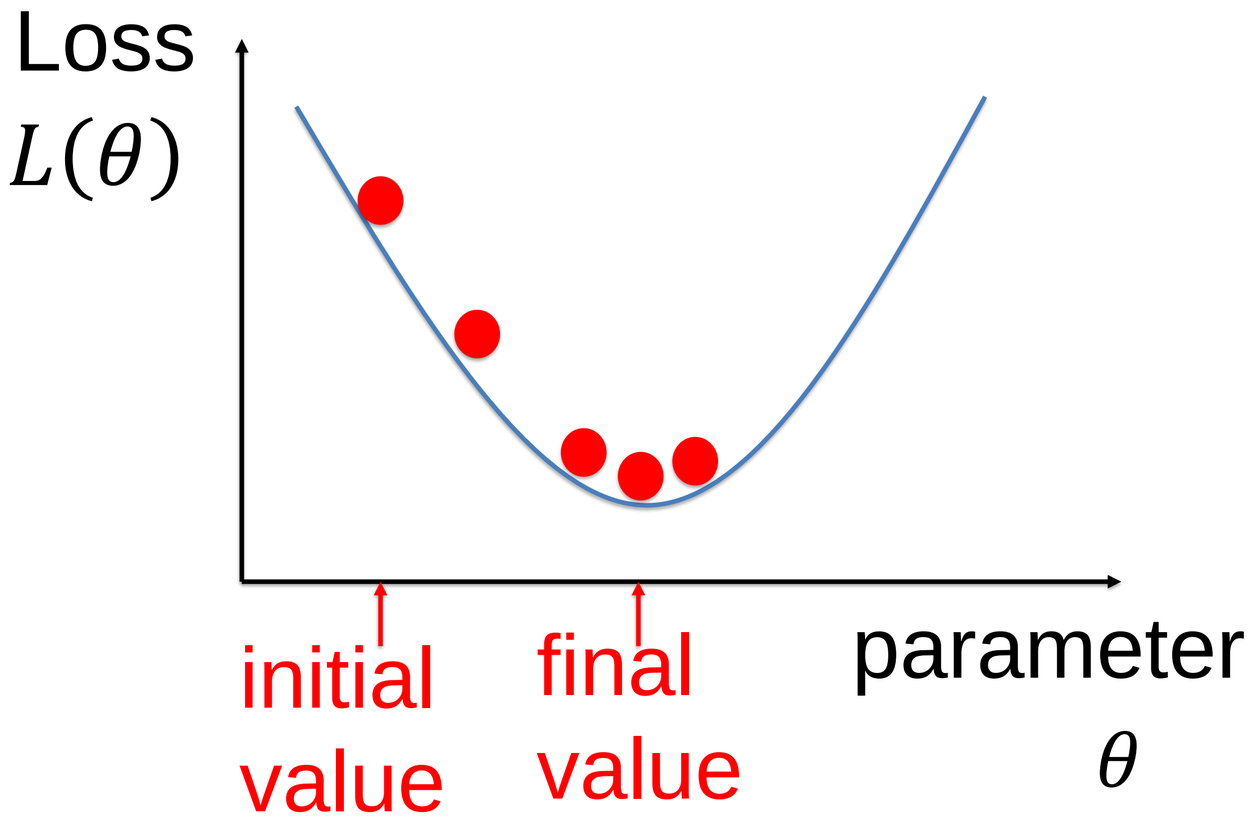
https://www.youtube.com/watch?v=BgqRW5rp8ac&list=PL7SYVykTNxXbu7EZTleyrJUNMwbg37WG3&index=7

Previous slide.

Last week we did a first calculation with expectations and I argued that this is ‘batch-like’. Since the type of calculation is important, but since this comparison caused many questions after class, I add a detour.

All the material in this part is in principle standard material in Machine Learning classes, even though I put the accent on aspects that are important for Reinforcement Learning.

Detour Machine Learning ReCap: Online, Batch, Expectation



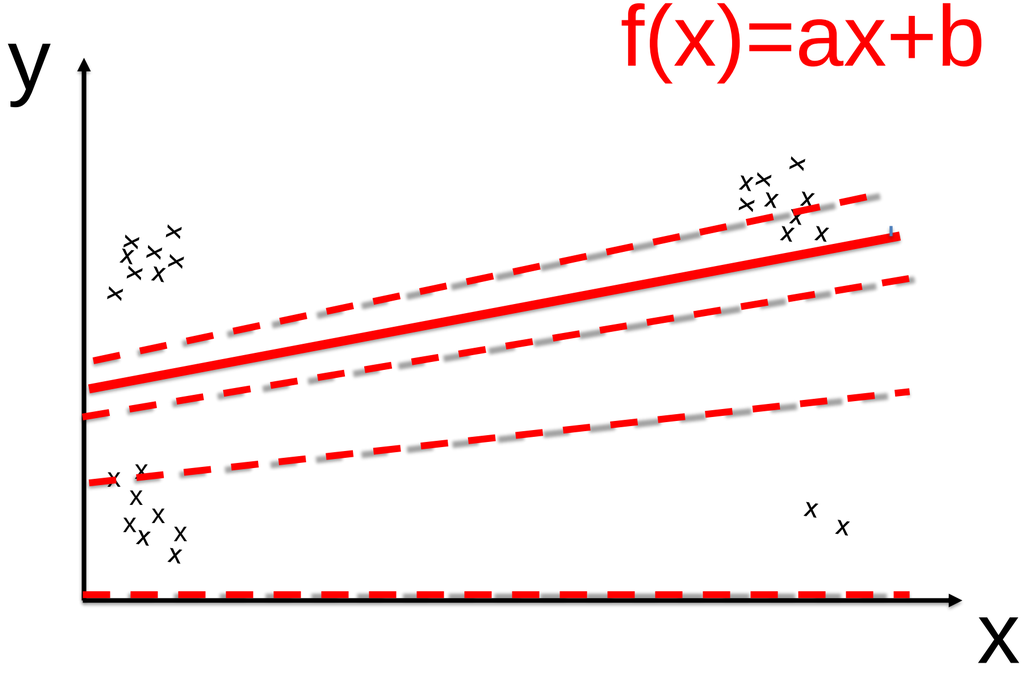
Your notes. (Review of gradient Descent)

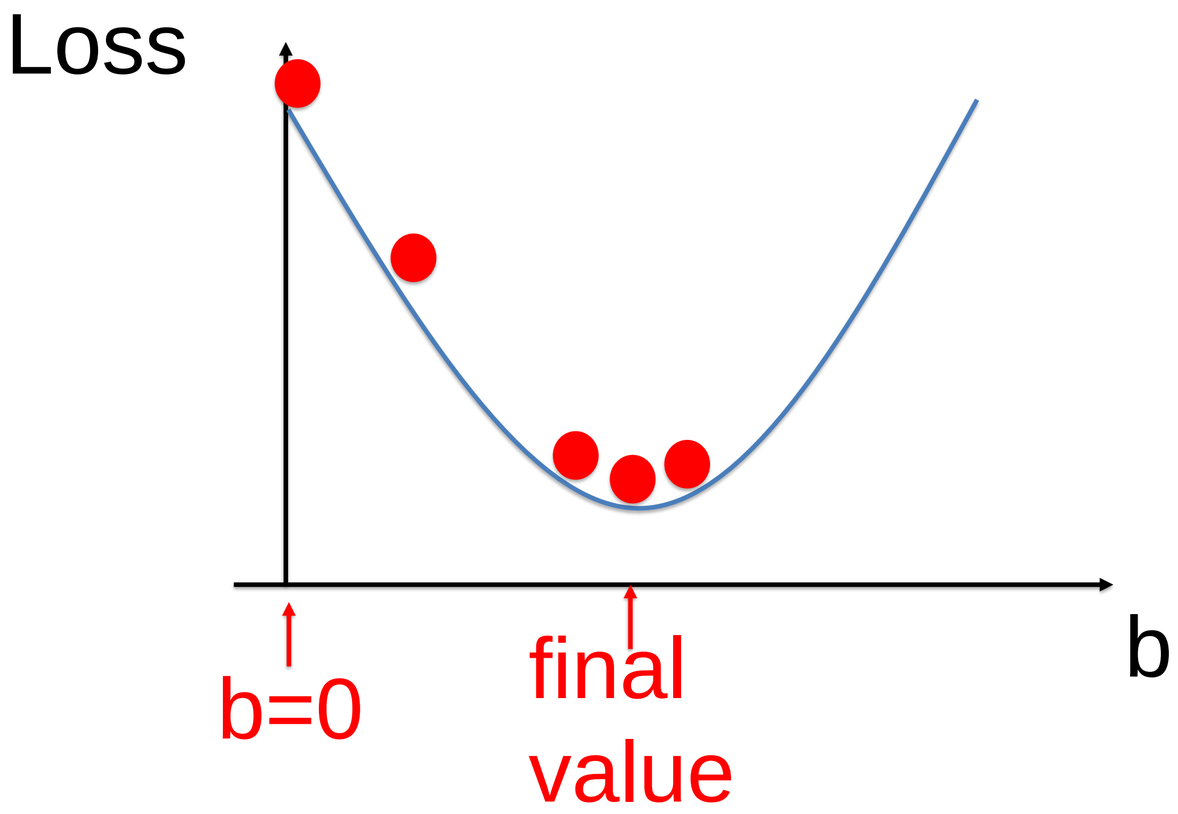
The set of parameters (also called parameter vector) is generically denoted by θ.

The loss function (error function) is denoted by capital L.

.

Detour Machine Learning ReCap: Online, Batch, Expectation





Gradient descent in ‘batch mode’

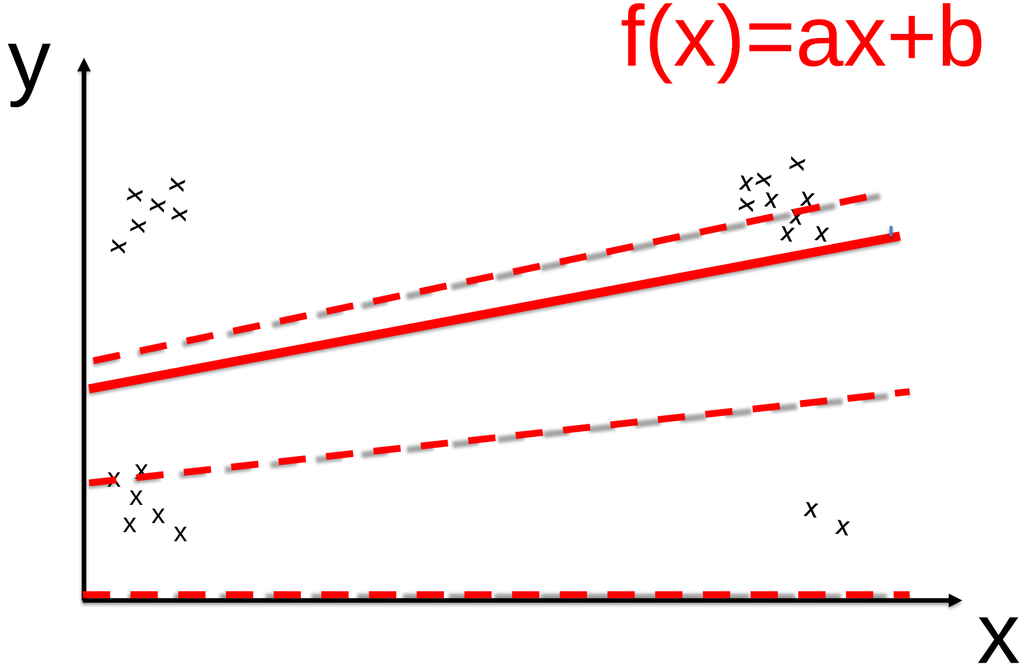
Your notes. (Review of gradient Descent)

The specific parameters of the linear function are a and b. For the drawing of the loss function, only one of the two parameters is plotted.

The aim is to fit a set of data points by the linear function.
Dashed red lines show intermediate update steps.

Note: For the learning rate I will often use the symbol α instead of η

Detour Machine Learning ReCap: Online, Batch, Expectation



$$f(x|\theta) = f(x|a,b) = ax + b$$

Parameters $\theta = (a, b)$

$$\Delta\theta = -\alpha \frac{d}{d\theta} L(\theta)$$

$$\Delta b = -\alpha \frac{\partial}{\partial b} L(a,b)$$

algo (iterative update)

iterate to convergence criteria

$$b^{old} \leftarrow b$$

$$b = b^{old} + \Delta b$$

analogously for a

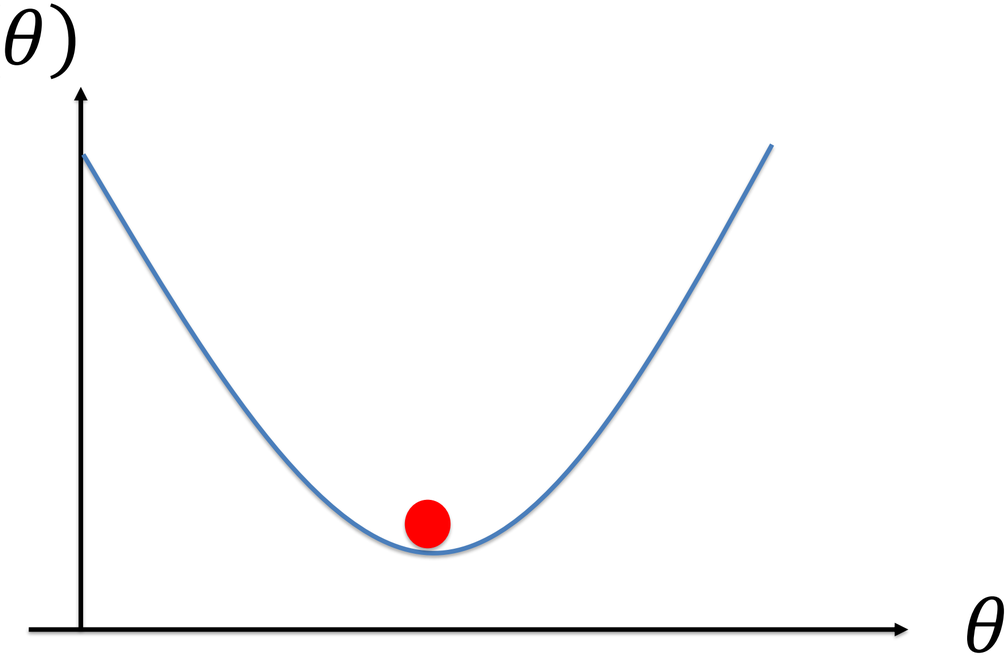
Your notes. (Review of gradient Descent)

The set of parameters (also called parameter vector) is generically denoted by θ.
And then the parameters are specified to be a and b.

Changes of parameters are calculated by gradient descent on the Loss function.

Detour Machine Learning ReCap: Online, Batch, Expectation

CONCLUSION 1, from rule (1):
if $\Delta\theta$=0 then
- **parameter θ no longer changes**
- **(local) minimum at θ^{old}**



Gradient is always evaluated at θ^{old}

$$\Delta\theta = -\alpha \frac{d}{d\theta} L(\theta)|_{\theta^{old}} \qquad (1)$$

Is this ‘batch mode’ or ‘online mode’?

Gradient descent in ‘batch mode’

algo (iterative update)

iterate to convergence criteria

$$\theta^{old} \leftarrow \theta$$

$$\theta = \theta^{old} + \Delta\theta$$

Your notes. (Review of gradient Descent)

The update rule tells us immediately that the update vanishes at a parameter value that has zero gradient. Only minima can be generically approached by gradient descent (not the maxima).

Detour Machine Learning ReCap: Online, Batch, Expectation

Loss function

$$L(\theta) = \mathbf{E}[l(\mathrm{f}(\mathrm{x}|\theta), \mathrm{y})]$$

$$L(\theta) = \frac{1}{N}\sum_{k}^{N}[l(\mathrm{f}(x_k|\theta), y_k)]$$

loss per data point

Gradient descent (batch)

$$\Delta\theta = -\alpha \frac{d}{d\theta} L(\theta)$$

Example: l =L2 loss, linear model

$$\mathrm{f}(x_k|\theta) = \mathrm{f}(x_k|a, b) = ax_k + b$$

$$L(\theta) = \frac{1}{N}\sum_{k}^{N}(ax_k + b - y_k)^2$$

$$\Delta\theta = -\alpha \boldsymbol{E}[\frac{d}{d\theta} l(f(x|\theta), y)]$$

$$\Delta\theta = -\alpha \frac{1}{N}\sum_{k}^{N}\frac{d}{d\theta}[l(\mathrm{f}(x_k|\theta), y_k)]$$

Your notes. (Review of gradient Descent)

The loss function is the expectation across all possible pairs (x,y) with the appropriate statistical weight. The loss per data point is denote by a small character l .

Often a large batch of N data points is taken instead. These N data points must be representative for the statistical distribution p(x,y).

The example shows the linear function that we considered earlier.

Detour Machine Learning ReCap: Online, Batch, Expectation

Loss function

$$L(\theta) = \mathbf{E}[l(\mathrm{f}(\mathrm{x}|\theta), \mathrm{y})]$$

$$L(\theta) = \int dxdy\, p(x,y)[l(\mathrm{f}(\mathrm{x}|\theta), \mathrm{y})]$$

$$L(\theta) = \frac{1}{N}\sum_{k}^{N}[l(\mathrm{f}(x_k|\theta), y_k)]$$

loss per data point

Gradient descent (batch)

$$\Delta\theta = -\alpha \frac{d}{d\theta} L(\theta)$$

$$\Delta\theta = -\alpha \int \boldsymbol{dxdy}\, \boldsymbol{p}(\boldsymbol{x},\boldsymbol{y})[\frac{d}{d\theta}\, l(f(x|\theta), y]$$

$$\Delta\theta = -\alpha\, \boldsymbol{E}[\frac{d}{d\theta}[l(f(x|\theta), y)]]$$

$$\Delta\theta = -\alpha \frac{1}{N}\sum_{k}^{N}\frac{d}{d\theta}[l(\mathrm{f}(x_k|\theta), y_k)]$$

Your notes. (Review of gradient Descent)

On the left:
The loss function is the expectation across all possible pairs (x,y) with the appropriate statistical weight. The gradient operation is linear and can be exchanged with the expectation (which is also a linear operation).

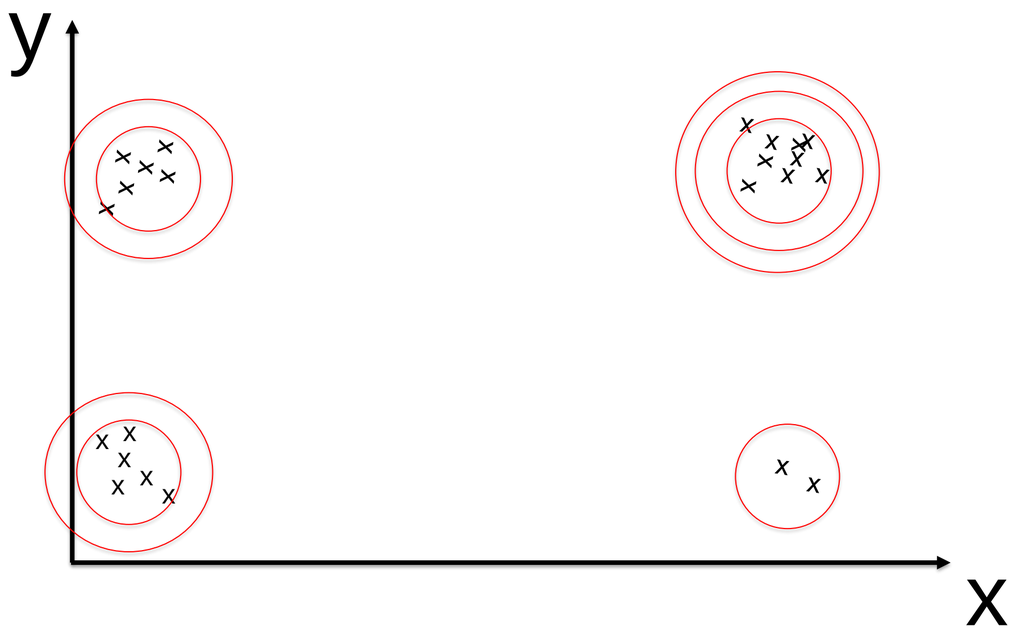
On the right:
The same calculation with a large batch of N data points.
The average of N in the gradient is analogous to the expectation (red boxes).

Conclusion: Expectation = Batch size N to infinity

$$\Delta\theta = -\alpha\, \boldsymbol{E}[\frac{d}{d\theta}\, l(f(x|\theta), y]$$

$$\Delta\theta = -\alpha \int \boldsymbol{dxdy}\, \boldsymbol{p}(\boldsymbol{x}, \boldsymbol{y})[\frac{d}{d\theta}\, l(f(x|\theta), y]$$

‘statistical formulation’ with ‘expectations’



choose N data points using the appropriate statistical weight

$$\Delta\theta = -\alpha\, \frac{\mathbf{1}}{\boldsymbol{N}} \sum_{\boldsymbol{k}}^{\boldsymbol{N}} \frac{d}{d\theta} [l(\mathrm{f}(x_k|\theta), y_k)]$$

$$\boldsymbol{E}[\dots] = \lim_{N\to\infty} \frac{\mathbf{1}}{\boldsymbol{N}} \sum_{\boldsymbol{k}}^{\boldsymbol{N}} [\dots]$$

Your notes. (Review of gradient Descent)

We said that the average of N in the gradient is ‘analogous’ to the expectation.
For the limit N to infinity batch and expectation are again identical.

The idea of the density p(x,y) is shown for the same example as before.

Conclusion: Expectation = Batch size N to infinity

$$\Delta\theta = -\alpha\, \boldsymbol{E}[\frac{d}{d\theta}\, l(f(x|\theta), y]$$

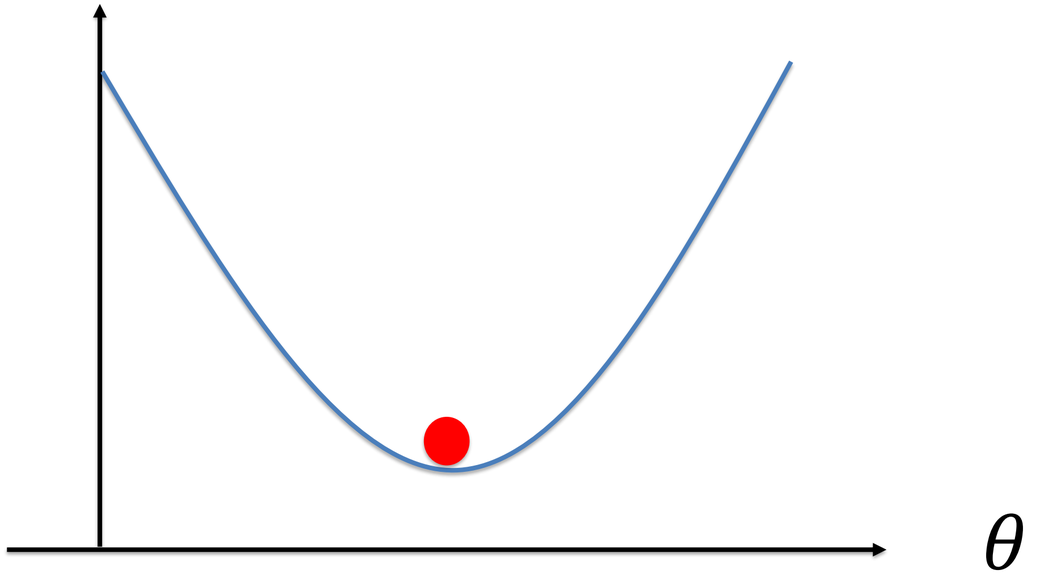
$$\boldsymbol{E}[\dots] = \lim_{N\to\infty} \frac{\mathbf{1}}{\boldsymbol{N}} \sum_{\boldsymbol{k}}^{\boldsymbol{N}} [\dots]$$

$$\Delta\theta = -\alpha\, \frac{\mathbf{1}}{\boldsymbol{N}} \sum_{\boldsymbol{k}}^{\boldsymbol{N}} \frac{d}{d\theta} [l(\mathrm{f}(x_k|\theta), y_k)]$$

CONCLUSION 1 from rule (1):

if $\Delta\theta$=0 (with N to infinity) then

- **$\boldsymbol{\theta}$ doesn't change**
- **(local) minimum at $\boldsymbol{\theta^{old}}$**
- **$\boldsymbol{\theta^{old}}$=$\boldsymbol{\theta^{optim}}$ in 'statistical' sense**



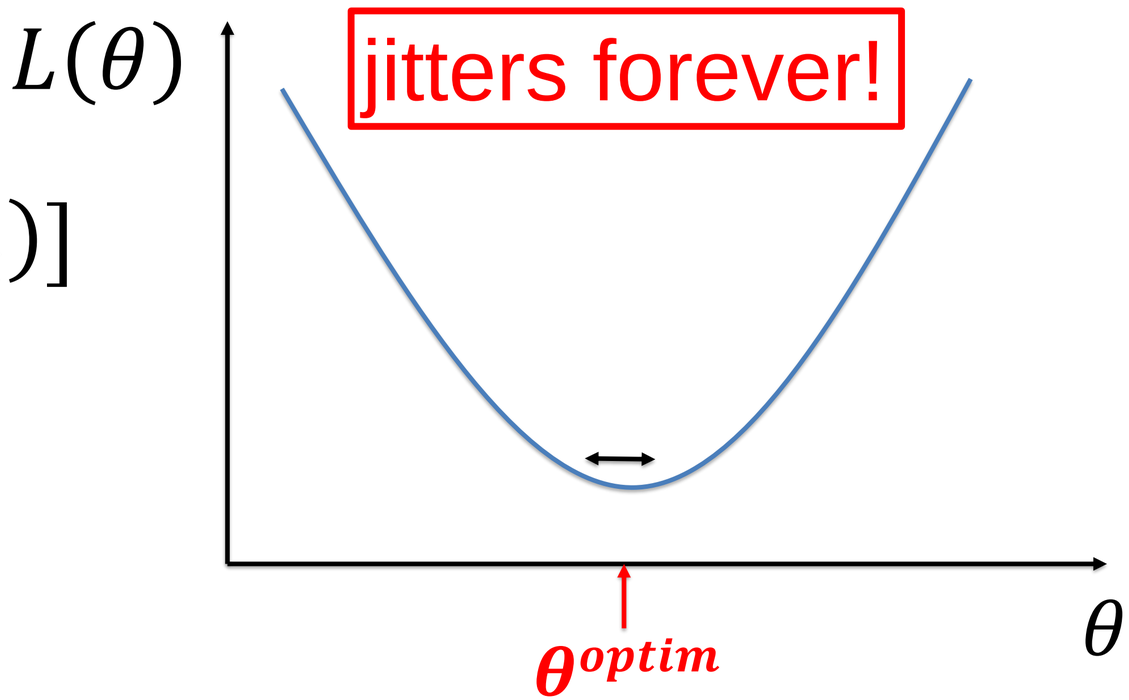
Your notes. (Review of gradient Descent)

A repetition of what we have seen before:
If the update step in the batch rule (N to infinity) vanishes, then we know that we are at a minimum of the loss function.

And this is equivalent to saying:
If the update step of the true loss function with the expectation sign vanishes, then we know that we are at a minimum of the loss function.

Detour Machine Learning ReCap: Batch versus 'Online'

$$\Delta\theta = -\alpha \frac{\mathbf{1}}{\boldsymbol{N}} \sum_{\boldsymbol{k}}^{\boldsymbol{N}} \frac{d}{d\theta} [l(\mathrm{f}(x_k|\theta), y_k)]$$



Online:

$$\Delta\theta = -\alpha \frac{d}{d\theta} [l(\mathrm{f}(x_k|\theta), y_k)]_{\theta=\theta^{old}}$$

Update after each data point

a.k.a. 'stochastic gradient descent'

algo (iterative update)

iterate to convergence criteria

$$\theta^{old} \leftarrow \theta$$

$$\theta = \theta^{old} + \Delta\theta$$

Your notes. (Review of gradient Descent)

An online rule means: drop the statistical averaging.

Here it means: drop the sum over data points.

As a result the parameter vector θ
can change after each data point!

And this is true even if we are already at the exact minimum of the true loss. The next data point might for example be an outlier and the parameter vector changes again. Therefore the gradient descent solution always jitters.

The size of the jitter depends on the learning rate (here called alpha).

Detour Machine Learning ReCap: Batch, 'Online', Expectation

Online:

$$\Delta\theta = -\alpha \frac{d}{d\theta}[l(\mathrm{f}(x_k|\theta), y_k)]_{\theta=\theta^{old}}$$

Update after each data point

Conclusion:

- Online update has jitter

BUT

- Expected update has no jitter

Expected Online Update ($\theta = \theta^{old}$ frozen):

$$E[\Delta\theta] = -\alpha E[\frac{d}{d\theta}[l(\mathrm{f}(x_k|\theta), y_k)]_{\theta=\theta^{old}}]$$

Conclusion:

Expected update of the online rule is identical to batch update with infinite data

Your notes. (Review of gradient Descent)

We can ask:
What would be the EXPECTATION of the update step.

Suppose we momentarily have the parameter $\theta = \theta^{old}$.
Then we ask what is the EXPECTED change at this location.

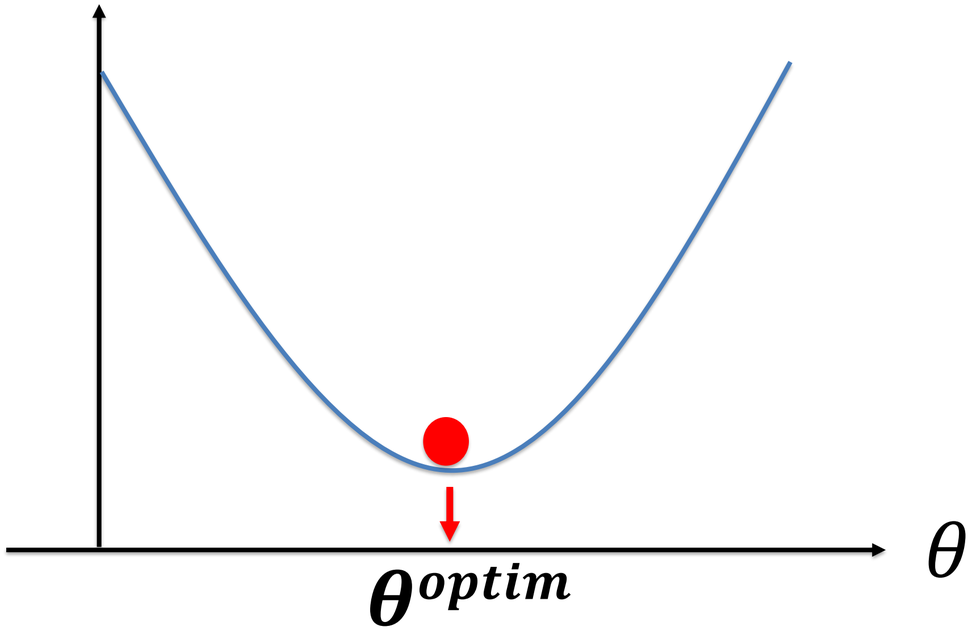
Comparison with the batch rule shows that the expected update of the online rule is identical to batch update with infinite data evaluated at $\theta = \theta^{old}$.

THIS IMPLIES:
If by chance $\boldsymbol{\theta = \theta^{old}}$ is the exact minimum, the expected update is zero; but the ACTUAL update can be nonzero!

This is also summarized in the next slide and the quiz.

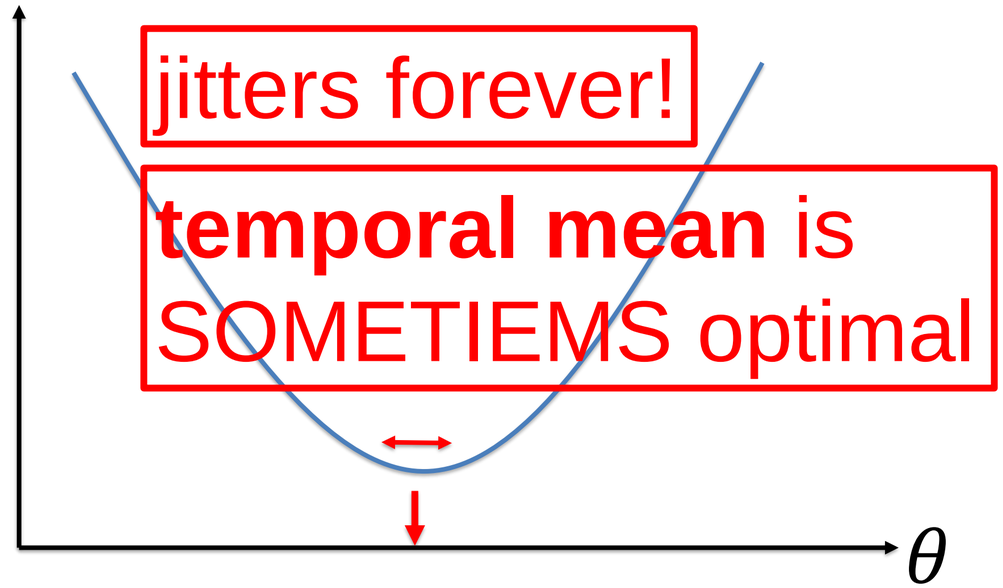
Batch rule with N to infinity

CONCLUSION 1 from rule (1):
if $\Delta\theta$=0 (with N to infinity) **then**
- $\boldsymbol{\theta}$ **doesn't change**
- **(local) minimum at** $\boldsymbol{\theta^{old}}$
- $\boldsymbol{\theta^{old}}$**=**$\boldsymbol{\theta^{optim}}$



Online Rule

CONCLUSION 2:
- $\boldsymbol{\theta}$ **jitters forever. BUT:**
- **if by chance** $\boldsymbol{\theta^{old}}$ **such that** $E(\Delta\theta)$=0 **then**
(local) minimum at $\boldsymbol{\theta^{old}} = \boldsymbol{\theta^{optim}}$



Previous slide/next slide. Summary slides:

If the expected update is zero [i.e., $\boldsymbol{E}(\Delta\theta)$=0] for a given set of parameter $\theta = \boldsymbol{\theta^{old}}$, then θ is a locally optimal parameter, **even for the online rule**:

$$\theta = \boldsymbol{\theta^{old}} = \boldsymbol{\theta^{optim}}$$

There is no statement how we would find this parameter $\theta = \boldsymbol{\theta^{optim}}$

A completely different statement concerns the **mean of the jittering parameter** $\boldsymbol{\theta}$.

If the **update steps are symmetric**, then the mean of the parameter θ is the optimal one: $\langle\theta\rangle = \boldsymbol{\theta^{optim}}$

However, if the update steps are asymmetric, then the mean $\langle\theta\rangle$ of the parameter θ is shifted compared to the optimal one (next slide).

For gradient descent on a loss function we recognize the asymmetry in loss curve.

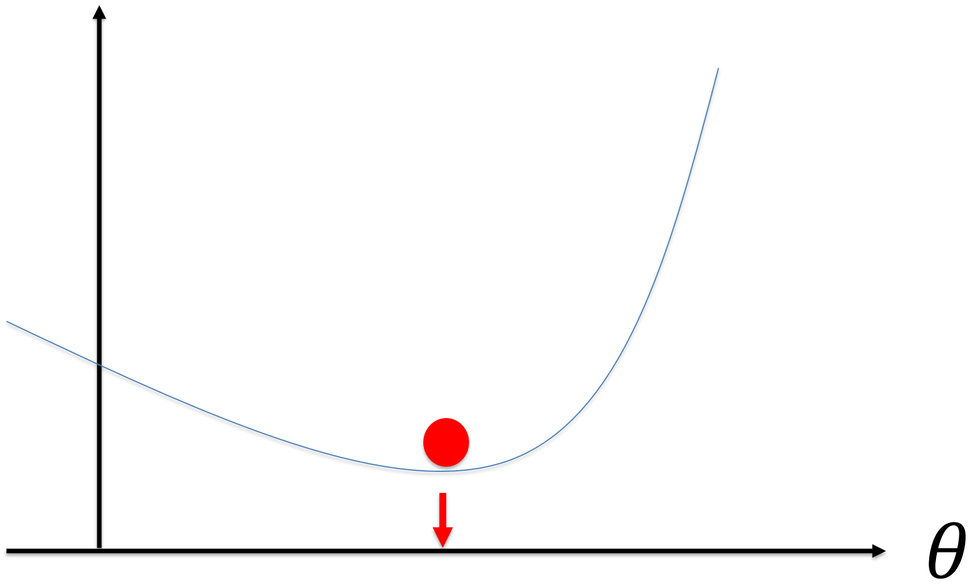
However (even in cases where we do not have a loss function) what really counts is whether the update steps are symmetric or not:

Suppose the current parameter is $\theta = \boldsymbol{\theta^{optim}} + \epsilon$ where is small, i.e. close to the optimum

Symmetry is guaranteed if update steps are linear $\Delta\theta = \alpha\epsilon$ with small constant α.

Batch rule with N to infinity

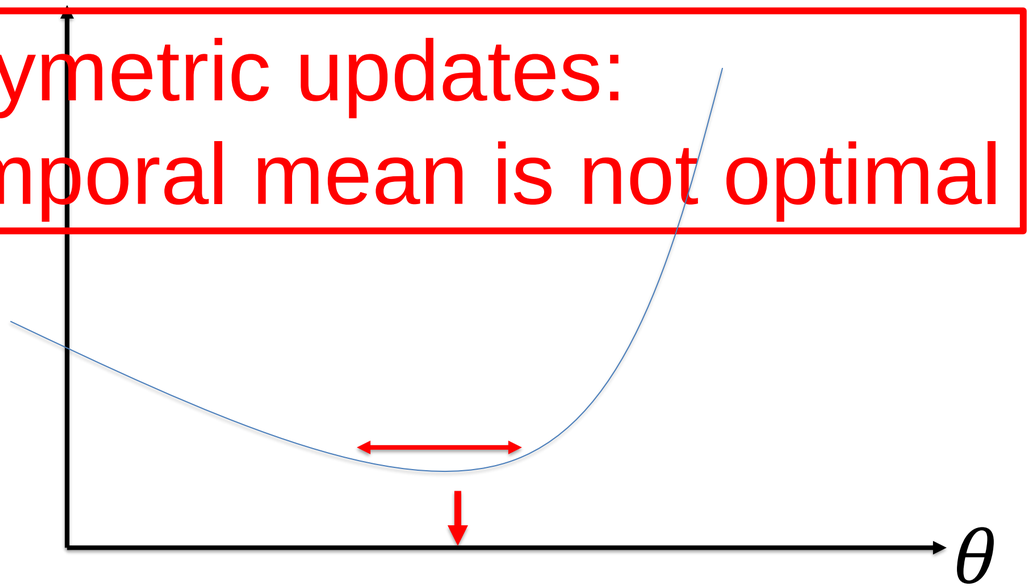
CONCLUSION 1 from rule (1):
if $\Delta\theta$=0 (with N to infinity) **then**
- $\boldsymbol{\theta}$ **doesn't change**
- **(local) minimum at** $\boldsymbol{\theta^{old}}$
- $\boldsymbol{\theta^{old}}$**=**$\boldsymbol{\theta^{optim}}$



Online Rule

CONCLUSION 2:
- $\boldsymbol{\theta}$ **jitters forever. BUT:**
- if by chance $\boldsymbol{\theta^{old}}$ **such that**
$E(\Delta\theta)$=0 **then**
(local) minimum at $\boldsymbol{\theta^{old}} = \boldsymbol{\theta^{optim}}$

Asymetric updates:
temporal mean is not optimal



Quiz: Expectation, Batch, Online (Recap of ML)

[ ] With a **batch rule** and small learning rate, I sometimes reach a local minimum without remaining parameter jitter.

[ ] With a **batch rule at a local minimum** I never have any remaining parameter jitter

[ ] With an **online rule at a local minimum** I never have any remaining parameter jitter

[ ] With an **online rule** at a local minimum the **expectation of the online update step vanishes.**

[ ] The **expectation of the online update step** is equivalent to a very large **batch** (N to infinity)

[ ] With an online rule jittering round the minimum, the temporal mean is guaranteed to be at the location of the minimum

Teaching monitoring – monitoring of understanding

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

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

End of Detour:
Apply to Q-values in the Bandit problem.

ML: parameters are called θ

Function fitting: parameters are a and b

Bandit problem: parameters are Q(s,a)