Reinforcement Learning Lecture 4
Policy Gradient Methods

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Part 1: Review of TD methods

Objectives of this lecture:

- **basic idea of policy gradient: learn actions, not Q-values**
- **log-likelihood trick: getting the correct statistical weight**
- **policy gradient algorithms**
- **why subtract the mean reward?**
- **REINFOCE algorithm**

Reading for this week:

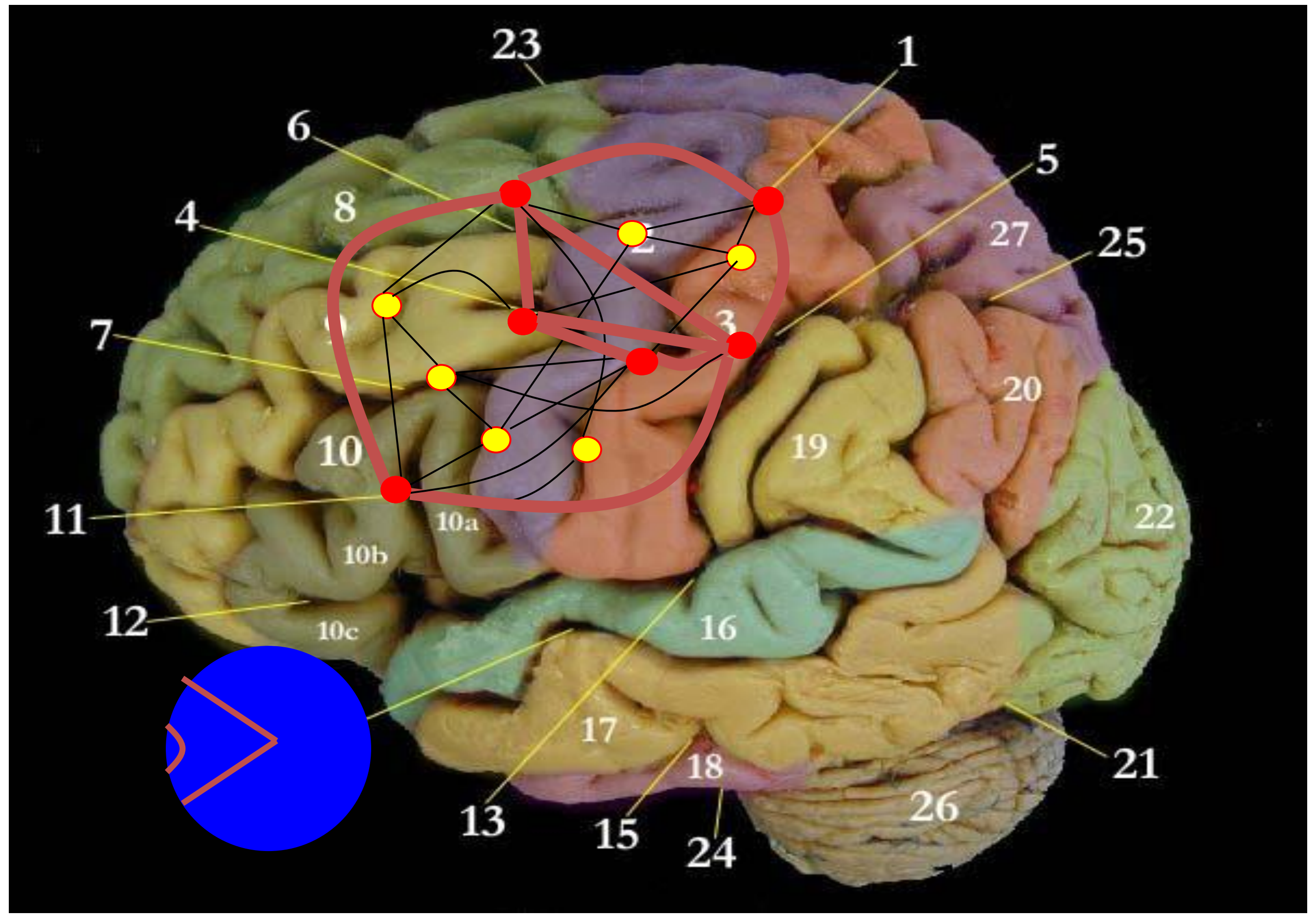
Sutton and Barto, Reinforcement Learning (MIT Press, 2nd edition 2018, also online)

Chapter: 13.1-13.5

Background reading: none

Review: Artificial Neural Networks for action learning

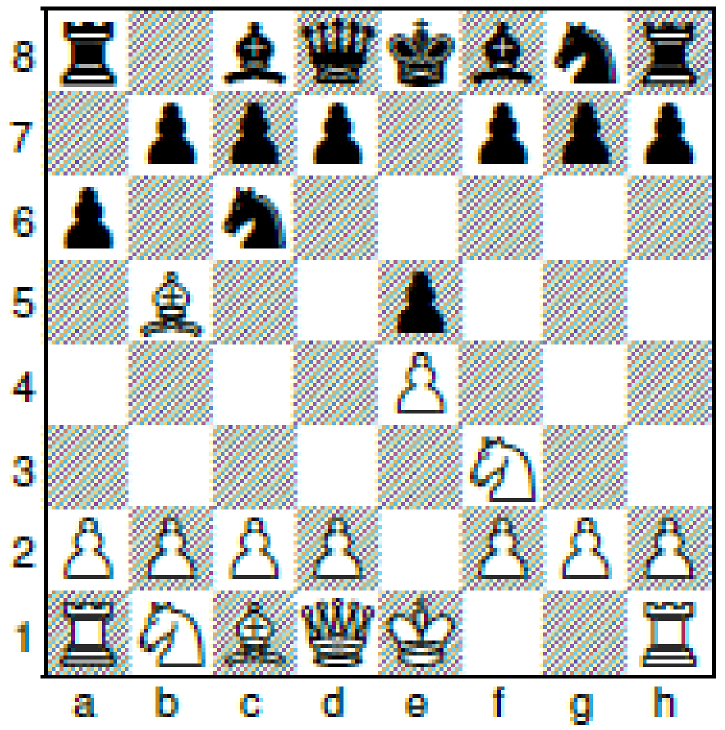
Learning without labeled data
Learning by ‘reward’



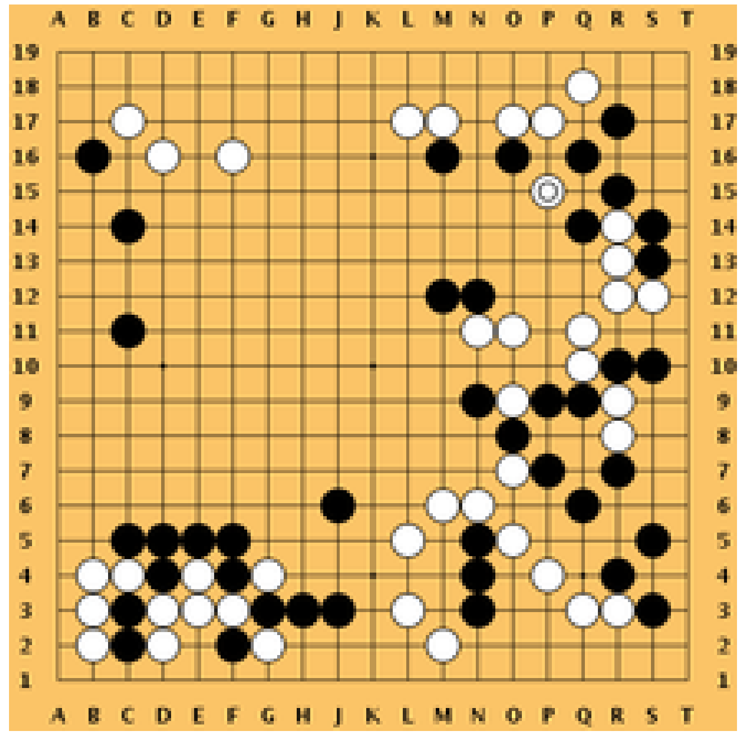
BUT: Reward is rare:
‘sparse feedback’ after
a long action sequence

First steps toward Deep reinforcement learning

Chess



Go



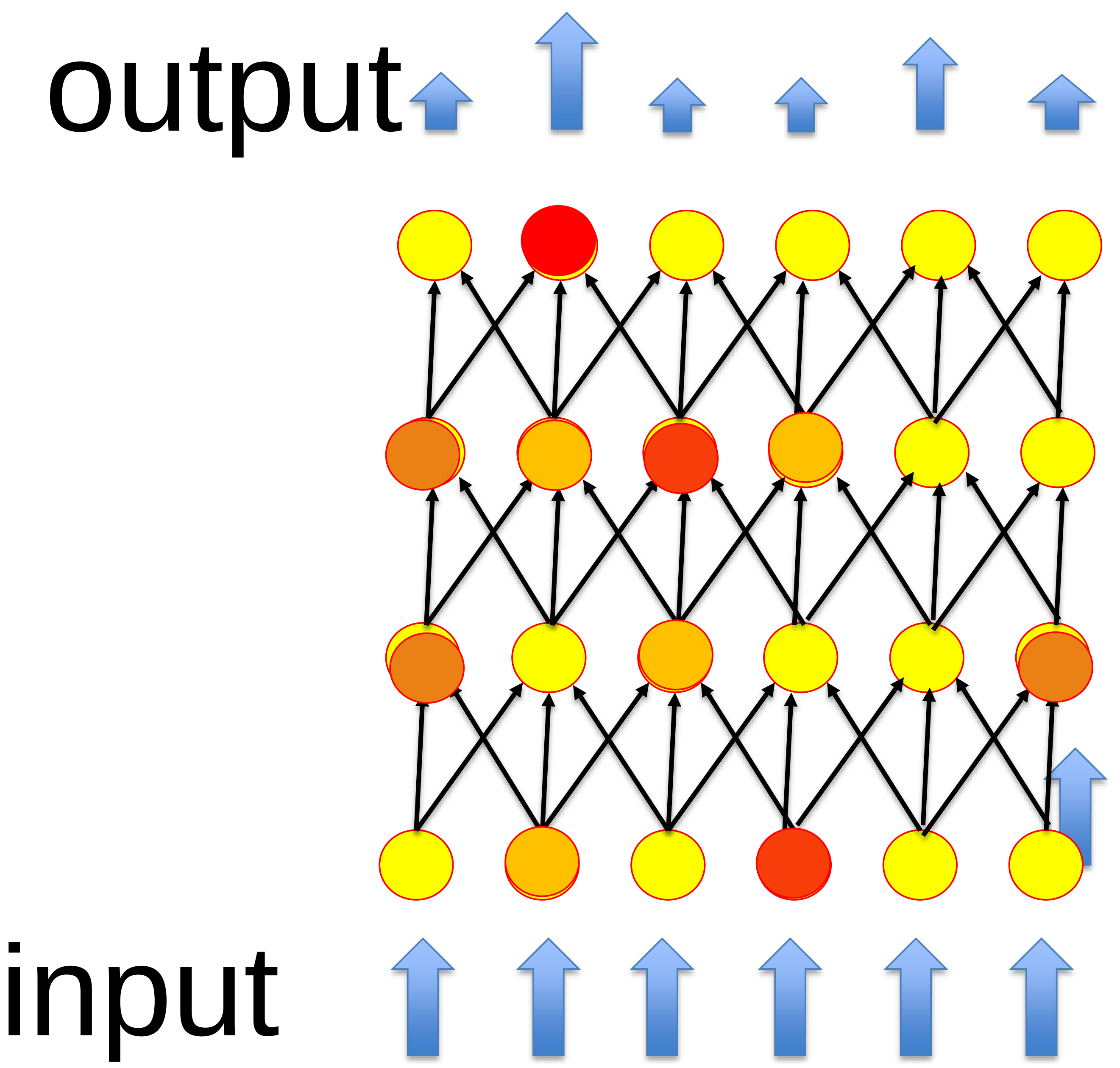
Artificial neural network (*AlphaZero*) discovers different strategies by playing against itself.

In Go, it beats Lee Sedol

Review: Backprop for deep Q-learning

(Backprop = gradient descent rule in multilayer networks)

action and Q-values:



Neural network parameterizes Q-values as a function of continuous state s.

- One output for each action a.

- Learn weights by semi-gradient on loss function

Error function for SARSA

$$E = 0.5\,[\,r + \gamma\, Q(s',a') - Q(s,a)\,]^2$$

(previous slide)

Suppose that each output corresponds to one action (e.g. one type of move in chess). Parameters are now the weights of the artificial neural network.

Actions are chosen, for example, by softmax on the Q-values in the output.

Weights are learned by playing against itself – doing gradient descent on an error function E.

Last week we finished by stating the error function:

$$E = 0.5\, [\, r + \gamma\, Q(s', a') - Q(s,a)\,]^2$$

This error function will depend on the weights w (since $Q(s,a)$ depends on w).
We can change the weights by gradient descent on the error function. This leads to the Backpropagation algorithm of 'Deep learning' (will be discussed next week).

Summary: Deep Neural Network for TD learning

In all TD learning methods
(includes n-step SARSA, Q-learning, TD(λ))

- V-values OR Q-values are the central quantities

- actions are taken with softmax, greedy, or
  epsilon-greedy policy **derived from Q-values/V-values**

(previous slide)
In the previous two weeks, we have seen many different versions of TD learning. This includes SARSA and Q-learning, TD learning, with eligibility traces (decay factor lambda<1) or without, or n-step V-learning.

In all of these algorithms the V-values or Q-values are the central quantities. We first learn the V-values (or Q-values) and then the policy is based on these values.

TD learning versus Policy Gradient

Aim of this lecture:
- learn actions directly
- no need for Q-value estimation

→ Policy Gradient

→ A glimpse into Deep Reinforcement Learning

(previous slide)
The question for today is: Can we learn directly the policy – without taking the detour via the Q-values or V-values? The answer is yes and leads to a family of methods that are called ‘policy gradient’.

A secondary aim is to give a preparation of modern developments in Deep Reinforcement Learning.

Reinforcement Learning Lecture 4
Policy Gradient Methods

Wulfram Gerstner
EPFL, Lausanne, Switzerland

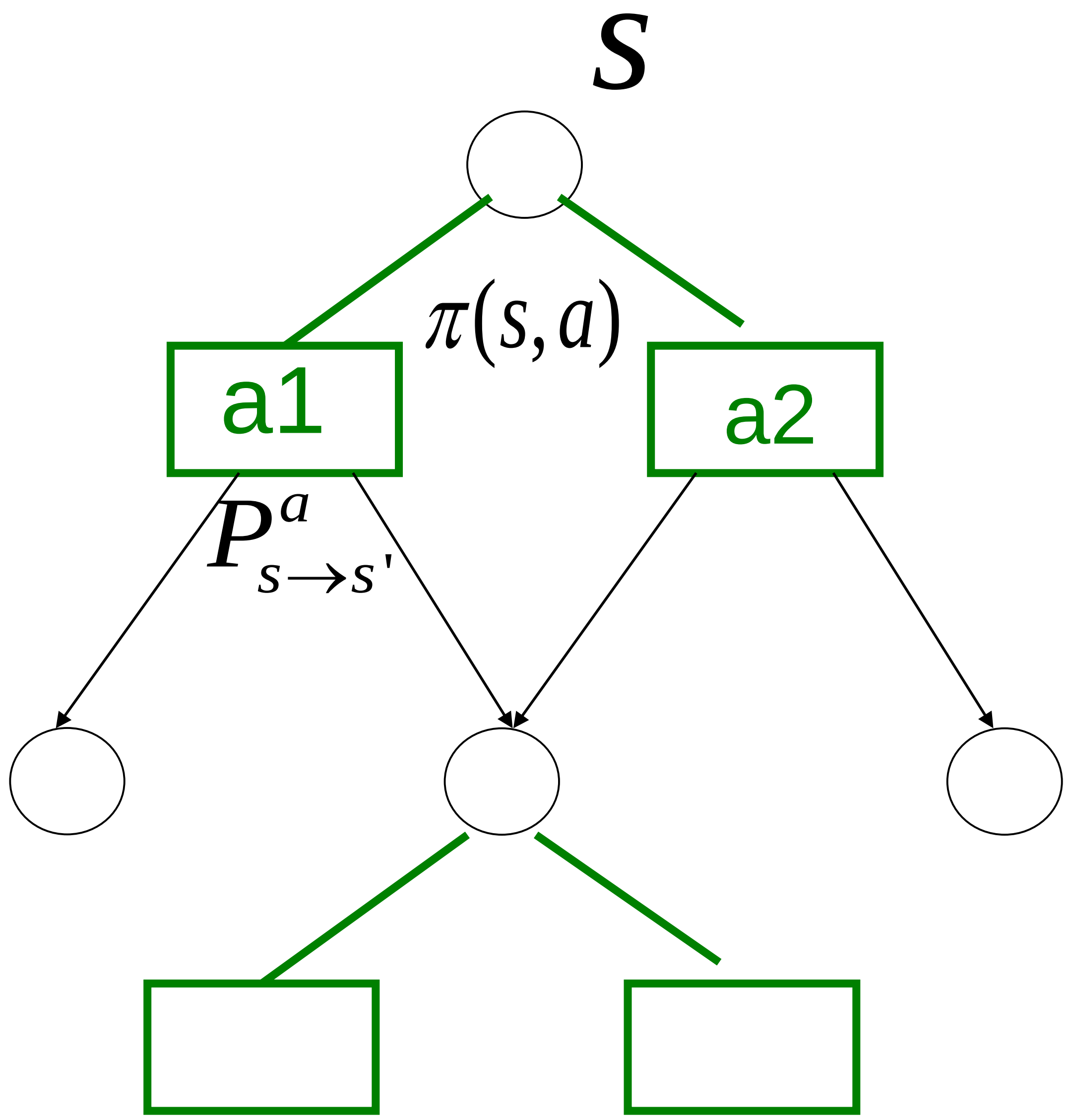
Part 2: Basic idea of policy gradient

1. First steps toward deep reinforcement learning
2. **Basic idea of policy gradient**

(previous slide)
Let us start with the reasons to work with policy gradients rather than V-values or Q-values.

Disadvantages of Q-learning, SARSA, or TD-learning

- For **continuous states**, **function approximation** is necessary (which is potentially unstable).
- Even in fully observable (Markov) settings, off-policy TD algorithms (e.g. Q-learning) can diverge using **function approximation**.
- In **partially observable** environments (non-Markov), TD algorithms are problematic
- **Continuous actions** are difficult to represent using TD methods.



World is not a Markov Process
World is not fully observable
World is not tabular (not discrete states)

(previous slide)
Q-values and V-values work best in an environment that has Markov properties, in particular discrete, distinguishable states, and transition probabilities between these states. Building V-values (or Q-values) then means building a table of these states (or state-action pairs).

But the world is not Markovian; however, if we use the Markovian assumptions in an environment where this is not true, then there is no guarantee that these algorithms converge.

Policy Gradient methods: basic idea

- Forget Q-values
- Optimize directly the reward
- Associate actions with stimuli stochastically

Table in Q-learning:
(state,action) → Q

	a_1	a_2	a_3
s_1	$Q(s_1,a_1)$	$Q(s_1,a_2)$	
s_2	$Q(s_2,a_1)$		
s_3			
s_4			

Table in Policy gradient:
state → Prob(action|state)

	a_1	a_2	a_3
s_1	0.1	0.8	0.1
s_2	0.75	0.1	0.15
s_3	0.01	0.02	0.97
s_4	0.5	0.5	0.0

(previous slide)
Difference between Q-learning and policy gradient:

In Q- learning you build a table of Q(s,a) for each state-action pair.
Then you derive the policy from this (e.g., epsilon-greedy).

In policy gradient you learn directly the probability of taking action a in state s.
Since these are probabilities, they must sum to one.

Policy Gradient methods: basic idea

- Forget Q-values
- Optimize directly the reward
- Associate actions with stimuli using a stochastic policy
- **Change parameters so as to maximize rewards**

Table in Policy gradient: $\pi(a|s,\theta)$

state → Prob(action|state,parameters)

	a_1	a_2	a_3
s_1	0.1	0.8	0.1
s_2	0.75	0.1	0.15
s_3	0.01	0.02	0.97
s_4	0.5	0.5	0.0

stochastic policy

$\pi(a|s,\theta)$

parameter

(previous slide)
The basic ideas are now that
(i) these probabilities will depend on a set of parameters θ
(ii) these probabilities can be directly interpreted as the policy $\pi(a|s,\theta)$

Note sometimes the policy is written with parameters suppressed, or parameters added as an index:

$$\pi(a|s,\theta) \rightarrow \pi_\theta(a|s)$$

Summary: idea of Policy Gradient

1. stochastic policy

$\pi(a|s,\theta)$ Prob(action|state,parameters)

parameter

2. Change parameters so as to maximize rewards

3. Different from TD learning: No need for Q-values or V-values

Summary.

Reinforcement Learning Lecture 4
Policy Gradient Methods

Wulfram Gerstner

EPFL, Lausanne, Switzerland

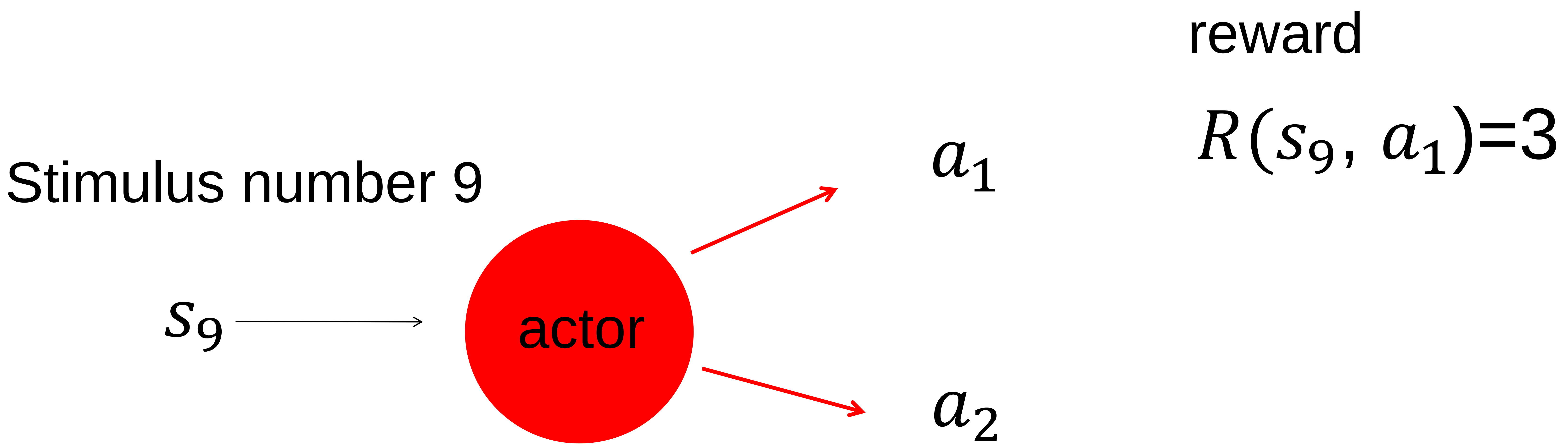
Part 3: Policy gradient with 1-step horizon

1. First steps toward deep reinforcement learning
2. Basic idea of policy gradient
3. **Example: 1-step horizon**

(previous slide)
To make these abstract notions concrete, we start with a simple example.

Policy Gradient methods: 1-step horizon

- Associate actions with stimuli
- Optimize directly the reward



(previous slide)

As always in reinforcement learning, the goal is to optimize rewards. We start with a one-step horizon and a binary choice.

For each stimulus (here stimulus number 9) there is the choice of two actions.

For example if the agent takes action a_1 in response to stimulus s_9, it receives a reward of value 3.

Policy Gradient methods: 1-step horizon

stimulus=state=input vector

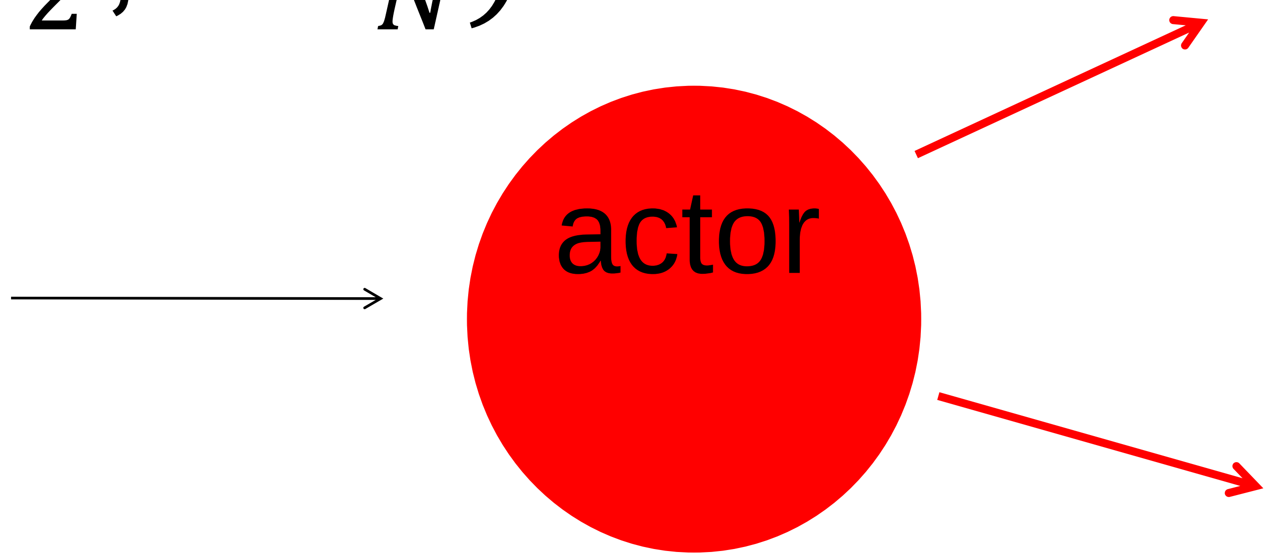
Stimulus number 9 is a vector

$$\vec{x}^9 = (x_1^9, x_2^9, \ldots x_N^9)^T$$

$R(\vec{x}^9,\, a_1)$=3

a_1

$s_9 = \vec{x}^9$



a_2

(previous slide)
We model the stimulus s as in input vector (input pattern $\vec{x}$).

The actor can take two possible actions.

Policy Gradient methods: 1-step horizon

Aim: change weights of neuron

→Maximize expected reward!

$$\langle R \rangle = \sum_x \sum_{y=\{0,1\}} \pi(y|\vec{x})\, p(\vec{x}) R(y,\vec{x})$$

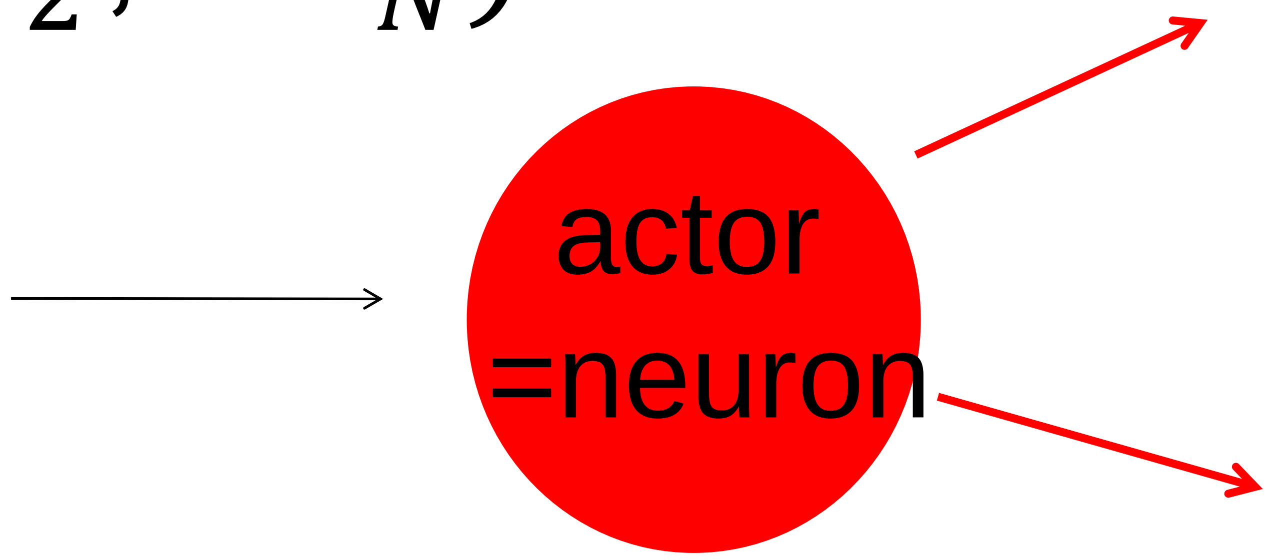
Stimulus number 9

$$\vec{x}^9 = (x_1^9, x_2^9, \ldots x_N^9)^T$$

Output of neuron

$$a_1 \rightarrow y = 1$$

$$s_9 = \vec{x}^9$$



$$a_2 \rightarrow y = 0$$

Choice of actions

policy: $\pi(a1|\vec{x},\vec{w}) = prob(y = 1|\vec{x},\vec{w}) = g(\sum_k^N w_k x_k)$

(previous slide)
We now model the policy as a single sigmoidal neuron with transfer function g and weight vector $\vec{w}$.

It is convenient to introduce a binary output variable: y takes the value of 1 if action a1 is taken and zero otherwise.

The question now is:
How should we adapt the weight vector so that (averaged over all possible stimuli) the reward is maximal?

Define the mean reward as
$$\langle R \rangle = \sum_{x} \sum_{y=\{0,1\}} \pi(y|\vec{x})\, p(\vec{x}) R(y, \vec{x})$$

and use $\pi(y = 1|\vec{x}) = \; g(\sum_k^N w_k x_k)$

Exercise 1: maximize expected reward

Exercise 1 now (12min)
Next Lecture at 11h36

Exercise 1. (in Class): Single neuron as an actor

Assume an agent with binary actions $Y \in \{0, 1\}$. Action $y = 1$ is taken with a probability $\pi(Y = 1|\vec{x}; \vec{w}) = g(\vec{w} \cdot \vec{x})$, where $\vec{w}$ are a set of weights and $\vec{x}$ is the input signal that contains the state information. The function g is monotonically increasing and limited by the bounds $0 \leq g \leq 1$.

For each action, the agent receives a reward $R(Y, \vec{x})$.

a. Calculate the gradient of the mean reward $\langle R \rangle = \sum_{Y,\vec{x}} R(Y, \vec{x})\pi(Y|\vec{x}; \vec{w})P(\vec{x})$ with respect to the weight w_j.

Hint: Insert the policy $\pi(Y = 1|\vec{x}; \vec{w}) = g(\sum_k w_k x_k)$ and $\pi(Y = 0|\vec{x}; \vec{w}) = 1 - g(\sum_k w_k x_k)$. Then take the gradient.

b. The rule derived in (a) is a batch rule. Can you transform this into an 'online rule'?

Hint: Pay attention to the following question: what is the condition that we can simply 'drop the summation signs'?

online = 'stochastic gradient ascent'

$$\vec{x}^9 = (x_1^9, x_2^9, \ldots x_N^9)^T$$

$\vec{x}^9 \longrightarrow$ actor =neuron

$$a_1 \rightarrow y = 1$$

$$\pi(y = 1|\vec{x}, \vec{w}) = g(\sum_k^N w_k x_k)$$

$$a_2 \rightarrow y = 0$$

(your calculations)

Policy Gradient methods: 1-step horizon

blackboard1

$$\langle R \rangle = \sum_{x} \sum_{y=\{0,1\}} \pi(y|\vec{x})\, p(\vec{x}) R(y,\vec{x})$$

reward

$R(y,\vec{x})$

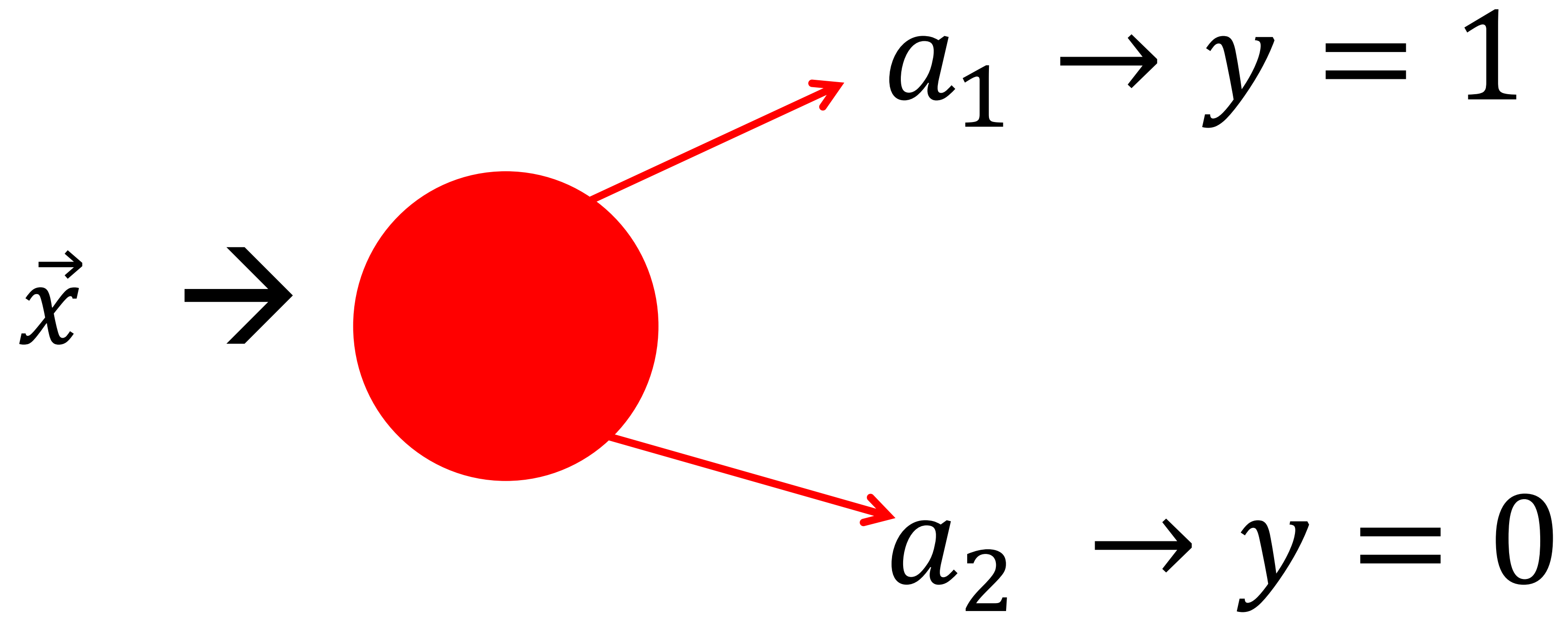
policy

$$\pi(y=1|\vec{x},\vec{w}) \;= g(\vec{w}^{\top}\vec{x})$$

$$\pi(y=0|\vec{x},\vec{w}) \;= 1 - g(\vec{w}^{\top}\vec{x})$$

policy

$$\pi(y=1|\vec{x},\vec{w}) \;= g\Big(\sum_{k}^{N} w_k x_k\Big)$$



(your calculations)

From Batch rule to Online rule (pedestrian approach)

$$\langle R \rangle = \sum_{x} \sum_{y=\{0,1\}} \pi(y|\vec{x})\, p(\vec{x}) R(y, \vec{x}) \qquad (*)$$

blackboard1

policy

$$\pi\left(y = 1|\vec{x}, \vec{w}\right) = g(\vec{w}^{\mathsf{T}}\vec{x}) \qquad (1)$$

$$\pi\left(y = 0|\vec{x}, \vec{w}\right) = 1 - g(\vec{w}^{\mathsf{T}}\vec{x}) \qquad (2)$$

batch

$$\Delta w_j = \alpha \; \boxed{\quad ? \quad} \qquad (3)$$

From Batch rule to Online rule

$$(*)\quad \langle R\rangle = \sum_{x}\sum_{y=\{0,1\}} \underbrace{\pi(y|\vec{x})\,p(\vec{x})}_{p(y,x)} R(y,\vec{x})$$

policy

$$\pi(y=1|\vec{x},\vec{w}) = g(\vec{w}^T\vec{x}) \quad (1)$$

$$\pi(y=0|\vec{x},\vec{w}) = 1-g(\vec{w}^T\vec{x}) \quad (2)$$

batch

$$(3)\quad \Delta w_j = \alpha \sum_x p(\vec{x})[\quad ? \quad]x_j$$

$$\Delta w_j = \alpha \sum_{\vec{x}} p(\vec{x})\, g'\left[\underbrace{\frac{\pi(y=1|\vec{x},\vec{\omega})}{g(\vec{\omega}^T\vec{x})} R(1,x)}_{y=1} - \underbrace{\frac{\pi(y=0|\vec{x})}{1-g(}R(0,x)}_{y=0}\right]\cdot x_j$$

$$= \alpha \cdot \sum_x p(\vec{x}) \sum_{y\in\{0,1\}} \pi(y|\vec{x})\cdot R(y,x)\cdot g'\left[\frac{y}{g(\vec{\omega}\cdot\vec{x})} - \frac{1-y}{1-g(\vec{\omega}^T\cdot x)}\right]\cdot x_j \quad (4)$$

Note: This is the pedestrian approach (see video for step-by step calculation) – there are more elegant ways of arriving at this result

Policy Gradient methods: 1-step horizon (summary)

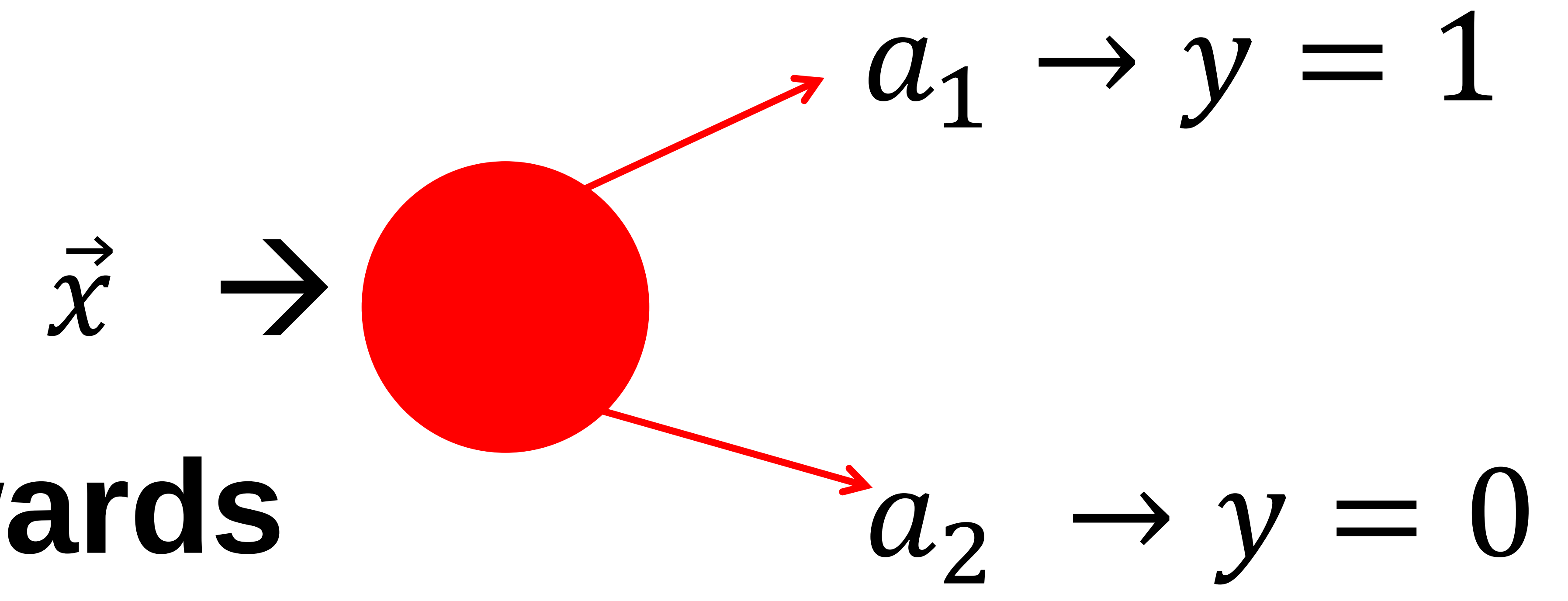
reward

$R(y, \vec{x})$

policy

$$\pi(y = 1|s, \vec{w}) = g\left(\sum_{k}^{N} w_k x_k\right)$$

Online rule (=stochastic gradient ascent)



Update parameters to maximize rewards

$$\left.\begin{aligned} &\text{If } y = 1: \quad \Delta w_j = \alpha \frac{g'}{g} R(1, \vec{x}) x_j \\ &\text{If } y = 0: \quad \Delta w_j = \alpha \frac{-g'}{(1-g)} R(0, \vec{x}) x_j \end{aligned}\right\} \quad (4)$$

(previous slide)
The optimal update rule (last two lines) has a simple interpretation:
The weight w_j in moved in direction of x_j if the reward is positive.
The notation g' refers to the derivative of the sigmoidal function g.

Equation (4) can be written using the if condition (main slide), or alternatively

$$\Delta w_j = \alpha \; g' \, R(y, \vec{x}) \; [\frac{y}{g} - \frac{(1-y)}{(1-g)}] \, x_j \qquad (4)$$

Here, the if condition is implemented by y and (1-y), respectively.

Summary: Policy Gradient methods, from Batch-to-Online

Attention at transition 'Batch to Online':
→ natural statistical weight must be correct!

We have a stochastic starting point with weight $p(s)$ as well as $\underbrace{\text{stochastic transitions}}$ and a $\underbrace{\text{stochastic policy}}$

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

weighting factor
for 'next state'

$$\sum_{a'} \pi(a'|s, \vec{w})$$

weighting factor
for 'next action'

(previous slide)
Batch rule (like in standard ANN): a single update is performed after having processed many patterns (minibatch) or all patterns (standard batch rule).
Online rule (like in standard ANN): an update is performed at every time step (after each pattern.

The example (and your calculations in the exercise) show that the transition from batch to online is not always possible by deleting the sum signs. In fact, it is only possible if the statistical weighting factor is correct.

Reinforcement Learning Lecture 4
Policy Gradient Methods

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Part 4: From Batch to Online: Log-likelihood trick

1. First steps toward deep reinforcement learning
2. Basic idea of policy gradient
3. Example: 1-step horizon
4. **From Batch to Online: Log-likelihood trick**

(previous slide)
Is there a more systematic way to perform the transition from batch to online?
The answer is yes and given by (what I call) the log-likelihood trick

From Batch to Online

Change parameters w to maximize average reward

$$\langle R \rangle_w = \sum_x \sum_y p(\vec{x})\, \pi_w(y|\vec{x})\, R(y,\vec{x})$$

Is there an ‘elegant’ way to keep a correct statistical weight when
averaging a gradient in ‘online’ mode?

(previous slide)
The aim is to maximize the gradient of a reward function which involves averaging.

Is there a more systematic way to perform the transition from batch to online?
The answer is yes. How to implement this elegant derivation is explained next.

Log-likelihood trick

Blackboard 2

$$\langle R \rangle_w = \sum_x \sum_y p(\vec{x})\, \pi_w(y|\vec{x})\, R(y,\vec{x})$$

optimization yields online rule

$$\Delta w_j = \alpha R(y,\vec{x}) \frac{d}{dw_j} \ln \pi_w(y|\vec{x})$$

(your comments)

Summary: Log-likelihood trick (1-step horizon)

Aim: change weights so as to maximize

$$\langle R_w \rangle = \sum_x \sum_y p(\vec{x})\, \pi_w(y|\vec{x})\, R(y,\vec{x})$$

Optimization by gradient decent yields online rule

$$\Delta w_j = \alpha R(y,\vec{x}) \frac{d}{dw_j} \ln \pi_w(y|\vec{x}) \qquad (5) \qquad \text{(online policy gradient)}$$

The derivative of log of policy plays an important role!

(previous slide)

In the setting of a 1-step-horizon, a policy gradient algo adapts the parameters w so as to maximize the expected reward

$$\langle R_w \rangle = \sum_x \sum_y p(\vec{x})\, \pi_w(y|\vec{x})\, R(y,\vec{x})$$

Our aim is to arrive at an online update rule with the correct statistical weight. To achieve this we use the derivative of the logarithm of the policy. Then the we can cut out the natural statistical weight and find the online rule

$$\Delta w_j = \alpha R(y,\vec{x}) \frac{d}{dw_j} \ln \pi_w(y|\vec{x})$$

Note that w_j is one of the many parameters that together form the parameter vector $\boldsymbol{w}$

Reinforcement Learning Lecture 4
Policy Gradient Methods

EXERCISES 1-3 now.
Next Lecture at 14h15

Part 4*: From Batch to Online: Log-likelihood trick

1. First steps toward deep reinforcement learning
2. Basic idea of policy gradient
3. Example: 1-step horizon
4. From Batch to Online: Log-likelihood trick

4* Example (1-step horizon) revisited

(your comments)

Log-likelihood trick

$$\langle R\rangle = J = E[R] = \int p(H)R(H)dH$$

depends on θ

$$\begin{aligned}
\nabla_\theta J &= \int \nabla_\theta p(H)R(H)dH \\
&= \int \frac{p(H)}{p(H)}\nabla_\theta p(H)R(H)dH \\
&= \int p(H)\nabla_\theta \log p(H)R(H)dH
\end{aligned}$$

$$= E[R\nabla_\theta \log p] = \int p(H)R(H)\nabla_\theta \log p(H)\, dH$$

J = function you want to optimize

H = ensemble over which you integrate

(previous slide)
From BATCH to ONLINE (review of calculation with different notation).

Suppose you want to optimize some function J which is given by the integral over the **statistical ensemble *H***. Instead of an integral you often have the discrete sum over all possible patterns, for example.
You want to do optimization by gradient ascent, therefore you need to calculate the gradient.
For the correct statistical weight you need the weight factor *p(H)*. But it is exactly the weight factor that depends on the parameters.
Normally this factor disappears (is invisble) when you naively take the gradient.

However, if you rewrite this as the gradient of (log p) and then multiply by p(H), you have the exactly the same result – but now the correct weight factor p(H) is explicit. Once the statistical weight factor is visible, you can cut out the integral and p(H) and get a valid online rule.

Policy gradient derivation

$$\nabla_\theta J = \int p(H) \nabla_\theta \log p(H) R(H) dH.$$

Taking the sample average as Monte Carlo (MC) approximation of this expectation by taking N trial histories we get

$$\nabla_\theta J = \mathbf{E}_H \Big[\nabla_\theta \log p(H) R(H) \Big] \approx \frac{1}{N} \sum_{n=1}^{N} \nabla_\theta \log p(H^n) R(H^n).$$

which is a fast approximation of the policy gradient for the current policy

(previous slide)
Delete the integral and p(H) and sum over all examples, and you have a good approximation to your original integral. It works because the examples appear with their natural statistical weight!

Policy gradient evaluation: Example (1-step horizon)

Claim: log-likelihood trick yields online rule

$$\Delta w_j = \alpha g' \, R(y,\vec{x}) \, [\frac{y}{g} - \frac{(1-y)}{(1-g)}] \, x_j \qquad (4)$$

reward

EXCERCISES

$R(y,\vec{x})$

policy

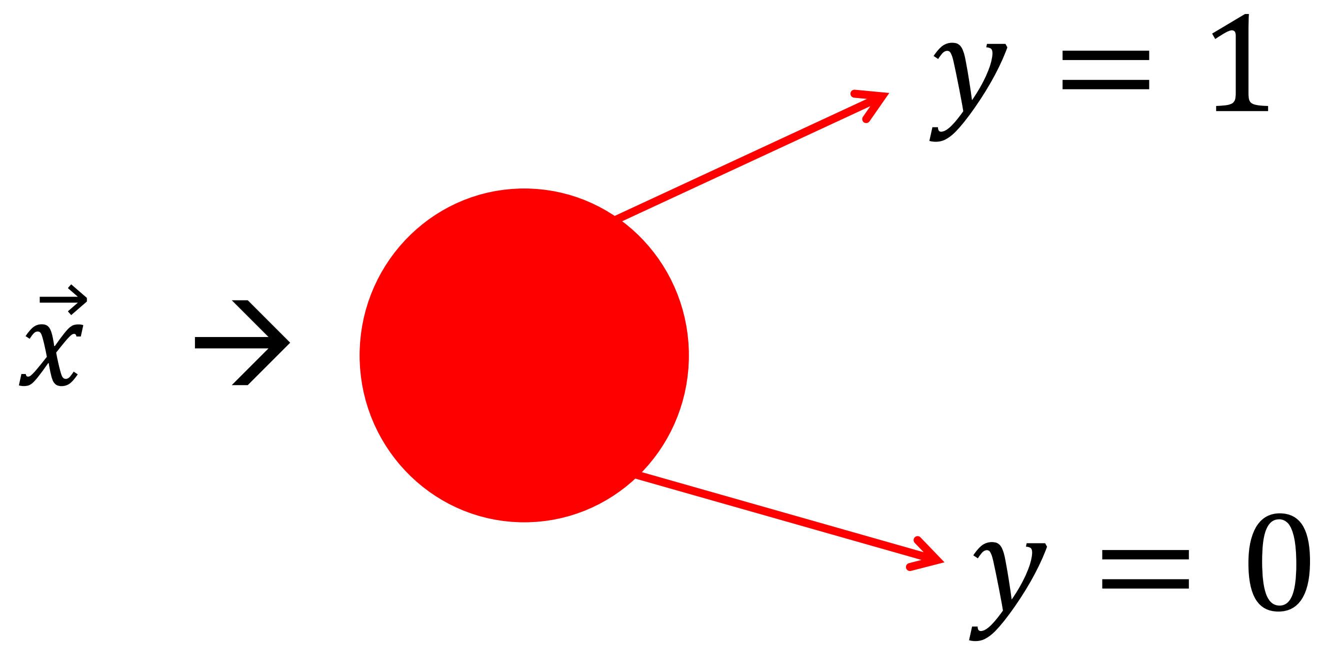
$$\pi(y=1|\vec{x},\vec{w}) = g(\vec{w}^{\top}\vec{x})$$

$$\pi(y=0|\vec{x},\vec{w}) = 1 - g(\vec{w}^{\top}\vec{x})$$

Proof: (Exercise)

$$\Delta w_j = \alpha R(y,\vec{x}) \frac{d}{dw_j} \ln \pi_w(y|\vec{x})$$

(online policy gradient)



(previous slide).
We now return to our one-dimensional example!
The calculations have been done in the Exercises.

Update rule of example

observe input $\vec{x}$, output y, and reward $R(y, \vec{x})$

EXERCISES

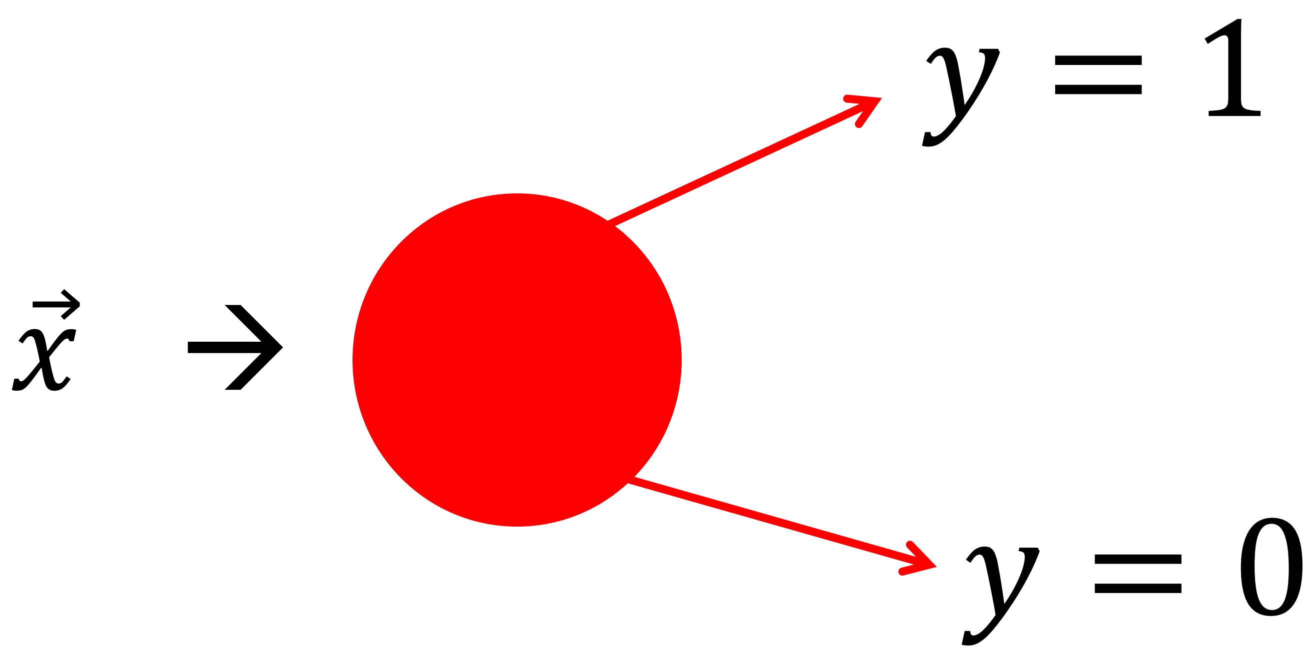
Earlier result, Eq. (4):

If $y = 1$: $\quad \Delta w_j = \alpha \dfrac{g'}{g} \cdot R(1, \vec{x}) x_j$

If $y = 0$: $\quad \Delta w_j = \alpha \dfrac{-g'}{(1-g)} R(0, \vec{x}) x_j$

policy

$$\pi(y = 1 | s, \vec{w}) = g\left(\sum_k^N w_k x_k\right)$$



Now rewritten as:

$$\Delta w_j = \alpha \frac{g'}{g(1-g)} R(y, \vec{x}) [\, y - \langle y \rangle] x_j \quad (6)$$

Note: $\langle y \rangle = g(\sum_k^N w_k x_k)$

(previous slide)
Using the log-likelihood trick we arrive at the same result as before but faster and, importantly, via a systematic sequence of steps.

Last line – two important comments:

(i) The two cases (y=+1) and (y=0) can be summarized in a single update rule
(ii) <y> is the expectation of the output, given the input vector $\vec{x}$

Quiz: Policy Gradient and Reinforcement learning

Your friend has followed over the weekend a tutorial in reinforcement learning and claims the following. Is he right?

[ ] All reinforcement learning algorithms work either with Q-values or V-values

[ ] The transition from batch to online is always easy: you just drop the summation signs and bingo!

[ ] Both TD algorithms and policy gradient algorithms aim to optimize the expected total reward (potentially discounted if there are multiple time steps)

(your comments)

First Interpretation: Comparison with Perceptron

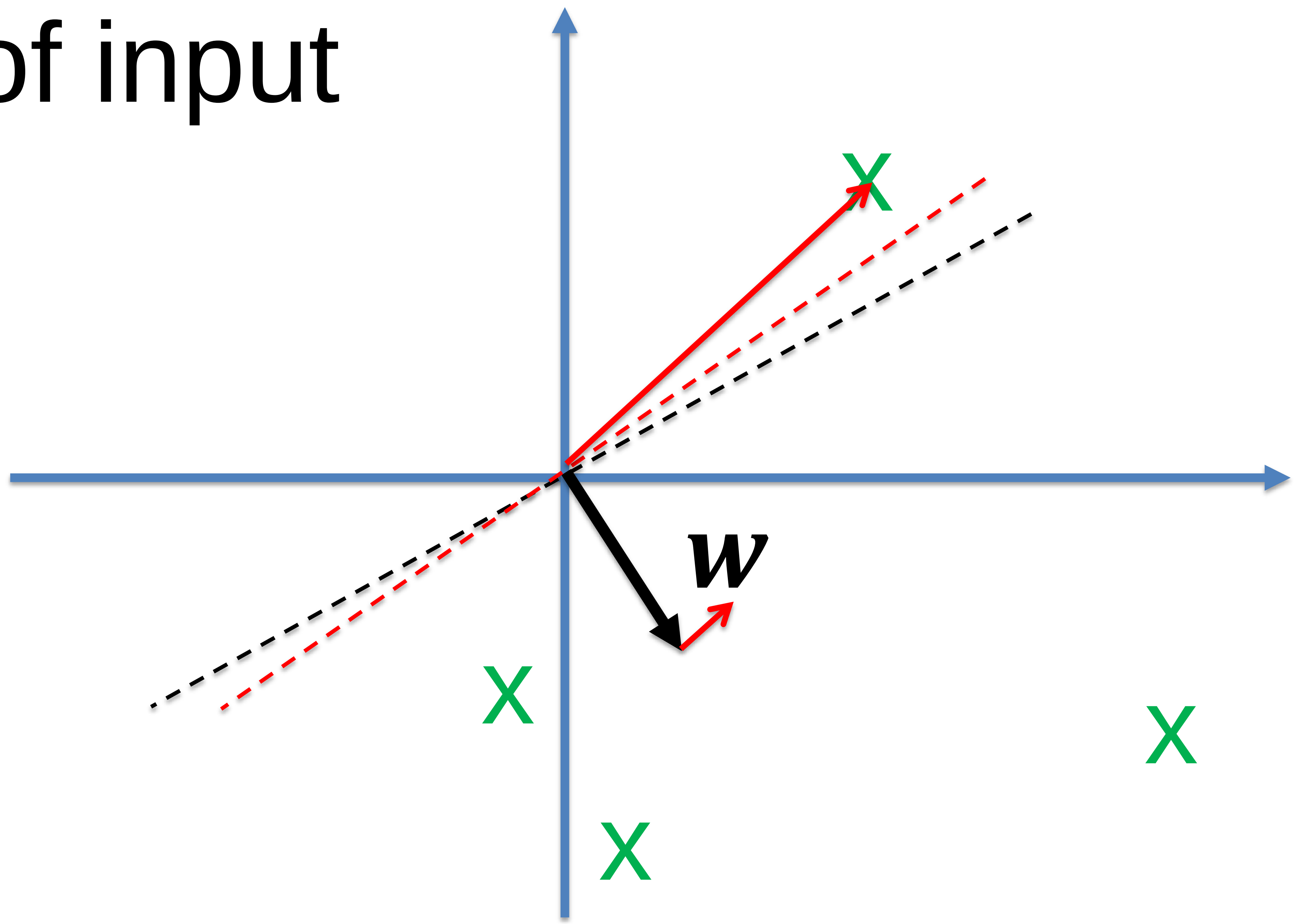
parameter = weight w_j

$$\Delta w_j \propto R(y, \vec{x})[\,y - \langle y \rangle]x_j$$

Weight vector turns in direction of input

$$\Delta \boldsymbol{w} \propto \pm \boldsymbol{x}$$

R>0 and y=1 → $\Delta \boldsymbol{w} \propto +\boldsymbol{x}$



(previous slide)

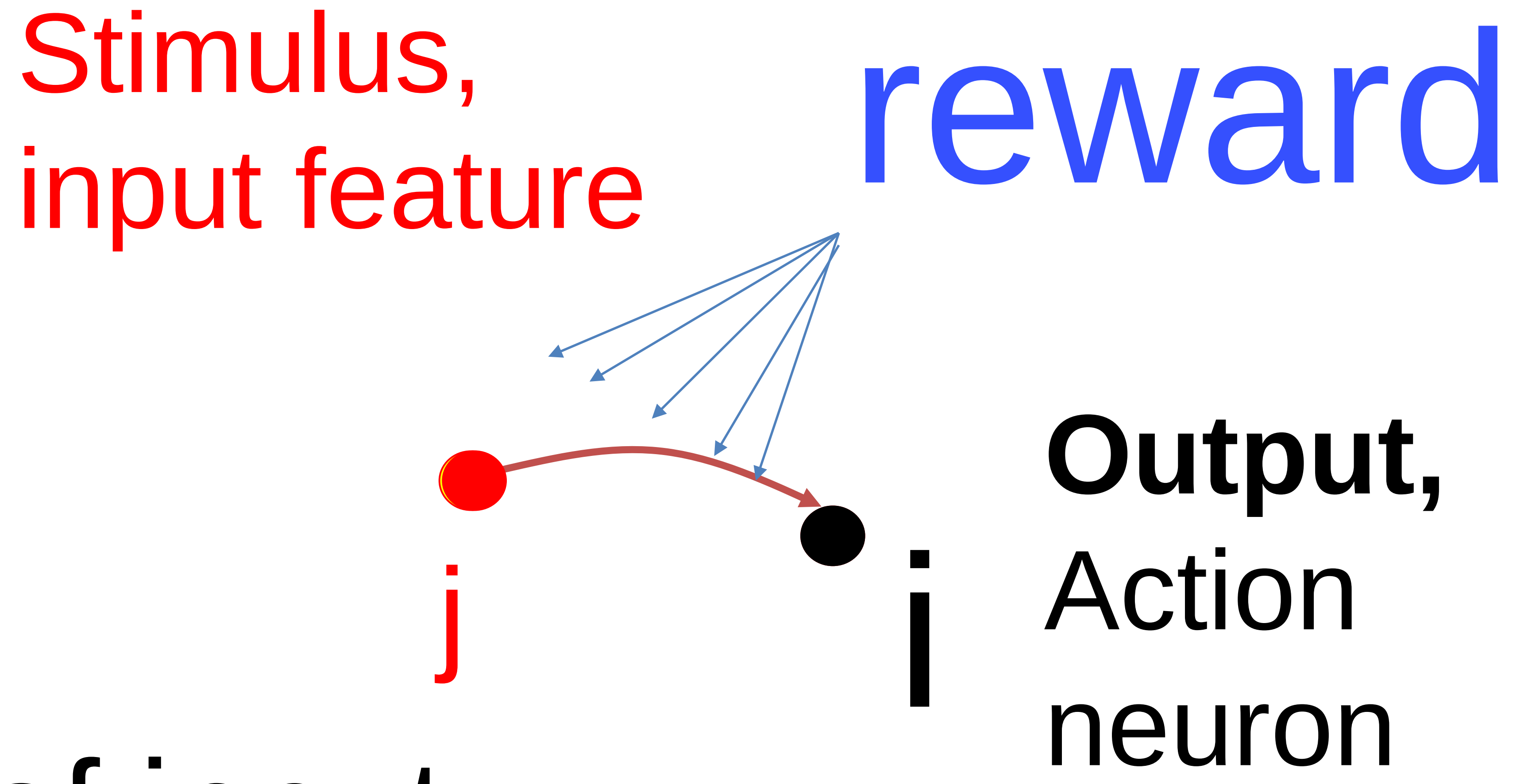
Similar to the perceptron update rule, the update with gradient descent can be interpreted as a weight vector that turns in direction of an input pattern (with positive or negative sign)

Second Interpretation: activity minus expected activity

parameter = weight w_j

$$\Delta w_j \propto R(y, \vec{x})[\, y - \langle y \rangle] x_j$$

Weight vector turns in direction of input



Three factors: **reward** **output** **stimulus**

$$\Delta w_{ij} = \eta \frac{g'}{g(1-g)} R(\vec{y}, \vec{x}) \overbrace{[\, y_i - \langle y_i \rangle]}^{\text{output}} x_j$$

output factor is

'activity – expected activity'

(previous slide)

The update rule gives also rise to an interesting interpretation that is useful to understand biology, but also to develop neuromorphic hardware chips (chips for non-standard computing principles)

The learning rule depends on three factors:

(i) The reward

(ii) The 'state' of the output neuron where 'state'=activity minus expected activity

(iii) The input feature (e.g., one of the pixels)

The connection weight is increased if the output in a given trial is +1 and hence larger than the expected output, and a positive reward was received in this trial. If the reward was negative in that trial, the output +1 was bad and the weight is decreased. As we will see in the following slides,
a further step is to also subtract the expected reward that is received on average for this input

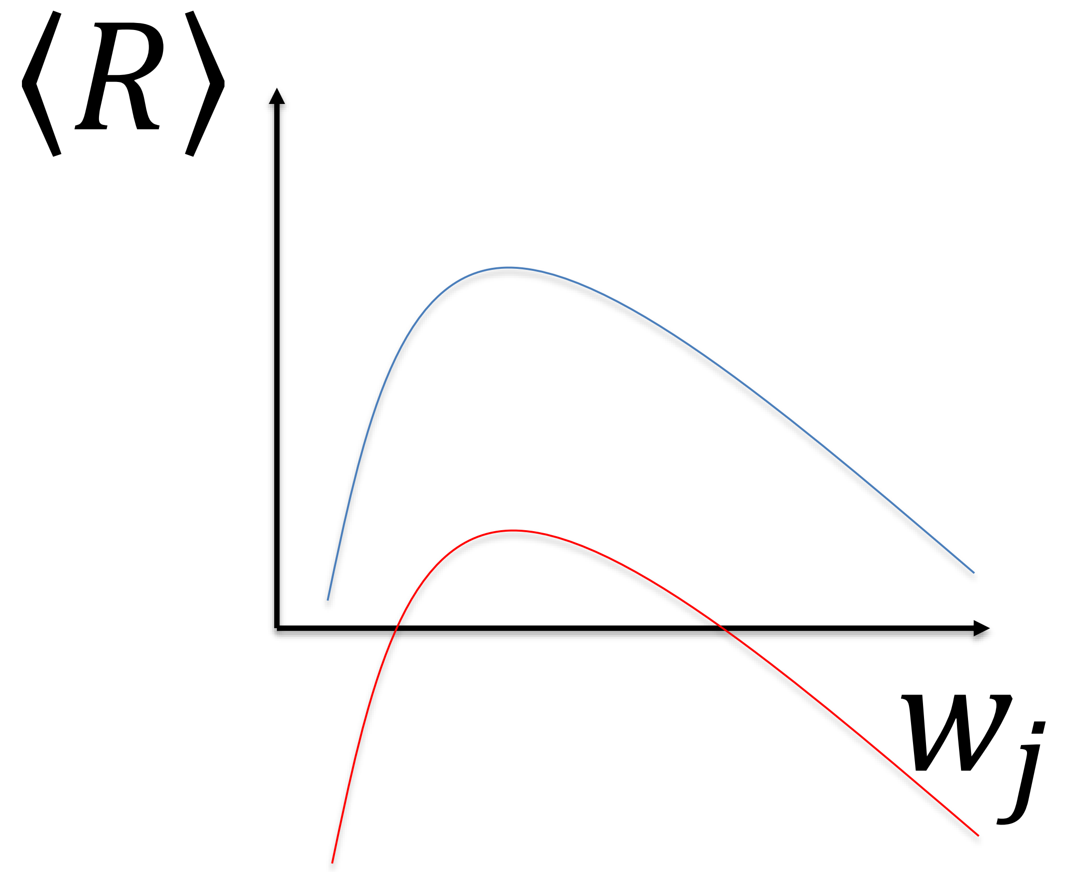
$$\Delta w_{ij} \sim \; [R(\vec{y},\vec{x}) - \langle R(x)\rangle [\, y_i - \langle y_i \rangle] x_j$$

A connection is defined by the two neurons that it connects. The change of the connections depends on the input x_j (i.e. the state of the sending neuron) and the output y_i $(i.e.$ the state of the receiving neuron) neuron-specific. Information about the reward is broadcasted across the network and shared by many (or even all) neurons. The broadcast signal can be positive or negative.

Several research groups (e.g. at INTEL, at the Institute of Neuroinformatics in Zurich, at University of Heidelberg) develop neuromorphic hardware that is inspired by these principles!

We will come back to this at the very end of the semester.

Generalization: subtract a reward baseline



maximizing $\langle R \rangle = \sum_x \sum_y p(\vec{x})\, \pi(y|\vec{x})\, R(y,\vec{x})$

we derived this online gradient rule

$$\Delta w_j \propto R(y,\vec{x})[\,y - \langle y \rangle]x_j$$

But then this rule is also an online gradient rule

$$\Delta w_j \propto [R(y,\vec{x}) - b][\,y - \langle y \rangle]x_j$$

with the same location of maximum

(start with $\langle R - b \rangle$, but the baseline shift is irrelevant if we take the gradient)

(previous slide)
Note that the we are interested in finding the set of weights that optimize the expected reward <R>.
The update rule has been derived by taking the gradient on the mean reward <R>.

But a function <R-b> with constant bias b would have exactly the same location of the maximum.
If we repeated the gradient steps, the results would lead to an update rule with a factor [R-b] instead of R. Therefore, the rule with [R-b] is also a valid online rule.

Why subtract a baseline?

Subtracting an appropriate baseline makes an online algorithm less noisy so that it converges better. Good baseline is mean.

Example: **estimate mean of product** $x(y-\bar{y})$ **of two indep. variables**

with substraction

mean:

$$\langle (x-\bar{x})(y-\bar{y})\rangle = 0$$

Sample k:

$$(x_k-\bar{x})(y_k-\bar{y})$$

$$x_k = 5 \pm 1$$
$$y_k = 8 \pm 1$$

without substraction

mean:

$$\langle x(y-\bar{y})\rangle = 0$$

Sample k:

$$x_k(y_k-\bar{y})$$

$$= (x_k-\bar{x})(y_k-\bar{y}) + \bar{x}(y_k-\bar{y})$$

$$\text{order} = \pm 1 \qquad noise = \pm 5$$

(previous slide)
Why is it useful to subtract the mean?

Whatever the choice of baseline, the algorithm should eventually converge to the same set of parameters. However, since the algorithm is based on stochastic gradient descent (i.e., the online rule instead of the full batch rule), the algorithm makes **noisy steps** that only go on average in the right direction.

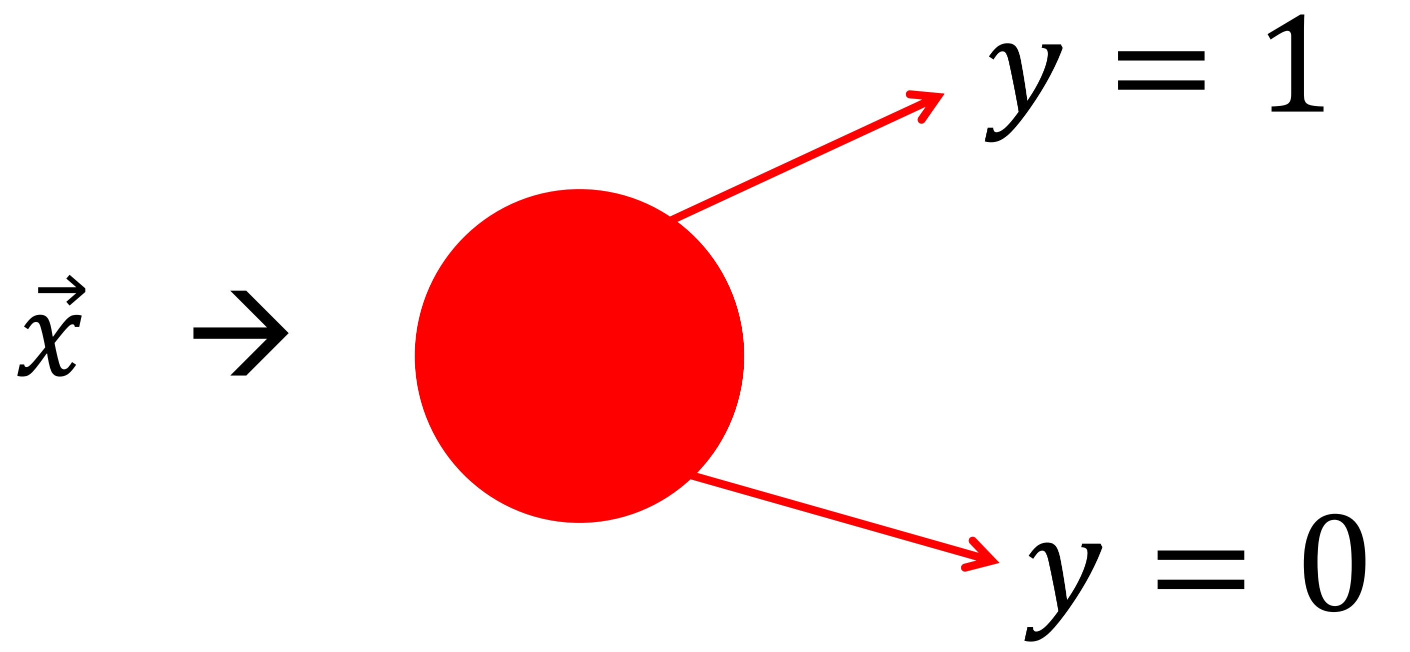
Subtracting a baseline that is close to the mean generally reduces the noise.
The example with a product of independent variables shows that by subtracting the mean of x, the noise is considerable reduced in each of the samples!

Note that we have seen earlier that the update rule of policy gradient can be written as a product of R and something like $(y - \bar{y})$ for fixed input. Replace x by R and you are done, assuming independence of the two variables.

Third Interpretation: Detect covariance (1-step horizon)

policy

$$\pi(y=1|s,\vec{w}) = g(\sum_{k}^{N} w_k x_k)$$



Example: in trial n

- input x_j>0
- output y=1 → $[y - \langle y \rangle] > 0$
- more reward than on average
  → $R(y,\vec{x}) - \langle R \rangle > 0$

Hence:

weight increases

$$\Delta w_j \propto [R(y,\vec{x}) - \langle R \rangle][y - \langle y \rangle] x_j$$

Detects co-variance between reward and actions (given stimulus)

(previous slide)
The update rule gives also rise to an interesting interpretation

The learning rule depends on three factors:

(i) The reward
(ii) The ‘state’ of the output neuron where ‘state’=activity minus expected activity
(iii) The input feature (e.g., one of the pixels)

For positive input, the connection weight is increased if the output in a given trial is +1 and hence larger than the expected output and the reward larger than the expected reward.

Quiz: Policy Gradient and Reinforcement learning

[ ] For the 1-step horizon and binary action choice, the derivative of the log-policy has an intuitive interpretation

Teaching monitoring – monitoring of understanding

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

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

Reinforcement Learning Lecture 4
Policy Gradient Methods

Wulfram Gerstner
EPFL, Lausanne, Switzerland

Part 5: Multiple time steps

1. First steps toward deep reinforcement learning
2. Basic idea of policy gradient
3. Example: 1-step horizon
4. Log-likelihood trick
5. **Multiple time steps**

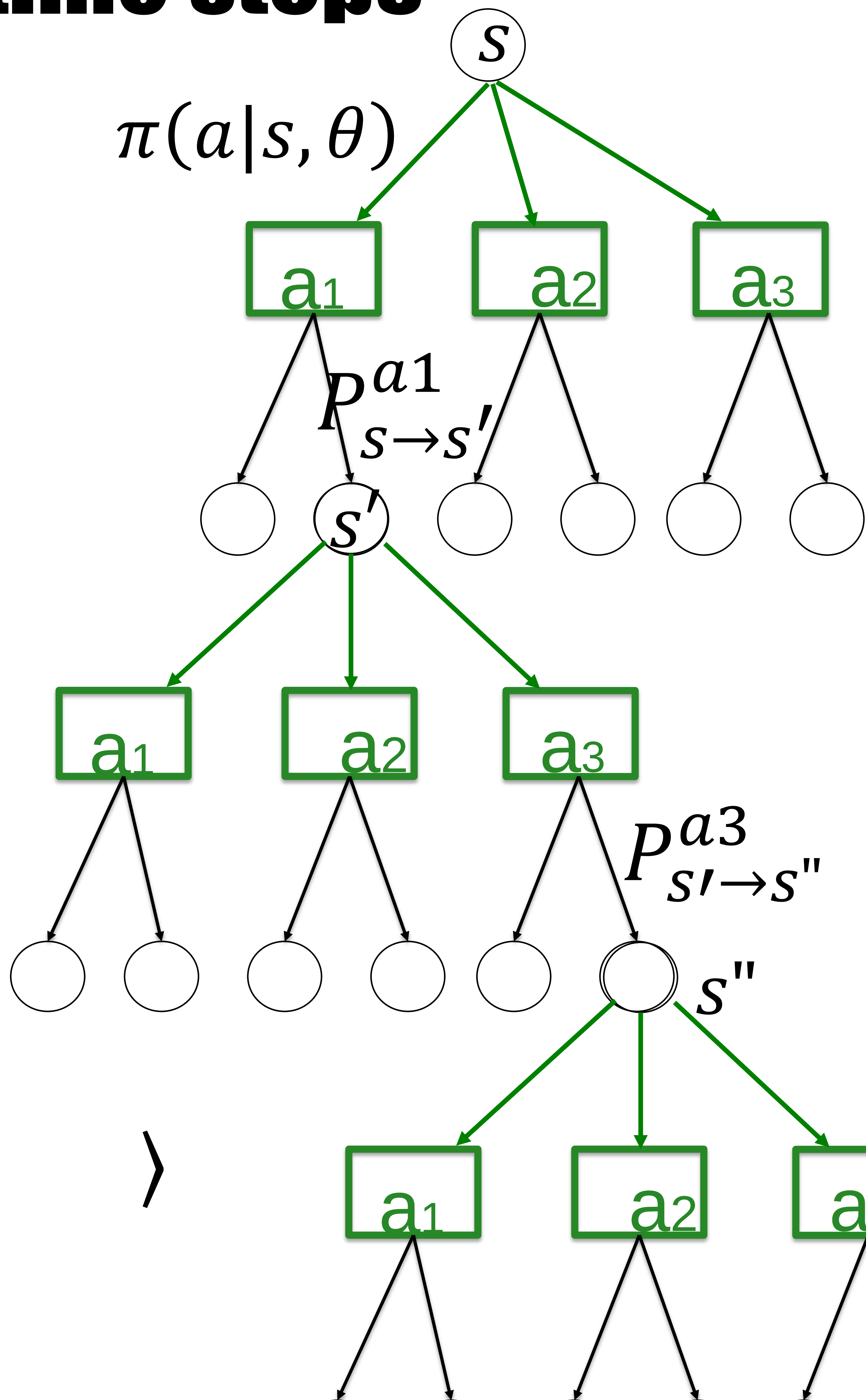
(previous slide)
So far the discussion has been restricted to scenarios with a one-step horizon.
The agent takes an action, gets a reward, and the episode ends.
Now we need to generalize to scenarios that extend over multiple time steps.

Policy Gradient methods over multiple time steps

Aim:
update the parameters θ
of the policy $\pi(a|s,\theta)$

so as to maximize the average
total discounted reward from s
(expected ***Return***)

$$\langle R_{s_t \to s_{end}} \rangle = \langle r_t + \gamma^1 r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} \quad \rangle$$



(previous slide)

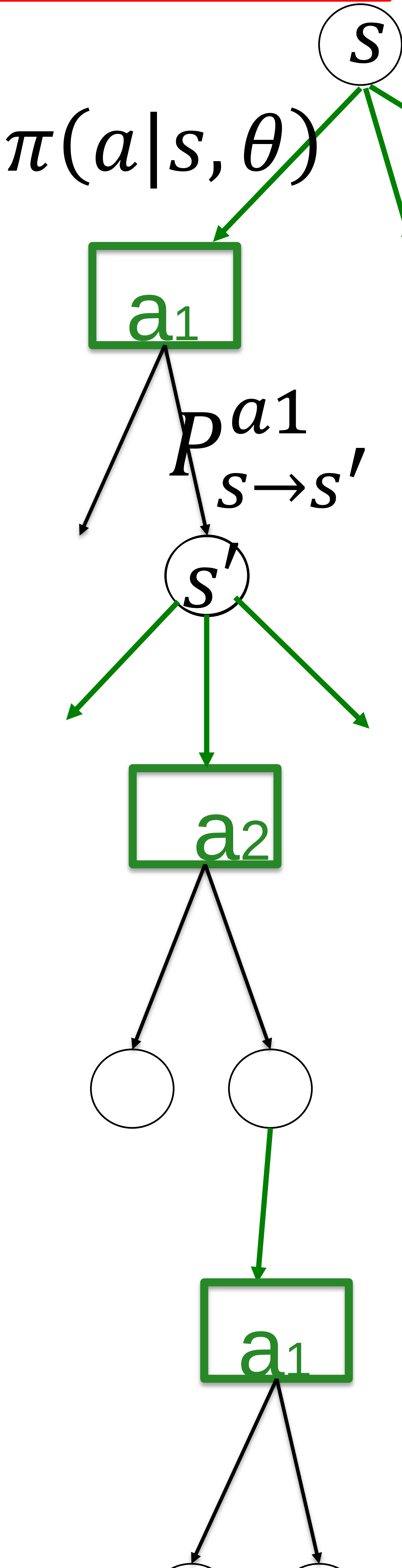
We use the same graph of the multistep Markov decision model as for the derivation of the Bellman equation.

However, now we work directly on a policy $\pi(a|s,\theta)$ which depends on parameters θ.

Policy Gradient methods over multiple time steps

Blackboard 3

$$\langle R_{s_t \to s_{end}} \rangle = \langle r_t + \gamma^1 r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} \rangle$$



Policy Gradient methods over multiple time steps

Calculation yields several terms of the form

Total accumulated discounted reward
collected in one episode starting at s_t, a_t

$$\Delta\theta_j \propto \left[R^{a_t}_{s_t \to s_{end}}\right] \frac{d}{d\theta_j} \ln[\pi(a_t|s_t,\theta)]$$

$$+\gamma\left[R^{a_{t+1}}_{s_{t+1} \to s_{end}}\right] \frac{d}{d\theta_j} \ln[\pi(a_{t+1}|s_{t+1},\theta)]$$

$+ \ldots$

(previous slide)

We consider a single episode that started in state s_t with action a_t and ends after several steps in the terminal state s_{end}

The result of the calculation gives an update rule for each of the parameters.

The update of the parameter θ_j contains several terms.

(i) the first term is proportional to the total accumulated (discounted) reward, also called return $R^{a_t}_{s_t \to s_{end}}$

(ii) the second term is proportional to γ times the total accumulated (discounted) reward but starting in state s_{t+1}

(iii) the third term is proportional to γ -squared times the total accumulated (discounted) reward but starting in state s_{t+2}

(iv) We can think of this update as one update step for one episode. Analogous to the terminology last week, Sutton and Barto call this the Monte-Carlo update for one episode.

Note that each of the terms is proportional to ln π

Pseudo-code for algo REINFORCE (Policy Gradient)

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

From book:
Sutton and Barto, 2018

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, r_1, \ldots, S_{T-1}, A_{T-1}, r_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$ (G_t)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Different states S_0, S_1, S_2, ... during one episode

G = total accumulated reward during the episode starting at S_t;

All updates done AT THE END of the episode

Algorithm maximizes expected discounted rewards starting at S_0

(previous slide)
The algorithm in Pseudocode taken from the book of Sutton and Barto. The update concerns a single episode.

The only notational difference with respect to the earlier slide is a rewrite of the factors gamma – you can check the equivalence by taking a piece of paper.

Note that for an implementation it would be most convenient to start at the terminal state of the episode and work backwards so as to reuse the return calculations.

Variations of this algorithm are the basis of policy gradient methods and widely used in applications.

IMPORTANT: This version of the algo is derived for the situation where we optimize the return from a known starting state and a known terminal state.
In practice it works better to optimize the return from ALL possible states (appropriately weighted). This will be treated in a separate lecture.

Summary: Policy Gradient methods over multiple time steps:

-starting at s_t

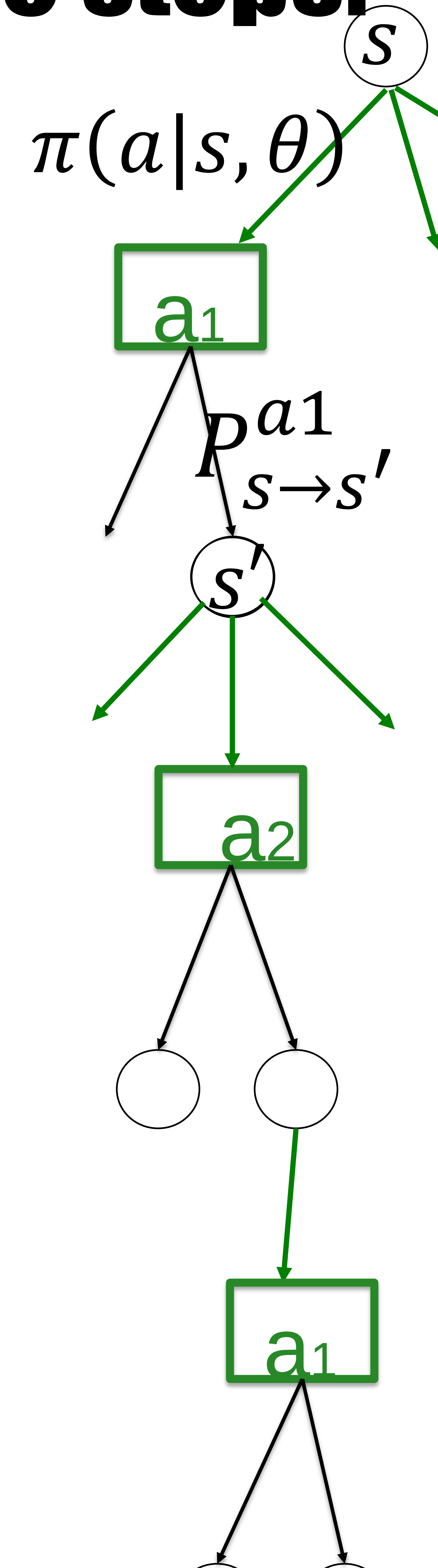
-derivative of log-policy at different states visited during episode,

$$\frac{d}{d\theta_j}\ln[\pi(a_t|s_t,\theta)] \;\; R^{a_t}_{s_t \to s_{end}}$$

$$+\gamma^1 \frac{d}{d\theta_j}\ln[\pi(a_{t+1}|s_{t+1},\theta)] \;\; R^{a_{t+1}}_{s_{t+1} \to s_{end}}$$

$$+\gamma^2 \frac{d}{d\theta_j}\ln[\pi(a_{t+2}|s_{t+2},\theta)] \;\; R^{a_{t+2}}_{s_{t+2} \to s_{end}}$$

- Multiplied with the returns from each state
- discounted with γ



(previous slide)
Summary of the basic algorithmic principle of a policy gradient method over multiple time steps, if many episodes start in the same state s, and you optimize return for this state s. Episodes end at the terminal state. This is called the episodic case.

In practice algorithms that optimize the return from all states work better, because this version of the episodic algorithm puts most weight on rewards in states close to the starting state. However, if the only reward is at the end (at the terminal state) then it is better to either use a discount very close to 1, or to optimize the return from ALL states (i.e., also from those closer to the target).
We will come back to this in the lecture on deep reinforcement learning

Here we optimize $\langle R_{s_t \to s_{end}} \rangle$ where the return is averaged over paths starting in s_t

Later we optimize $\langle R_{s \to s_{end}} \rangle$ where the return is averaged over all states encountered during the path. Thus we optimize the MEAN return. This usually works better.

Learning outcomes and Conclusions:

- **basic idea of policy gradient: learn actions, not Q-values**
  → gradient ascent of total expected discounted reward
- **log-likelihood trick: getting the correct statistical weight**
  → enables transition from batch to online
- **policy gradient algorithms**
  → updates of parameter propto $[R\ \ ]\frac{d}{d\theta_j}\ln[\pi]$ (several terms)
- **why subtract the mean reward?**
  → reduces noise of the online stochastic gradient
- **Reinforce with baseline**
  → a further output to subtract the mean reward

$$[R(s)\ \ -V(s)]\frac{d}{d\theta_j}\ln[\pi]$$

(previous slide) Your notes

Teaching monitoring – monitoring of understanding

The End

[ ] 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.

Exercise : Subtract baseline

THE END
Exercise at 15h15

Exercise 4. Subtracting the mean

You have two stochastic variables, x and y with means $\langle x \rangle$ and $\langle y \rangle$. Angles denote expectations. We are interested in the product $z = (x - b)(y - \langle y \rangle)$ with a fixed parameter b.

a. Show that $\langle z \rangle$ is independent of the choice of the parameter b.

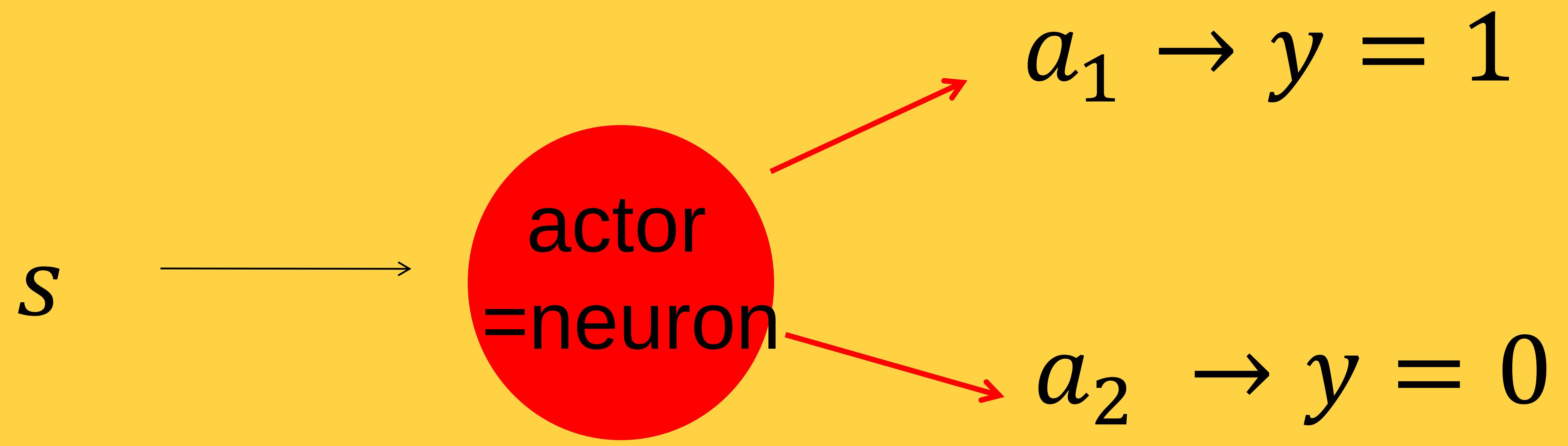
b. Show that $\langle z^2 \rangle$ is minimal if $b = \frac{\langle x f(y) \rangle}{\langle f(y) \rangle}$, where $f(y) = (y - \langle y \rangle)^2$.

   Hint: write $\langle z^2 \rangle = F(b)$ and set $dF/db = 0$.

c. What is the optimal b, if x and $f(y)$ are approximately independent?

d. Make the connection to policy gradient rules.

   Hint: take $x = r$ (reward) and y the action taken in state s. Compare with the policy gradient formula of the simple 1-neuron actor. What can you conclude for the best value of b? Consider different states s. Why should b depend on s?



Reinforcement Learning Lecture 4
Policy Gradient Methods

Wulfram Gerstner
EPFL, Lausanne, Switzerland

Part 6: Subtracting the mean via the value function

1. First steps toward deep reinforcement learning
2. Basic idea of policy gradient
3. Example: 1-step horizon
4. Log-likelihood trick
5. Multiple time steps
6. **Subtracting the mean via the value function**

(previous slide)

The REMAINING SLIDES ARE SHIFTED TO THE FOLLOWING WEEK!!!!

Wait for next Lecture.

Review: subtract a reward baseline (1-step horizon)

we derived this online gradient rule (for 1-step horizon)

$$\Delta w_j \propto R(y, \vec{x})[\, y - \langle y \rangle] x_j$$

But then this rule is also an online gradient rule

$$\Delta w_j \propto [R(y, \vec{x}) - b][\, y - \langle y \rangle] x_j$$

with the same expectation

(because a baseline shift drops out if we take the gradient)

(previous slide)
In the simple one-step scenario we have seen that we can subtract a bias b from the reward.

The question arises whether the same is true in the multi-step episodes.
The answer is YES.

Subtract a reward baseline

online gradient rule for multi-step horizon
has many terms of form

$$\Delta\theta_j \propto \; [R_{s_t \to s_{end}}^{a_t}] \, \frac{d}{d\theta_j} \ln[\pi(a_t|s_t,\theta)]$$

But then this rule is also an online gradient rule

$$\Delta\theta_j \propto \; [R_{s_t \to s_{end}}^{a_t} - b(s_t)] \, \frac{d}{d\theta_j} \ln[\pi(a_t|s_t,\theta)]$$

with the same optimum

(because a baseline shift drops out if we take the gradient)

(previous slide)
Please remember that the full update rule for the parameter θ_j in a multi-step episode contains several terms of this form; here only the first of these terms is shown.

Similar to the case of the one-step horizon, we can subtract a bias b from the reward without changing the location of the maximum of the total expected return.

Moreover, this bias $b(s_t)$ can itself depend on the state s_t.
Thus the update rule now has terms

$$
\begin{aligned}
\Delta\theta_j \;\propto\; & [R_{s_t \to s_{end}}^{a_t} - b(s_t)] \tfrac{d}{d\theta_j} \ln[\pi(a_t|s_t,\theta)] \\
& + \gamma [R_{s_{t+1} \to s_{end}}^{a_{t+1}} - b(s_{t+1})] \tfrac{d}{d\theta_j} \ln[\pi(a_{t+1}|s_{t+1},\theta)] \\
& + \gamma^2 [R_{s_{t+2} \to s_{end}}^{a_{t+2}} - b(s_{t+2})] \tfrac{d}{d\theta_j} \ln[\pi(a_{t+2}|s_{t+2},\theta)] \\
& + \dots
\end{aligned}
$$

Subtract a reward baseline

Total accumulated discounted reward
collected in one episode starting at s_t, a_t

$$\Delta\theta_j \propto [R^{a_t}_{s_t \to s_{end}} - b(s_t)] \frac{d}{d\theta_j} \ln[\pi(a_t|s_t,\theta)] + \ldots$$

- The bias b can depend on state s
- Good choice is b ='mean of $[R_{s_t \to s_{end}}]$'

→ take $b(s_t) = V(s_t)$

→ learn value function *V(s)*

(previous slide
Is there a choice of the bias $b(s_t)$ that is particularly good?

One attractive choice is to take the bias equal to the expectation (or empirical mean). The logic is that if you take an action that gives more accumulated discounted reward than your empirical mean in the past, then this action was good and should be reinforced.
If you take an action that gives less accumulated discounted reward than your empirical mean in the past, then this action was not good and should be weakened.

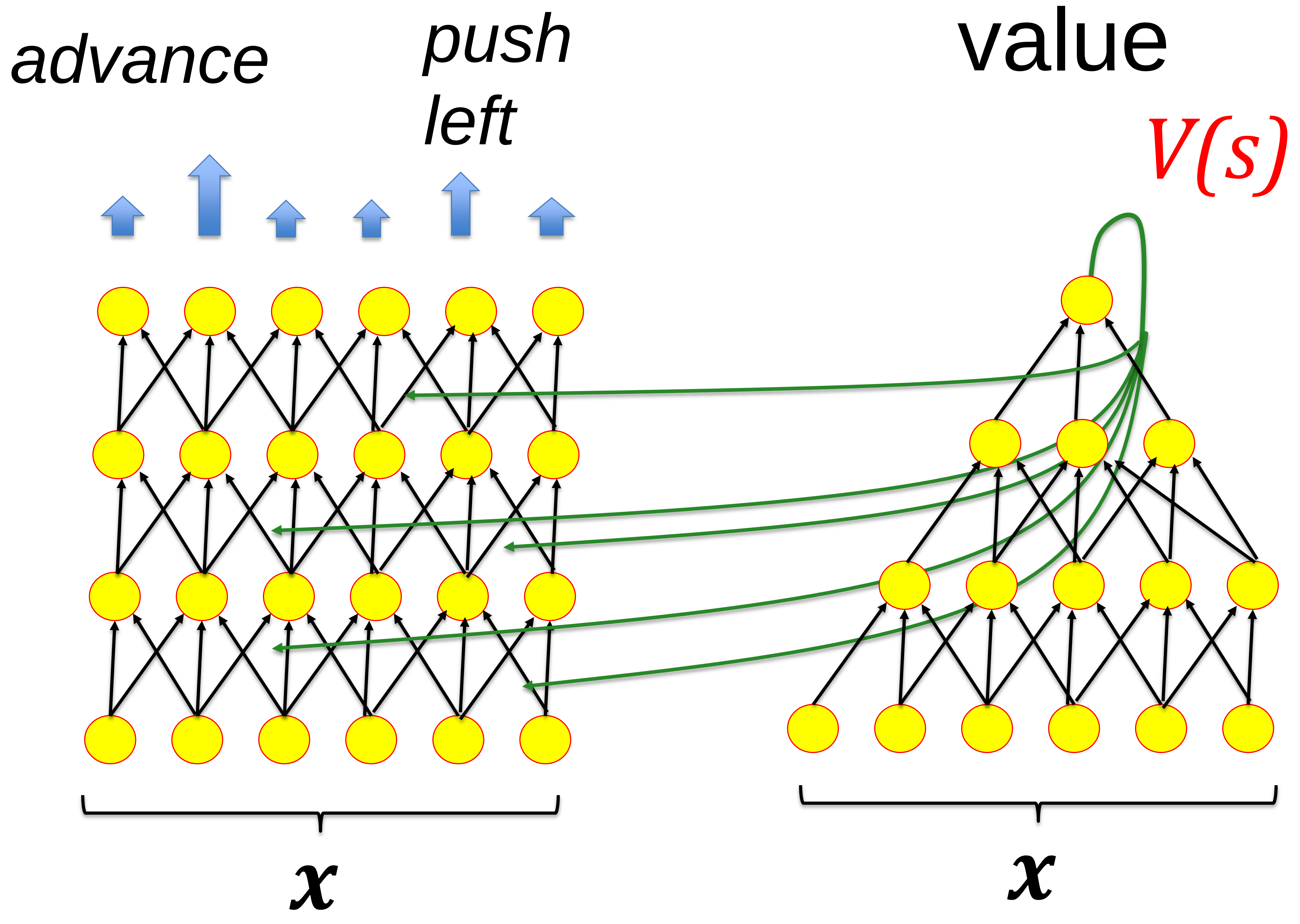
But what is the expected discounted accumulated reward? This is, by definition, exactly the value of the state. Hence a good choice is to subtract the V-value.

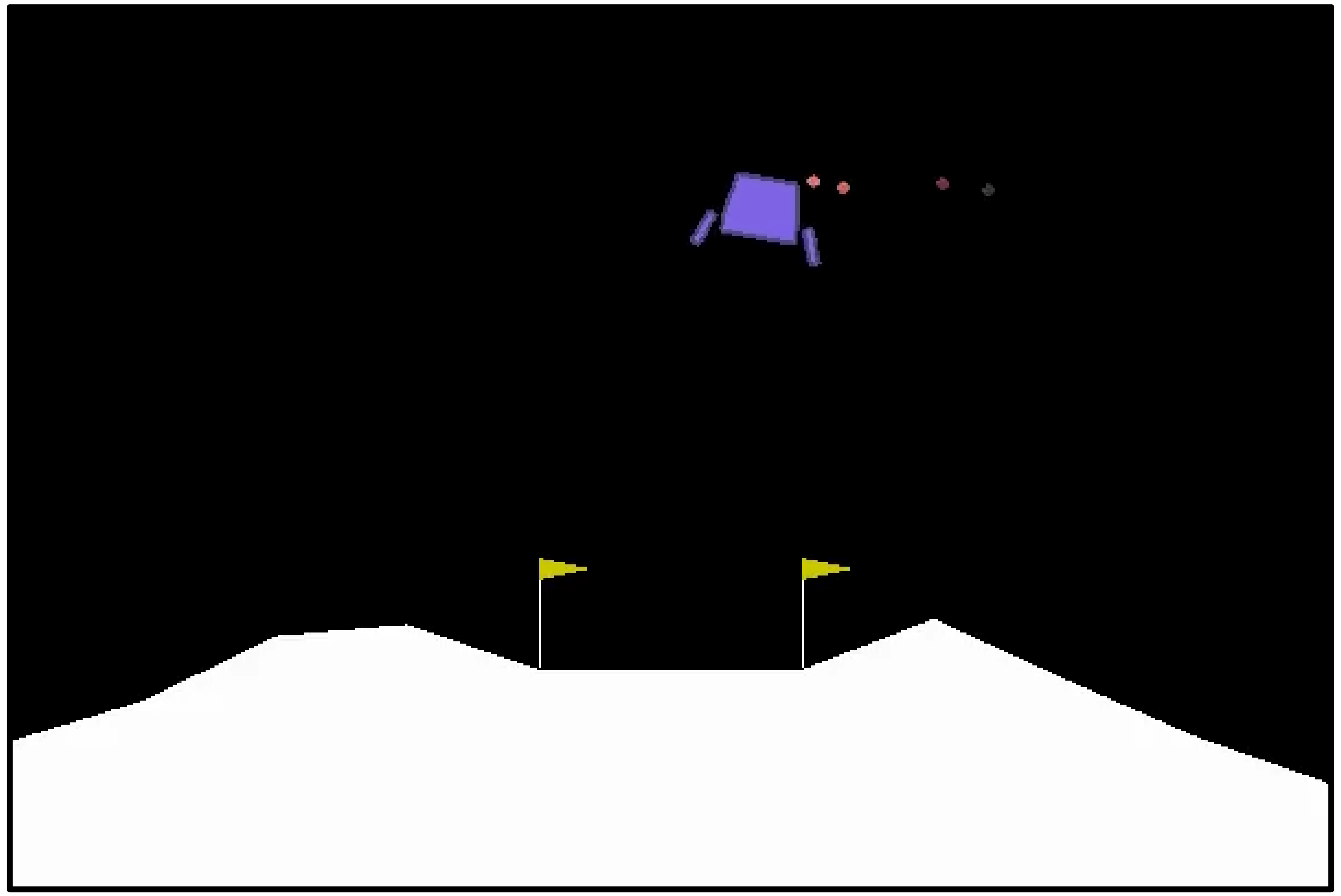
And here is where finally the idea of Bellman equation and TD learning comes in through the backdoor: we can learn the V-value, and then use it as a bias in policy gradient.

Deep Reinforcement Learning: Lunar Lander and other games

actions

Aim: land between poles





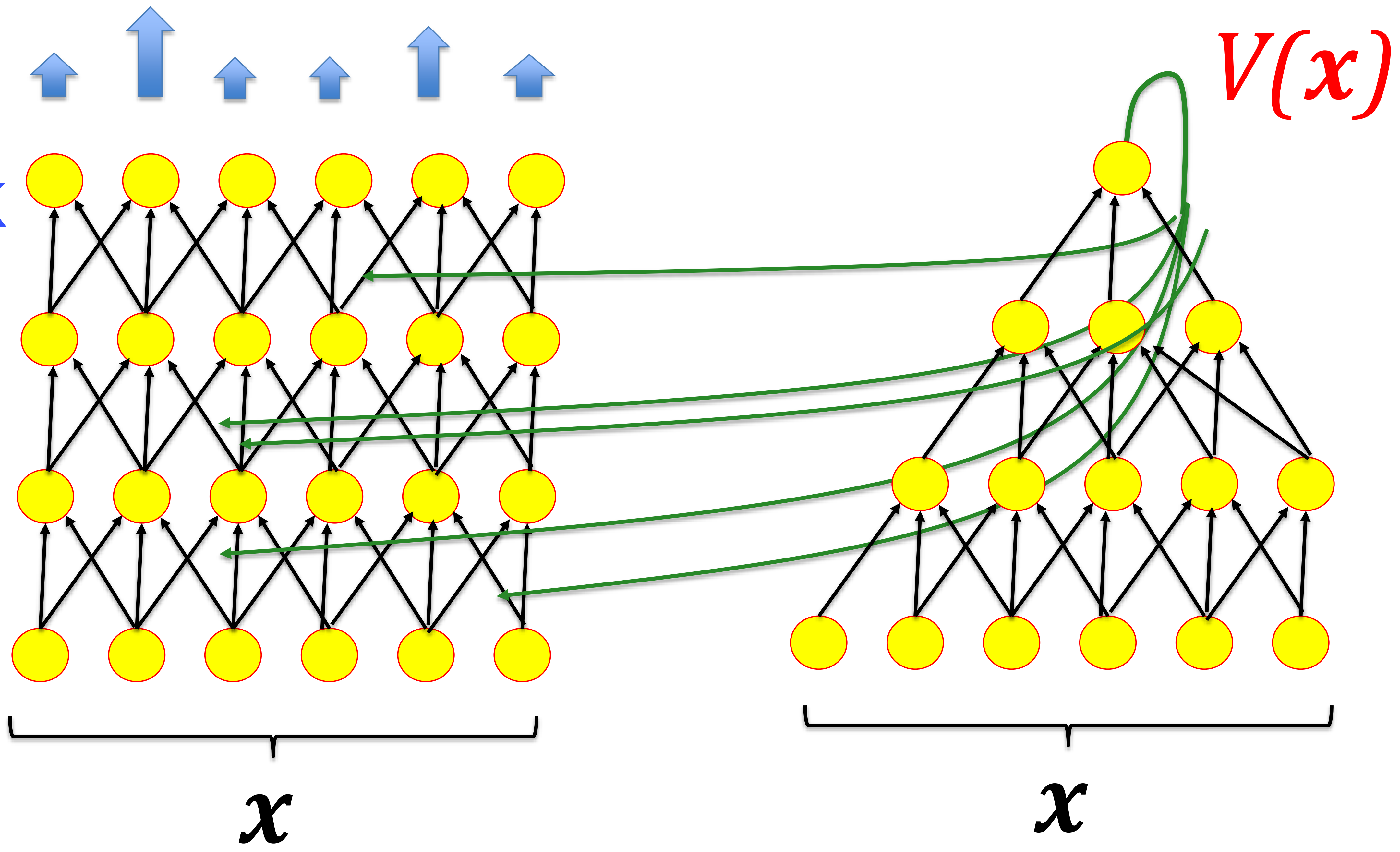
(previous slide)
And the value can for example be estimated (=learned) in a separate network.

Learning two Neural Networks: actor and value

Actions:
-Learned by
Policy gradient
- Uses $V(\boldsymbol{x})$ as baseline

Value function:
- Estimated by Monte-Carlo
-provides baseline $b=V(\boldsymbol{x})$
for action learning

Parameters
are the network
weights θ



Parameters
are the weights w

$\boldsymbol{x}$ = states from
episode:
$s_t, s_{t+1}, s_{t+2},$

(previous slide)
In the latter case we have two networks:

The actor network learns a first set of parameters, called θ in the algorithm of Sutton and Barto.
The value network learns a second set of parameters, with the label w .

The value $b(x = s_{t+n}) = V(\boldsymbol{x})$ is the estimated total accumulated discounted reward of an episode starting at $x = s_{t+n}$

The total accumulated discounted ACTUAL reward in ONE episode is $R_{s_{t+n} \to s_{end}}^{a_{t+n}}$

'REINFORCE' with baseline

From book:
Sutton and Barto, 2018

REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, r_1 \ldots, S_{T-1}, A_{T-1}, r_T$ following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
    Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$ (G_t)
        $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \gamma^t \delta \nabla \hat{v}(S_t,\mathbf{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

(previous slide)
Algorithm in pseudocode taken from the book of Sutton and Barto.
For the actor, the algorithm evaluates terms of the form

$$[R_{s_{t+n}\to s_{end}}^{a_{t+n}} - b(s_{t+n})] \frac{d}{d\theta_j} \ln[\pi(a_{t+n}|s_{t+n}, \theta)]$$

Where the return is $G = R_{s_{t+n}\to s_{end}}^{a_{t+n}}$

And the bias estimate is $\mathrm{v}(s_{t+n}) = b(s_{t+n})$

The terminal state in their notation occurs at time T and
the initial state has index 0.

For the value function, they use Monte-Carlo estimation of the total accumulated reward in one episode (see last week).

Why subtract the mean?

Subtracting the expectation provides estimates that have (normally) smaller variance (look less noisy)

Note: in multi-step RL, the minimal variance is not exactly at bias=expection.

Reason: correlations

(previous slide)
Why is it useful to subtract the mean?

Whatever the choice of baseline, the algorithm should eventually converge to the same set of parameters. However, since the algorithm is based on stochastic gradient descent (i.e., the online rule instead of the full batch rule), the algorithm makes **noisy steps** that only go on average in the right direction.

Subtracting a baseline that is close to the mean generally reduces the noise.
The example with a product of independent variables shows that by subtracting the mean of x, the noise is considerable reduced in each of the samples! (Exercise 2)

Unfortunately, in a multi-step reinforcement learning scenario, the minimal noise is not exactly the situation where one subtracts the mean because of correlations, but it is close to it.

(your calculations)

Outlook: Deep Reinforcement Learning

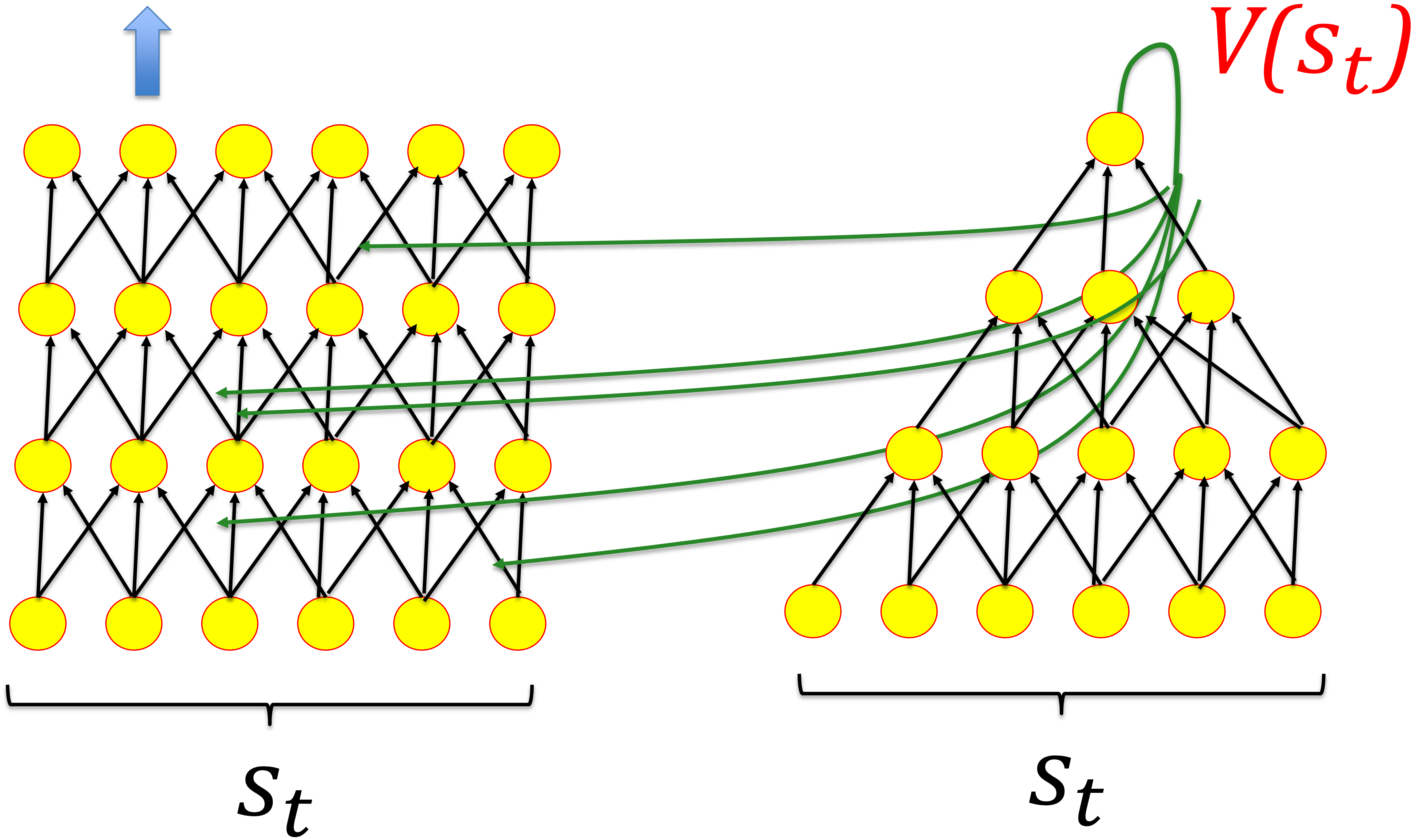
Policy gradient involves many terms of the form:

$$\frac{d}{d\theta_j}\ln[\pi(a_t|s_t,\theta)] \qquad [R^{a_t}_{s_t\rightarrow s_{end}} - (V(s_t))]$$

Output of network: policy
Parameters are the weights of the network.

Output of network:
value (baseline)

$\pi(a_t|s_t,\theta)$



Learning signal:
[actual Return - $V(s)$]

(previous slide)

Both actor and critic are optimized by changing the parameters according to a gradient descent rule.
Gradient descent in a multi-layer network is called BackProp or Deep Learning.

We start with Deep Learning next week.Algorithm in pseudocode taken from the book of Sutton and Barto.
For the actor, the algorithm evaluates terms of the form

Outlook: Deep reinforcement learning: alpha-zero

Network for choosing action

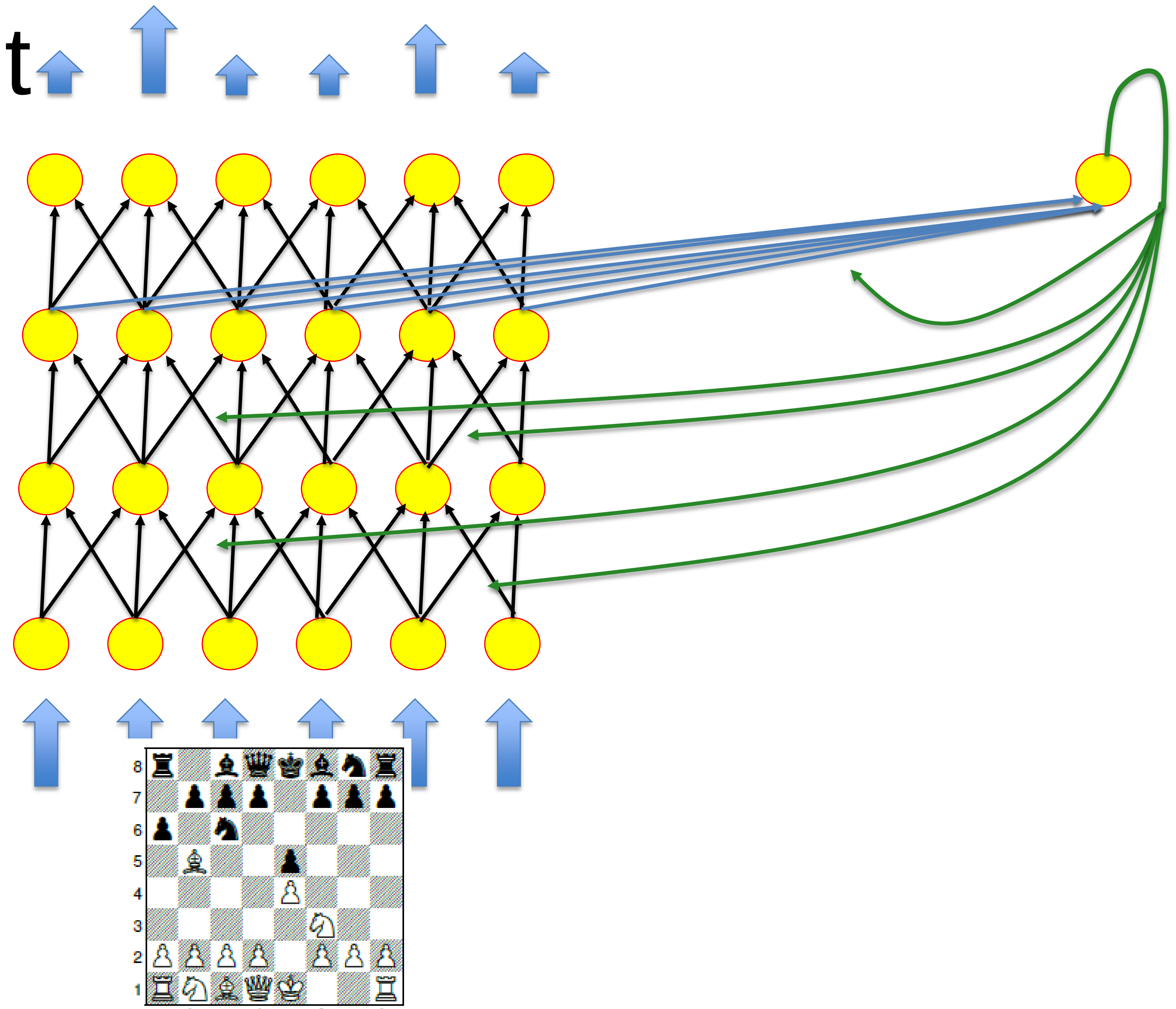
action:

Advance king

2^{e} output for **value** of state:

$V(s)$

output



input

learning:

→ change connections

aims:

- learn value $V(s)$ of position
- learn action policy to win

Learn V(s) with TD learning!

(previous slide)
Very schematically is this one of the ideas of deep reinforcement learning. We construct a deep network with multiple layers. We use the output units for action choice and optimize the parameters via policy gradient. We have a further output unit to estimate the V-value, and use it as a bias.

- The model of the V-value can share some units with the model of the actions
- The model of the V-value can be learned with tools from TD learning

This gives rise to actor-critic and ‘advantage actor critic’, to be discussed in the lecture on Deep Reinforcement Learning.

Quiz: Policy Gradient and Reinforcement learning

Your friend has followed over the weekend a tutorial in reinforcement learning and claims the following. Is she right?

[ ] Even some policy gradient algorithms use V-values

[ ] V-values for policy gradient are calculated in a separate network (but some parameters can be shared with the actor network)

Teaching monitoring – monitoring of understanding

The End

[ ] 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.