Reinforcement Learning Lecture 3

Wulfram Gerstner

EPFL, Lausanne, Switzerland

TD-learning in continuous space: function approximation

Part 1: Modeling the Input Space

1. Modeling the input space

Sutton and Barto, Reinforcement Learning (MIT Press, 2nd ed. 2018),
Chapters 9.3

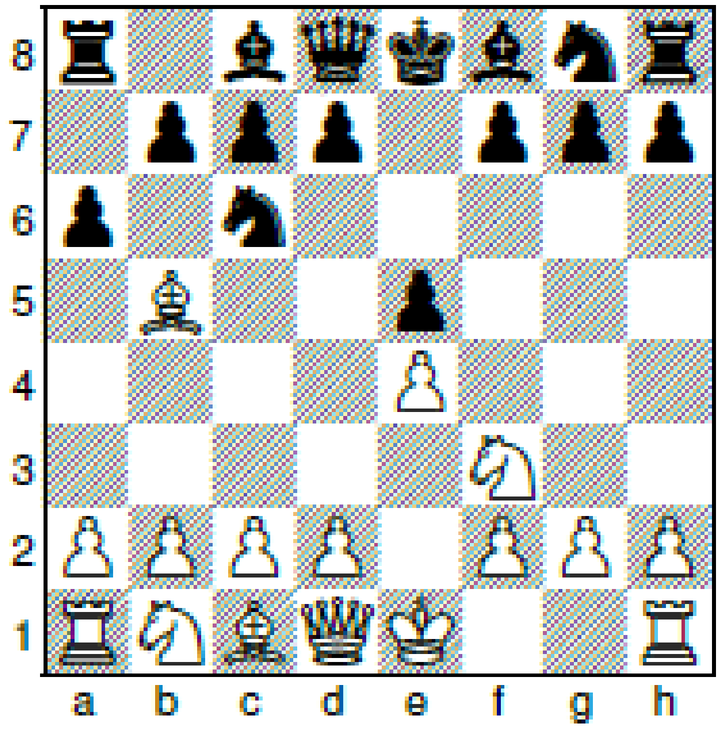
Background Reading:
Mnih et al. 2015, Nature, Vol 5018, doi:10.1038/nature14236
Tesauro 1995, https://www.csd.uwo.ca/~xling/cs346a/extra/tdgammon.pdf
Strosslin et al, 2005, Neural Networks 18: 1125–1140
Sheynikhovich et al, 2009, Psychological Review, 116:540

(previous slide)

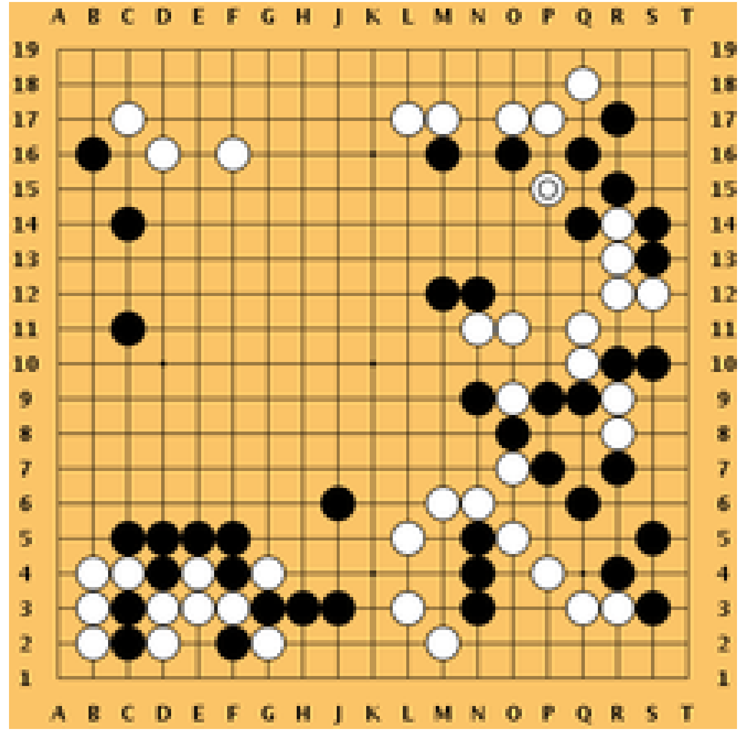
Continuous input spaces have a second problem: there are many Q-values are V-values that you need to compute.

Review: Deep reinforcement learning

Chess



Go

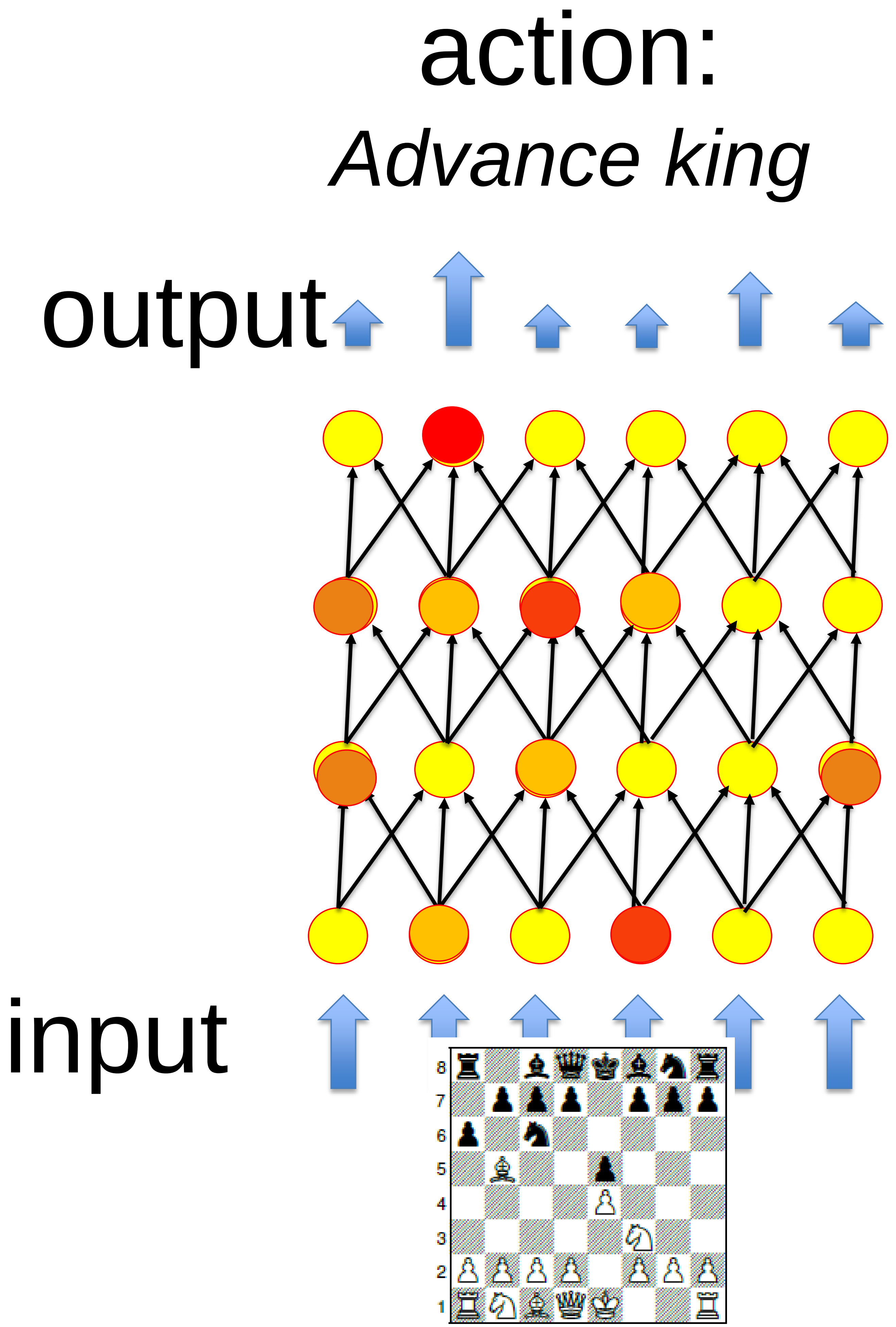


Artificial neural network (*AlphaZero*) discovers different strategies by playing against itself.

In Go, it beats Lee Sedol

Review: Deep reinforcement learning

Network for choosing action



Today

first steps toward learning action choices in a small network:

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

→ Continuous/Large State space

Problem of TD algorithms: representation of input

All algorithms so far are 'tabular':

Q-learning or SARSA:
→ build a table $Q(s,a)$ with entries
for all states s and actions a

TD-learning of V-values:
→ build a table $V(s)$ for all states s

discrete states and actions

→ many entries,
→ 'independent' (apart from self-consistency of Bellman)

(previous slide)

Two observations:

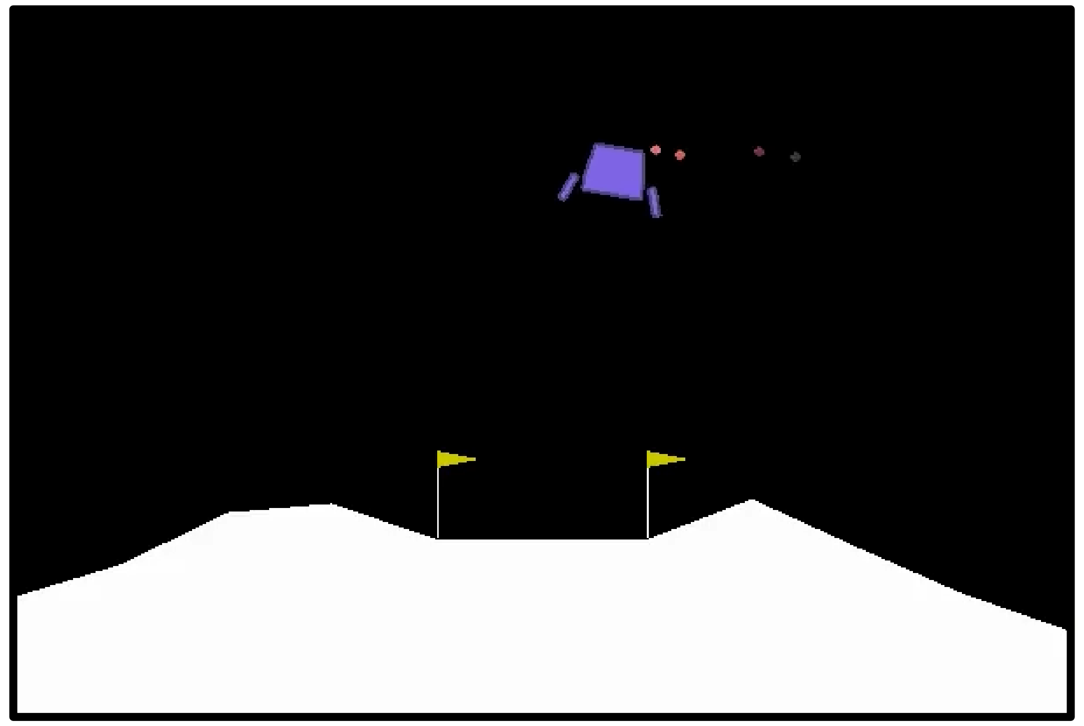
First, in a table all entries are independent – the only relation between Q-values or V-values arises from the self-consistency condition of the Bellman equation.

Second, there are many (!) entries.

Problem of TD algorithms: representation of input

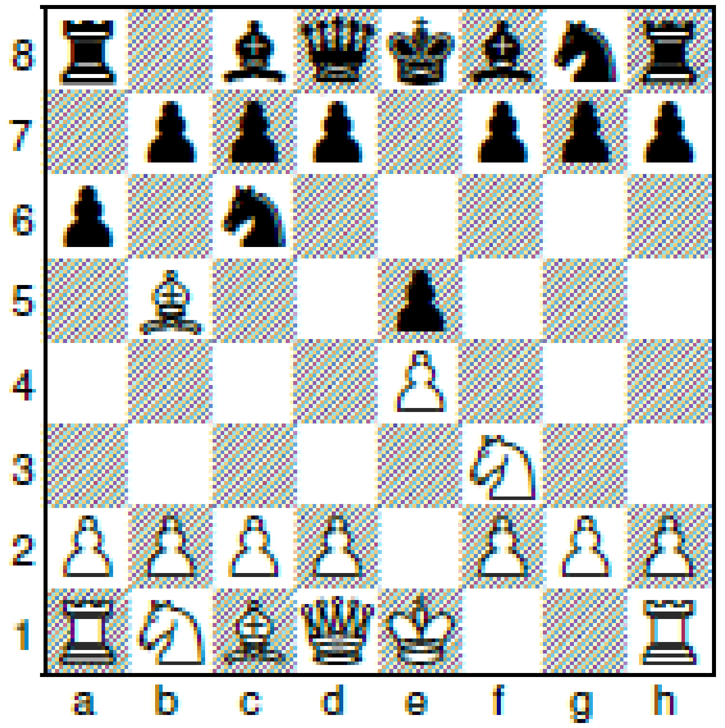
- for control problems, input space is naturally continuous

Moon lander
Aim: land between poles

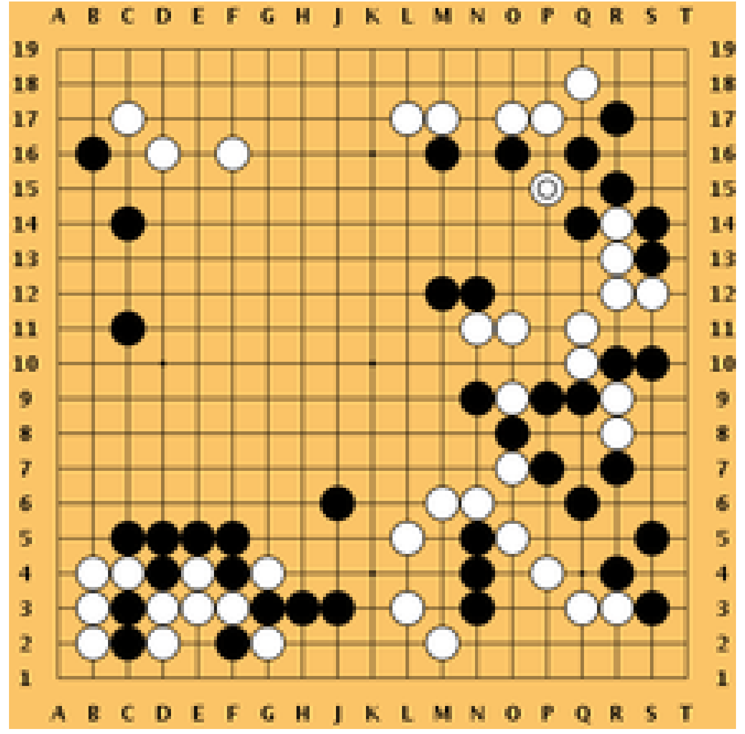


- for discrete games, the input space is often too big

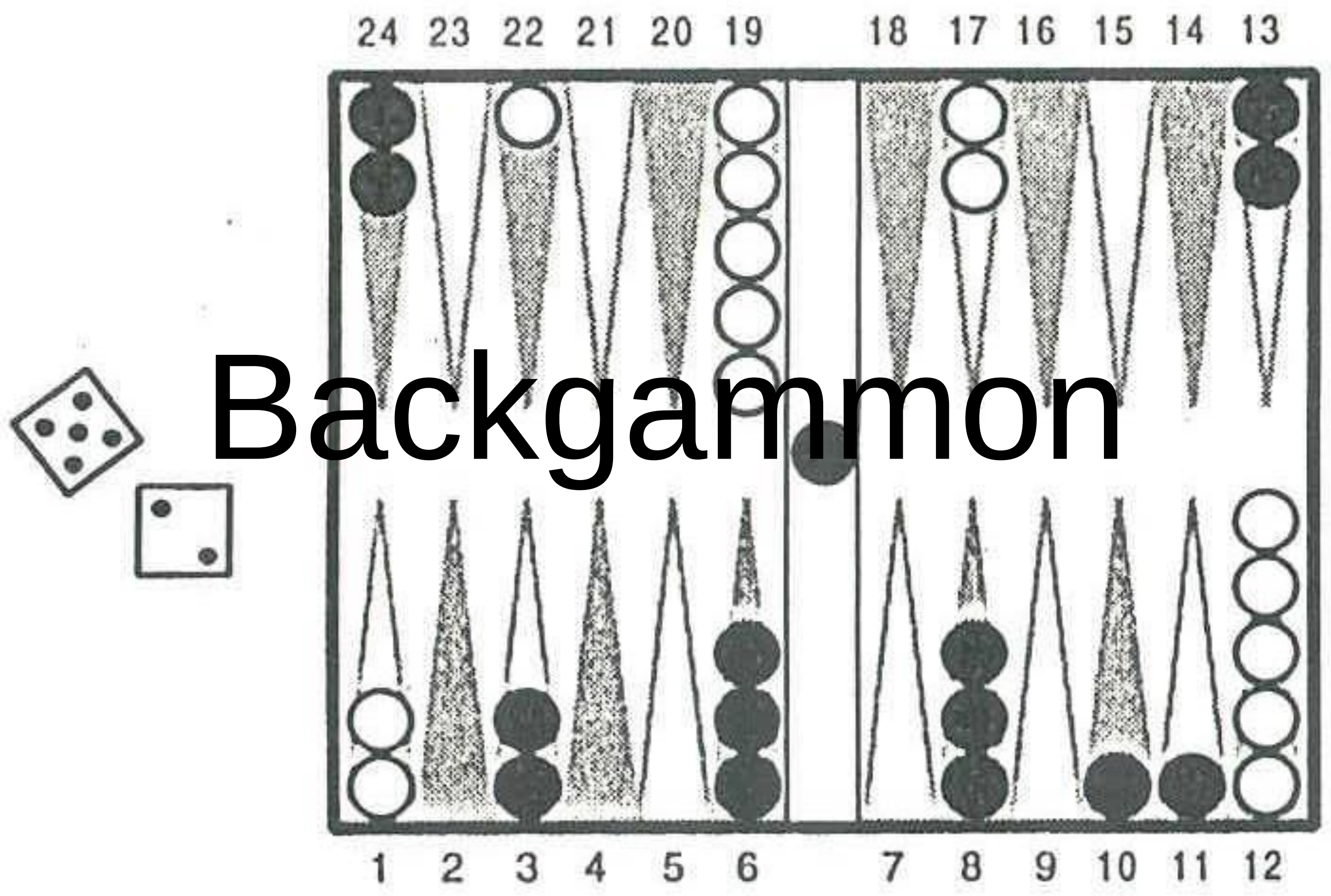
Chess



Go



Backgammon



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

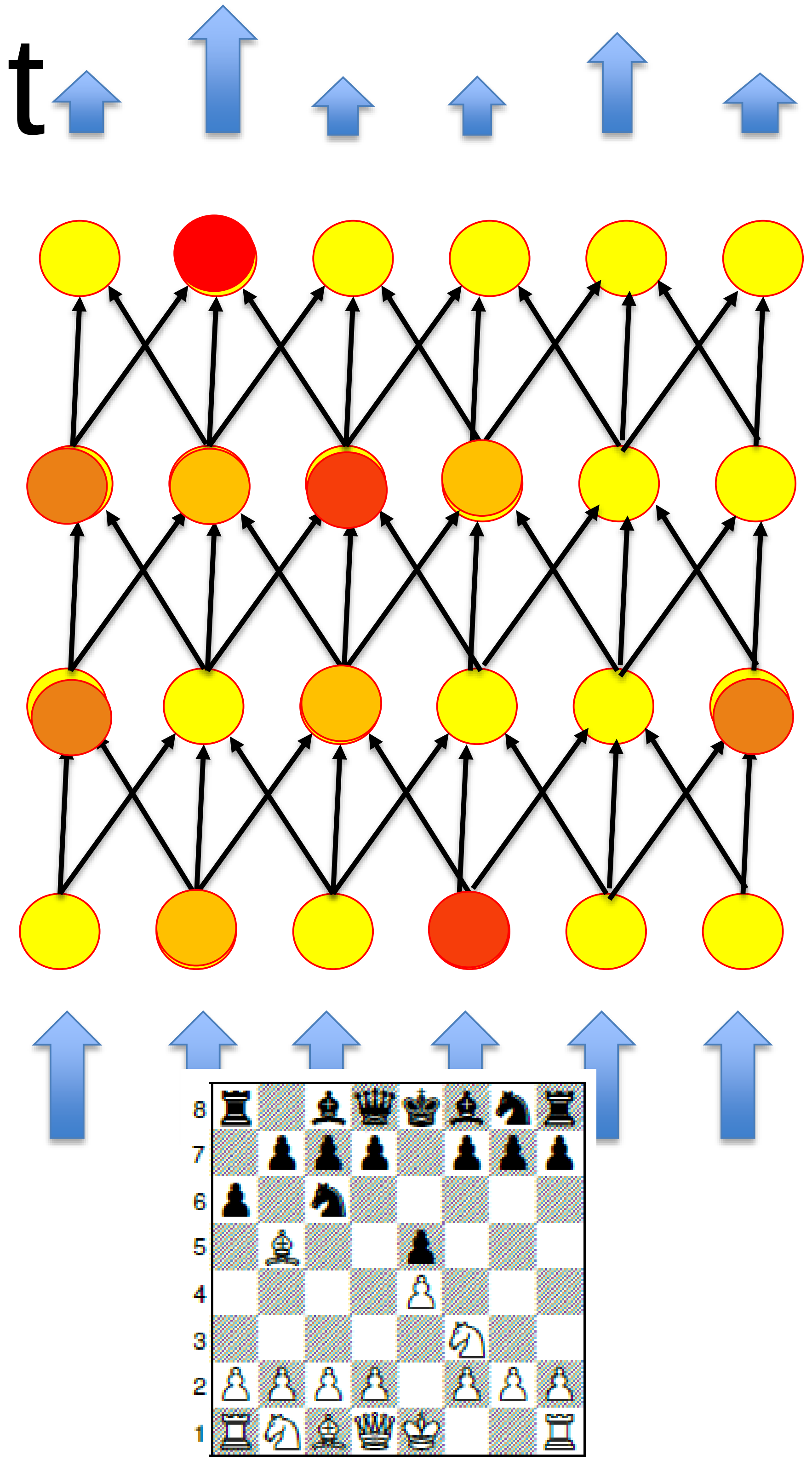
Solution: Neural Network to represent input configuration

Schematically (theory will follow):

action:

Advance king

output



input

action output units represent Q-values

learning:

- change connections

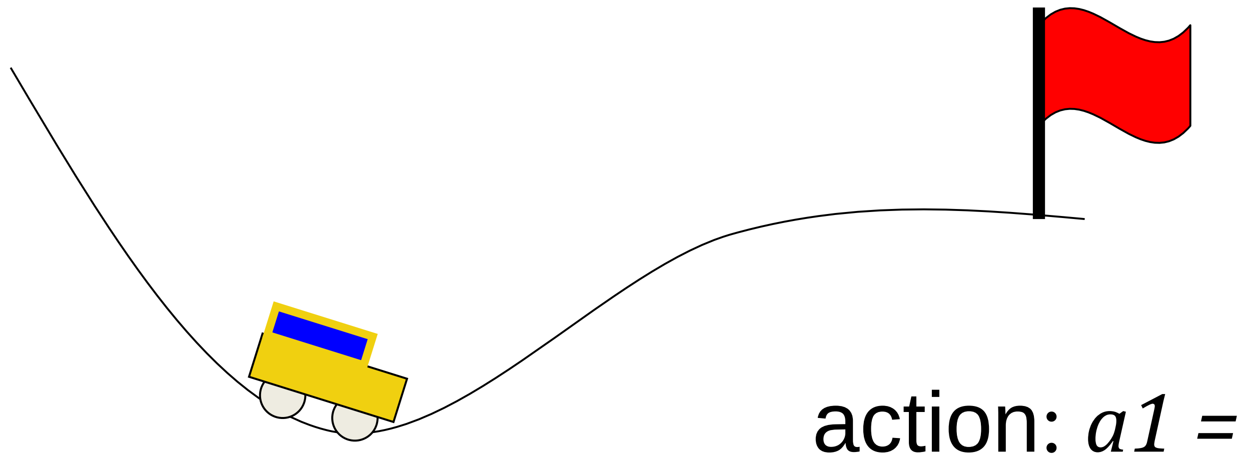
aim:

- Choose next action to win
- Optimize return

  = probability to win

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

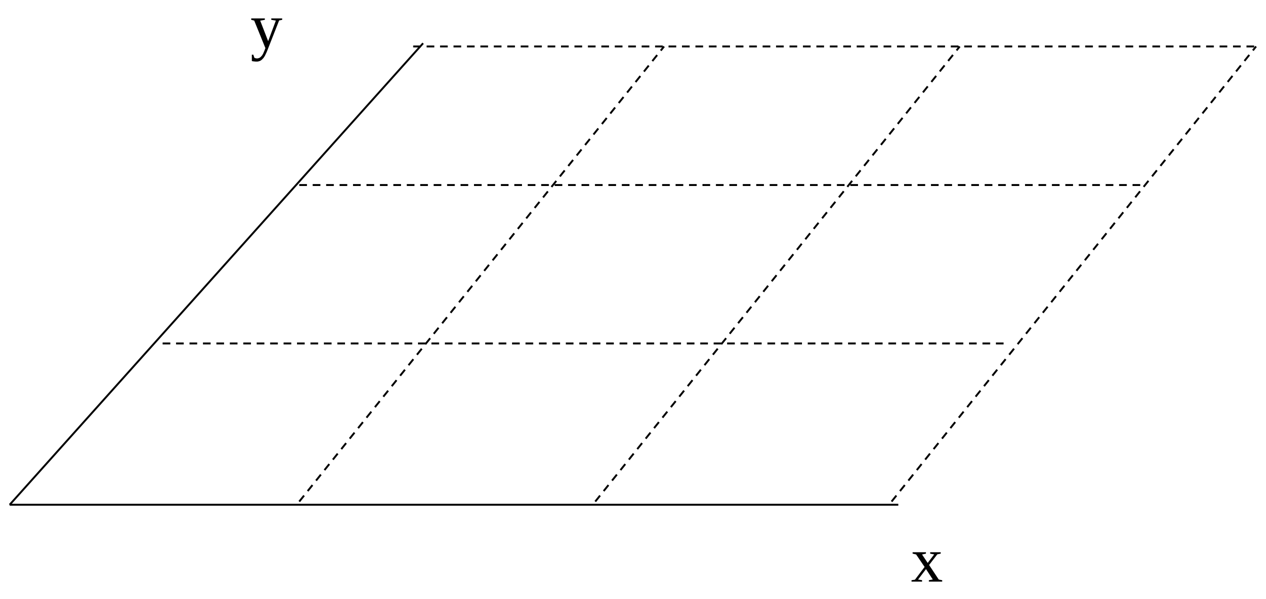
Solution: Continuous input representation



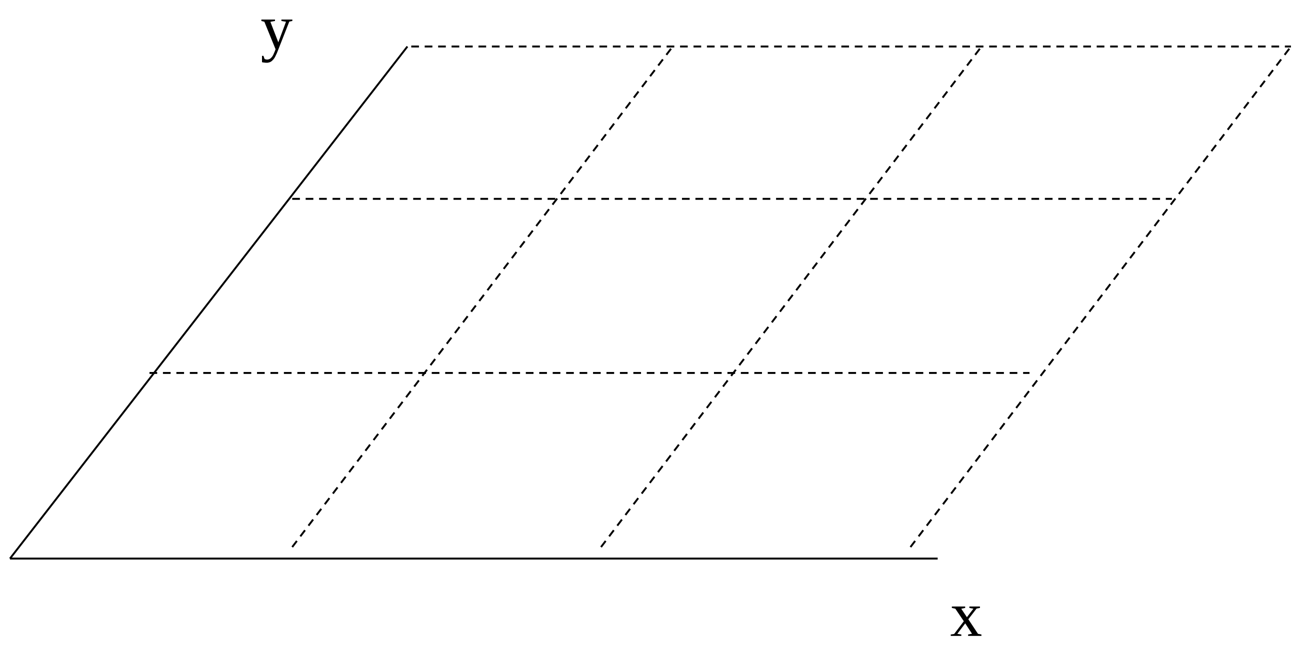
Example: Mountain Car

action: *a1 = right*
a2 = left

for action *a1*



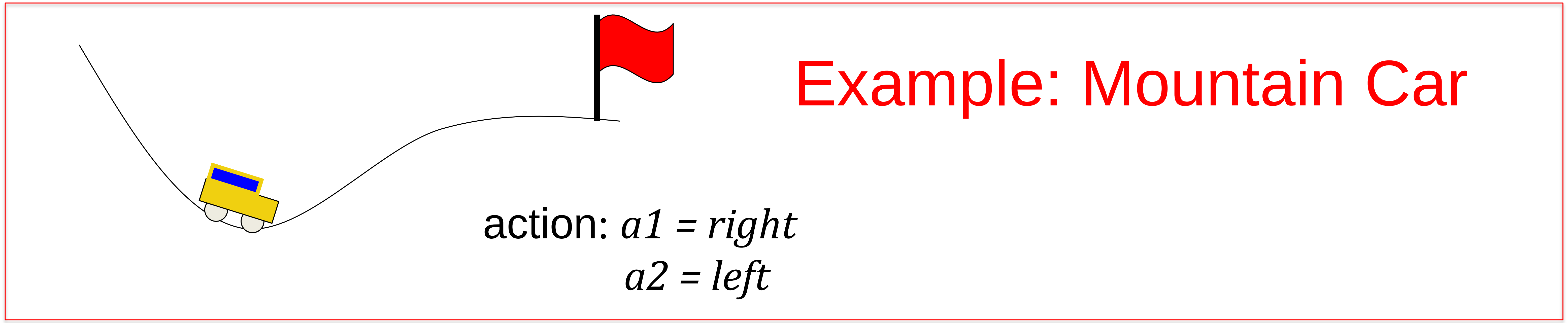
for action *a2*



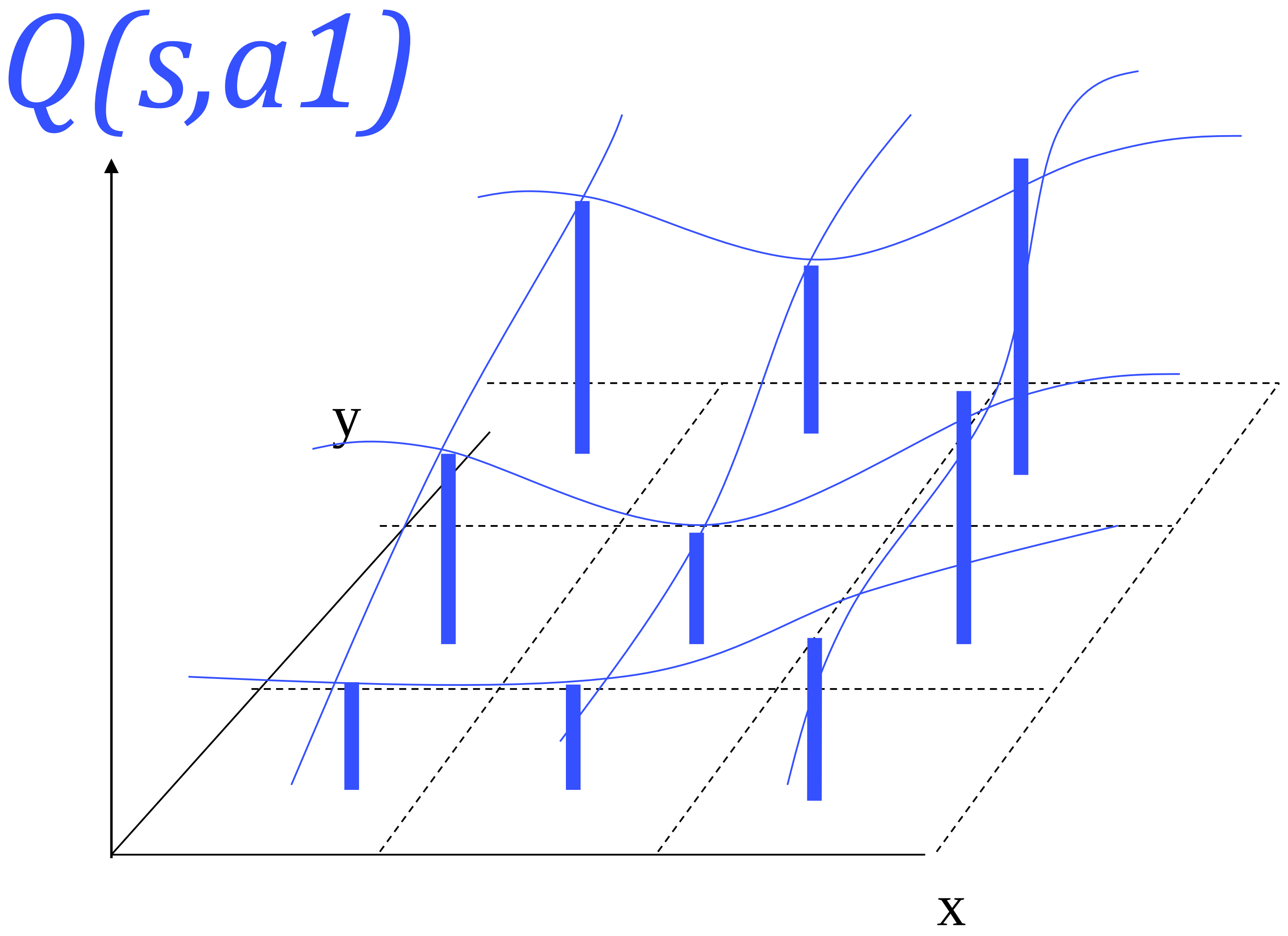
(previous slide)
In the mountain car task, the input space is two dimensionals: the position x and the speed.
Suppose both dimensions are discretized into 3 values. The Q-values therefore have 9 entries for action a1 (force to the left) and 9 further entries for action a2 (force to the right).

Solution: Continuous input representation

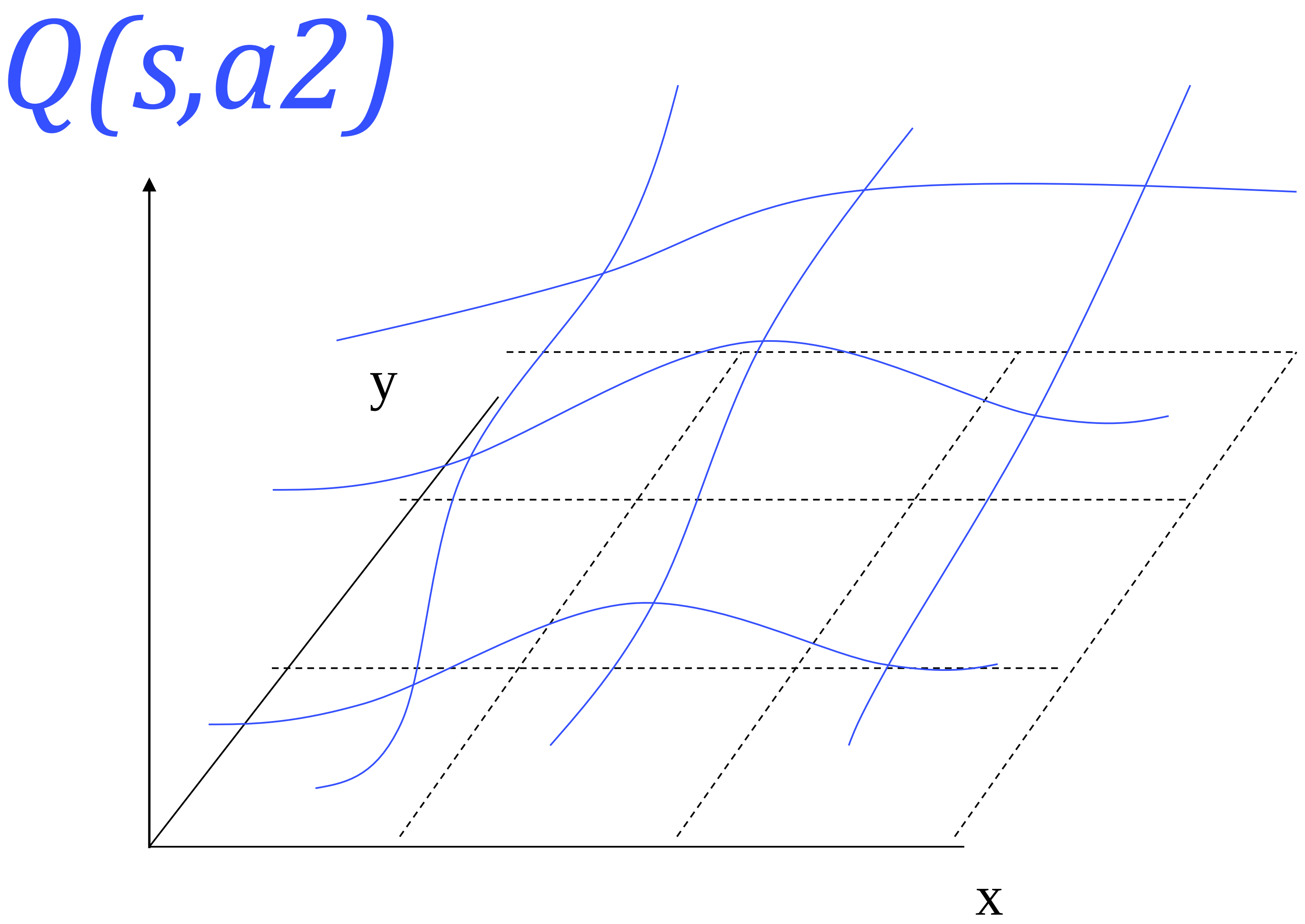
Blackboard 3: Radial Basis functions



for action $a1$



for action $a2$



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

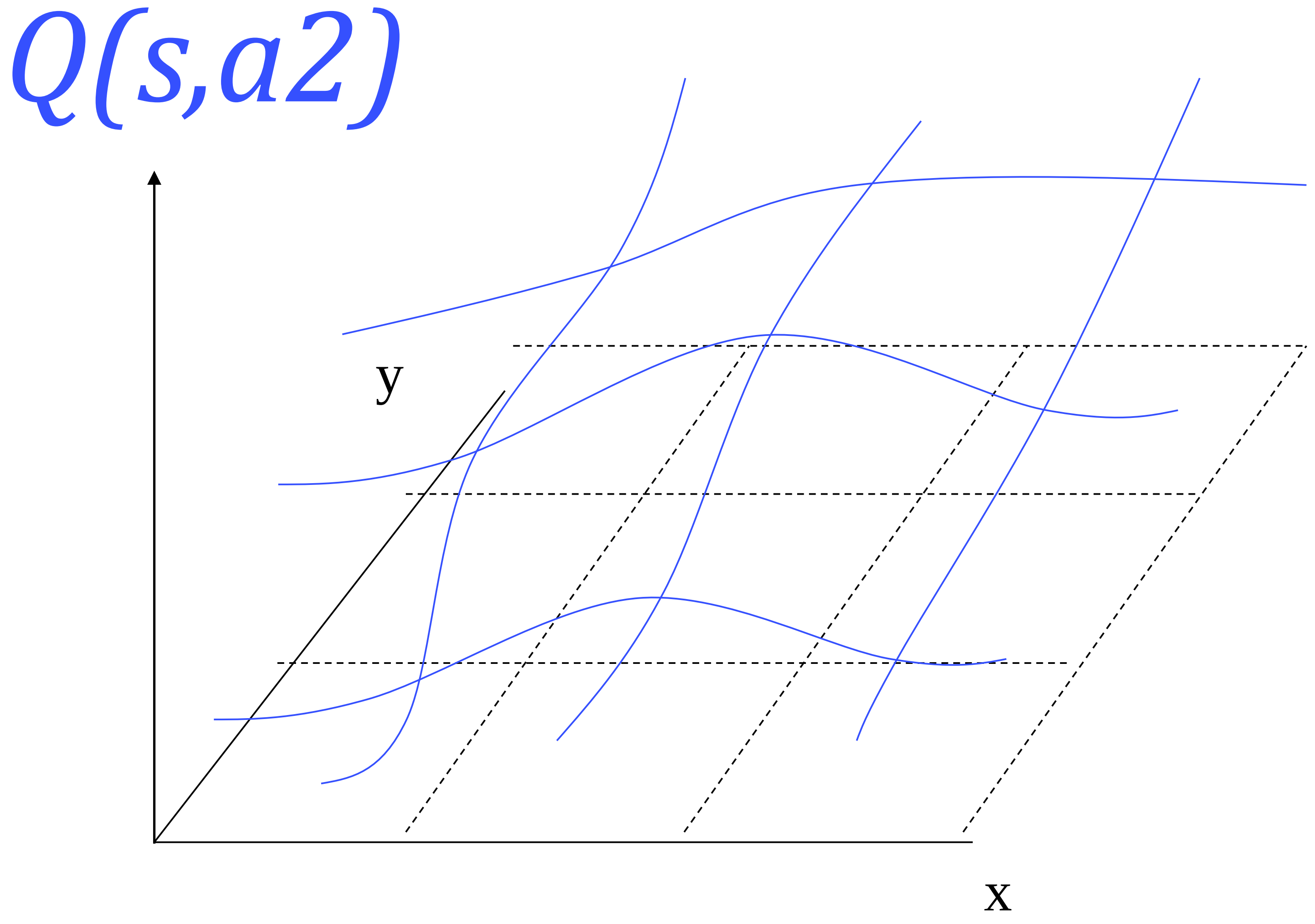
A first advantage is, that the question of discretization of the input space has now disappeared, since we can model the Q-values as a function of the continuous state variable s=(x,y).

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

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

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

Blackboard 3: Radial Basis functions



Summary:

If input space is continuous (or discrete but large),
we model for each action a’ the Q-values as a function
of the input state s

$$\mathrm{Q}(s, a'|\boldsymbol{\theta})$$

This function has parameters $\boldsymbol{\theta}$, i.e., the weights $\boldsymbol{w}$.

Good Q-values imply a good choice of the parameters $\boldsymbol{w}$.

Reinforcement Learning Lecture 3

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Continuous input space: function approximation

Part 2: Loss Function and Semi-Gradient

1. Modeling the input space
2. **Loss Function and Semi-Gradient for SARSA**

Previous slide:
Now we have a function that maps the state-space to Q-values.
This function depends on parameters.
How can we learn these parameters? Via minimization of a Loss Function.

From Bellman equation to Error function.

Consistency condition of Bellman Eq.

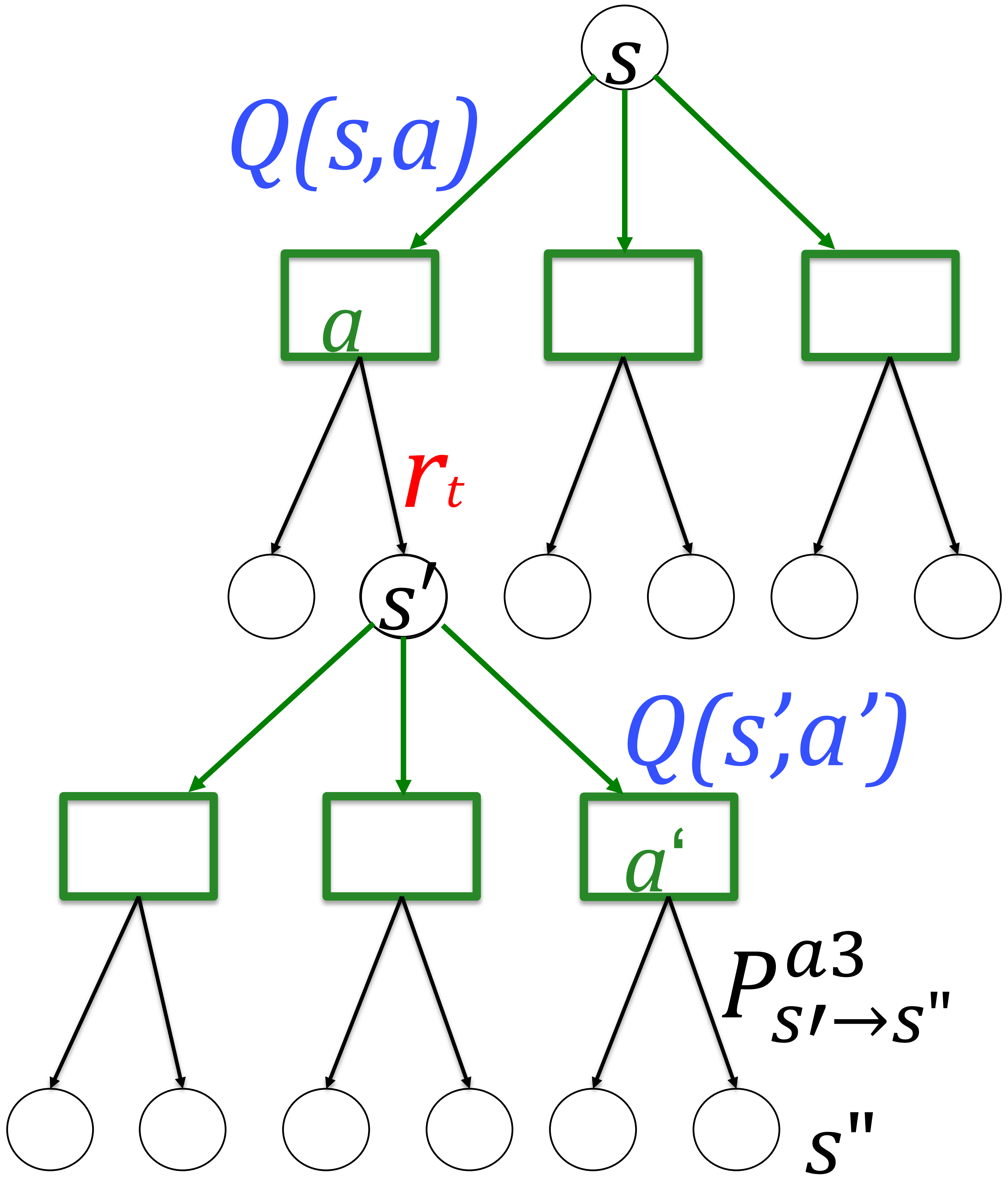
$$Q(s,a) = \sum_{s'} P^{a}_{s\to s'} \left[R^{a}_{s\to s'} + \gamma \sum_{a'} \pi(s',a') Q(s',a') \right]$$

On-line consistency condition

(should hold on average)

$$Q(s,a) = \overbrace{r_t + \gamma\, Q(s',a')}^{\text{target}}$$

yields online Error function (loss)



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

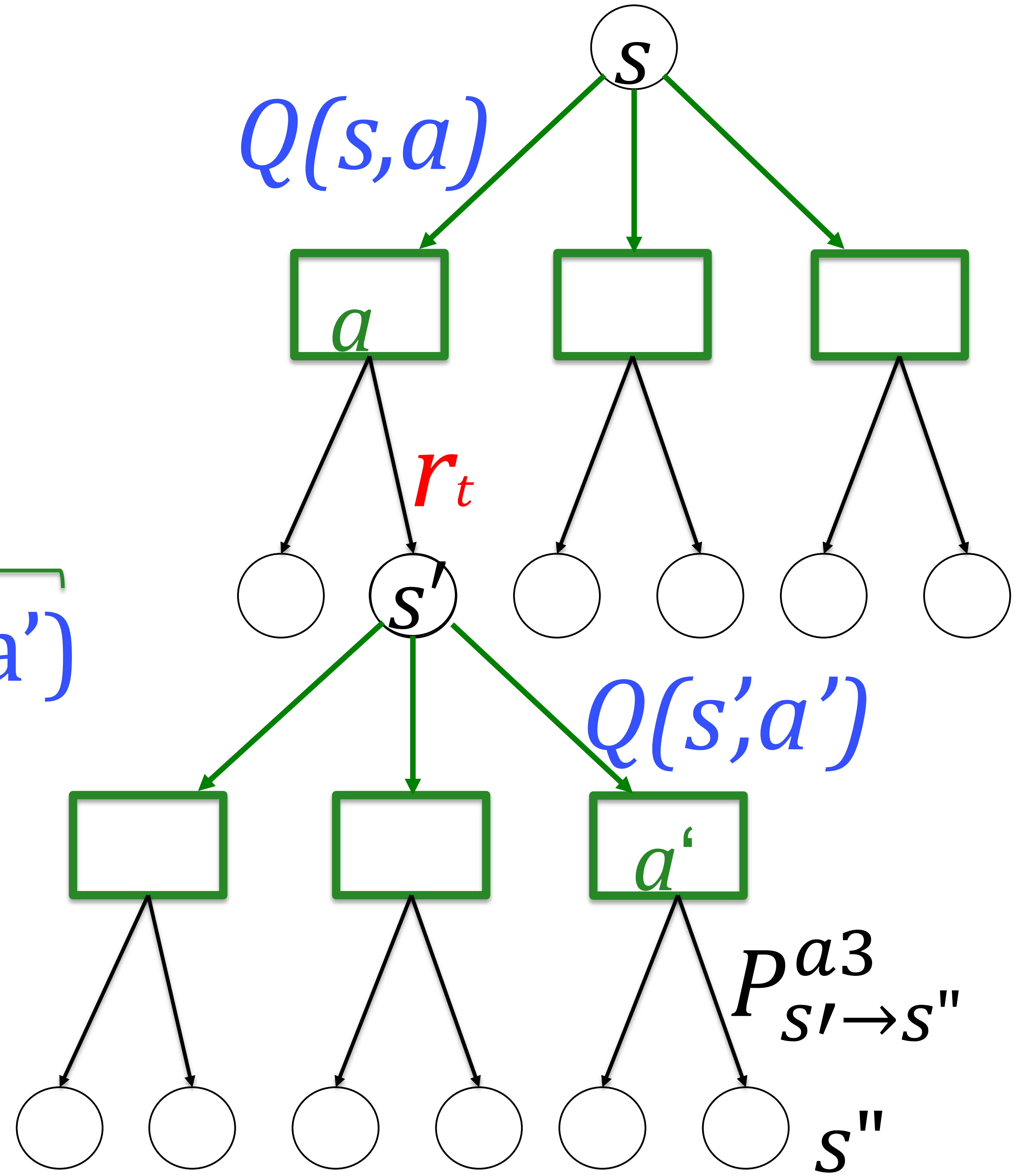
$$Q(s,a) = r + \gamma\, Q(s',a')$$

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

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

Blackboard 4: Error function

$$Q(s,a) \leftarrow \overbrace{r + \gamma\, Q(s',a')}^{\text{target}}$$



your notes

Error function: full gradient and semi-gradient

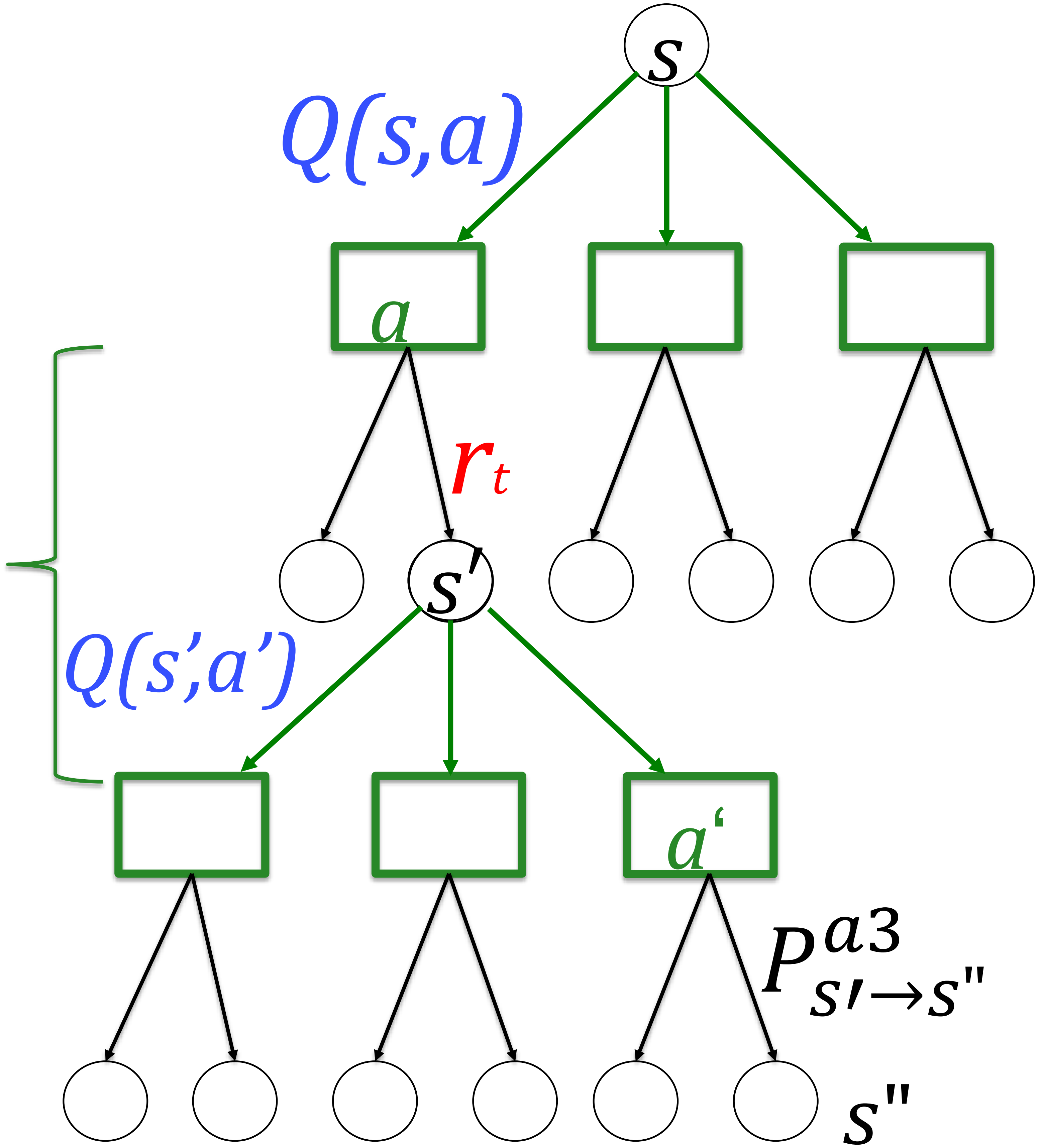
Blackboard 4: Gradient

Discrete time steps: s,a → s',a'

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

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

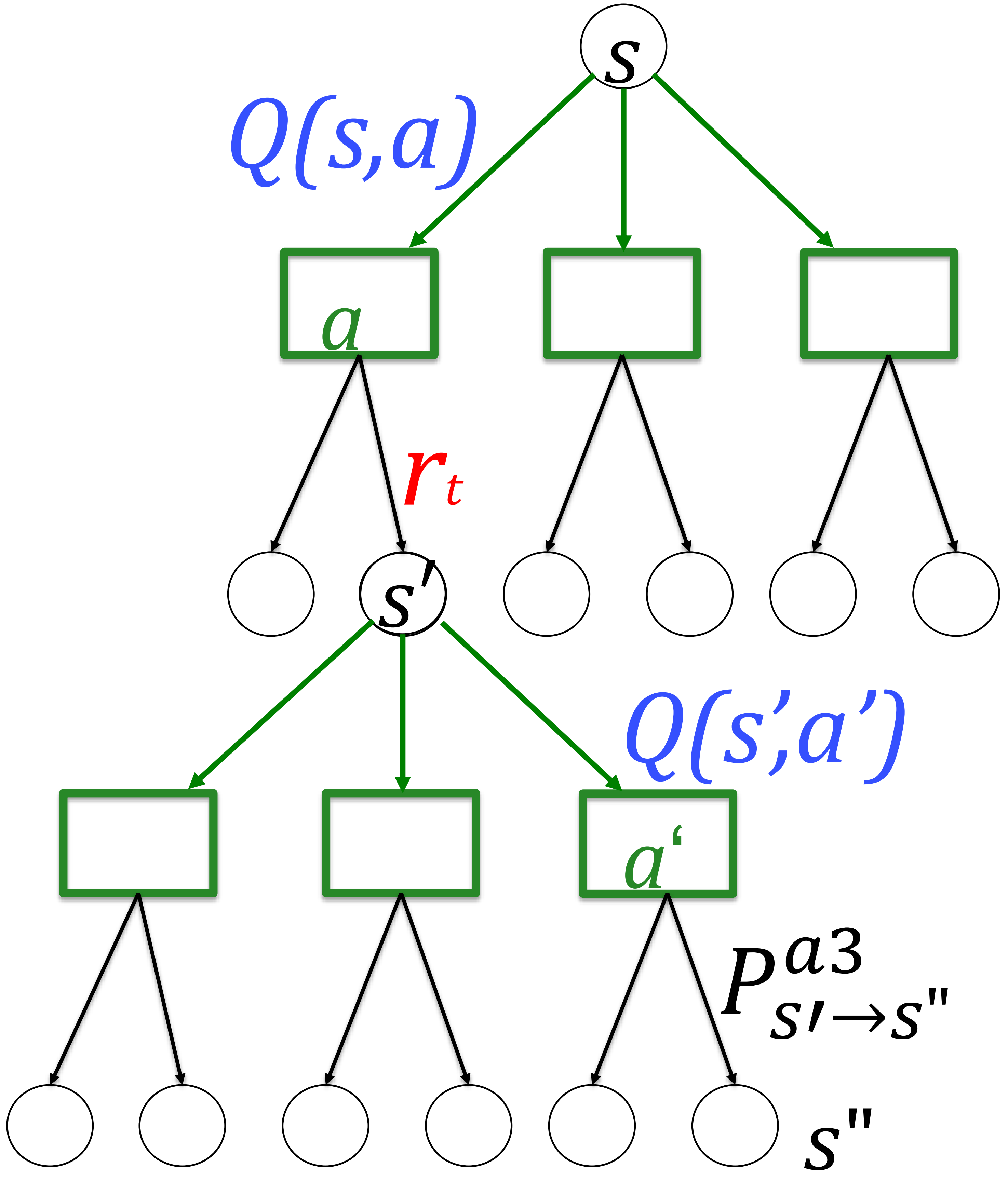
$$Q(s,a) \leftarrow \overbrace{r + \gamma\, Q(s',a')}^{\text{target}}$$



Summary: From Bellman equation to Error function.

Consistency condition of Bellman Eq.

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



On-line consistency condition

(should hold on average)

$$Q(s,a) = \overbrace{r_t + \gamma\, Q(s',a')}^{\text{target}}$$

yields online Error function (loss)

$$E(\boldsymbol{w}) = \frac{1}{2}\Big[\overbrace{r_t + \gamma Q(s',a'|\cancel{\boldsymbol{w}})}^{\text{target}} - Q(s,a|\boldsymbol{w})\Big]^2$$

'semi-gradient' ignore

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

Summary of Error function: full gradient and semi-gradient

Discrete time steps: s,a → s’,a’

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

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

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

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

(This is a heuristic trick to stabilize learning)

Calculate semi-gradient: relation to TD error

Discrete time steps: s,a → s’,a’

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

take gradient w.r.t. this ***w***

$$\frac{d}{d\boldsymbol{w}}E(\boldsymbol{w}) = \underbrace{[r_t + \gamma Q(s',a'|\boldsymbol{w})\text{- } \mathrm{Q}(s,a|\boldsymbol{w})]}_{\text{TD error}}\frac{d}{d\boldsymbol{w}}Q(s,a|\boldsymbol{w})$$

TD-error controls the ‘direction’ and ‘amount’ of update

Implement semi-gradient with SG operator

Discrete time steps: s,a → s’,a’

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

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

$\mathbf{SG}$ = ‘StopGradient’ Operator

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

Take standard derivative (normal definition), but on a loss function that contains the StopGradient Opeartor

(previous slide)

In implementations of standard optimization packages, we have to tell that some parts of the loss should be ignored when calculating the derivative.
This is sometimes written with a StopGradient operator SG{ . }

Summary: Function approximation

Continuous state space

→ use a function with parameters $\boldsymbol{w}$ (or generally $\boldsymbol{\theta}$) to model Q-values and generalize to unseen parts of state space

→ Learn parameters with Loss Function/Semi Gradient

→ Loss function implements consistency condition of Bellman eq.

→ Loss function can also be used to train a deep neural network with Q-values as output variables

(previous slide)

With a neural network (or other function approximation methods), we can work with a continuous state space. We define a loss function, linked to the consistency of the Bellman equation.

For optimization we use the semi-gradient.
The symbol ‘SG’ stands for ‘Stop Gradient’ and is available in several implementation packages of deep learning.

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

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

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

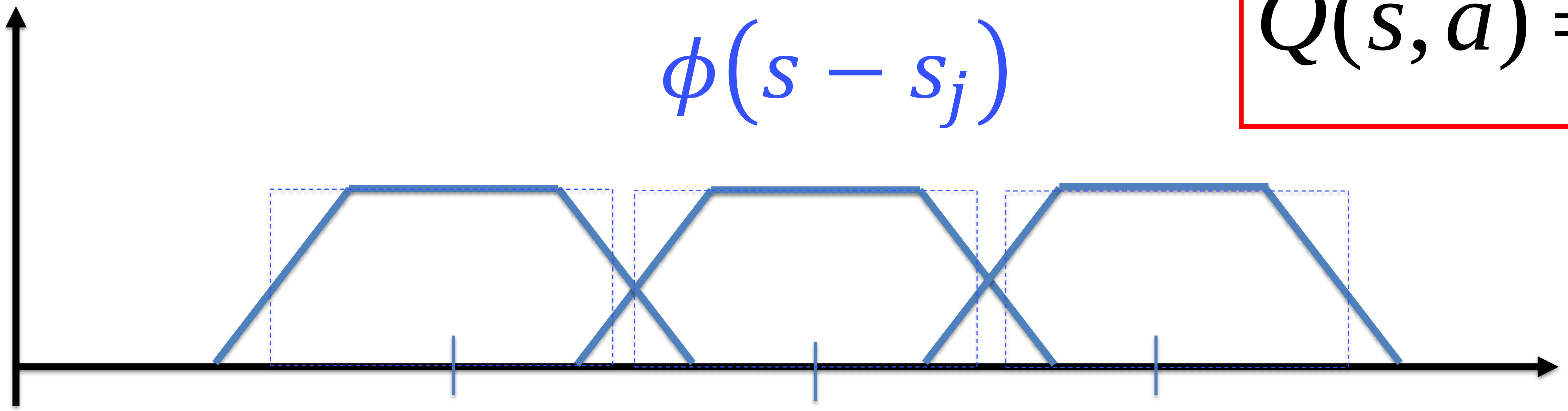
Teaching monitoring – monitoring of understanding

[ ] 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.

Theorem: Semi-Gradient versus Full Gradient (Exercise)

Continuous state space represented by localized basis functions:



$\phi(s - s_j)$

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

Claim: (1) If basis functions $\phi(s - s_j)$ become sharp and non-overlapping rectangles, function approximation turns into tabular TD-learning.

(2) In this limit, **Semi-Gradient yields SARSA** whereas Full Gradient does not!

(previous slide)

This is shown in the Exercise session.

Hence, semi-gradient is not just heuristics, but systematic in the sense that it relates to algorithms for which we know that they are consistent with the Bellman equation.

Reinforcement Learning Lecture 3

Wulfram Gerstner

EPFL, Lausanne, Switzerland

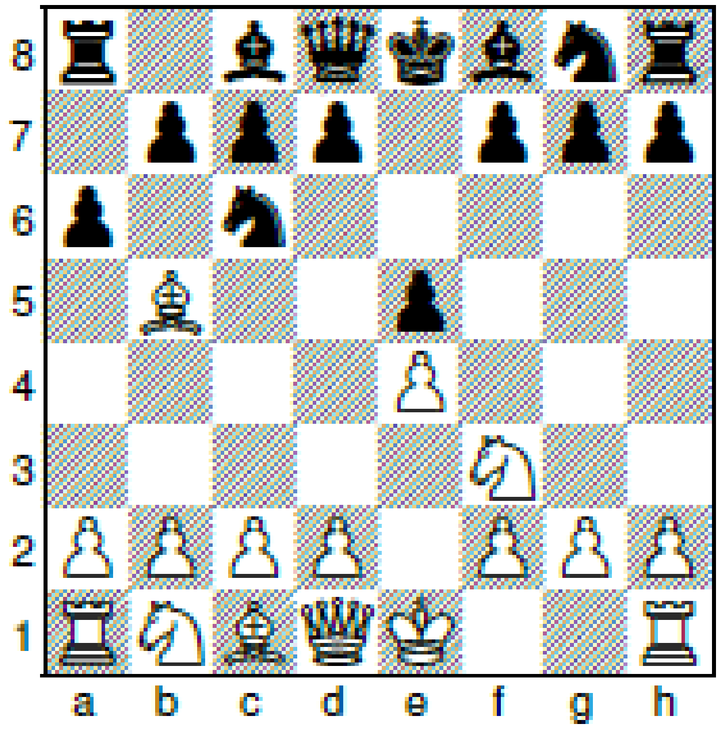
Continuous input space: function approximation

Part 3: Application to Artificial Neural Networks

1. Modeling the input space
2. Loss Function and Semi-Gradient for SARSA
3. **Application to Artificial Neural Networks**

First steps toward Deep reinforcement learning

Chess



Backgammon

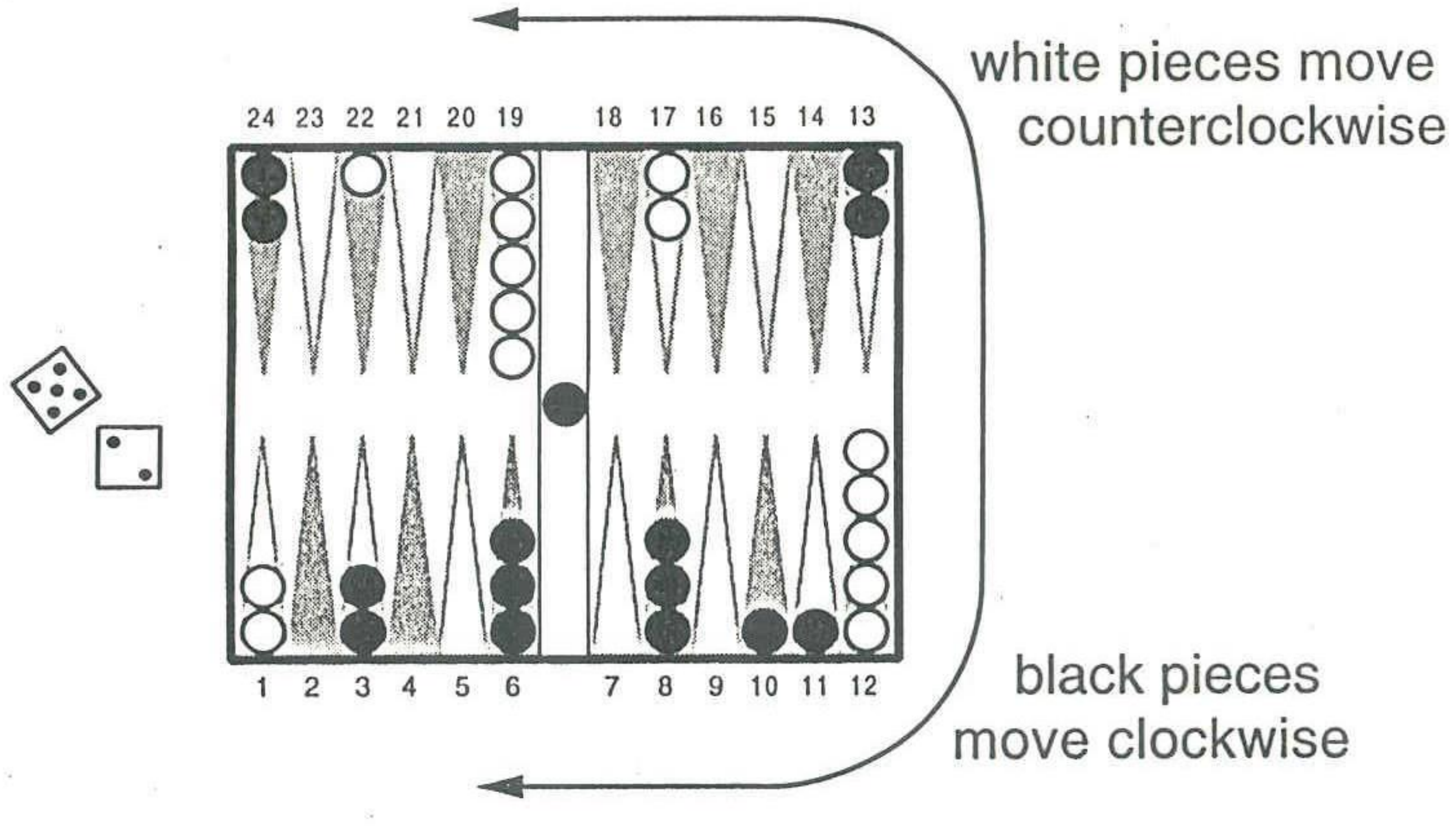
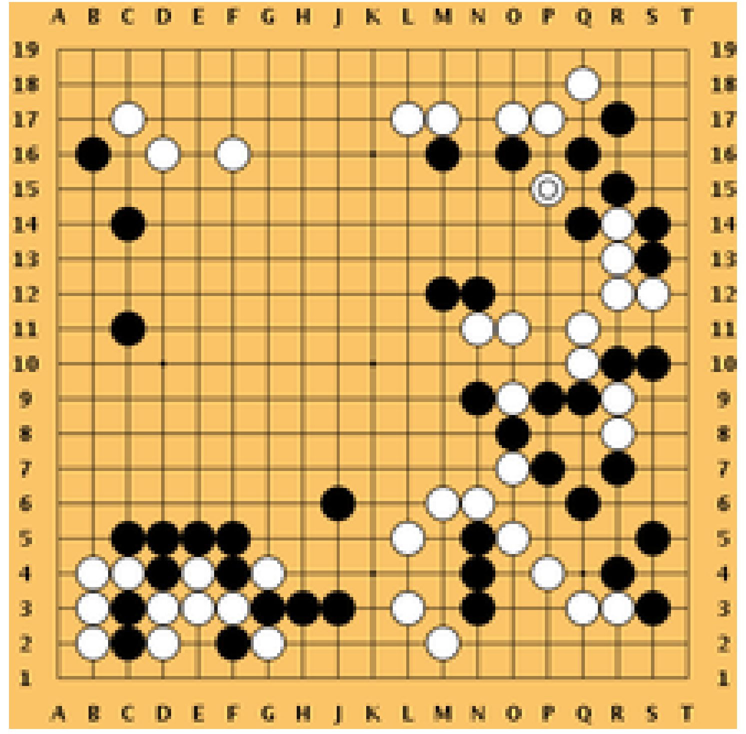

white pieces move counterclockwise
24 23 22 21 20 19
18 17 16 15 14 13
1 2 3 4 5 6
7 8 9 10 11 12
black pieces move clockwise

Figure 11.1 A backgammon position.

Go



since 1992 …

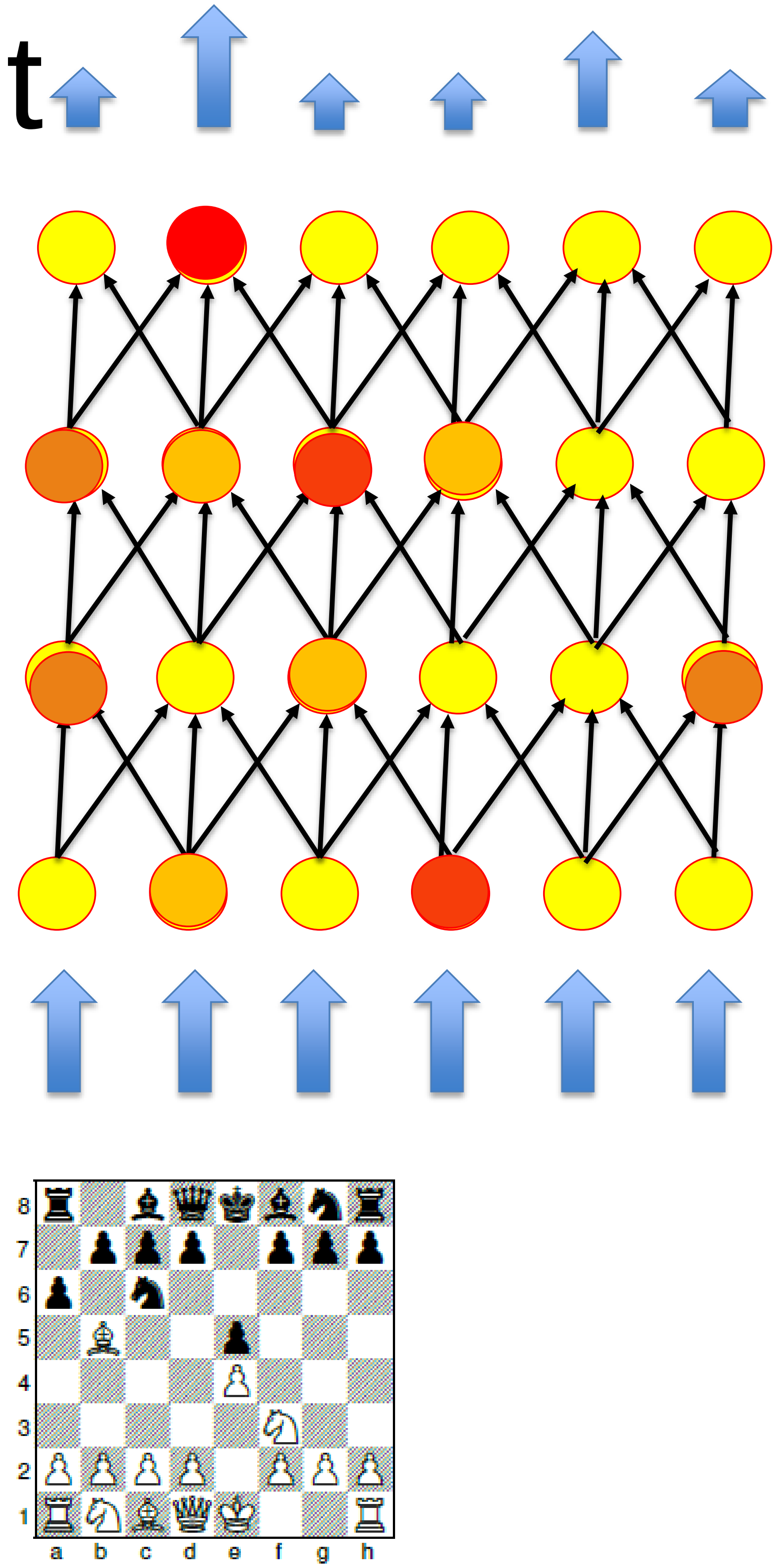
started 2015 (?)

Backprop for deep Q-learning

action and Q-values:

Advance king

output



input

Outputs are Q-values
→ actions are easy to choose

For example:
Softmax strategy: take action a'
with prob

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

(previous slide)
Last week we have seen that we can model Q-values in continuous state space as a function of the state s, and parameterized with weights w.

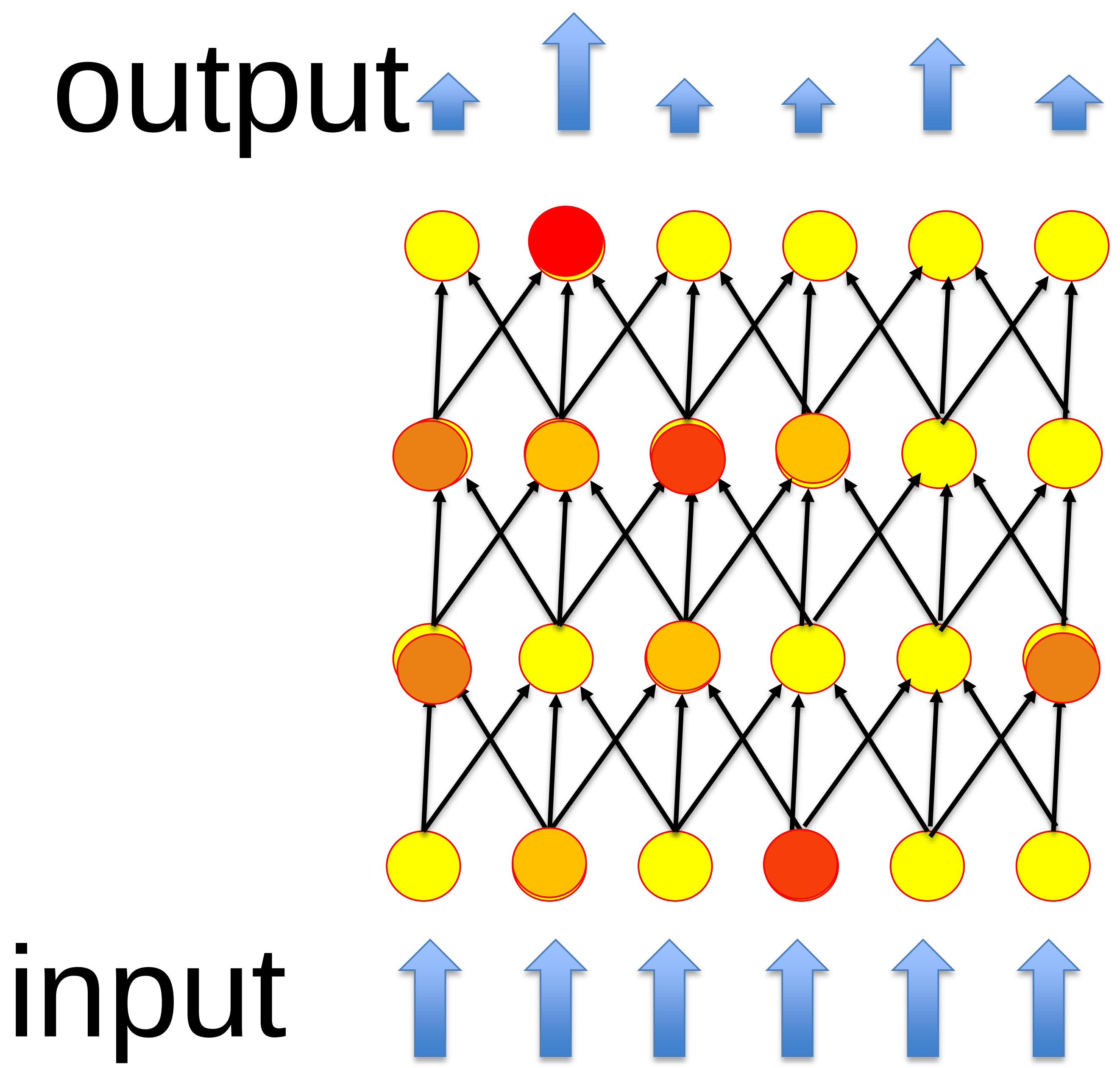
But in fact, a model of Q-values also works when the input space is discrete, such as it is in chess. Suppose that each output corresponds to one action (e.g. one type of move in chess).
We can use a neural network where the output are the Q-values of the different actions while the input represents the current state s.

Thus, an output unit n represents $Q(a_n,s)$.

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 playing against itself.

Error function for SARSA

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

Error function for Q-learning

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

(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.

We already discussed 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 (semi-)gradient descent on the error function. This leads to the Backpropagation algorithm of ‘Deep learning’).

Review Q-values and V-Values

expected total discounted reward starting in s with action a_1

$Q(s, a_1)$

expected total discounted reward starting in s

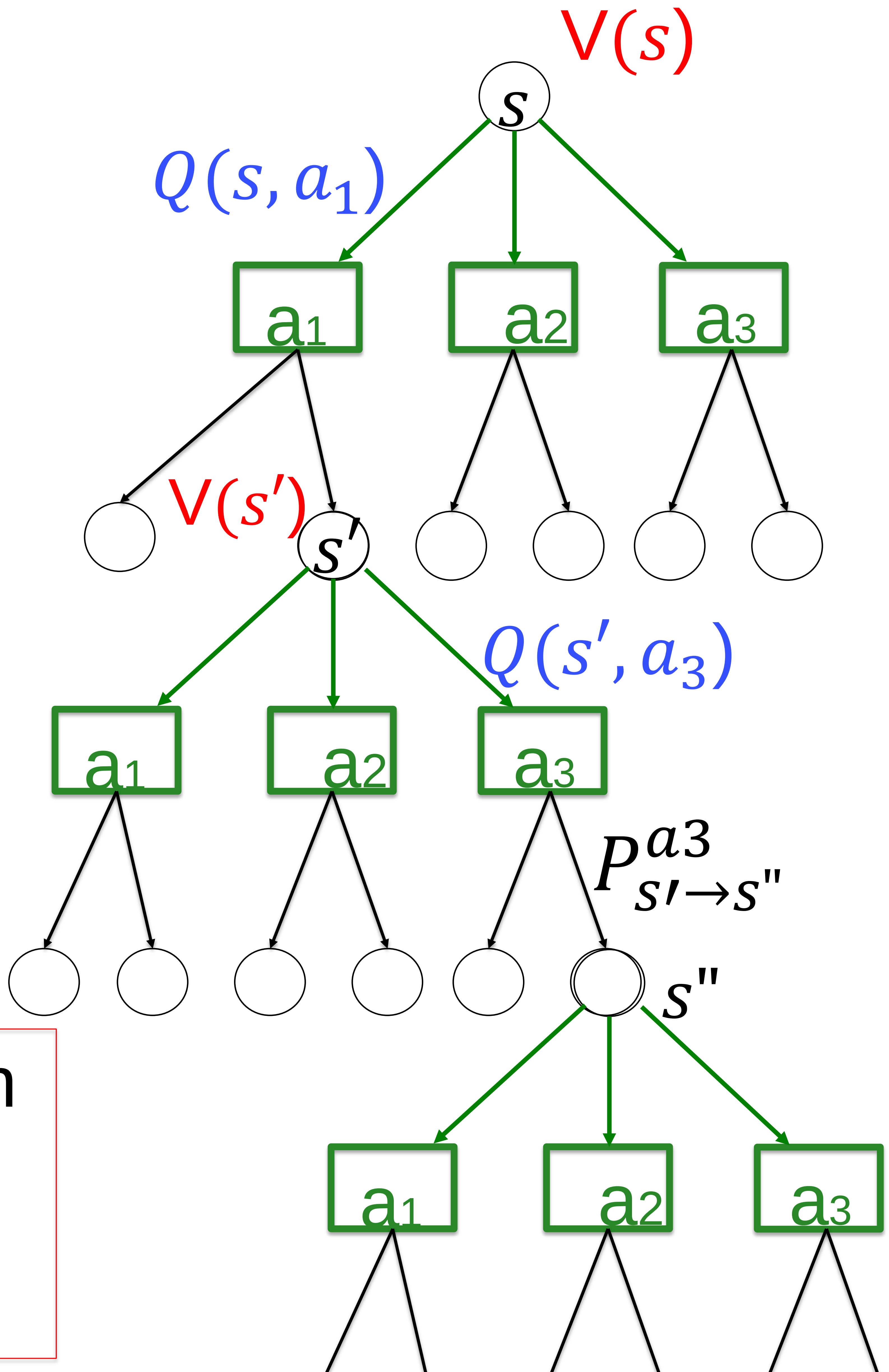
$$V(s) = \sum_k \pi(s, a_k)\ Q(s, a_k)$$

optimize by semigradient on Loss function

$$E(\mathbf{w}) = \frac{1}{2}[\underbrace{r_t + \gamma V(s'|\cancel{\mathbf{w}}}_{\text{target}}) - V(s|\mathbf{w})]^2$$

ignore

take gradient



(previous slide)

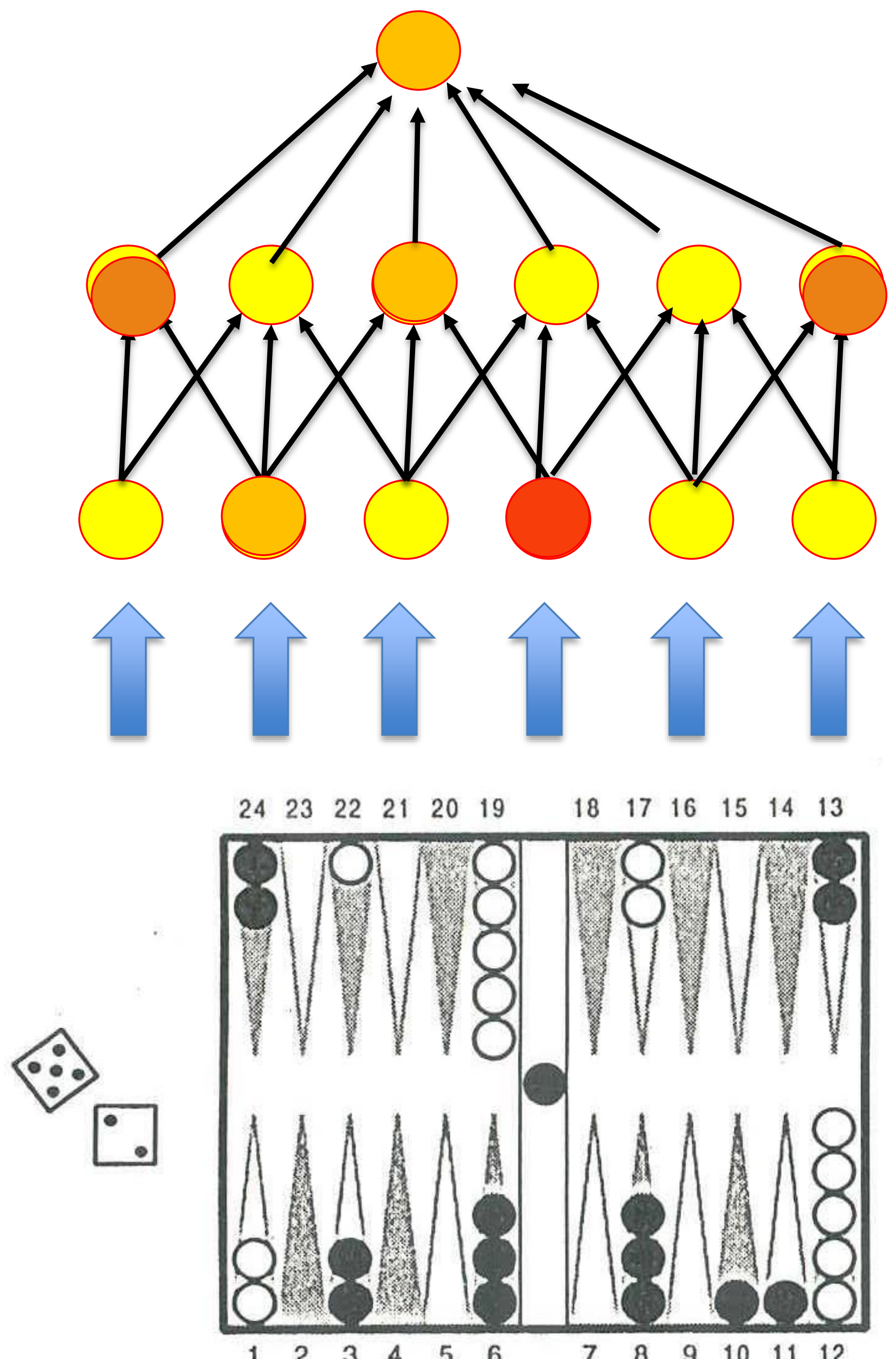
As an alternative to Q-values, the output of the Artificial Neural Network can also represent V-Values. The error function is constructed analogously.

Deep Neural Network for Value function

Action: move piece by greedy so as to increase V-value in each step (max across allowed moves)

output: V-values:

- **Neural network parameterizes V-values as a function of state s.**
- **One single output.**
- **Learn weights by playing against itself.**
- **Minimize TD-error of V-function**
- **use eligibility traces**



input

TD-Gammon

Tesauro, 1992,1994, **1995**, 2002

(previous slide)
The very same ideas can also be applied to learning the V-values, instead of Q-values. The advantage is that we have one single output. The disadvantage is that we need to look ahead (next possible states) to choose the action. But for games with a small number of ‘possible next states’ this is not a problem.

The analogous Bellman equation for the V-values leads to a consistency condition characterized by an error function

$$E(\boldsymbol{w}) = \frac{1}{2}[r_t + \gamma V(s'|\boldsymbol{w}) \text{- } \mathrm{V}(s|\boldsymbol{w})]^2$$

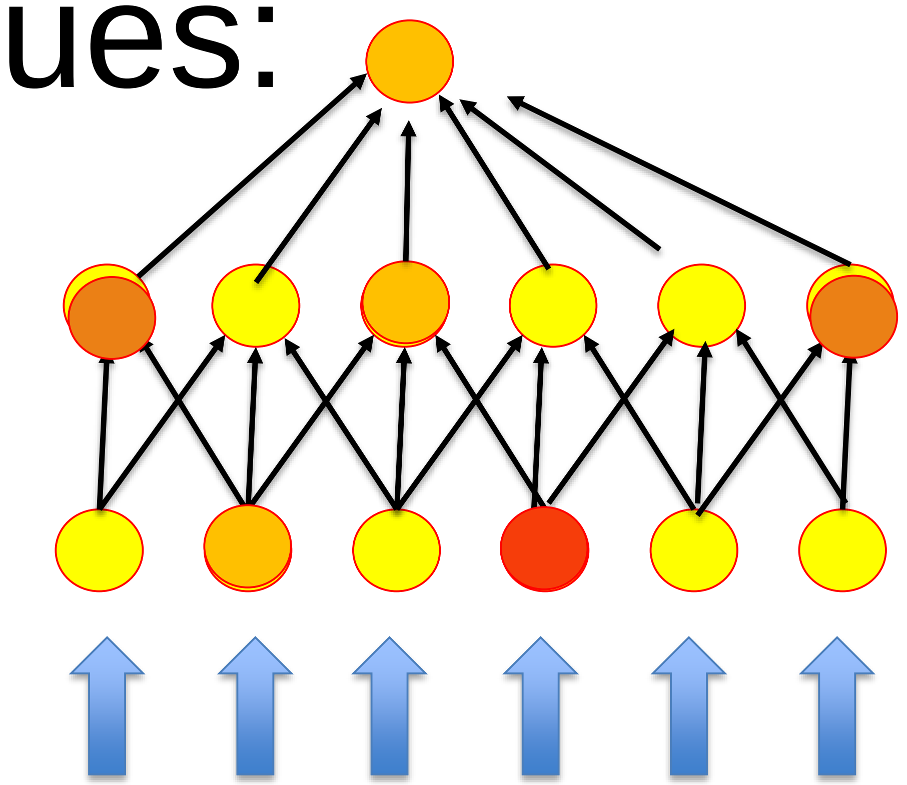
Eligibility traces enable to connect the reward at the end to states several steps before.

Deep Neural Network for Value function

TD-Gammon

Tesauro, **1995**,

output: V-values:



input

24 locations. For each location 8 inputs (4 for white/4 for black).

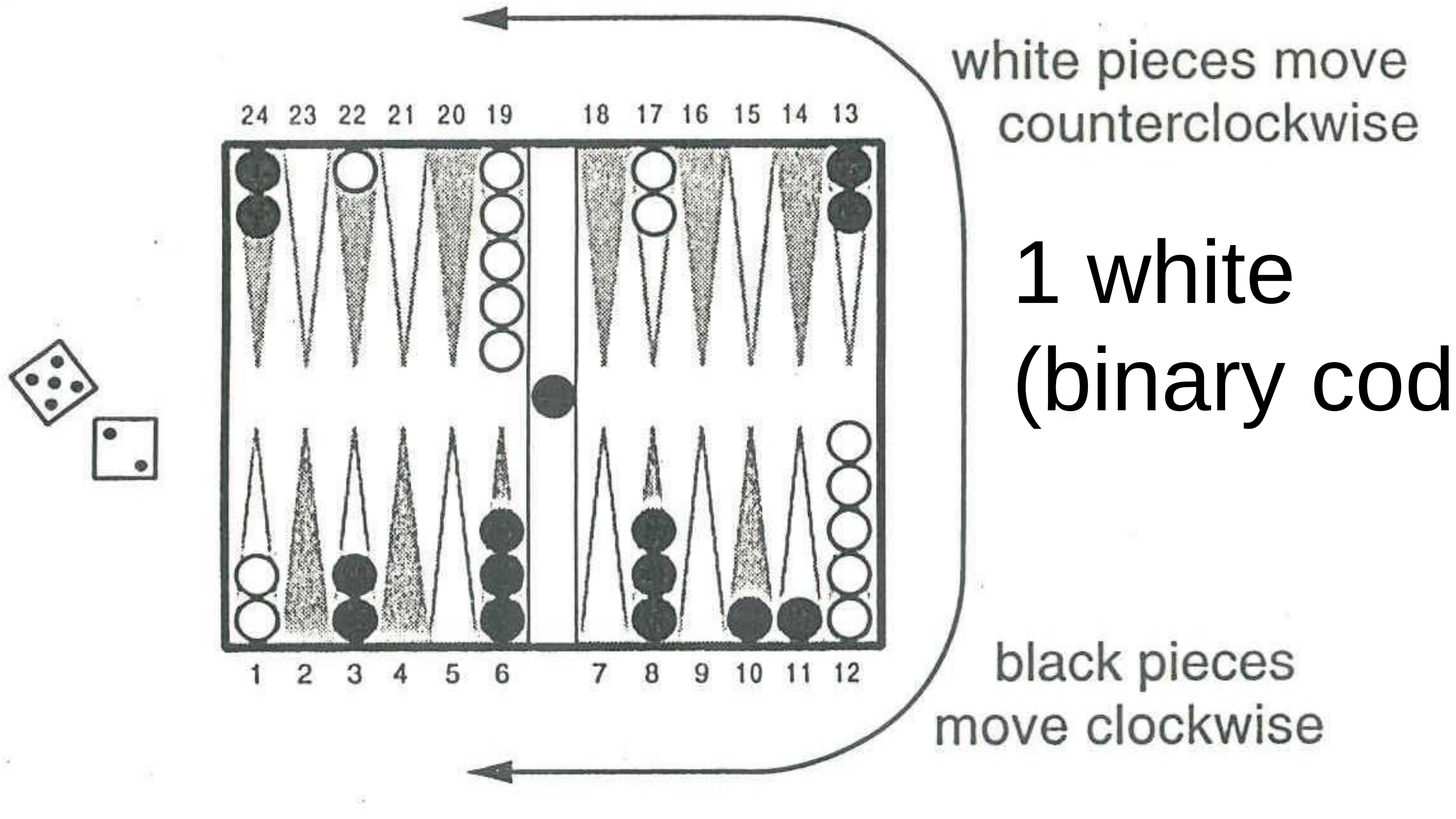
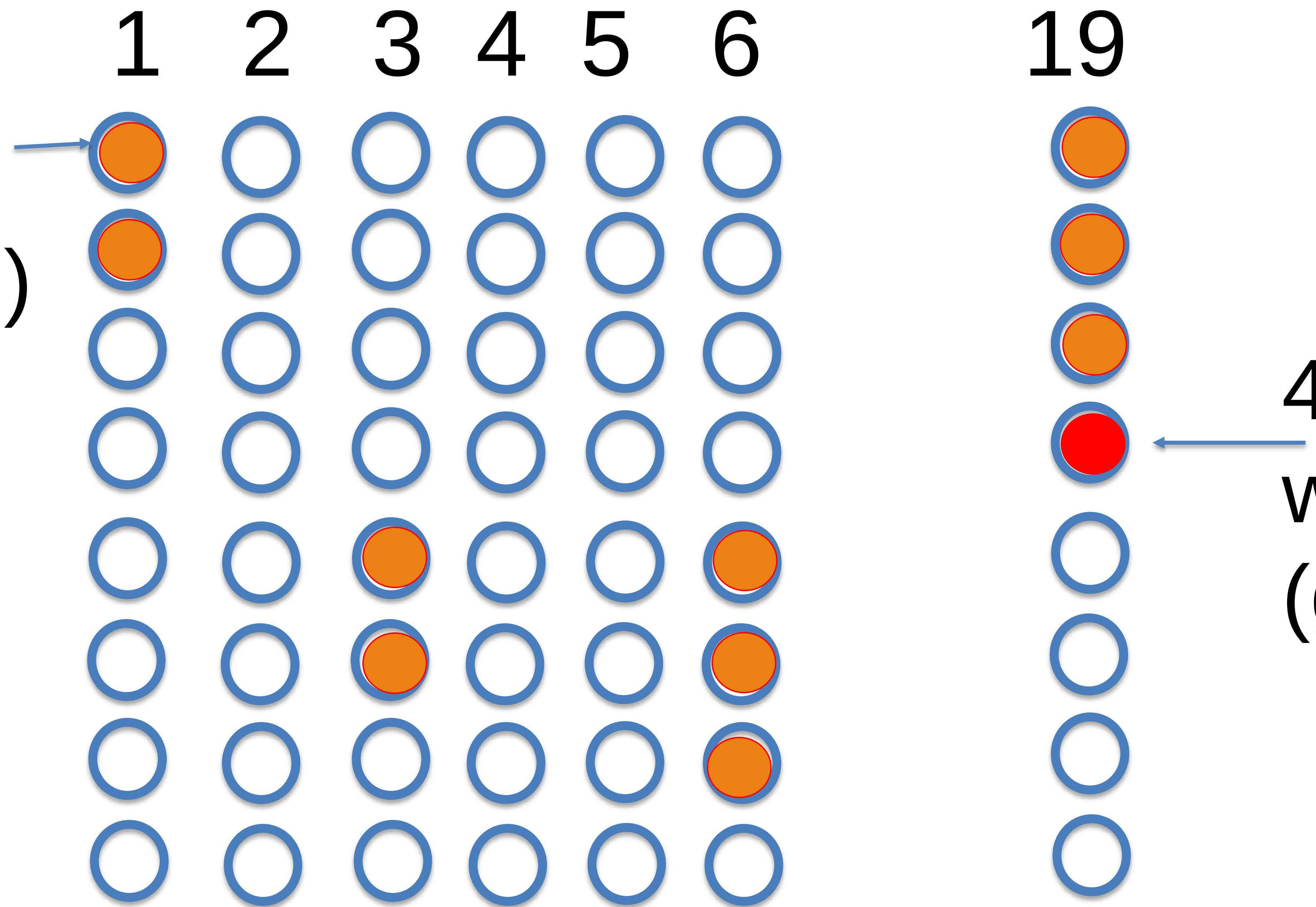

white pieces move counterclockwise
24 23 22 21 20 19 18 17 16 15 14 13
1 2 3 4 5 6 7 8 9 10 11 12
black pieces move clockwise

Figure 11.1 A backgammon position.

1 white
(binary code)



4 or more
white pieces
(graded code)

(previous slide)

Even though Backgammon is a discrete stochastic game with Markov properties (i.e., the perfect example of a Markov Decision Problem), Garry Tesauro decided to use an ANN to encode the input position.

This incoding makes maximal use of known properties of the game:

- A single piece on a position is not protected, and therefore very different from
- Two pieces on a position that are protected against attack.
- 4 or more pieces on a position give the freedom to move with two of the pieces and leave the remaining ones in the protected state.

He used an encoding that had separate inputs meaning for each of which and black (at least one), (at least two), (at least three), (at least four) where the last one had an additional linear intensity with value (n-3)/2.

A single hidden layer with 40 units was used.

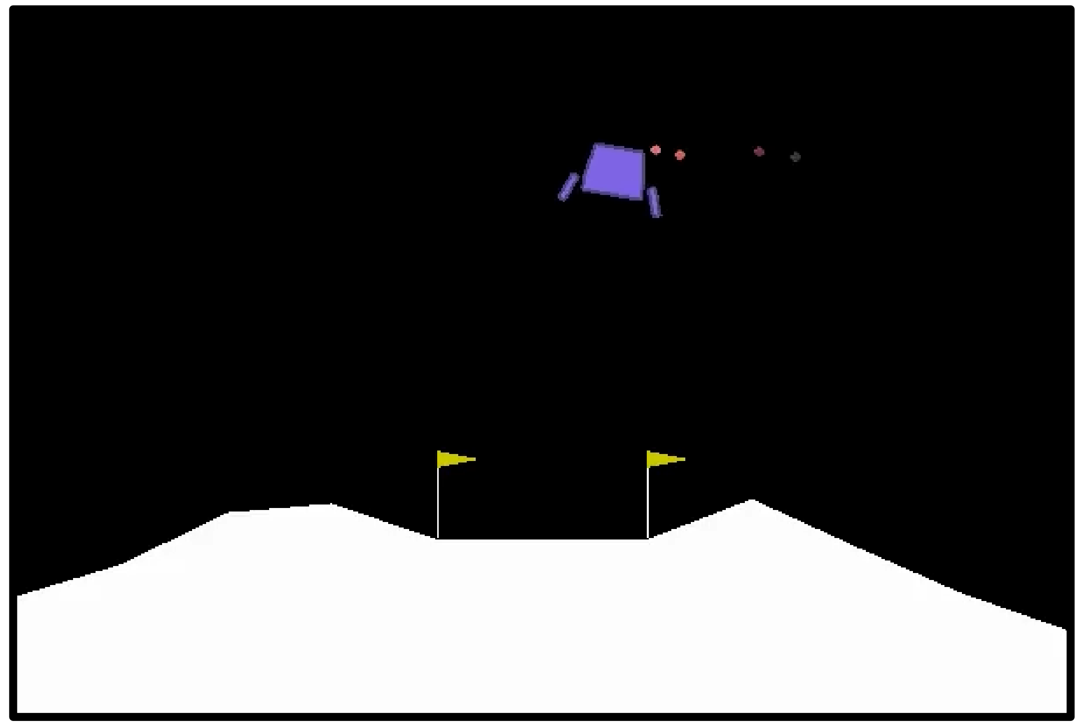
Neural networks to model input space

- for control problems, input space is naturally **continuous**

  Example: moon lander

  Aim: land between poles

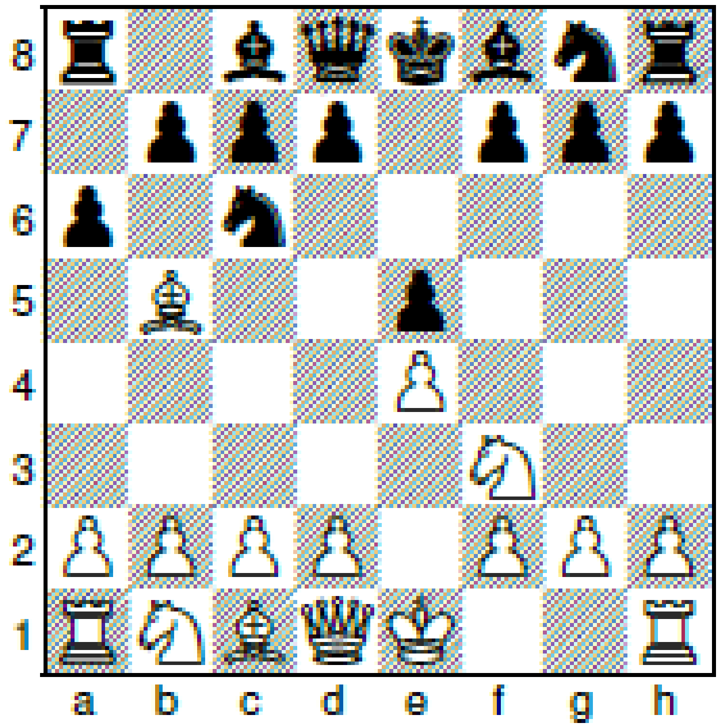
  →generalize to neighboring states



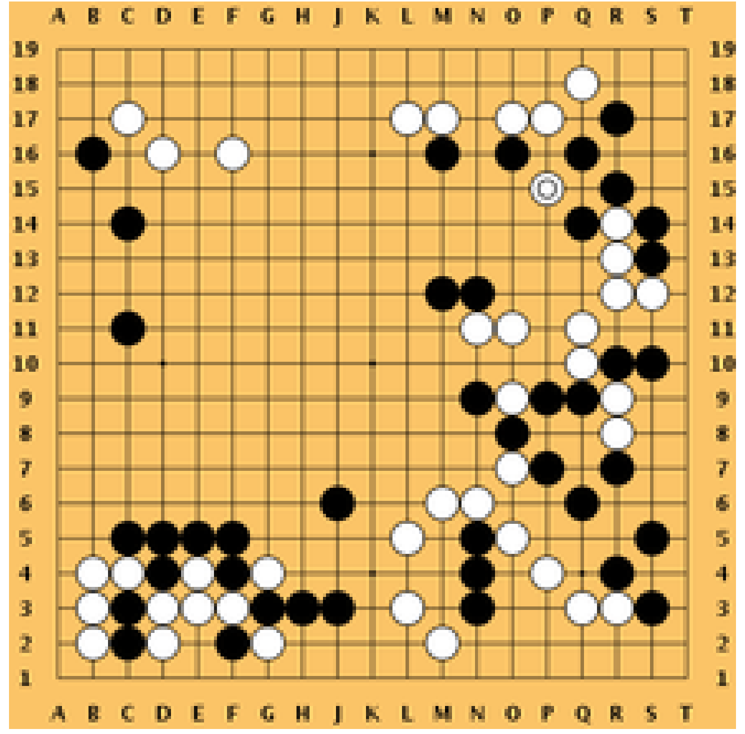
- for **discrete** games, the input space is often too big

  → generalize via hidden states in neural networks

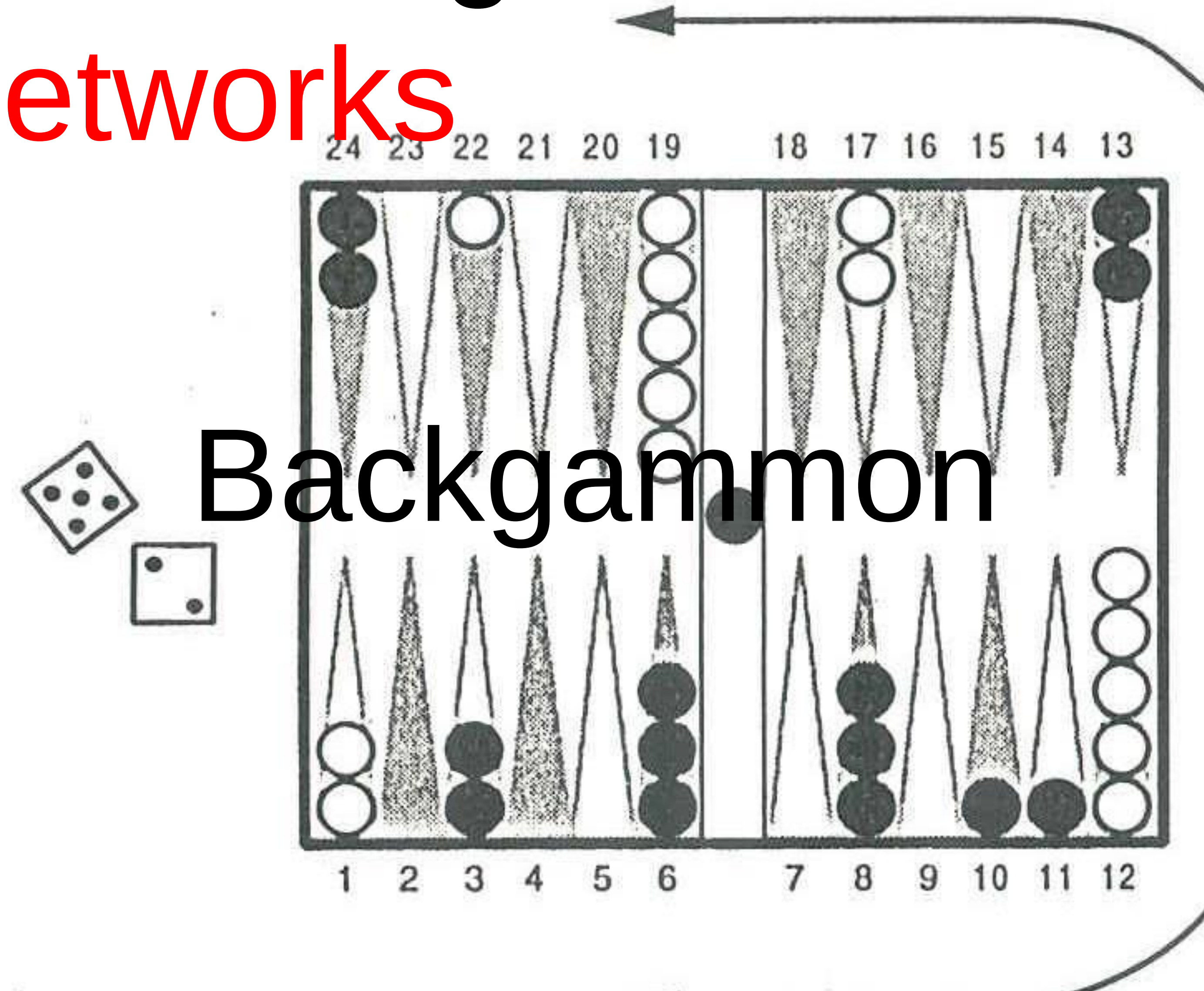
Chess



Go



Backgammon



(previous slide)

Why is it useful to use a continuous (as apposed to tabular) description of input space even in cases where the input is naturally discrete such as in games?

The reason is that describing Q-values as a SMOOTH function of the input enables generalization. Hidden layers of neural networks are able to extract compressed representations of the input space that introduce heuristic but useful notion of what it means that two states are ‘similar’ or ‘neighbors.

Related ideas have been used in many other applications, beyond chess backgrammon or Go. We will study some of these later in this class.

TD learning where Q-values are V-values are described by a smooth function, is also called ‘function approximation in TD learning’. The family of functions can be defined by the parameters of a Neural Network or by the parameters of a linear superposition of basis functions.

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**

- **Q-values can be represented as the output of an ANN**

(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.

Exercise 1-3 now : Q-values (continuous) and Semi-Gradient

Results of Exercise 1-3 are basis for next lecture.

Next Lecture at 14h15

Exercise 1. Consistency condition for 3-step SARSA

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

$$E = \frac{1}{2}\sum \delta_t^2$$

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

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

Exercise 2. Q-values for continuous states

We approximate the state-action value function $Q(s, a)$ by a weighted sum of basis functions (BF):

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

where $\Phi(x)$ is the BF "shape", and the s_j's represent the centers of the BFs.

Calculate

$$\frac{\partial Q(s, a)}{\partial w_{\tilde{a}i}}\,,$$

the gradient of $Q(s, a)$ along $w_{\tilde{a}i}$ for a specific weight linking the basis function i to the action $\tilde{a}$.

Exercise 3. Gradient-based learning of Q-values

Assume again that the Q-values are expressed as a weighted sum of 400 basis functions:

$$Q(s, a) = \sum_{k=1}^{400} w_a^k \Phi(s - s_k).$$

For this exercise the function Φ is arbitrary, but you may think of it as a Gaussian function. Note that s and s_k are usually vectors in $\mathbb{R}^N$ in this case. There are 3 different actions so that the total number of weights is 1200. Now consider the error function $E_t = \frac{1}{2}\delta_t^2$, where

$$\delta_t = r_t + \gamma \cdot Q(s', a') - Q(s, a) \tag{1}$$

is the reward prediction error. Our aim is to optimize $Q(s, a)$ for all s, a by changing the parameters w. We consider $\eta \in [0, 1)$ as the learning rate.

Exercise 3 (cont.)

a. Use the full gradient of the error function E_t and write down the learning rule based on gradient decent. Consider the case where the actions a and a' are different.

How many weights need to be updated in each time step?

b. Use the full gradient of the error function E_t and write down the learning rule based on gradient decent. Consider the case where the actions a and a' are the same.

Is there any difference to the case considered in (a)?

c. Repeat (a) and (b) by using the semi-gradient of the error function E_t. Do your answers change?

d. Suppose that the input space is two-dimensional and you discretize the input in 400 small square 'boxes' (i.e., 20 × 20). The basis function $\Phi(s - s_k)$ is now the indicator function: it has a value equal to one if the current state s is in 'box' k and zero otherwise.

How do your results from (a-c) look like in this case?

e. The learning rules in (d) are very similar to standard SARSA. What is the difference?

Hint: Consider the difference between Full Gradient and Semi-gradient.

f. Assume that $Q(s', a')$ in Equation 1 does not depend on the weights. For example $Q(s', a')$ could be extracted from a separate neural network with its own parameters. How is your result in (a-c) related to standard SARSA? What do you conclude regarding the choice of semi-gradient versus full gradient? What do you conclude regarding the choice of Mnih et al. (2015) to model $Q(s', a')$ by a separate network with parameters that are kept fixed for some time?

Reinforcement Learning Lecture 3

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Continuous input space: function approximation

Part 4: Deep Q-learning

(previous slide)
Deep Reinforcement Learning (DeepRL) is reinforcement learning in a deep network. Suppose that each output unit of the networkcorresponds to one action (e.g. one type of move in chess). Parameters are 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.

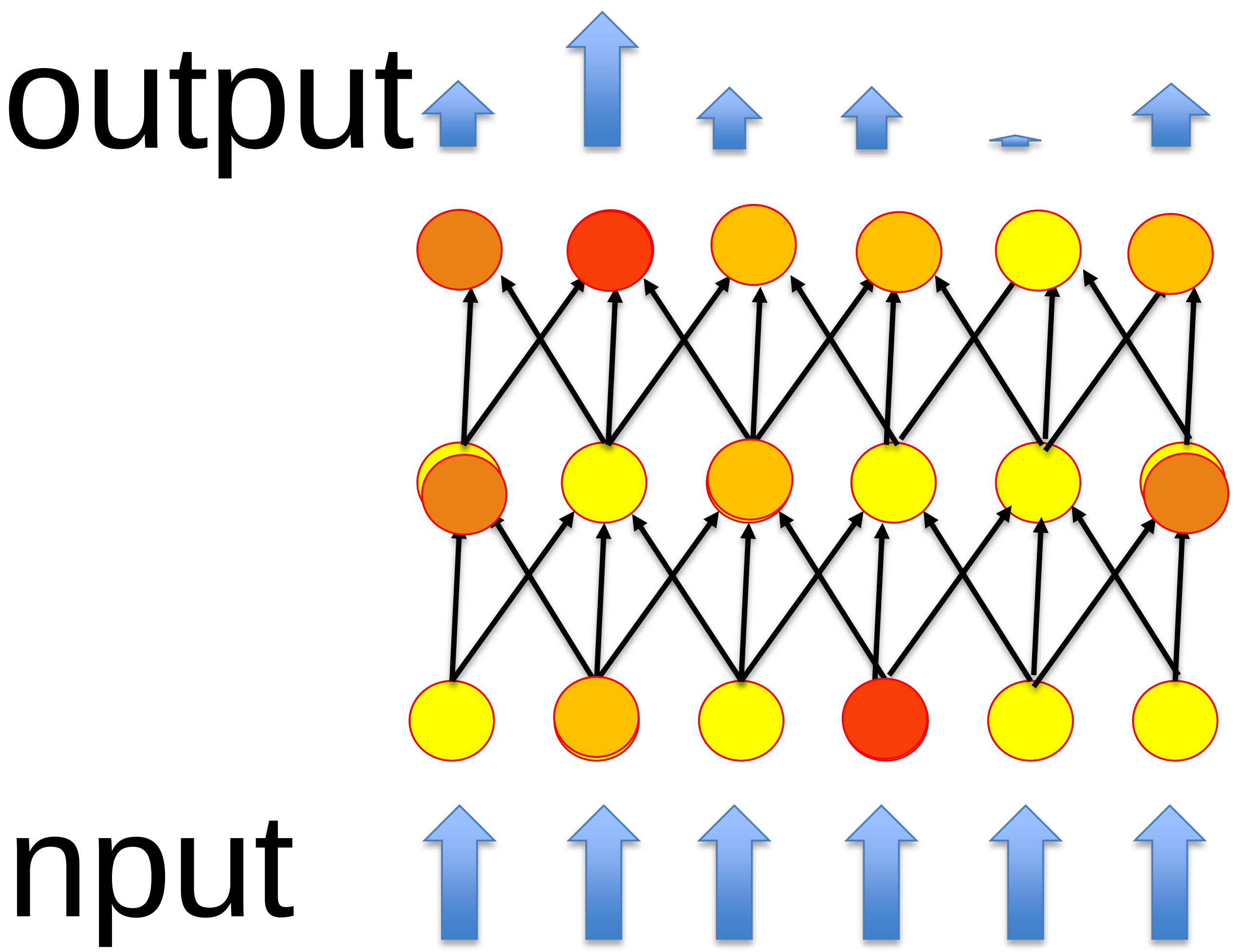
The consistency condition of TD learning, can be formulated by an error function:

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

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’

Deep Q-learning with maxQ: DeepQ/Atari games

output=Q-values



Input - states

Outputs are Q-values
→ actions are easy to choose
(e.g., softmax)

Neural network parameterizes Q-values as a function of continuous state s.
One output for one action a.
Reward = score increase

Error function for Q-learning (loss)

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

Example: Atari-video games (*Mnih et al. 2015*)

input = video screen; network = ConvNet; reward = score increase

Action = every 4th input. Additional tricks: Two networks, store and replay (s,a,r,s')

(previous slide)
Deep Q-Learning uses the a deep network which transforms the state (encoded in the input units) into Q-values in the output.

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

Weights are learned by taking the semi-gradient on the error function (symbol L)

$$L = 0.5\ [\ r + \gamma\ max_{a'} Q(s',a') - Q(s,a)\ ]^2$$

Recall that SARSA and Q-learning are TD algorithms. Recall also that the idea of the semi-gradient is to stabilize the target $r + \gamma\ max_{a'} Q(s',a')$

When Mnih et al. applied DeepQ to video games they used a few additional tricks: to stabilize the target even further, they kept target Q-values and current Q-values in two separate networks; and they stored past transitions s,a,r s', so that they could be replayed at any time (without actual action taking), so as to update Q-values. We will come back to DeepQ in a later lecture.

Semi-gradient: from online to expectation (SARSA)

Discrete time steps: s,a → r_t,s',a'

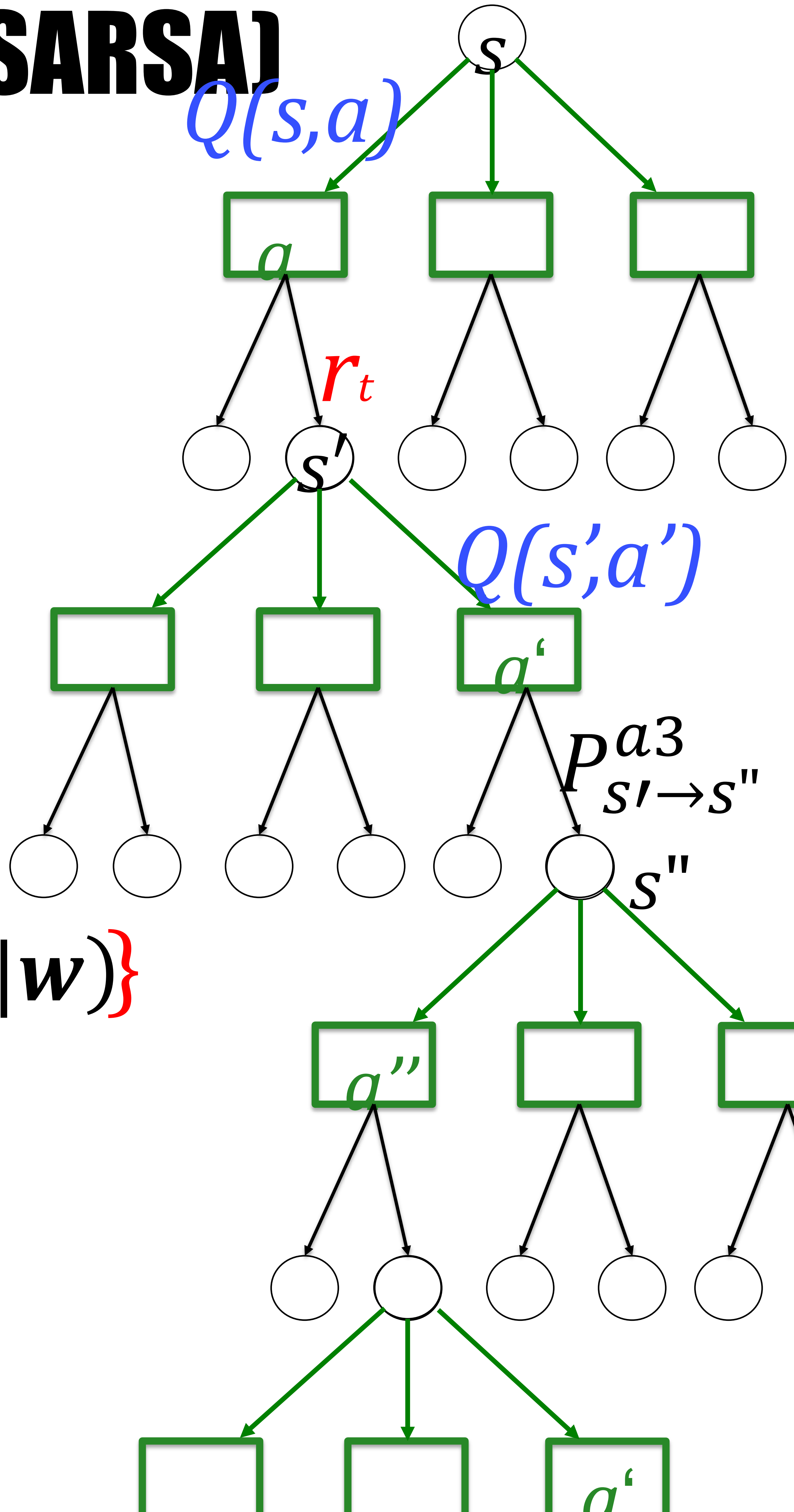
$$L(\boldsymbol{w}) = \boldsymbol{E}\{\tfrac{1}{2}[\overbrace{r_t + \gamma Q(s',a'|\boldsymbol{w})}^{\text{target}} - Q(s,a|\boldsymbol{w})]^2\}$$

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

$$\frac{d}{d\boldsymbol{w}} L(\boldsymbol{w}) = \boldsymbol{E}\{[r_t + \gamma Q(s',a'|\boldsymbol{w}) - Q(s,a|\boldsymbol{w})]\frac{d}{d\boldsymbol{w}} Q(s,a|\boldsymbol{w})\}$$

$$\boxed{\boldsymbol{E}\{\dots\} = \frac{\mathbf{1}}{\boldsymbol{N}} \sum_{\{all\ transitions\}}^{\boldsymbol{N}} \{\dots.\}}$$

How can we implement this ‘batch over transitions’?



(previous slide)
Semi-gradient implies that we only take the gradient of Q(s,a|w), but not that of Q(s’,a’).
We remark that the loss function that is usually written down refers to a SINGLE transition [i.e., (s,a,r, s’,a’) for the case of SARSA and (s,a,r, s’) for the case of Q-learning].

Hence this is the error function for ONLINE updates, one transition at the time.
However we may also consider BATCH updates (as approximations to expected updates).

How would you implement this???

Deep Q-learning with maxQ: use two networks for semi-gradient

1) Replay thousands of transitions (s,a,r,s') with parameters θ^{old}
2) Update parameters of 'fast network' $w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$
3) Update: $\theta^{old} \leftarrow w_{ij}$; play with new network; back to 1)

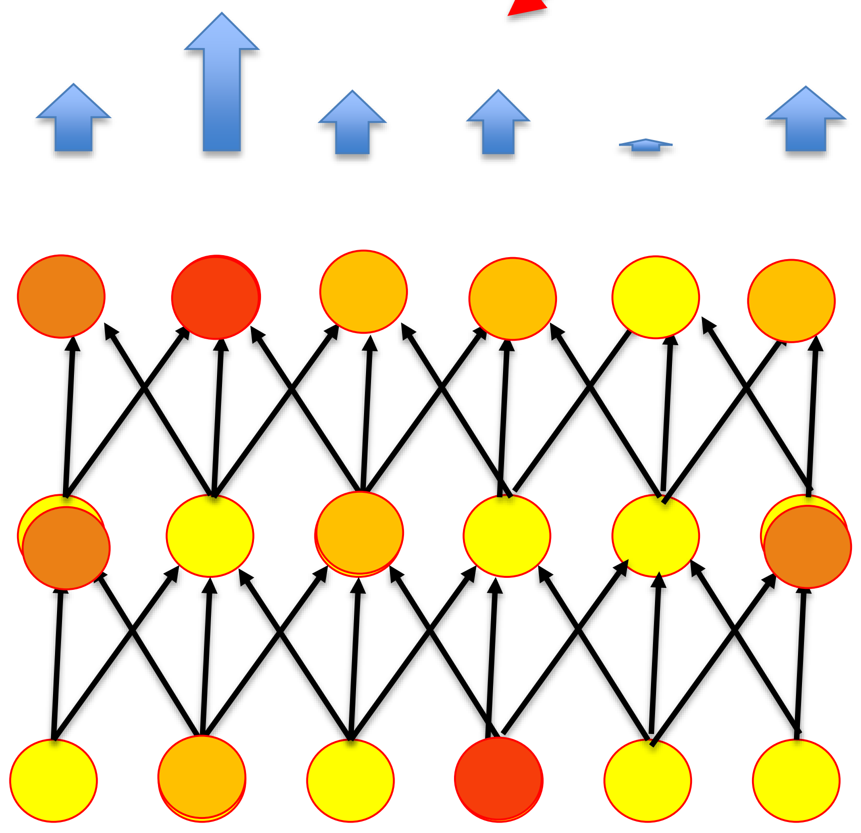
'batch update'

Loss function for Q-learning

$$L = 0.5\,[\, r + \gamma\, max_{a'} Q(s',a'|\theta^{old}) - Q(s,a|w_{ij})\,]^2$$

old/frozen Q-values

θ^{old}



input

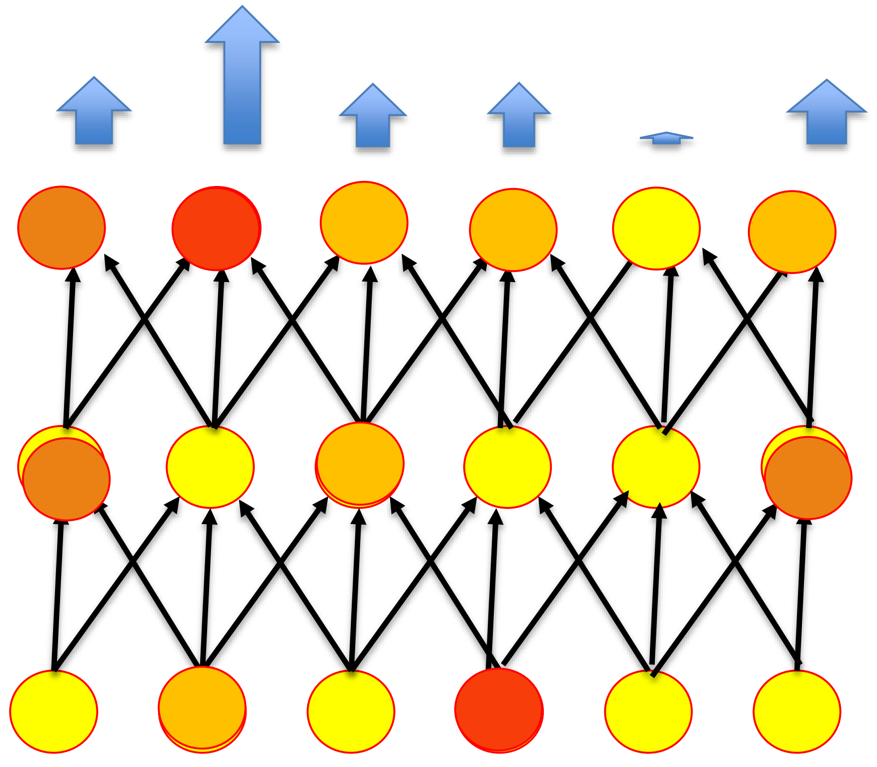
'slow target network'

Update

$w_{ij} \leftarrow w_{ij} + \Delta w_i$

new Q-values

w_{ij}



input

'fast update network'

(previous slide)

Semi-gradient implies that we only take the gradient of Q(s,a|w), but not that of Q(s',a'). The semi-gradient becomes natural if we consider that the two estimations come from different networks

$$E = 0.5\ [\ r + \gamma\ max_{a'} Q(s',a'\ |\theta^{old}) - Q(s,a|w)\ ]^2$$

The first network (green) now implements a stable target. The second one (blue) does rapid online updates over many samples.

Overall the resulting philosophy is that of a batch update:

0)While playing the task according to, say, epsilon-greedy resulting from the stable target network with parameters θ^{old}, we store thousands of transitions (s,a,r,s') .

1)We then use the stored transitions in random order to replay and

2) calculate intermediate updates by changing the weights w_{ij} in the 'fast update network'.

3) Only after several thousand of these replay segments we write the new parameters w_{ij} onto the 'stable' network which then gives an 'updated stable network'. We restart with step 0.

Hence, the gradient with respect to the weight w_{ij} with a stable target network with fixed parameters θ^{old}, is similar to semi-gradient. And also similar to batch updates of θ^{old}.

Remark: The above argument shows the relation of semi-gradient to a batch update via implementation in two networks (slow and fast).

We can also use an implementation with two networks (slow and fast), but choose the policy according to the fast network. This second option is the one implemented in the PyTorch tutorial.

Detour/Repetition: 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:

- With batch update we have less jitter.
- Semi-gradient with 2 networks = batch update with 1 network

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

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

Batch Update from $\theta = \theta^{old}$:

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

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

(previous slide)
This is a repetition from an earlier lecture.

Conclusion semi-gradient with the two networks can be interpreted as a batch update in a single network, i.e., in the slow network with parameters $\theta = \theta^{old}$.

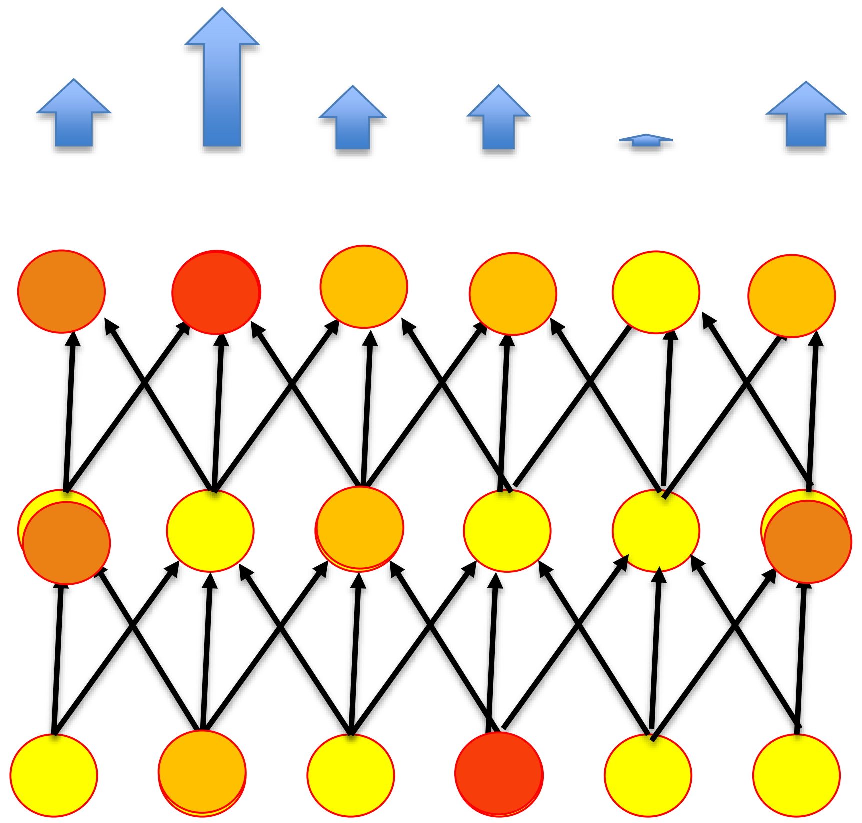
DQN: Batch semi-gradient is implemented by the slow network

DQN: *Mnih et al. 2015*

Batch updates:

$$\theta^{old} \leftarrow \theta^{new} = \{w_{ij}\}$$

Make slow target network consistent with Bellman equation

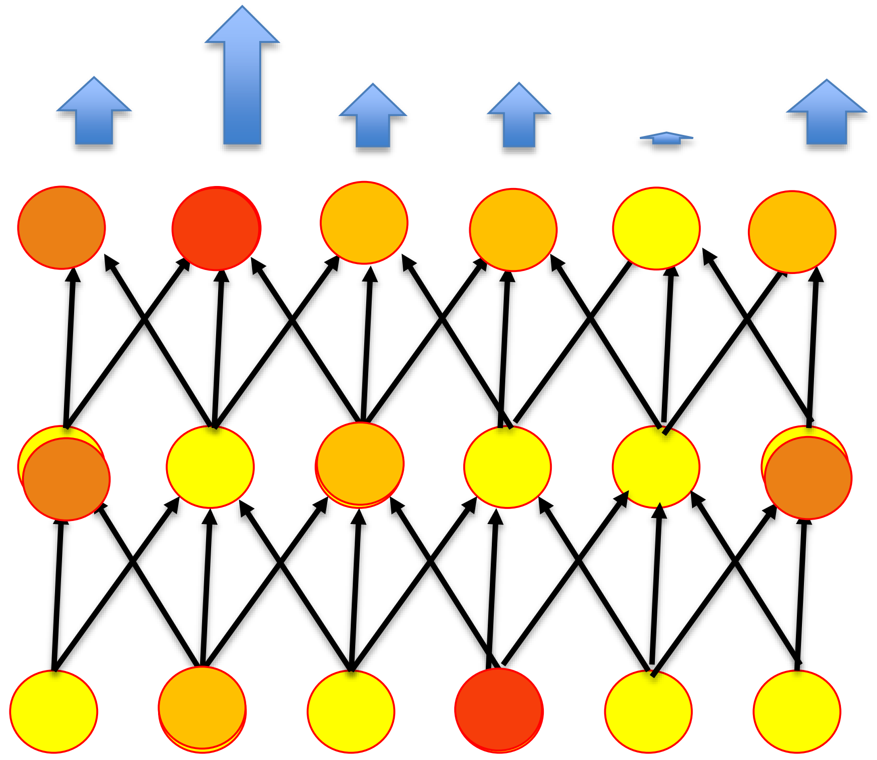


old/frozen Q-values

θ^{old}

input

'slow target network'



Update

$w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$

new Q-values

w_{ij}

input

'fast update network'

(previous slide)

Hence, the semi-gradient with the two networks can be interpreted as a batch update in a single network, i.e., batch update of the **slow network** with parameters $\theta = \theta^{old}$.

Summary: Deep Q-learning

- Q-learning with continuous (or high-dim.) state space
- Q-values represented by output of deep ANN
- Action choice (=policy) depends on Q-values
- For training use semi-gradient with error function

  either SARSA (online, on-policy)

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

  or Q-learning (off-policy)

  $L(\theta) = 0.5\ [\ r + \gamma\ \max_{a'} Q(s',a') - Q(s,a)\ ]^2$

- Further tricks (off-line updates, target stabilization)
  - store transitions *(s,a,r,s') or (s,a,r,s',a')* and replay offline

Why use deep networks? → generalization properties

(previous slide)
Deep Q-Learning is SARSA (or Q-learning) in a deep ANN.

For SARSA Weights are learned by taking the semi-gradient on the error function,

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

Recall that SARSA and Q-learning are TD algorithms.

In the next Lecture, we will go to Policy Gradient Methods; and in the third part we combine policy gradient methods with TD-learning!

But before that we explore function approximation further in view of its inherent inductive bias.

Reinforcement Learning Lecture 3

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Continuous input space: function approximation

Part 5: No Free Lunch Theorem

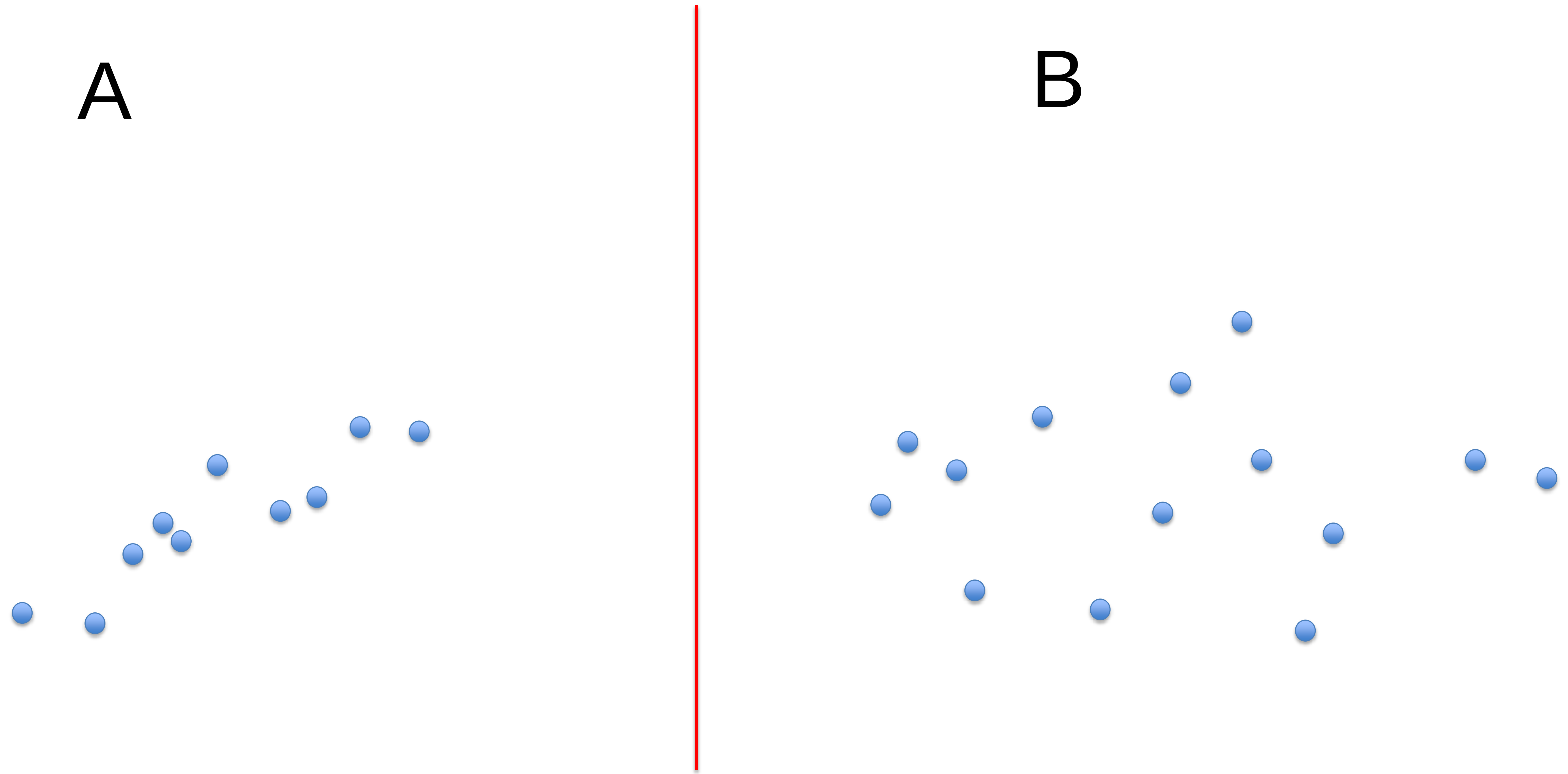
Previous slide.
No Free Lunch theorems (there are several variants) are foundational and philosophically important to answer the question: why do deep neural networks work so well?

The video for this part can be found on
https://lcnwww.epfl.ch/gerstner/VideoLecturesANN-Gerstner.html

Under
 ‘Deep Learning Lecture 3, part 6 (No free lunch theorems)

No Free Lunch Theorem

Which data set looks more noisy?



Commitment:
Thumbs up

Which data set is easier to fit?

Commitment:
Thumbs down

Previous slide.

Let us start with two data sets.

No Free Lunch Theorem

Previous slide.

And here a possible explanation (hidden behind the blue boxes).

No Free Lunch Theorem

Your notes

No Free Lunch Theorem

The NO FREE LUNCH THEOREM states
“ *that any two optimization algorithms are equivalent when their performance is averaged across* ***all possible problems***"

See Wikipedia/wiki/No_free_lunch_theorem

- Wolpert, D.H., Macready, W.G. (1997), "No Free Lunch Theorems for Optimization", *IEEE Transactions on Evolutionary Computation* **1**, 67.
- Wolpert, David (1996), "The Lack of *A Priori* Distinctions between Learning Algorithms", *Neural Computation*, pp. 1341-1390.

Previous slide.

The conclusion is: there is no reason to believe that an algorithm that works well on one data set will also work well on an arbitrarily chosen other data set.

No Free Lunch (NFL) Theorems

The mathematical statements are called

“NFL theorems because they demonstrate that if an algorithm performs well on a certain class of problems then it necessarily pays for that with degraded performance on the set of all remaining problems”

See Wikipedia/wiki/No_free_lunch_theorem

- Wolpert, D.H., Macready, W.G. (1997), "No Free Lunch Theorems for Optimization", *IEEE Transactions on Evolutionary Computation* **1**, 67.
- Wolpert, David (1996), "The Lack of *A Priori* Distinctions between Learning Algorithms", *Neural Computation*, pp. 1341-1390.

Previous slide.

Even worse, if the algo works well on some problem, there must exist another problem on which the algorithm works badly.

Quiz: No Free Lunch (NFL) Theorems

Take a neural networks with many layers, and many neurons, optimized by Backprop (with momentum/ADAM) as an example of deep learning

[ ] Deep learning performs better than most other algorithms on real world problems.

[ ] Deep learning can approximate to arbitrary degree any non-ambiguous (noise-free) data set (universal approximator theor)

[ ] Deep learning performs better than other algorithms on all problems.

Your notes.

No Free Lunch (NFL) Theorems

- Choosing a deep network and optimizing it with gradient descent is an algorithm

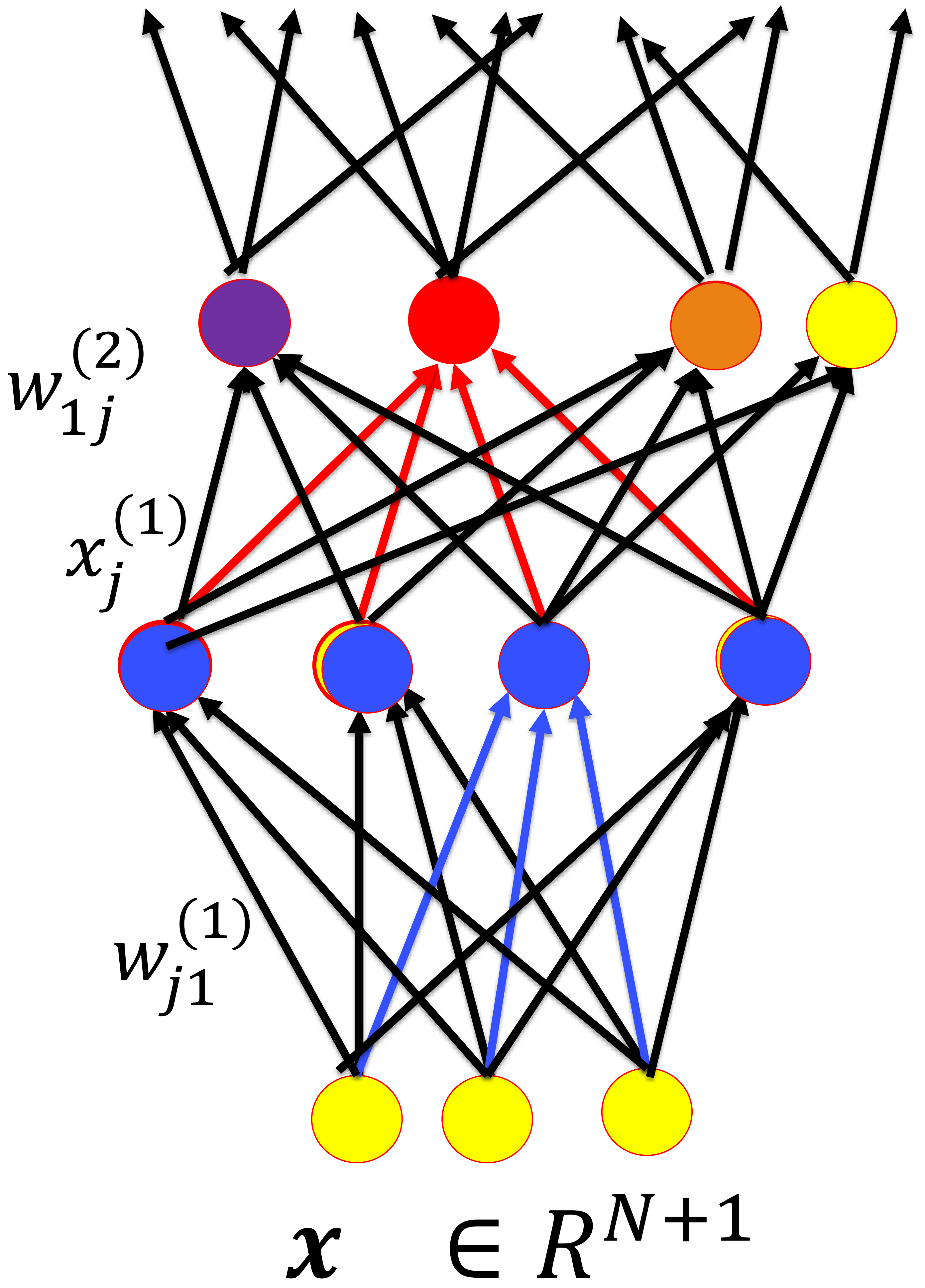
- Deep learning works well on many real-world problems

- Somehow the prior structure of the deep network matches the structure of the real-world problems we are interested in.

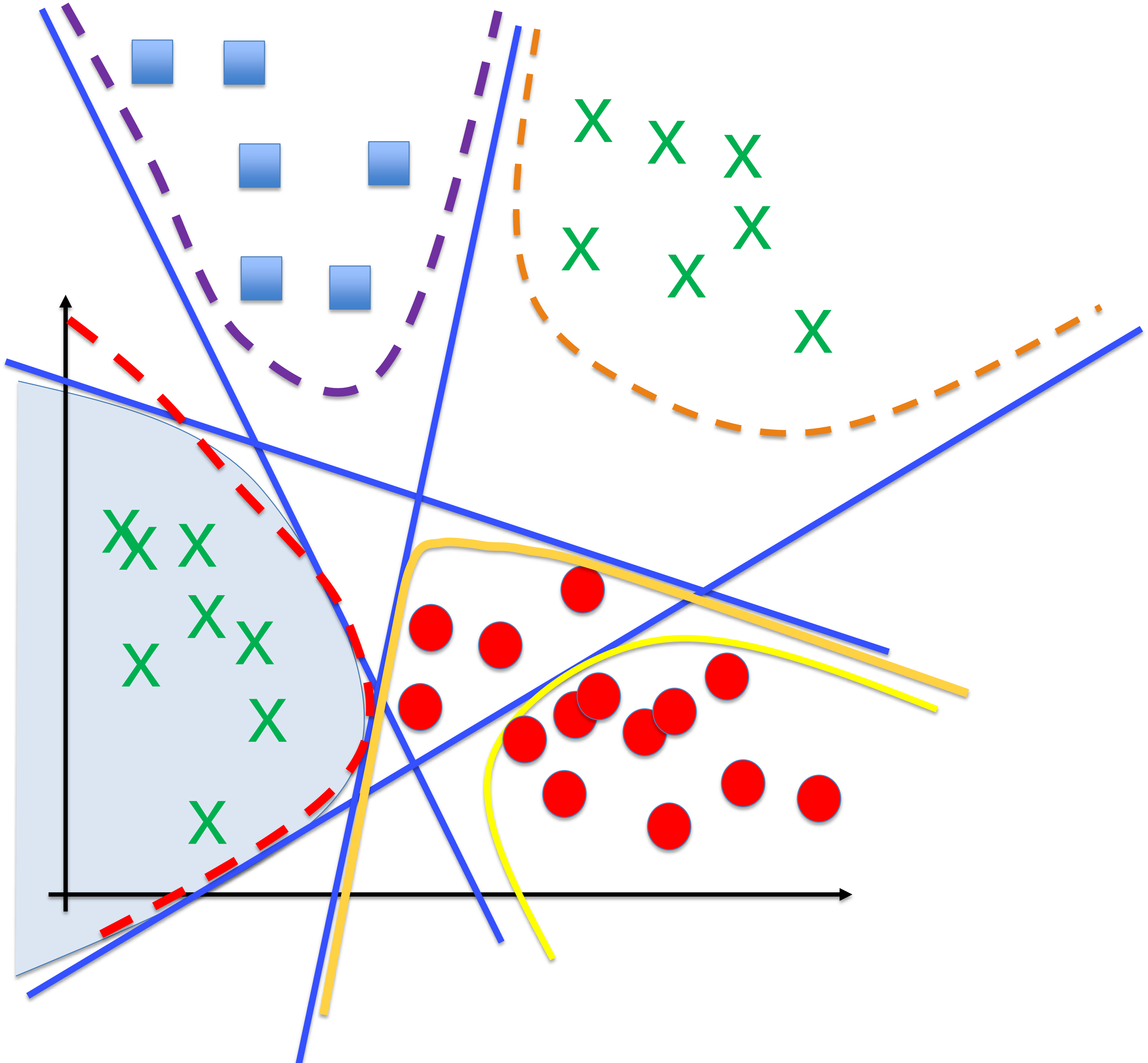
Previous slide.

The reason that deep networks work well must be linked to the type of data on which we test them.

No Free Lunch (NFL) Theorems



Geometry of the information flow in neural networks



Previous slide/next slide.

One possible explanation of why neural networks work well is the notion of hyperplanes. Even though the data is local, you make a cut through the whole space. This predefines additional ‘compartments’ that can be reused later for other data.

This argument might be applicable in the last few layers before the output. Suppose we look at layer 47 in a network of 50 layers. The previous layers have extracted high-level features (such as leg-detectors, fur-detectors etc). The last 3 layers before the output can then recombine these features in various ways to classify all sorts of animals.

Reuse of features in Deep Networks (schematic)

animals

birds

4 legs

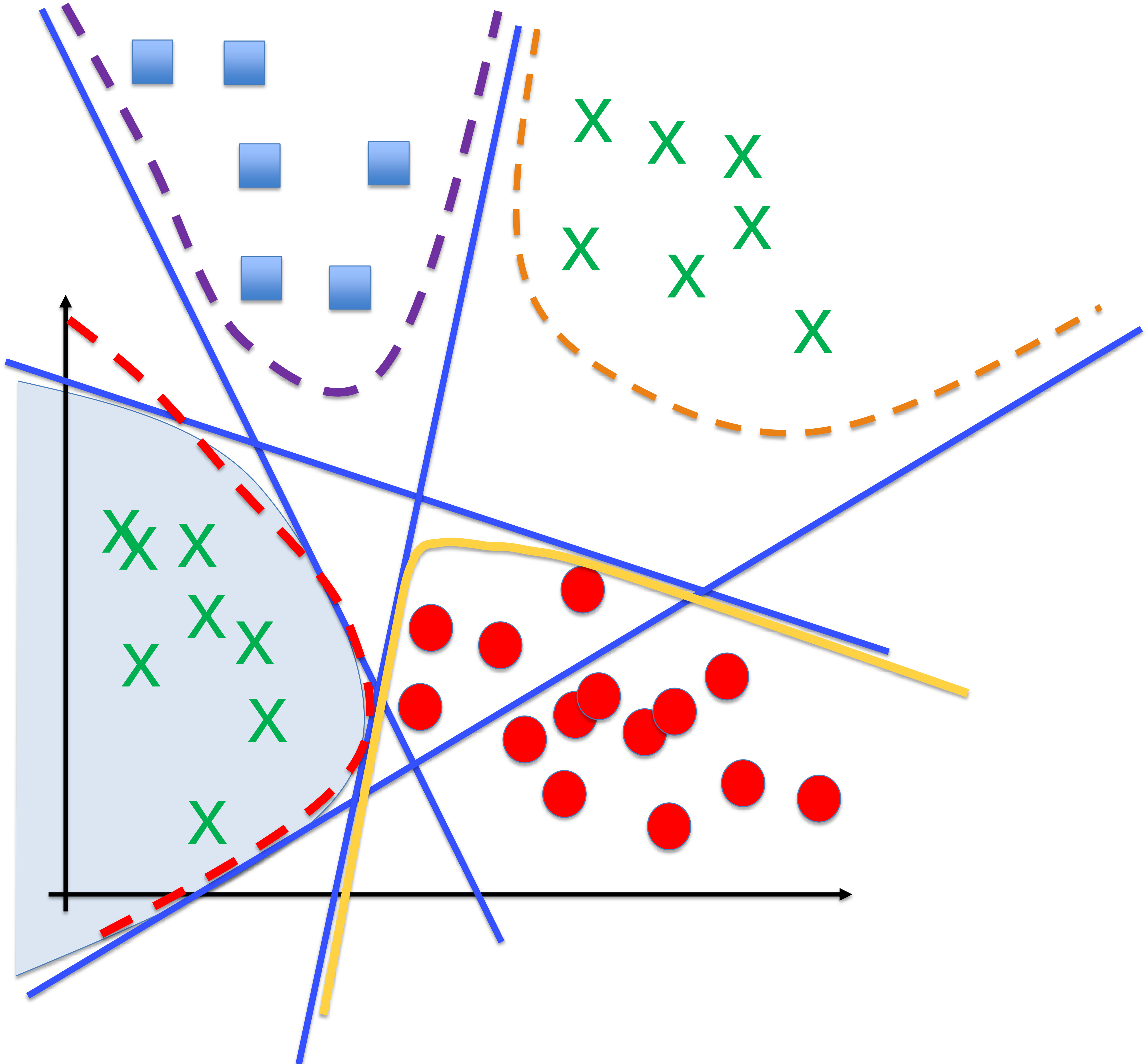
wings

snout

fur

eyes

tail



Summary: No Free Lunch (NFL) Theorems and Deep Networks

Somehow the prior structure of the deep network
matches the structure of the real-world problems
we are interested in.

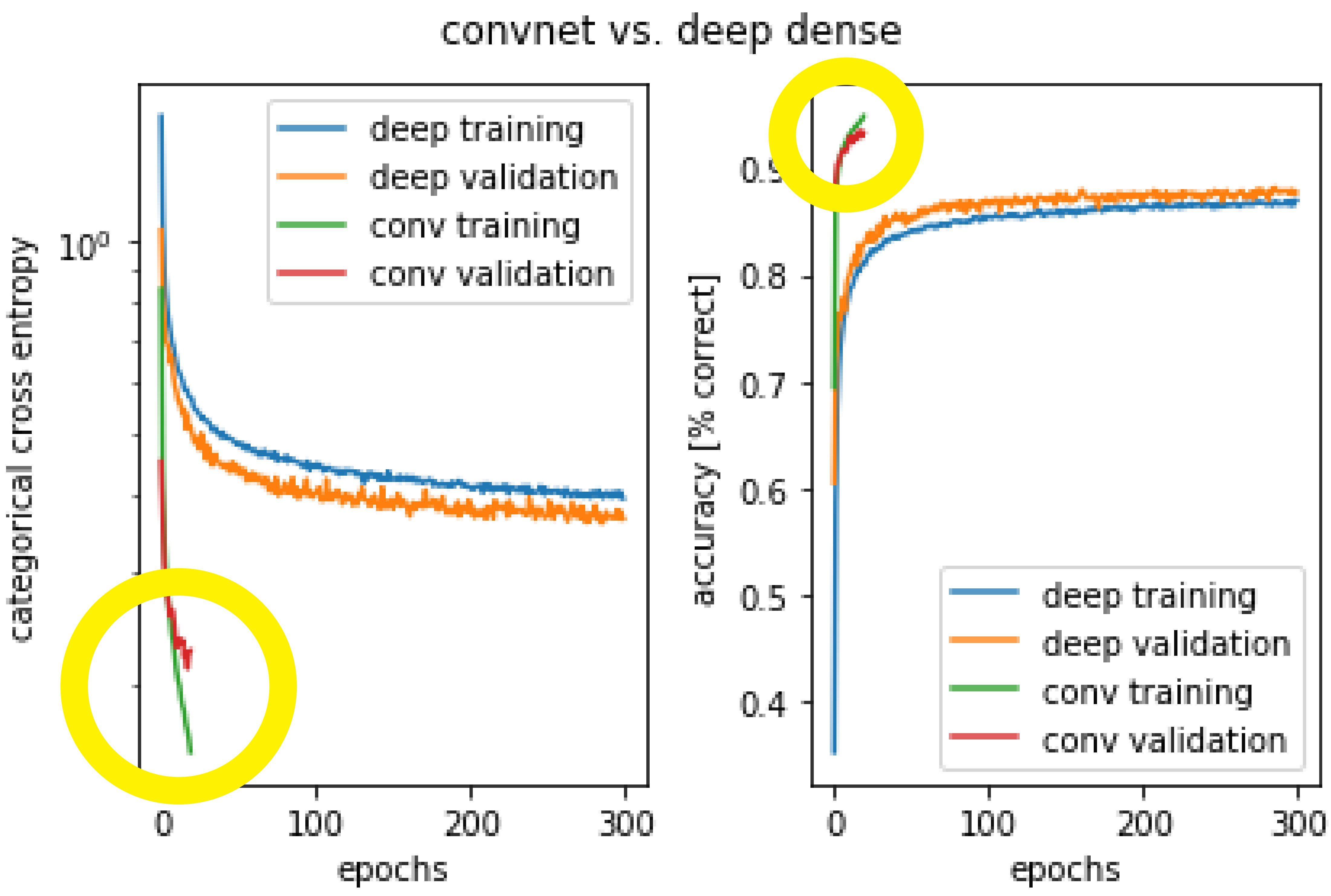
The above example is applicable to layers close to the output,
- relevant features have been extracted in earlier layers
- the network can recombine these in various ways

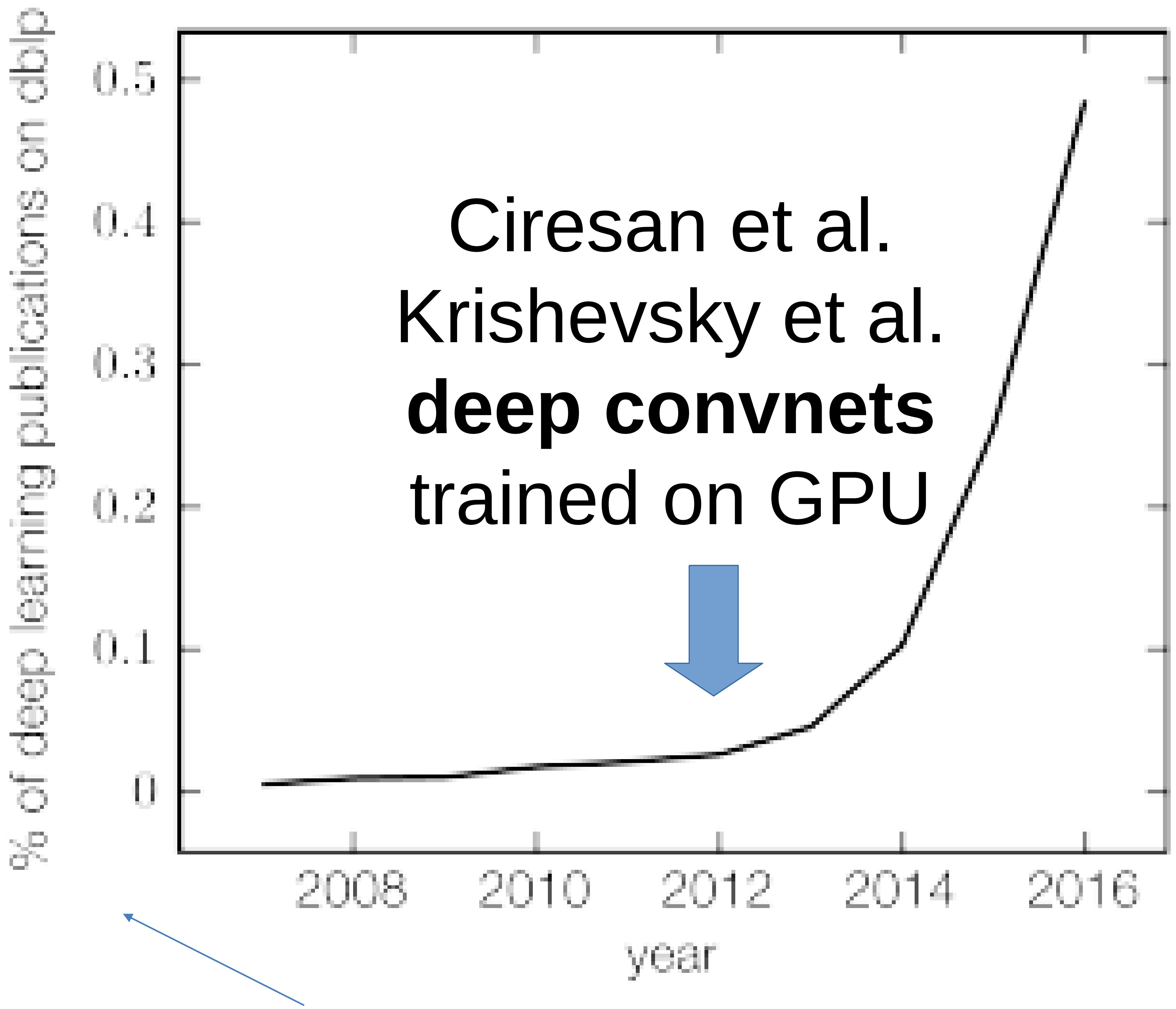
A good representation (in layers close to the output) ensures
- similar = neighbor in high-dimensional feature space

→This is the reason why transfer learning works:
- train a deep network on one data base (e.g. imageNet/LLM)
- retrain only a few layers close to output for new task

Motivation: Convolutional networks (convnet) work well!

An image recognition task





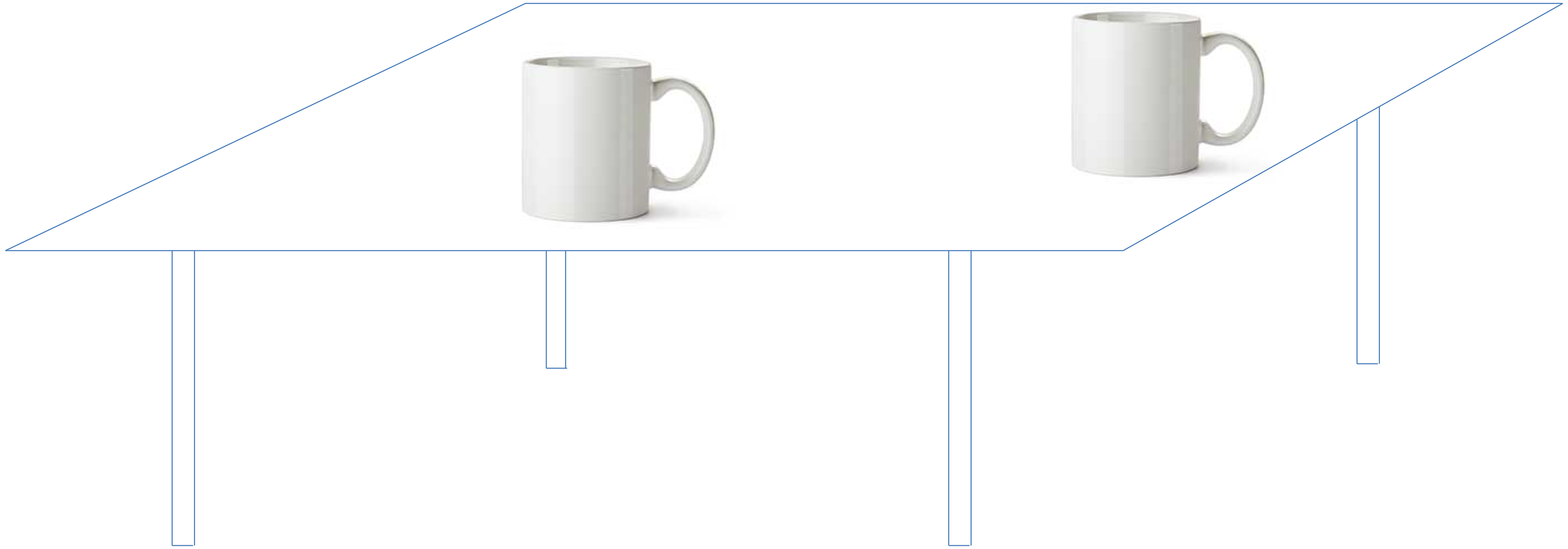
Ciresan et al.
Krishevsky et al.
deep convnets
trained on GPU

Fukushima (1982): Neocognitron
McClelland et al. (1996): Parallel Distributed Processing

Example: Why do convolutional networks work so well?

Convolutional networks provide an excellent inductive bias for image recognition:
object invariance to (local) translation

Inductive bias via network architecture (rather than data augmentation)



Convolutional networks (convnet) work well on images

- why do work well?
    - → answer: induce a good inductive bias
  - what is this inductive bias?
    - → local translation invariance of objects
  -

Quiz: Convolutional networks and No free lunch theorem

Why are convolutional networks better than other networks on image tasks?

[ ] They work better on images because they implement an explicit inductive bias that reflects known properties of images!

[ ] Classification of images needs nonlinear processing and this is the relevant inductive bias that distinguishes ConvNets from DeepReLu networks.

[ ] Classification of images needs to reflect local translation invariance of object classification.

Summary: No Free Lunch (NFL) Theorems and Deep Networks

More generally the prior structure of a deep network should match the structure of the real-world problems we are interested in.

→ Always use prior knowledge if you have some!

Example: - images: translation invariance (ConvNets)
- music: tone translation invariance, motif repetition
- known symmetries of tasks
- physics: energy conservation (Noether Networks)
- topological/hierarchical relations (Graph Neural Networks)

→ Use prior knowledge as ‘inductive bias’ in your algorithm!

Previous slide.

Prior knowledge is important. We can use prior knowledge when we design the network architecture.

Reinforcement Learning Lecture 3

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Continuous input space: function approximation

Part 6: Inductive Bias (Example)

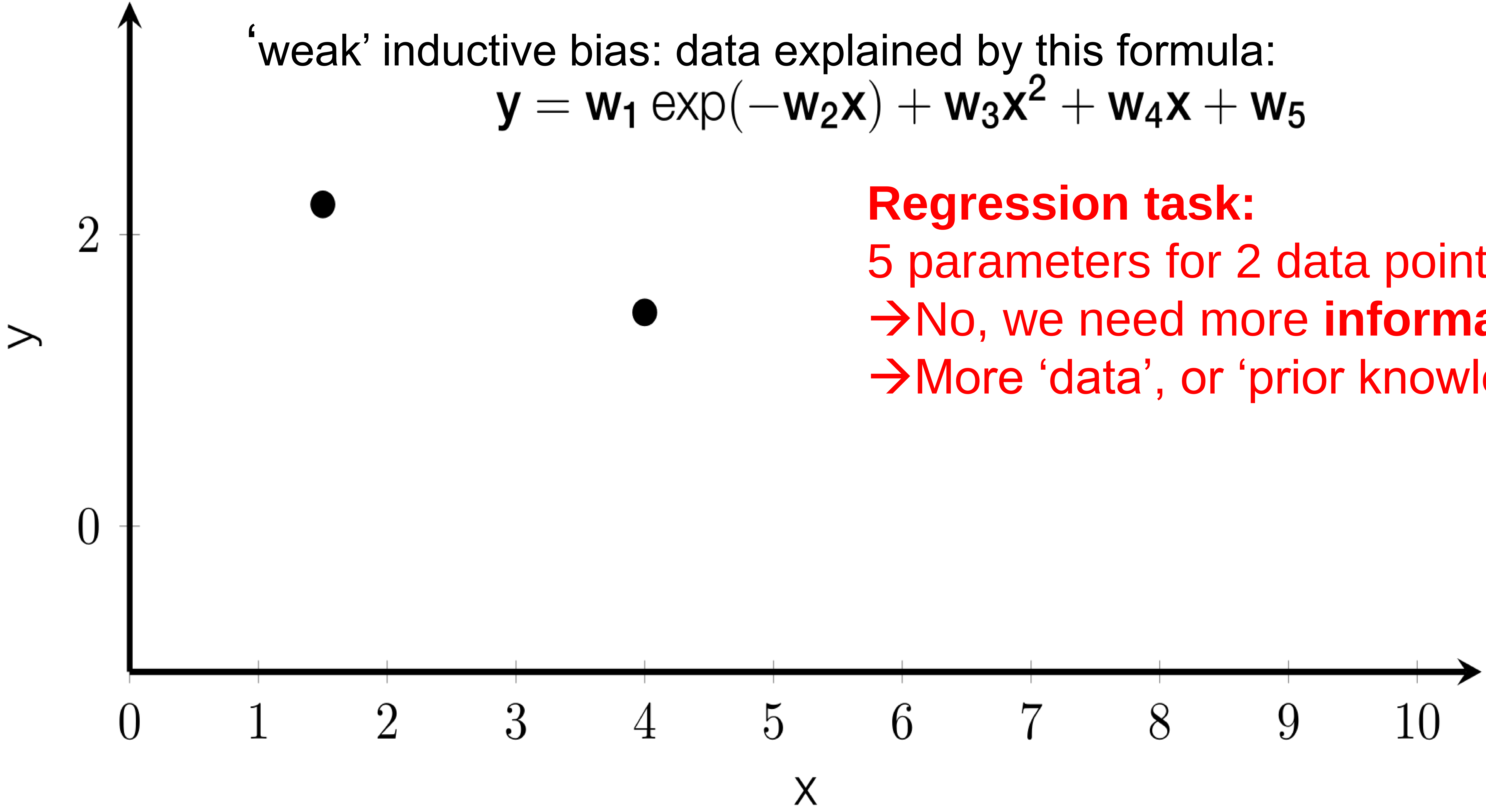
Previous slide.

We now turn to a small, but didactic example with only two data points.

review: No Free-lunch Theorem and Regularization

‘weak’ inductive bias: data explained by this formula:

$$y = w_1 \exp(-w_2 x) + w_3 x^2 + w_4 x + w_5$$



Regression task:

5 parameters for 2 data points?

→No, we need more **information!**

→More ‘data’, or ‘prior knowledge’

Previous slide.

The notion of Inductive Bias has a strong link to the problem of regularization.

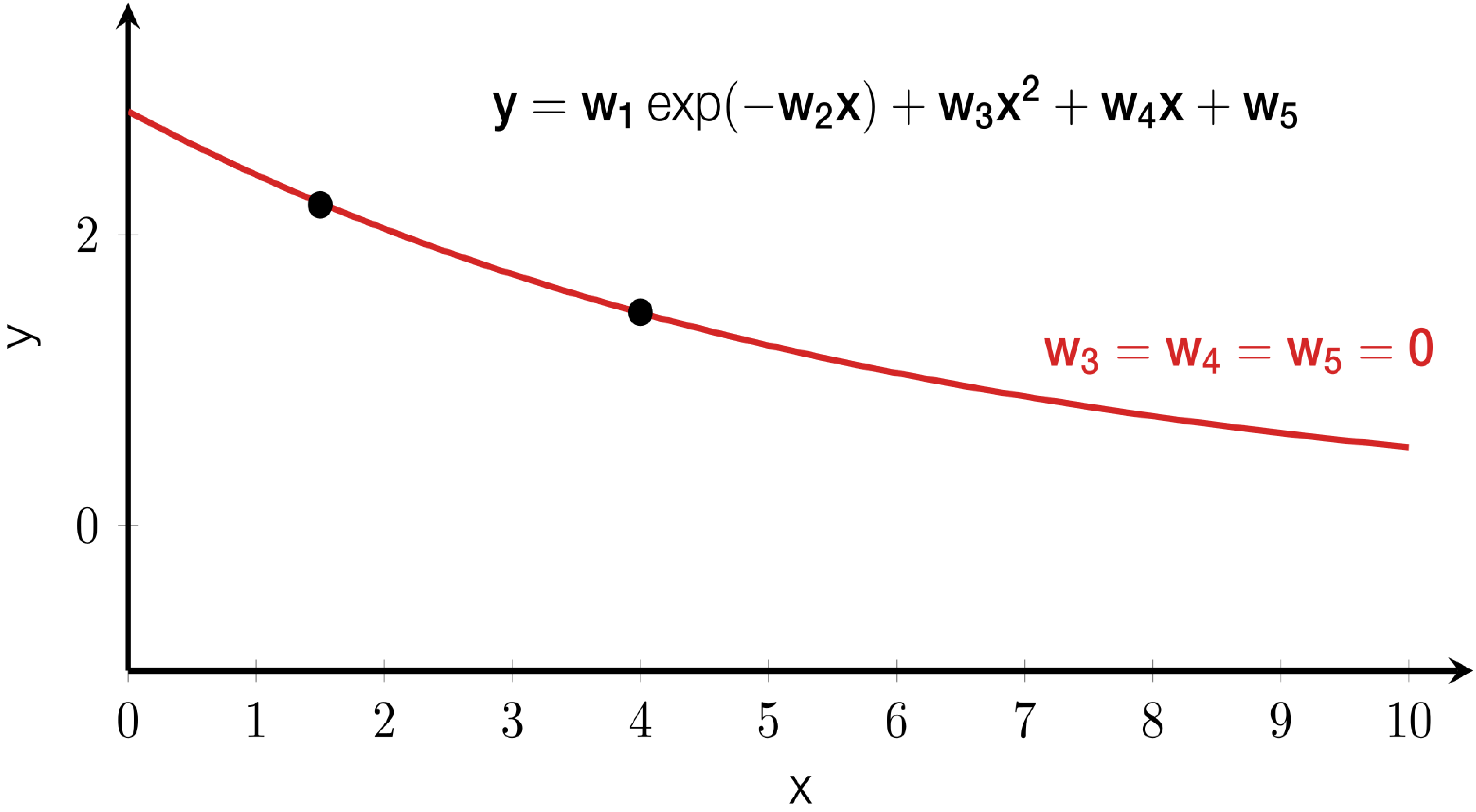
Let us suppose we have two data points in our training set. It is a regression task with input x and real-valued target y; if it were a classification task, y would be discrete.

There are infinitely many possibilities to fit these two data points, but let us assume that our initial inductive bias is that x and y can be linked through the displayed formula.

The fitting problem still is under-constraint, since we have more parameters than data points; we still have have infinitely many possibilities to fit these two data points with the displayed formula.

In other words our ‘inductive bias’ is ‘weak’, since the formula offers too much freedom.

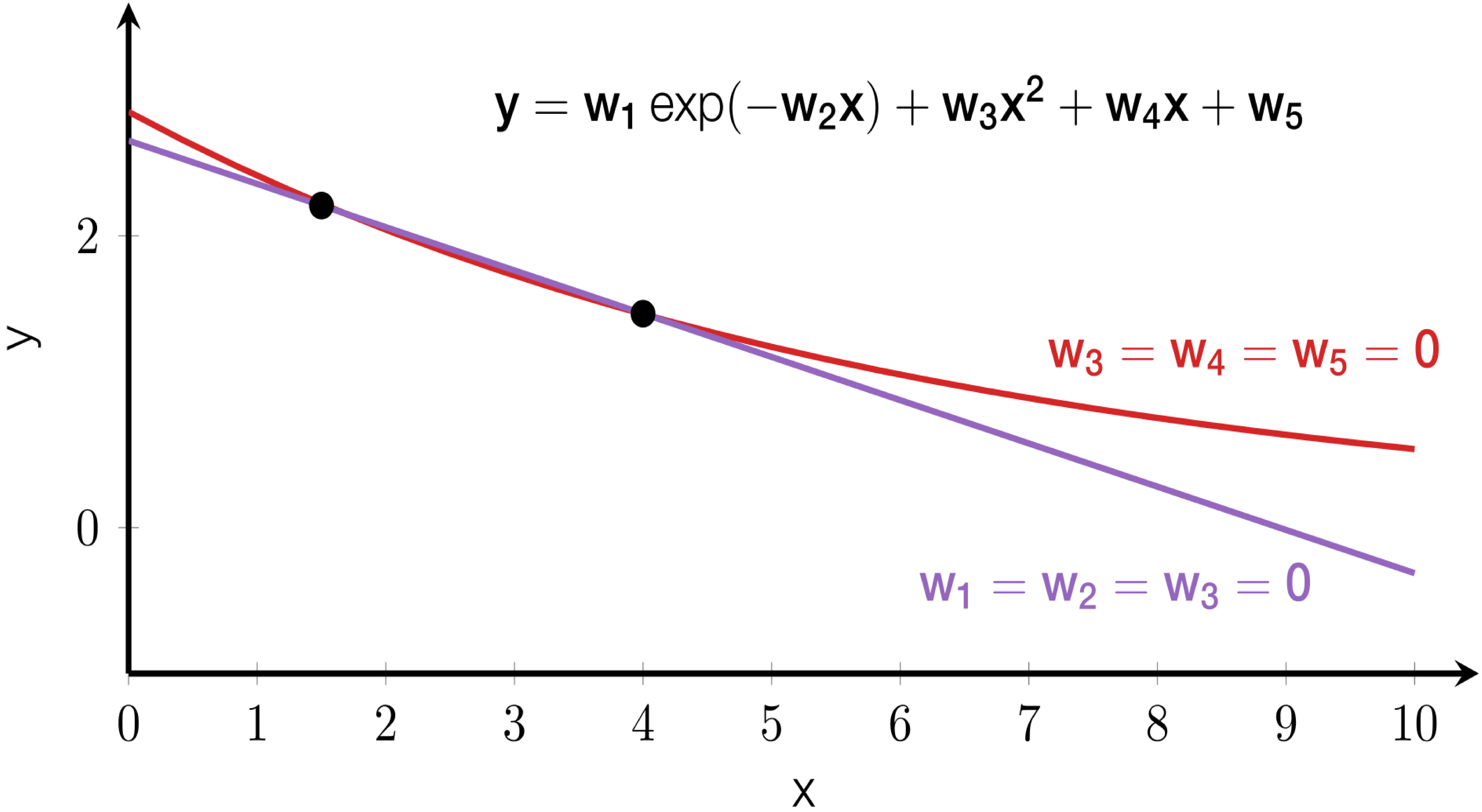
review: No Free-lunch Theorem (strong inductive bias 1)



Previous slide.

But let us assume we had reasons to consider a stronger inductive biases.
As a first example, exponential decay with $w_3 = w_4 = w_5 = 0$.

review: No Free-lunch Theorem (strong inductive bias 2)



$$y = w_1 \exp(-w_2 x) + w_3 x^2 + w_4 x + w_5$$

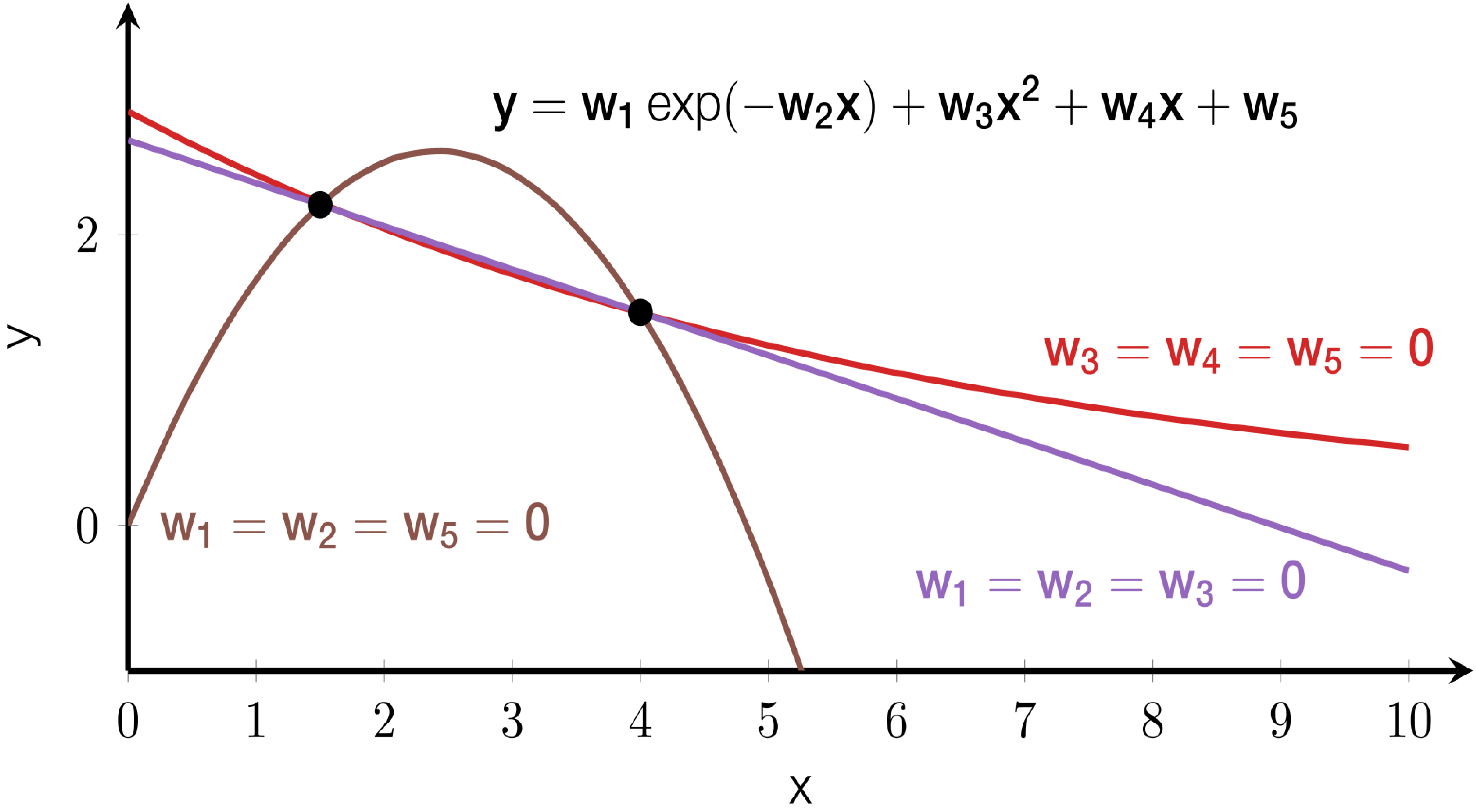
$w_3 = w_4 = w_5 = 0$

$w_1 = w_2 = w_3 = 0$

Previous slide.

Second option for strong inductive bias: we want to fit by a straight line, so that $w_1 = w_2 = w_3 = 0$.

review: No Free-lunch Theorem (strong inductive bias 3)



$$y = w_1 \exp(-w_2 x) + w_3 x^2 + w_4 x + w_5$$

$w_3 = w_4 = w_5 = 0$

$w_1 = w_2 = w_5 = 0$

$w_1 = w_2 = w_3 = 0$

Previous slide.

And third choice of a strong inductive bias, we assume that the data can be fit by a parabola through the origin with $w_1 = w_2 = w_5 = 0$.

Any other way of fixing at least three (of the five) parameters would be an valid alternative.

How should we choose between these inductive biases?

Inductive Bias: Use prior knowledge to constrain solutions

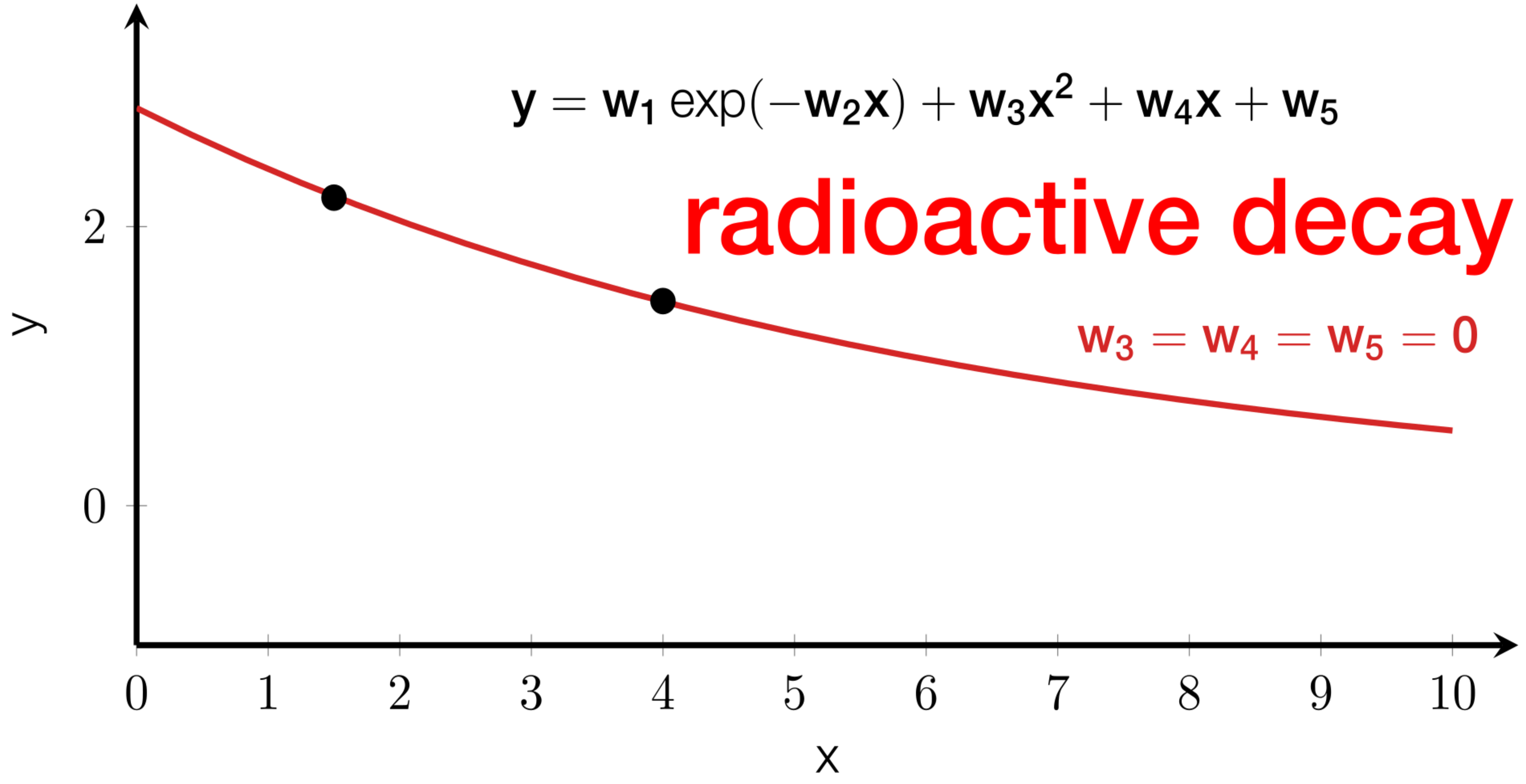
Question: What is a good inductive bias?

→ **Use your prior knowledge!**

Previous slide.

We always should choose our prior knowledge regarding the nature of the problem!

No Free-lunch Theorem: use inductive bias



Regularization with L1 norm on
w3
w4
w5

How can we transfer our knowledge about the task to the problem at hand?

1.Data scientist approach: fit many similar problems

transfer knowledge by identifying good regularization schemes

→ force w_3 and w_4 and w_5 to be (close to) zero

2. Explicit knowledge from physics: known law of nature

→ $w_3 = w_4 = w_5 = 0$

3. Data augmentation

Previous slide.

If we have more information about the data, e.g. it comes from measurements of radioactive decay, we can transfer our knowledge about this type of problem, select a strong inductive bias and fit the other parameters.

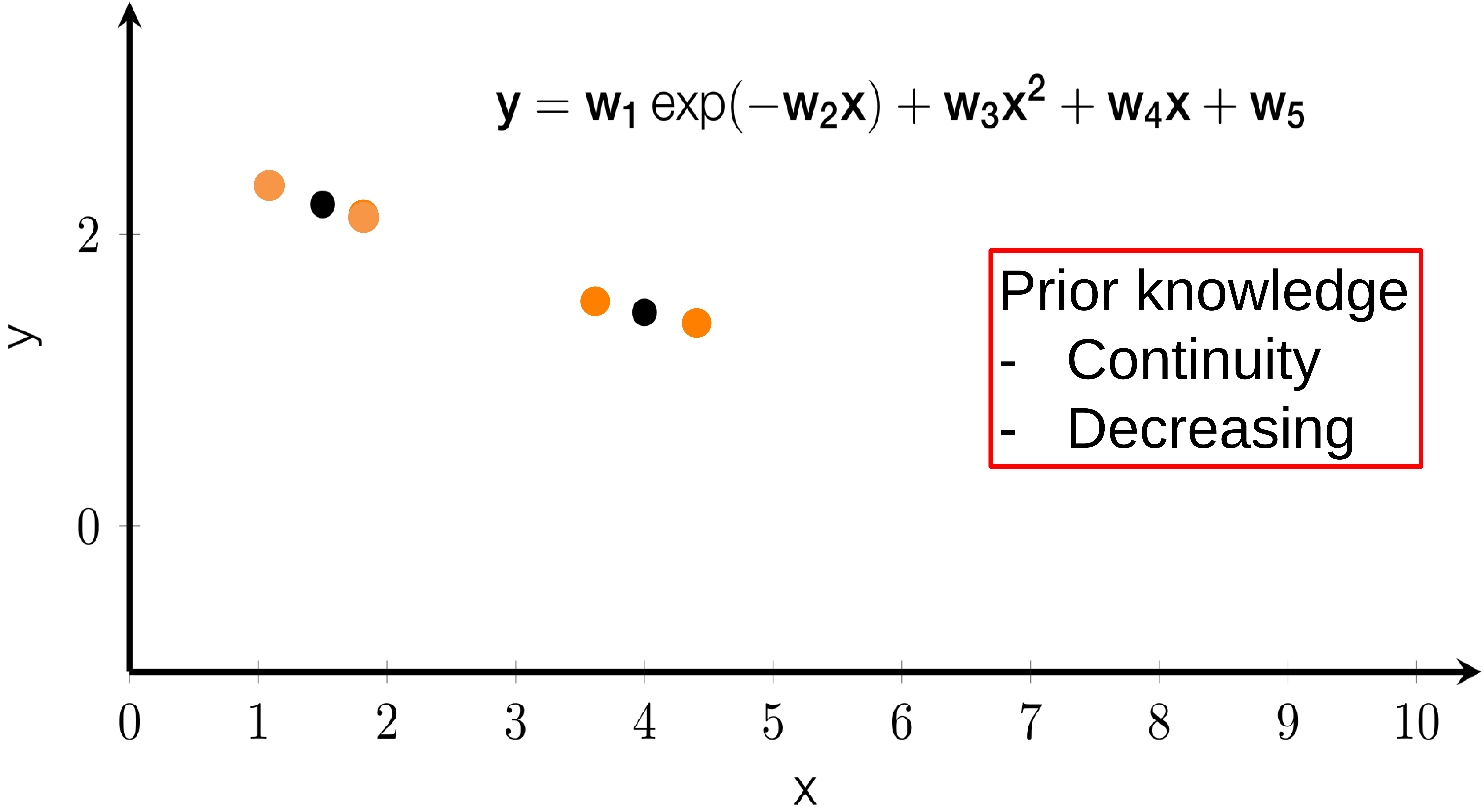
This transfer can happen in different ways.

1. (The data scientist approach) We have already several times fitted all parameters w_1 to w_5 to similar data and we have always observed that w_3 to w_5 were close to 0. Hence we decide to choose L1-regularization on w_3 to w_5 but leave w_1 and w_2 unconstrained.

2. (The physicist approach) There is some law that dictates a certain form of the function.

3. (Regularization by data augmentation) next slide.

review: No Free-lunch Theorem (data augmentation)



$$y = w_1 \exp(-w_2 x) + w_3 x^2 + w_4 x + w_5$$

Prior knowledge
- Continuity
- Decreasing

Previous slide.

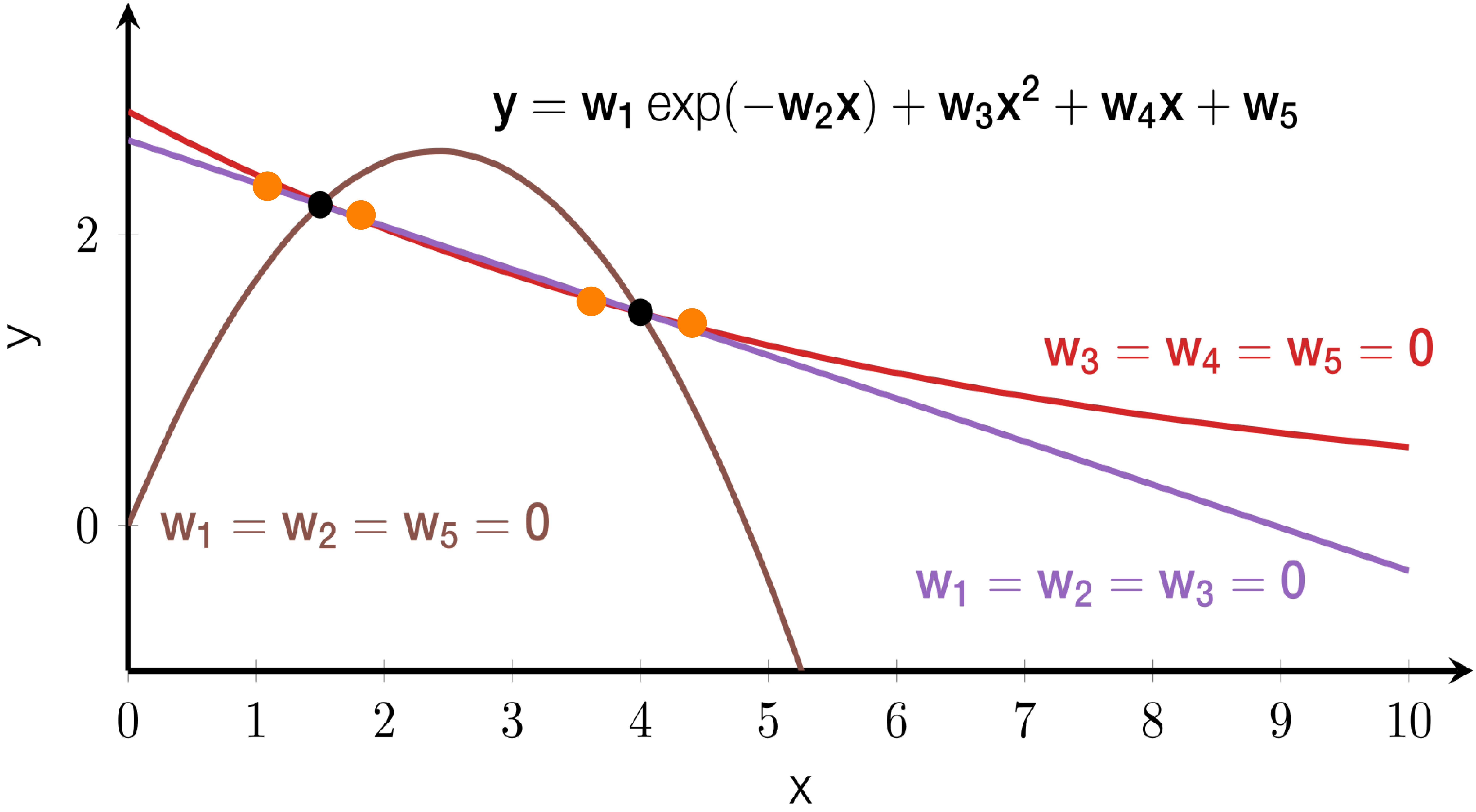
We may not know much about the data or find it difficult to define an explicit inductive bias but we have the intuition that

(i) the outcome should not change much if we transform the input in a certain way, e.g. we assume that if we would move the data points in the training set a bit to the left or to the right the outcome y should not be very different

(ii) The output should monotonically decrease as a function of x, as indicated with the orange data points.

With this augmented data set we can fit all 5 parameters!

review: No Free-lunch Theorem (data augmentation)

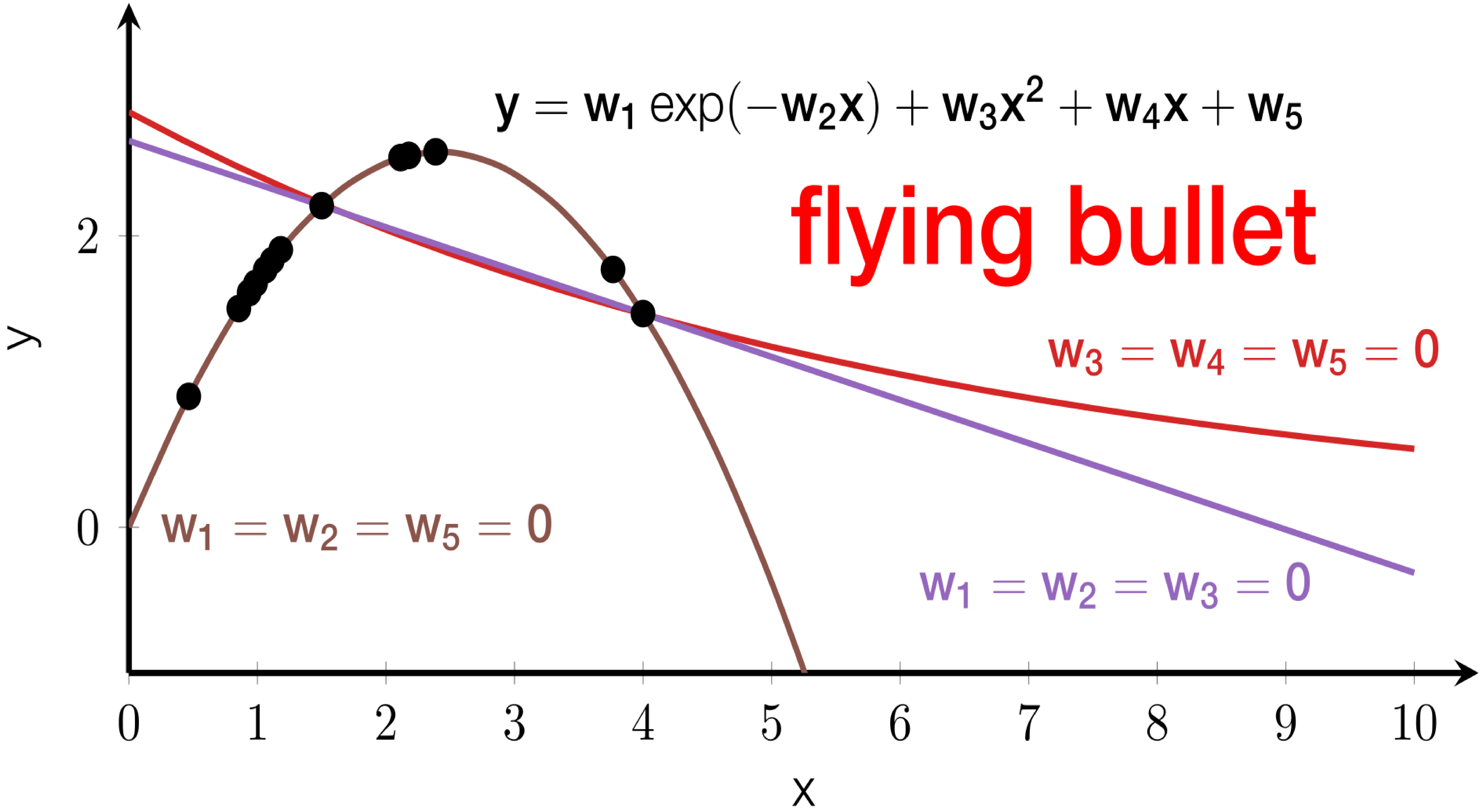


Previous slide.

Caveat: It may be difficult to find transformations that really leave the outcome invariant; getting more actual training data is typicall better than data augmentation.

Thanks to data augmentation (WITH SMART TRANSFORMATION!), or transfer from related problems we may be able to find a fit to only two data points that generalizes well if the data actually came from a measurement of radioactive decay.

review: No Free-lunch Theorem (the wrong inductive bias)



$$y = w_1 \exp(-w_2 x) + w_3 x^2 + w_4 x + w_5$$

flying bullet

Previous slide.

Even though the training error is zero for all three strong inductive biases considered here and we have the same degrees of freedom in each case (namely two parameters to be fitted) the performance on the test set can be terrible, if we pick the wrong inductive bias for the data at hand. If the data came from the measurement of the trajectory of a flying bullet the fit with the exponential decay or the straight line would not generalize well.
In other words, we don’t get good generalization for free; there is no free lunch here. Only if we choose the inductive bias that matches the data we get good generalization.

Inductive bias

Induction = finding a rule (function) from specific examples
Inductive bias = prior preference for specific rules (functions)

1) Explicit inductive bias (transfer reasoning)

“For radioactive decay I know that $w_3 = w_4 = w_5 = 0$”

2) Inductive bias through transfer learning

“I first train different models on data from other radioactive elements and choose the best model class for my current case”

3) Inductive bias through data augmentation

“For radioactive decay neighboring points have similar values and values can only go down”

Quiz: Inductive bias

□ [ ] With a strong (and correct) inductive bias, I can reach
a low test error with very little training data.

□ [ ] With a strong inductive bias the test error will always be low.

□ [ ] Data augmentation is a heuristic method to get more training data.

□ [ ] In data augmentation there is an inductive bias in the form of our assumptions
about reasonable transformations to be applied to the data

□ [ ] Choosing a specific neural network architecture is equivalent to
choosing an explicit inductive bias.

Reinforcement Learning Lecture 3

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Continuous input space: function approximation

Part 7: Inductive Bias in Reinforcement Learning (Examples)

Previous slide.

Using a specific example we want to illustrate why function approximation yields an inductive bias for generalization.

Inductive bias in Reinforcement Learning

Before you code an RL problem, try to answer the following questions:

1) Is the problem such that in similar (neighboring) input states the best action is (likely to be) the same?

2) Is the problem such that if I find the reward with action $a*$ from state s, then $a*$ is probably good in other states as well?

3) Do I expect rewards in many states or rather only in a few ‘goal states’?

4) Moreover, are rewards given for states or state-action transitions?

5) Is there a topology/neighborhood relation that would enable us to talk about two actions as being ‘similar’?

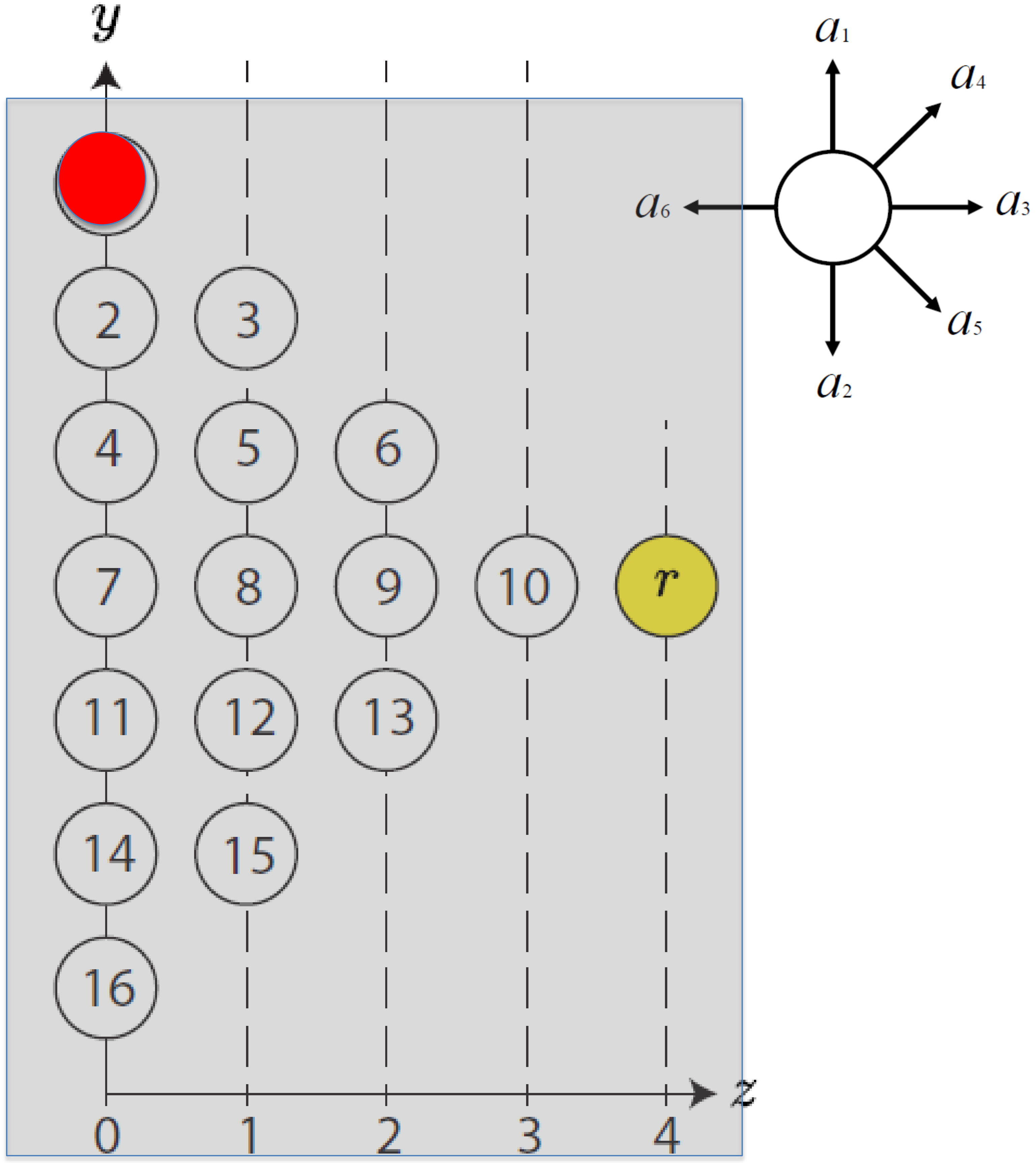
Previous slide.

If you know the answer to one of the questions you can use this knowledge to choose your coding scheme for inputs and for the action space.

Inductive bias in Reinforcement Learning (Example 1)

-16 discrete states + goal

- up to 6 actions per state



$$Q(a,X) = \sum_j w_j x_j$$

1-hot coded

$$X = (x_1, x_2 \dots , x_{18})$$

$$x_{17} = 0.5(z+1)$$

$$x_{18} = \ 0.1$$

More generally

$$x_{17} = \alpha(z-\beta)$$

Claim: this scheme encodes an inductive bias related to some of the questions

Previous slide.

We consider a specific example of how function approximation yields an inductive bias for generalization.

We have 18 parameters. The first 16 parameters are multiplied with one-hot encoded representations of the 16 states.
The other two parameters scale specific functions: an affine function and a bias.

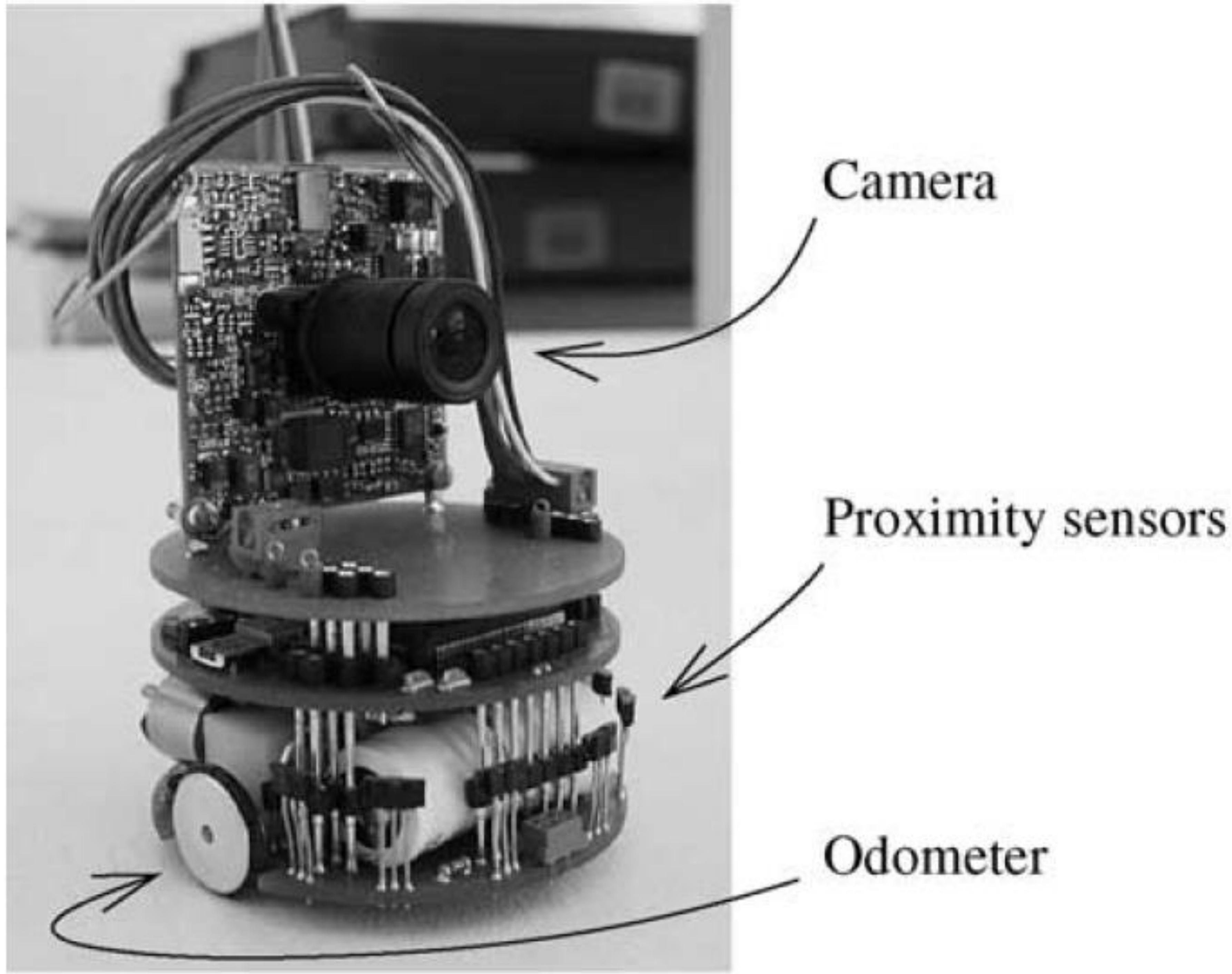
In the exercises you will explore how this choice implements an inductive bias.

Inductive bias in Reinforcement Learning (Example 2): Self-localization and Navigation to Goal

- 2-dimensional arena 80cmx60cm
- single goal location: reach goal from arbitrary start location
- 120 actions (=directions of movement)

Agent:
Khepera Robot

Camera:
240^0 view
>240 000 pixel
Preprocessing:
Gabor filter bank



Previous slide.

The camera of the Khepera robot makes snaptshots in 4 directions that are combined into a single ‘view’ covering a viewing field of 240 degree (total would be 360 degree). This corresponds to > 240 000 pixels per view.

The image serves as input that represents the present ‘state’.

The robot moves in a square (or circular) arena.

The task is to find an goal location (not marked!), from any possible start configuration.

Inductive bias in Reinforcement Learning (Example 2): Self-localization and Navigation to Goal

- 2-dimensional arena 80cmx60cm
- Task: reach goal location (5cmx5cm) from arbitrary start
- 120 actions (=directions of movement)
- state = camera input = >240’000 pixels

How many episodes do we need to train the agent to solve the task?

[ ] <20

[ ] 20 – 200

[ ] 200 – 1000

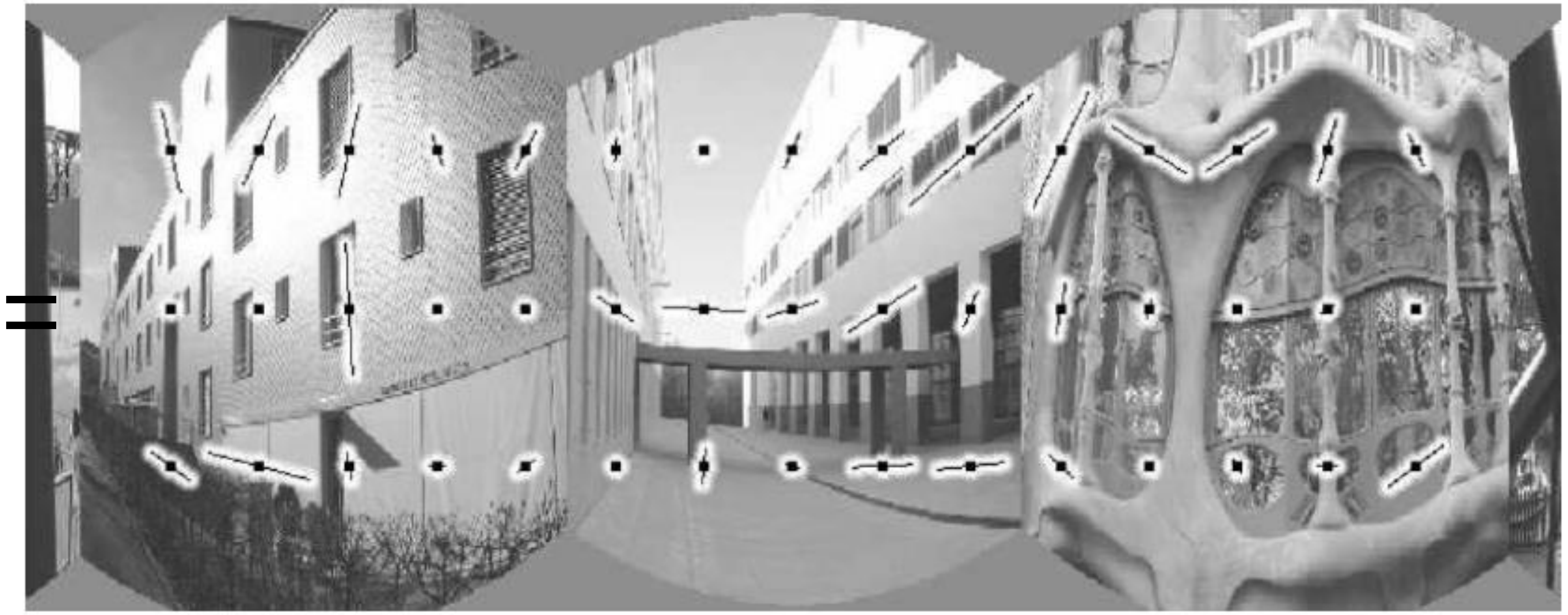
[ ] > 1000

What is a good Inductive bias for Self-localization and Navigation to Goal, starting from visual input?

We start with images of > 240’000 pixels, but

Inductive bias in Reinforcement Learning (Example 2)

- **Preprocessing Gabor filter bank:**
  Filters of several spatial frequency and orientation
  at 45 different locations,
  200 filters per location.



- **Snap-shot of environment =**
  store the vector F_j
  of 9000 filter responses

- **'Basis-function'** $\phi\left(F(t)-F_j\right)$
  similarity of current view $F(t)$ with stored view vector F_j
  after rotation to optimal matching angle

sample basis function

Previous slide.

The camera of the Khepera robot makes snaptshots in 4 directions that are combined into a single ‘view’ covering a viewing field of 240 degree (total would be 360 degree). This corresponds to > 240 000 pixels per view.

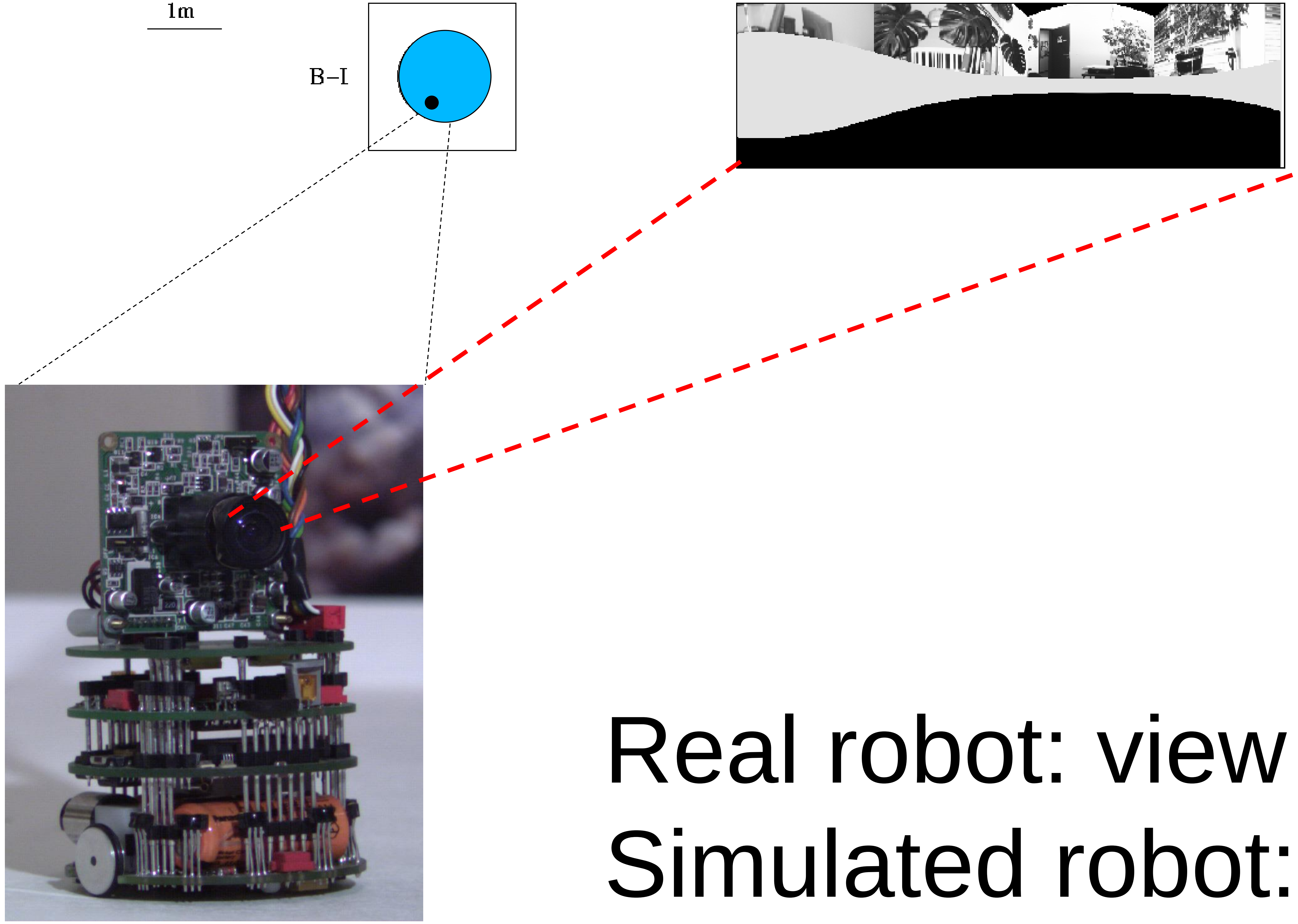
The sample image shows the orientation of the most strongly responding filter with the lowest spatial frequency at the 45 sampling locations.

The Gabor filters come as pairs of sine and cosine filters (or complex filters) and only the total amplitude, but not the phase of the response of the filter pair is recorded.

The set of filter responses at time t of all 9000 filters is denoted by $F(t)$

Details of the processing steps are explained in the next few slides

Self-localization and Navigation to Goal:
Details of visual processing and Extraction of Basis Function



Real robot: view field 4X60°

Simulated robot: view field 280°

Previous slide.

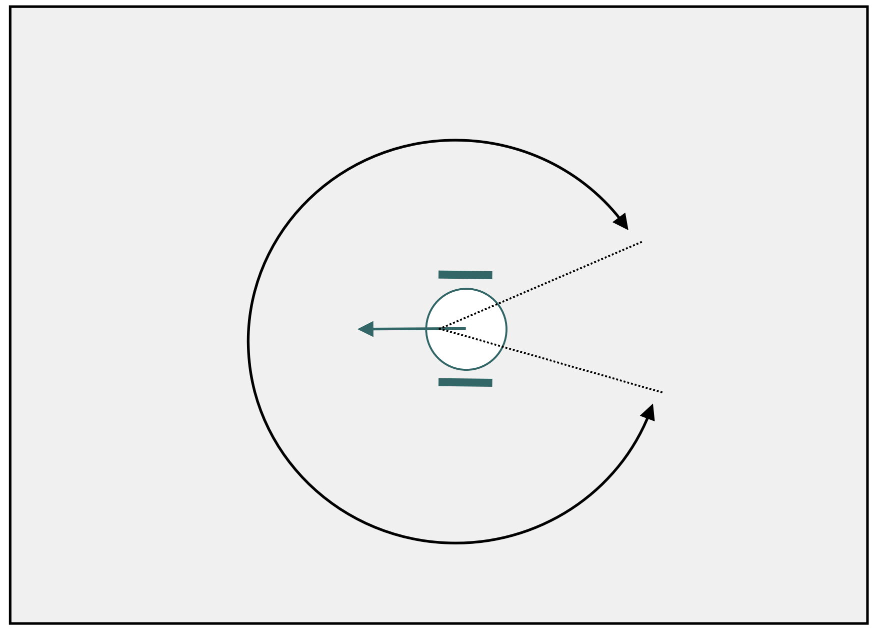
The robot takes a sample image.

With a real robot: we let the robot rotate around its own axis to take views in 4 directions, each view over 60 degree angle; the four views are considered as a single image of view angle 240 degrees.

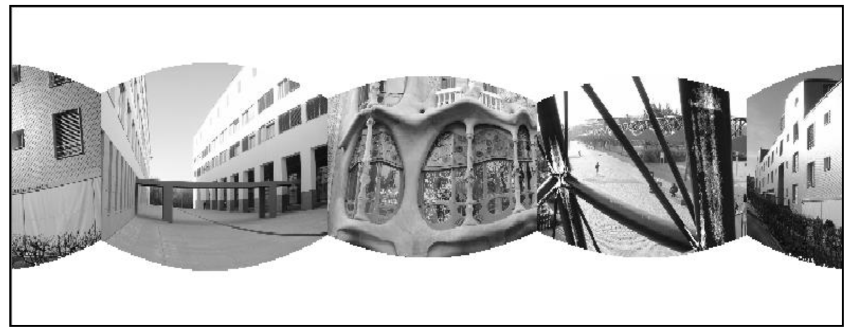
In simulated robots one case use directly 280 or 300 or even 360 degree as a viewing angle.

Model: stores views of visited places

Robot in an environment

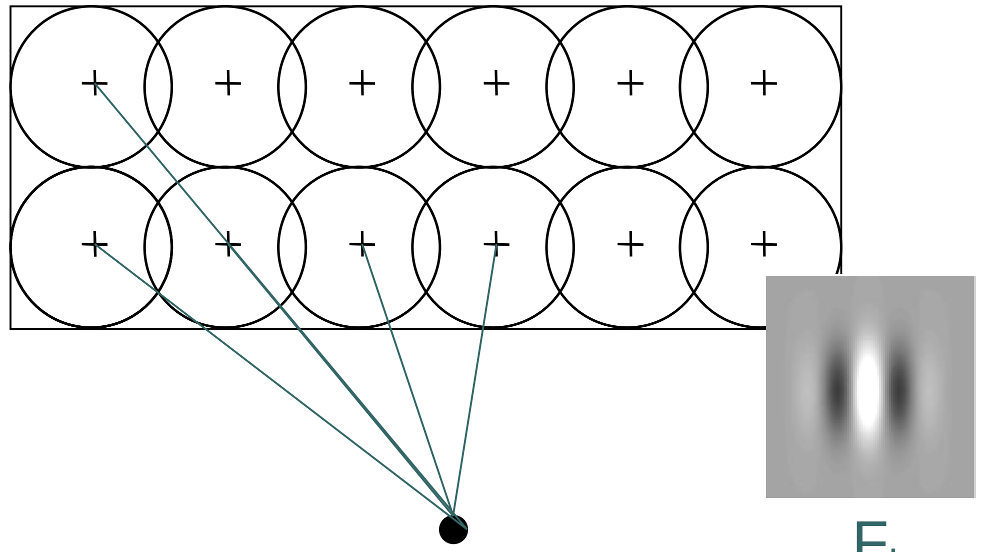


Visual input at each time step



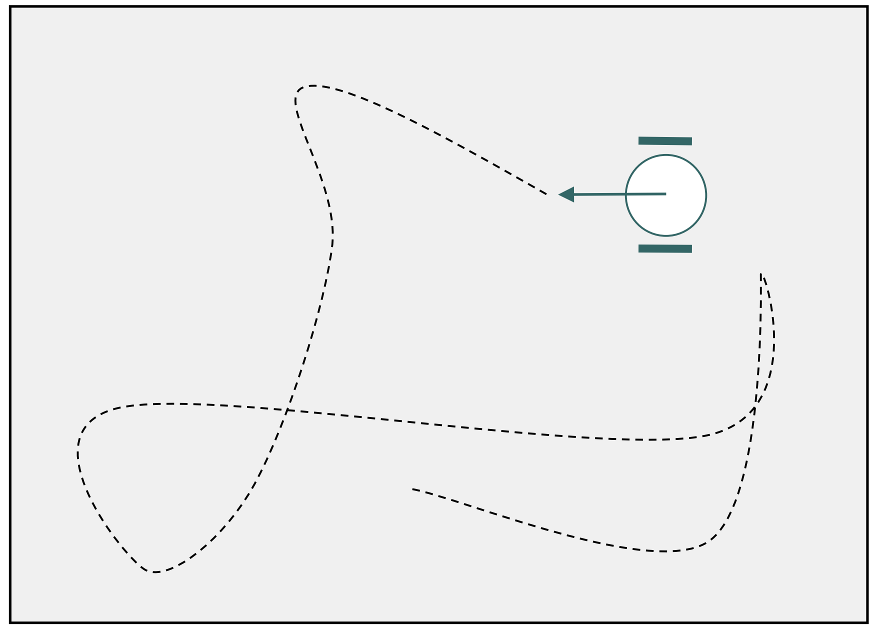
Local view : activation of set of 9000 Gabor wavelets

$$L(\vec{p},\Phi)=\{F_k\}$$

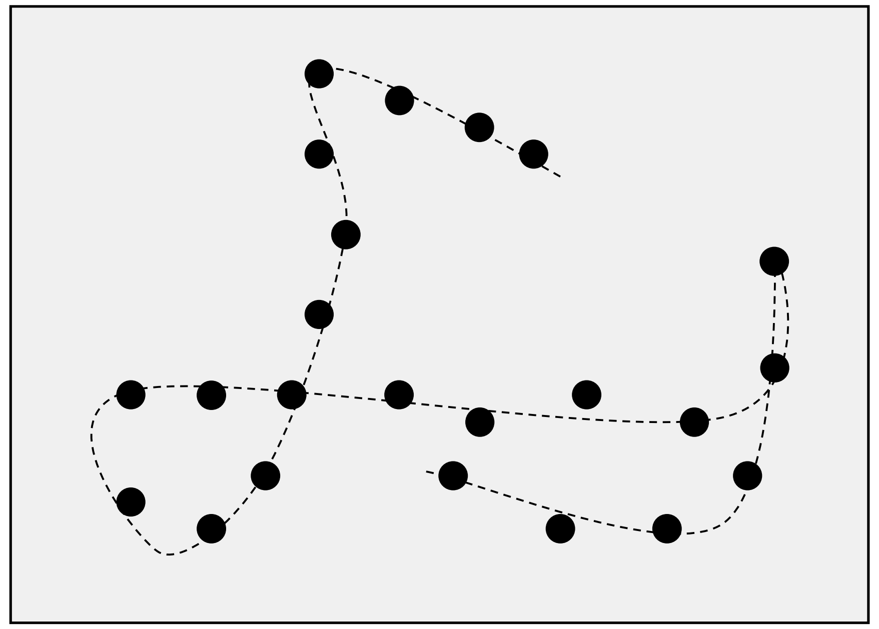


Single View Cell stores a local view

Environment exploration



Population of view cells



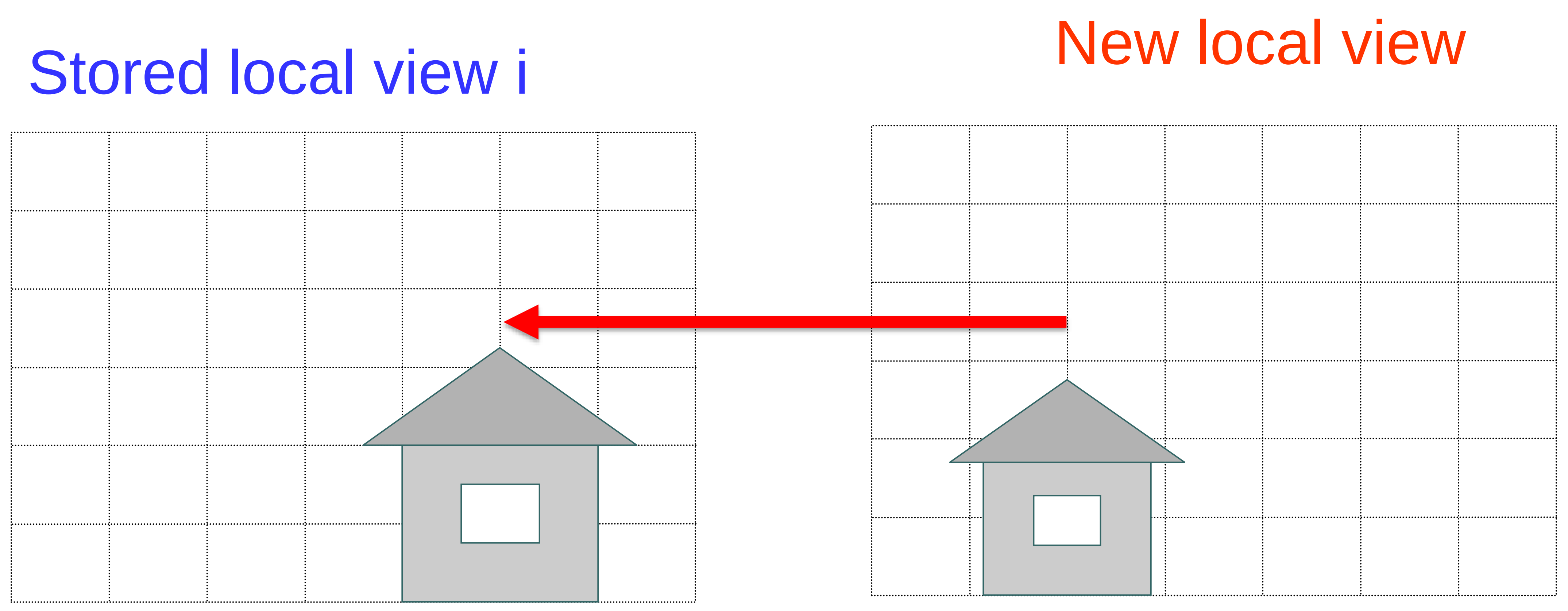
All local views are stored in an incrementally growing view cell population

Previous slide.

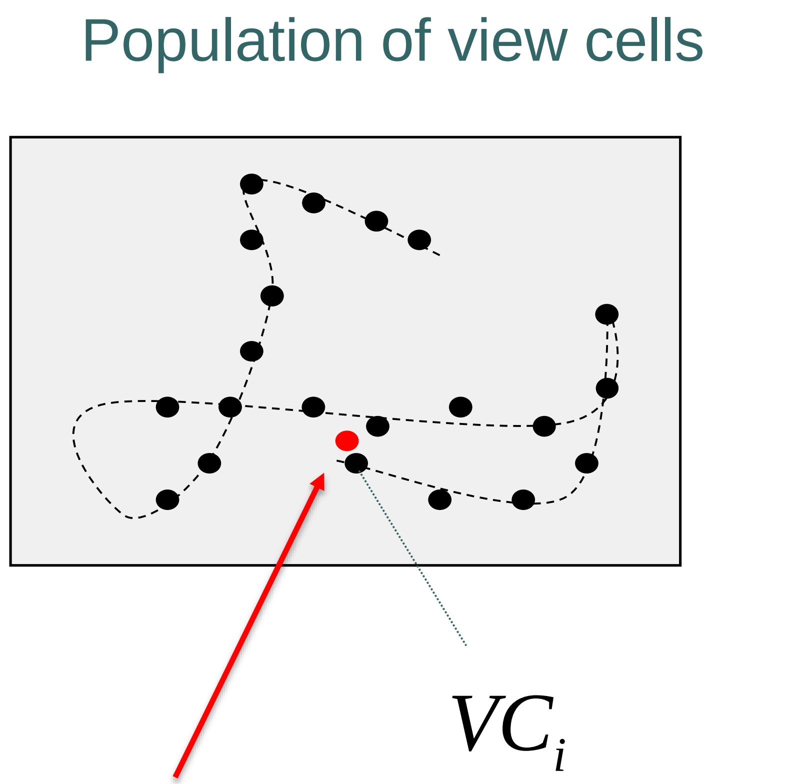
During exploration the robot takes a new sample image whenever it does not recognize the view. Recognition is defined that 10 or more cells strongly respond to the new image.

The sample image is memorized by storing the set of responses of the 9000 Gabor filters.

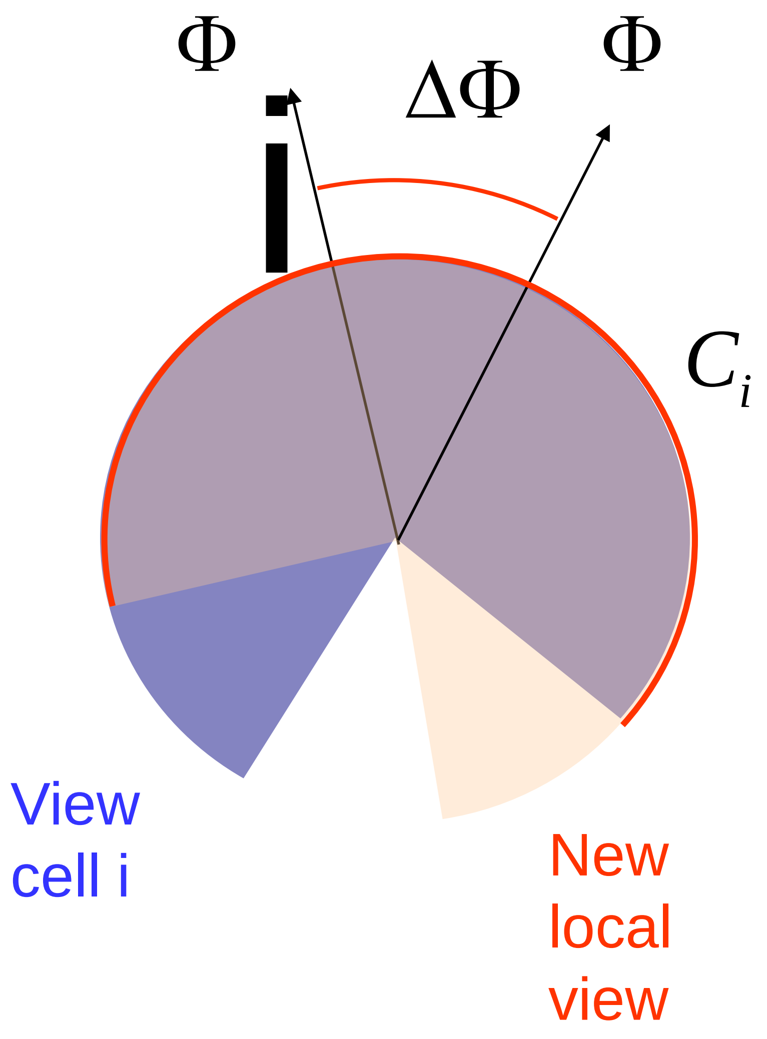
Model: extracting gaze orientation


Stored local view i

New local view


Population of view cells

position at new local view



View cell i

New local view

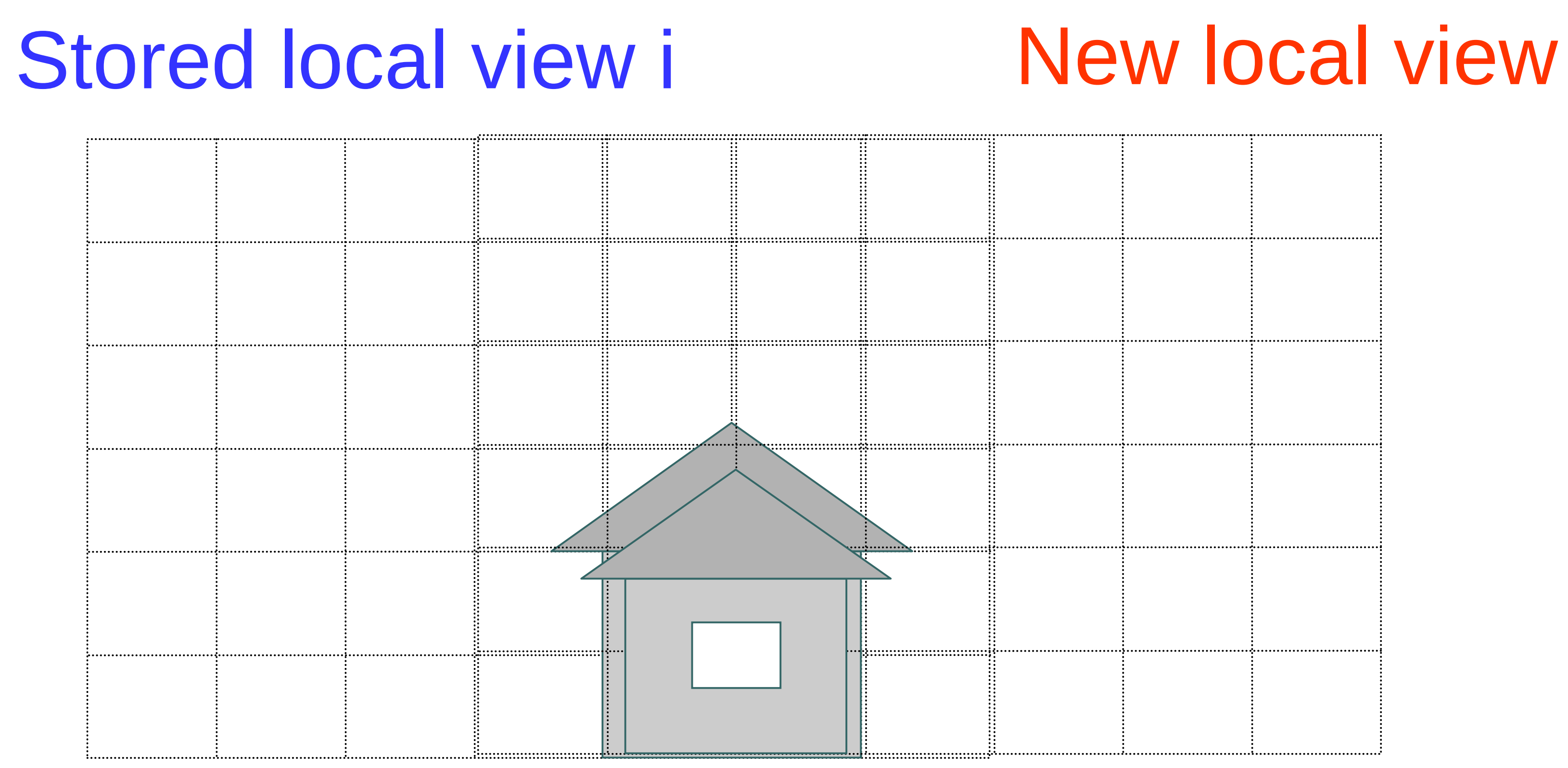
Alignment of views → current gaze direction

Previous slide.

The filter responses at time t are compared to the stored filter responses.
To find the best match the new image is rotated.

The angle of rotation (necessary to yield the best match) tells us about the direction of gaze compared to the gaze direction at the moment when the original image was stored.

Model: extracting position via basis functions



Small difference between local views – spatially close positions

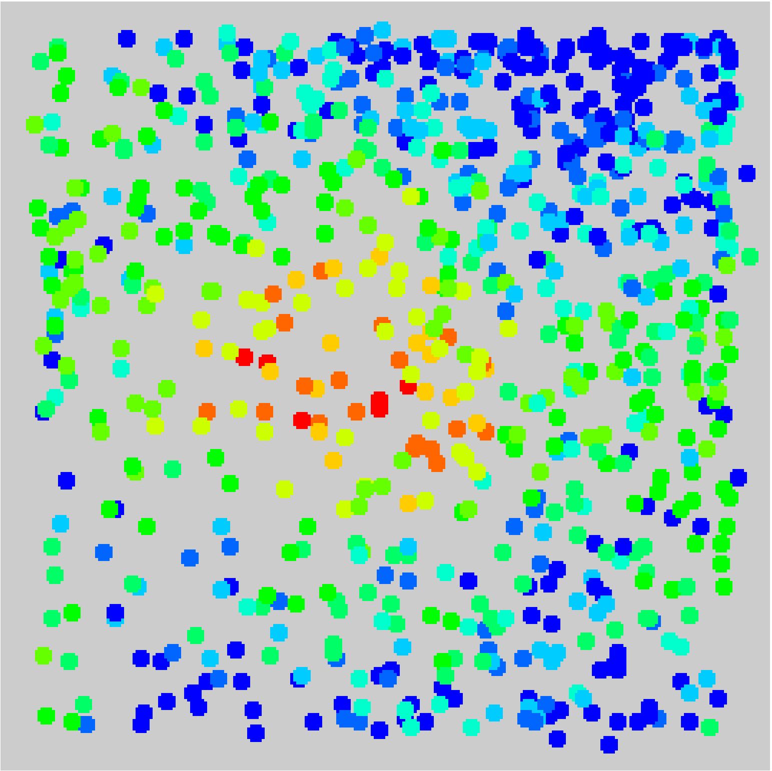
Vector of filter responses

Difference: $\Delta L = \left| F_i - F(t) \right|$

Similarity measure: $r_i^{VC} = \exp\left(-\frac{(\Delta L)^2}{2\sigma_{VC}^2} \right)$

Basis Function $\phi(F(t) - F_i) = r_i^{VC}$

population of view cells responding at red position



Previous slide.

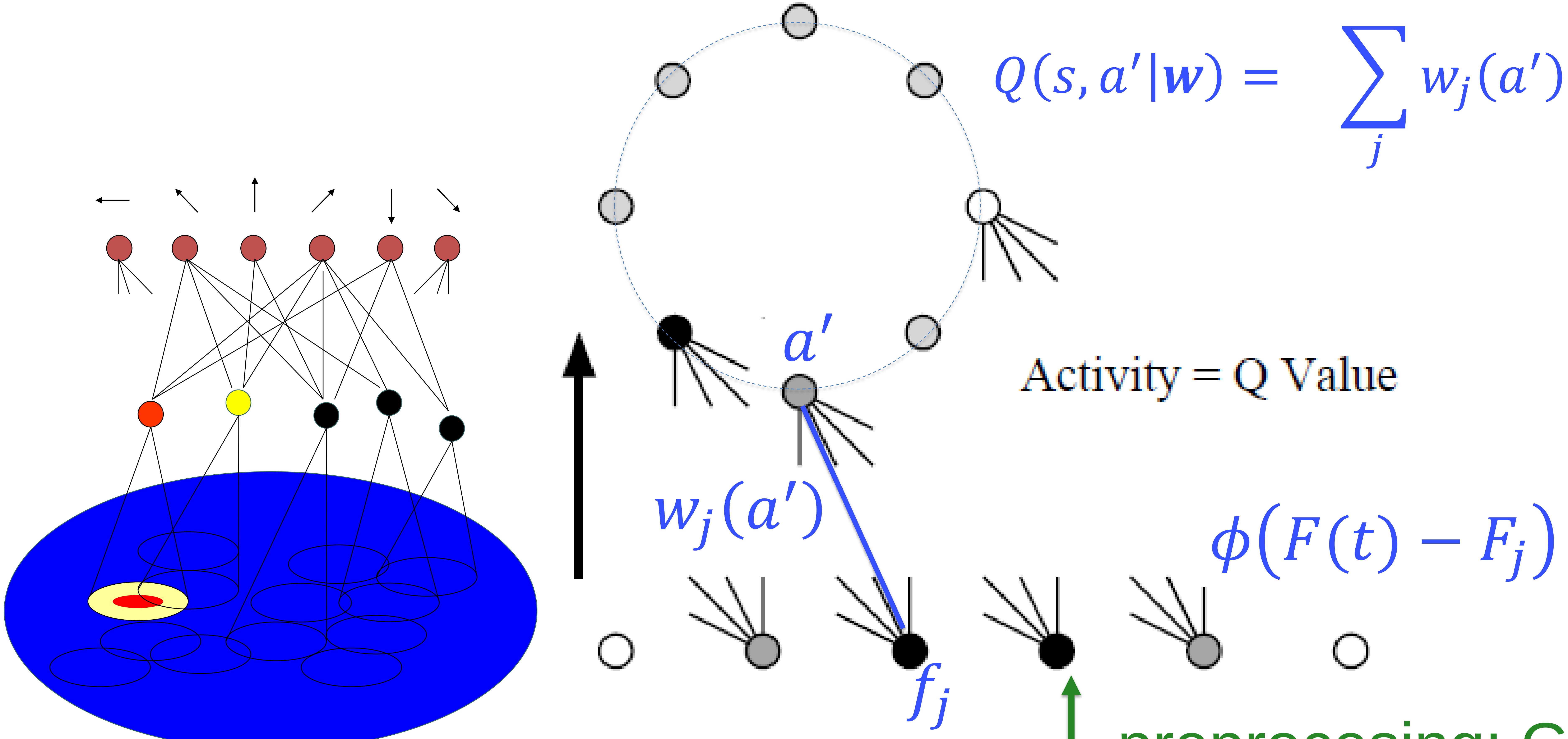
After the rotation to best-match position. the filter responses at time t are compared to the stored filter responses. This yields the basis function.

The image on the right shows which basis functions respond when the robot is at a specific location. Red indicates strong response.

Note that basis functions do not know where they are located in space (i.e. they have no spatial position label, but just their response profile and an index for each basis function). For this image we have plotted a dot at a place that corresponds to the location of the maximal response of a given basis function. But this is **for visualization purpose only.**

Summary: From Pixel input to Basis function to Q-values

$$Q(s, a'|\mathbf{w}) = \sum_j w_j(a')\phi\big(F(t) - F_j\big)$$



state s(t)

preprocesing: Gabor filters + rotation/alignment

Previous slide.

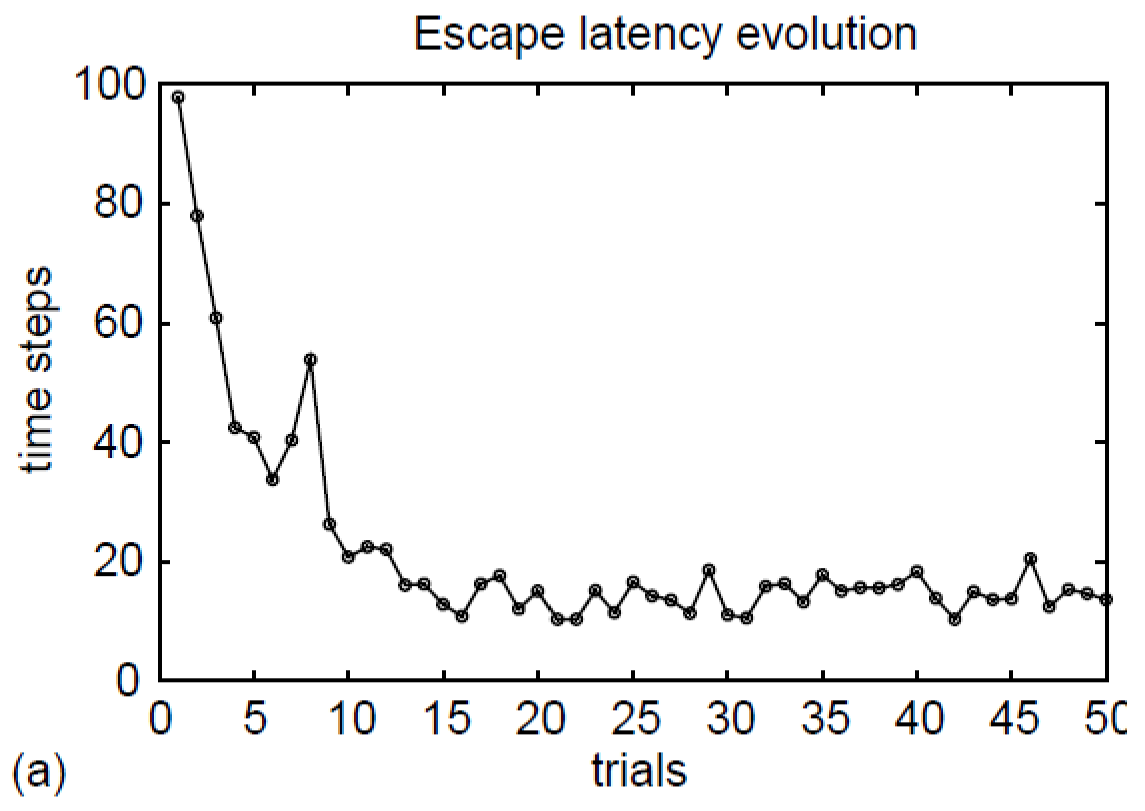
Action neurons represent the Q-values. In total there are 120 neurons. We may consider them to lie on a circle with a position on the circlue corresponding to the direction of movement triggered by the action.

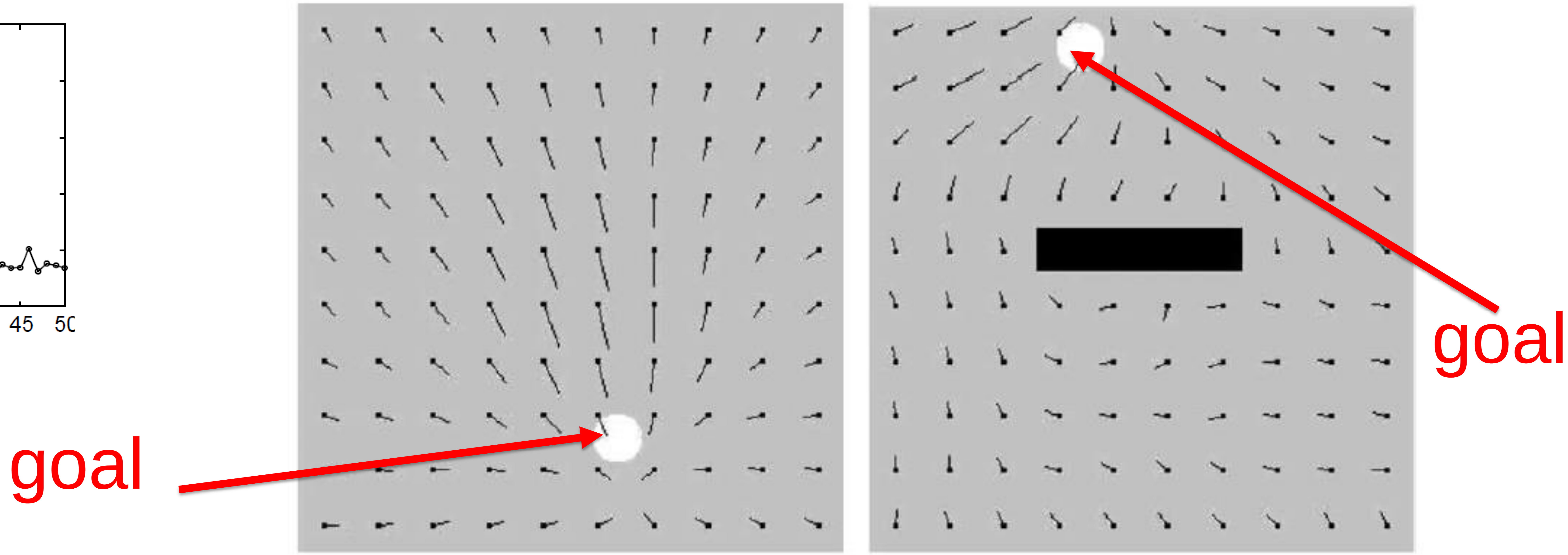
The center f_j of each basis function j corresponds to the (stored) response of thousands of Gabor filters recorded at some time t_j during exploration. The output of the basis function j measures the similarity with the current view, represent by the current response of the Gaborfilters, The vector of all Gabor filter responses at time t is f(t).

The figure on the left shows rather schematically the net result of the processing steps. The functions ϕ are visualized as local basis functions in the environment. Weights connect to actions that code for the different movement directions. The activiation of each action unit indicates its Q-value.

Inductive bias in Reinforcement Learning (Example 2): Self-localization and Navigation to Goal:

- While exploring: take new snapshot whenever less than 10 basis functions are active → creates new basis function
- Reinforcement Learning by Q-learning
- Final action directions after 20 trials (goal-findings)





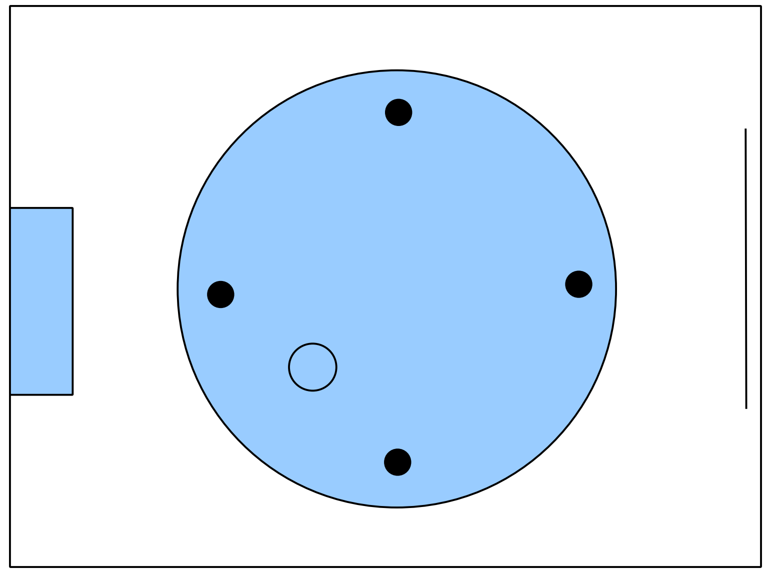
Previous slide.

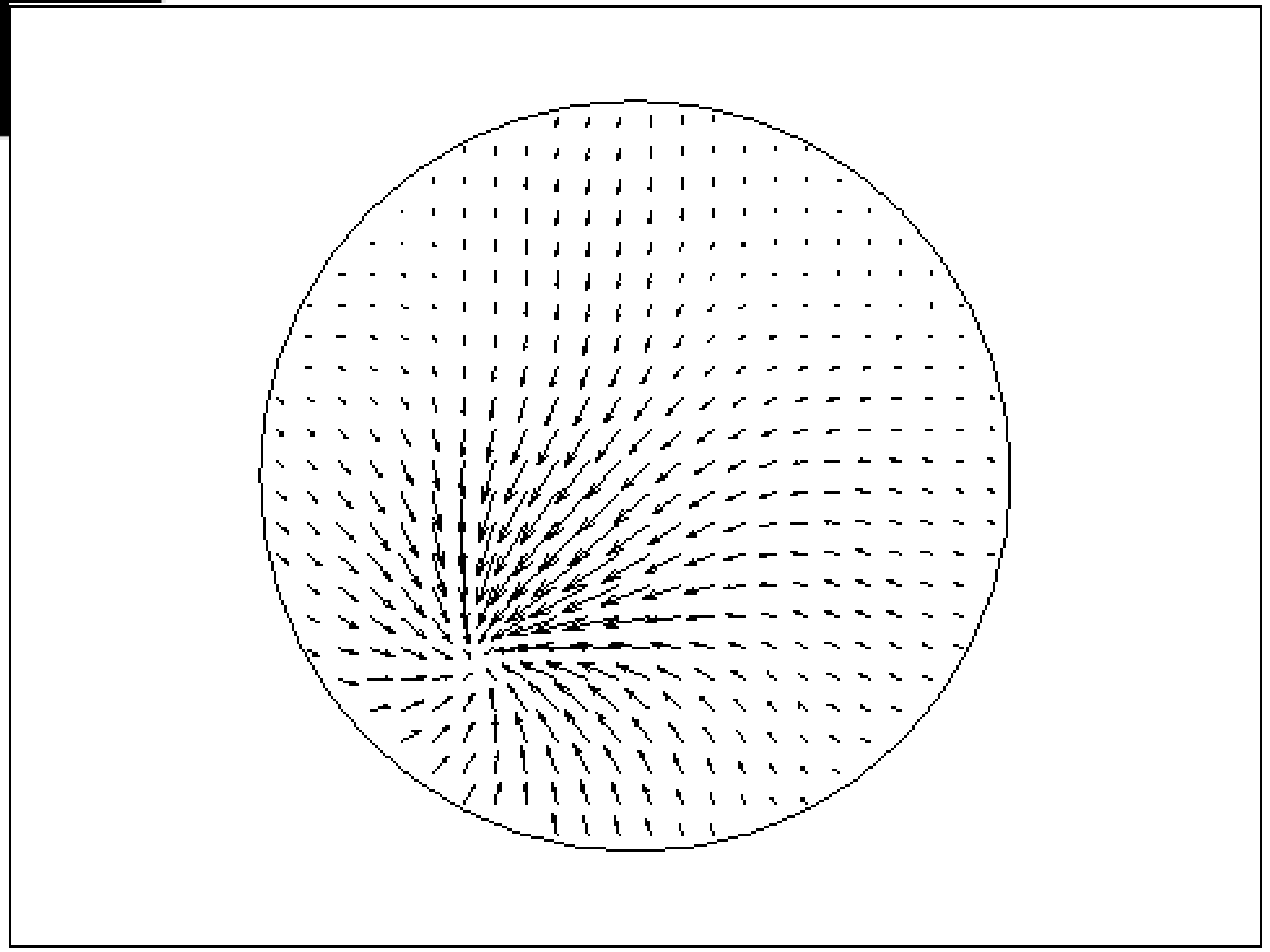
The left image shows the time it takes to find the goal, as a function of successful trials (episodes).

The right image shows needles that indicate the learned direction of movement after 20 trials.

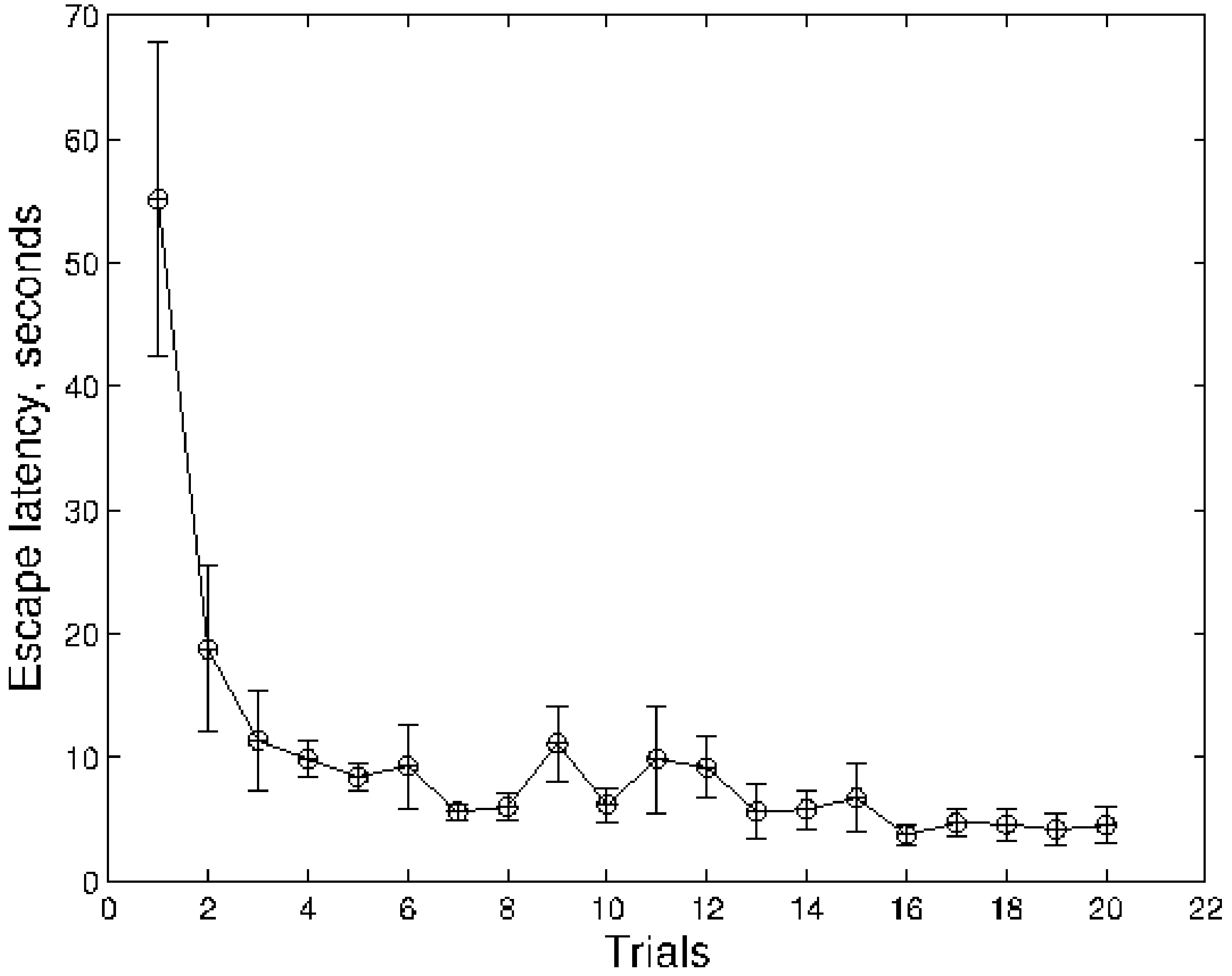
Navigation Results: Office environment

Sheynikhovich et al. Psychological Review, 2009





- Map after 10 trials



- Learning = relate present view (location) to movement direction
- Needs alignment of the views to know orientation

Previous slides.

- Coding of input space: we sample vectors of feature responses in the high-dimensional space, but we know that in the end they encode only two dimensions, so that sampling is indeed possible.

- The space is further reduced from 3 to 2 dimensions by algorithmic rotation of images (= shift of feature vectors) to get rid of difference due to orientation.

- We can work with relatively long eligibility traces, since there is a single goal state.

- We generalize across actions: we imagine actions forming a ring of possible directions. Neighboring actions should learn (in most states) similar behavior, hence if action a* is chosen with SARSA and learns (at rate eta), then all its neighbors learn as well (but with slightly reduced rate).

Inductive bias in Reinforcement Learning (Example 2)

- input: 240 000 pixels (or values of 9 000 Gabor filters)
  → high-dimensional!
- output: 120 actions
  → high-dimensional!

Why does it work?

What is the ‘real’ input dimension? (states)

What is the ‘real’ output-dimension? (actions)

Inductive bias in Reinforcement Learning

Before you code an RL problem, try to answer the following questions:

1) Is the problem such that in similar (neighboring) input states the best action is (likely to be) the same?

Yes → broad overlapping representation of states is possible.

low intrinsic dimensionality of state space → sampling possible

2) Is the problem such that if I find the reward with action a^* from state s, then a^* is probably good in all states?

No, not in presence of obstacles or objects in the middle

→ global representation of states is not useful.

Inductive bias in Reinforcement Learning

3) Do I expect rewards in many states or rather only in a few 'goal states'?

Single goal state → long eligibility traces possible.

4) Moreover, are rewards given for states or state-action transitions?

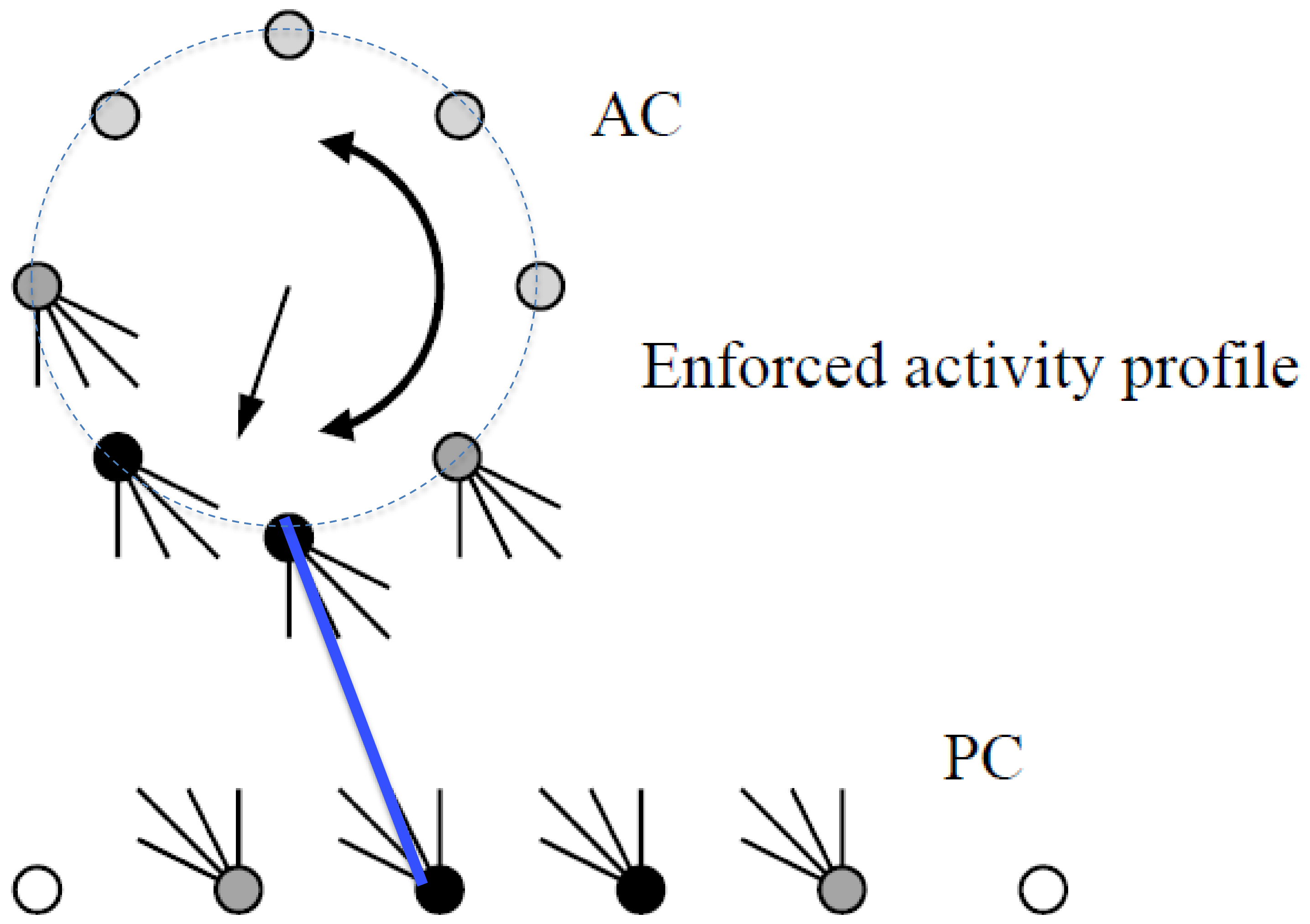
Rewards only in states → exploration easier: stop exploration if each state well represented

5) Is there a topology/neighborhood relation that would enable us to talk about two actions being 'similar'?

Inductive bias in Reinforcement Learning

5) Is there a topology/neighborhood relation that would enable us to talk about two actions being 'similar'?

Yes → Generalization across actions space possible.



Enforce activity profile
= spread Q-value activity from 'winning action' to neighbors
= learn neighbors at the same time
= learn as if all similar actions had been taken as well

Inductive bias in Reinforcement Learning

The EFFECTIVE number of parameters is much lower than the number of weights, since neighboring state neurons and neighboring action neurons learn similar things.

Additional inductive bias is also used in this example:

- Odometry (wheel turns) allows to give a noisy prediction of current location.
- This prediction can be combined with the filter response to give more localized filters
- The odometry in turn can be calibrated by the recognized filter responses.
- No stable compass, GPS, or knowledge of ‘where’ necessary

Previous slides.

- Coding of input space: we sample vectors of feature responses in the high-dimensional space, but we know that in the end they encode only two dimensions, so that sampling is indeed possible.

- The space is further reduced from 3 to 2 dimensions by algorithmic rotation of images (= shift of feature vectors) to get rid of difference due to orientation.

- We can work with relatively long eligibility traces, since there is a single goal state.

- We generalize across actions: we imagine actions forming a ring of possible directions. Neighboring actions should learn (in most states) similar behavior, hence if action a* is chosen with SARSA and learns (at rate eta), then all its neighbors learn as well (but with slightly reduced rate).

Inductive bias in Reinforcement Learning

Use all prior knowledge you have, before you start coding:

- No Free Lunch
- a generic neural network is rarely the best
- choose encoding and preprocessing so that generalization across ‘similar things’ becomes possible.

Reinforcement Learning can be extremely fast!!!