Artificial Neural Networks:

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Deep Nets 1: Backprop and multilayer networks

Part 1: - Background and Objectives
- Gradient descent: Batch versus Online

Overall Objectives for Today:

- XOR problem and the need for multiple layers
- understand backprop as a smart algorithmic implementation of the chain rule
- hidden neurons add flexibility, but flexibility is not always good: the problem of generalization
- training base, validation and test base: the need to predict well for future data

Previous slide.

The simple perceptron (see first week) is restricted to linearly separable problems. This week we will go beyond and look at neural networks with several layers. As an example we construct a solution to the XOR problem.

Updating parameters in a multi-layer network requires an efficient application of the chain rule known as backpropagation algorithm. We will study this algorithm.

Adding more neurons and layers is not necessarily good because the network needs to be used on future data that were not used during training. The problem of generalization is in practice handled by splitting the data base into two or even three parts, as known from introduction courses to machine learning.

Background: Data base for Supervised learning (single output)

P data points $\quad \{ \quad (\boldsymbol{x}^{\mu}, t^{\mu}) \quad , \qquad 1 \leq \mu \leq P \quad \};$

input target output

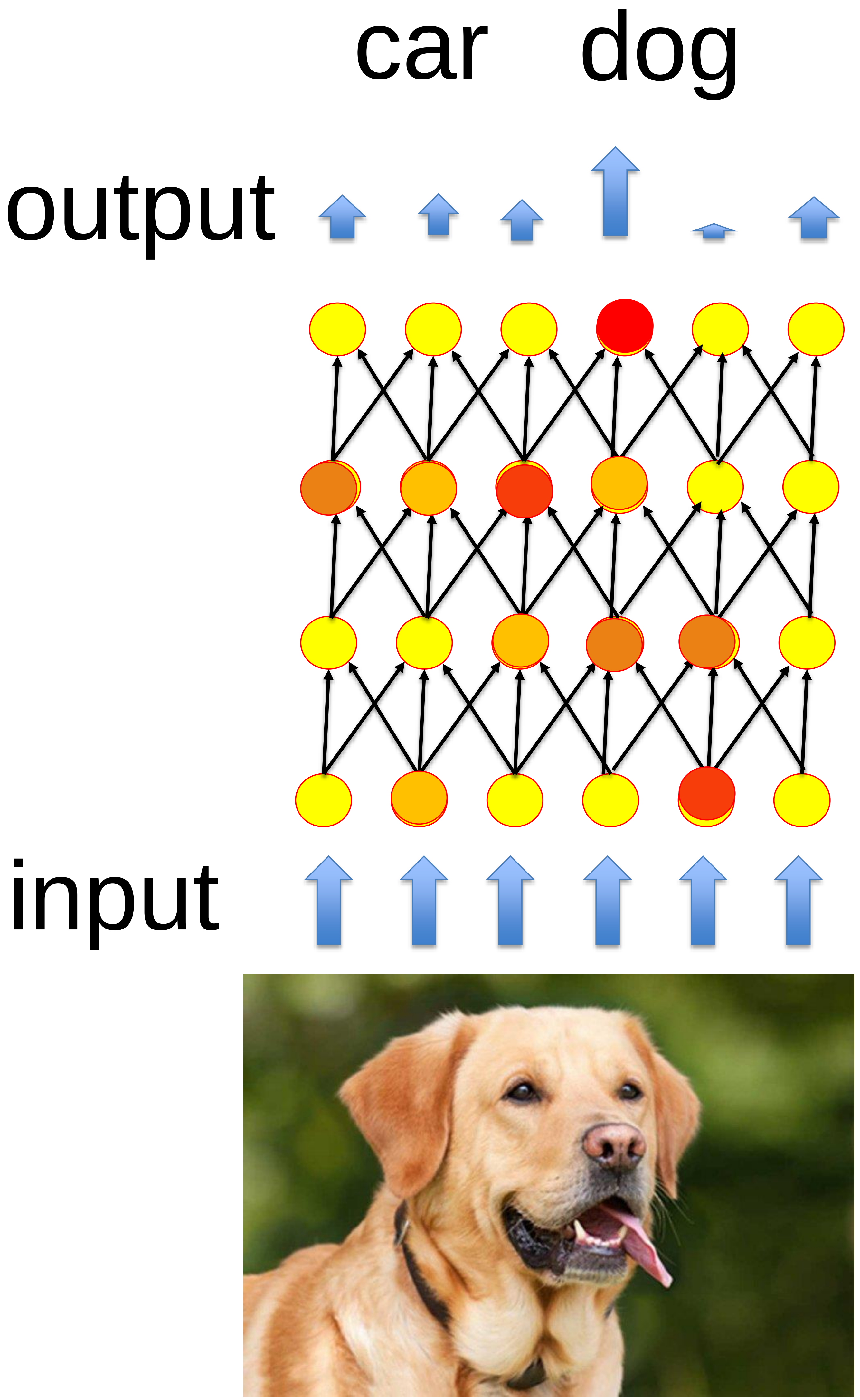
$t^{\mu} = 1$ car =yes

$t^{\mu} = 0$ car =no

Previous slide.

To train the network, we make use of a data based of supervised learning where each input pattern $\boldsymbol{x}^{\mu}$ is associated with the appropriate label t^{μ}which we consider as target output.

Background: Artificial Neural Networks for classification

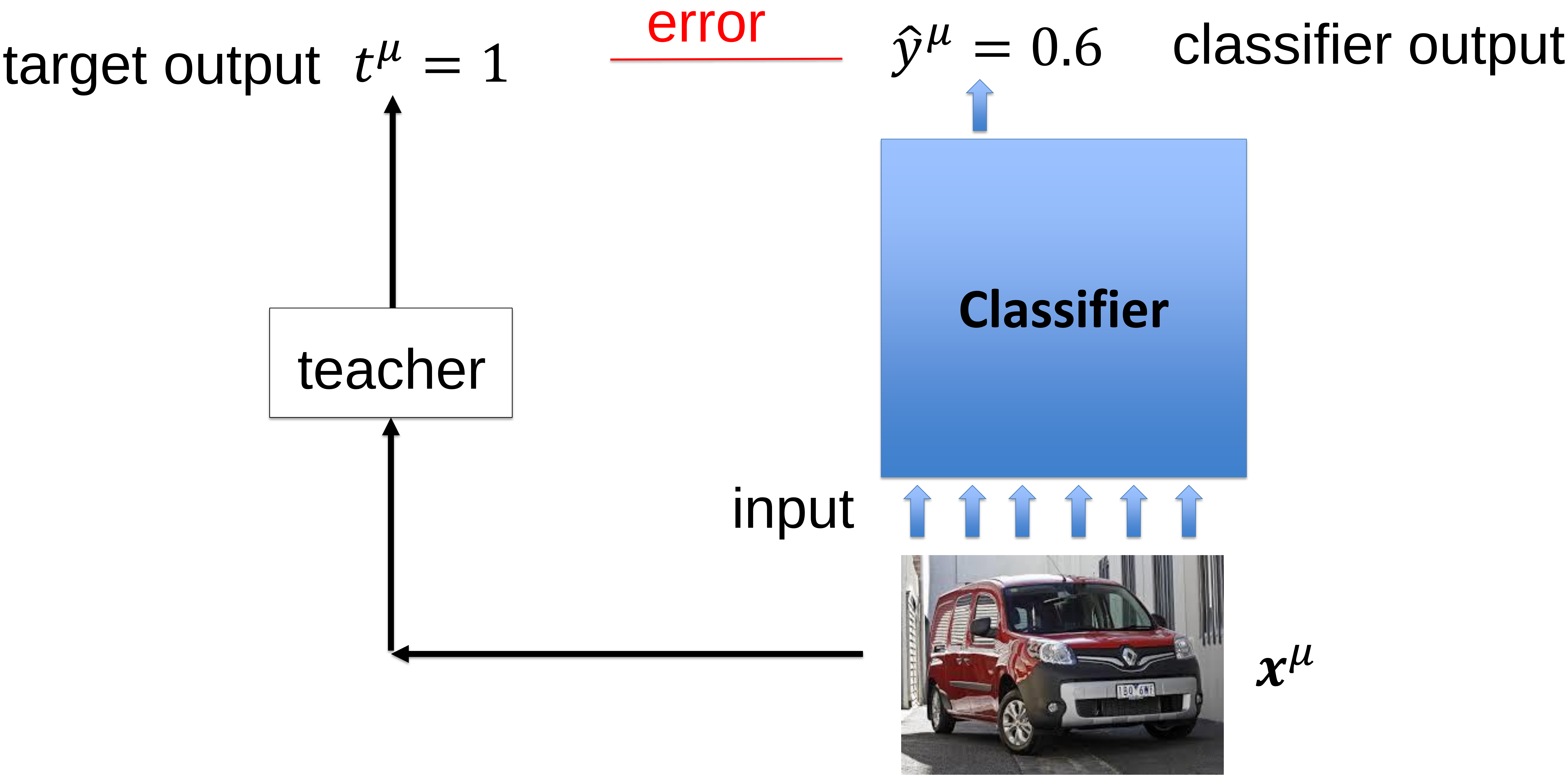


Aim of learning:
Adjust connections such that output is correct (for each input image, **even new ones**)

Previous slide.
As we have seen in week 1, artificial neural networks are often organized in layers. In the context of a classification task the output units indicate the class to which an input belongs.

To train the network we adjust the connection weights of the network.

Background: Supervised learning



error

target output $t^\mu = 1$

$\hat{y}^\mu = 0.6$ classifier output

teacher

Classifier

input

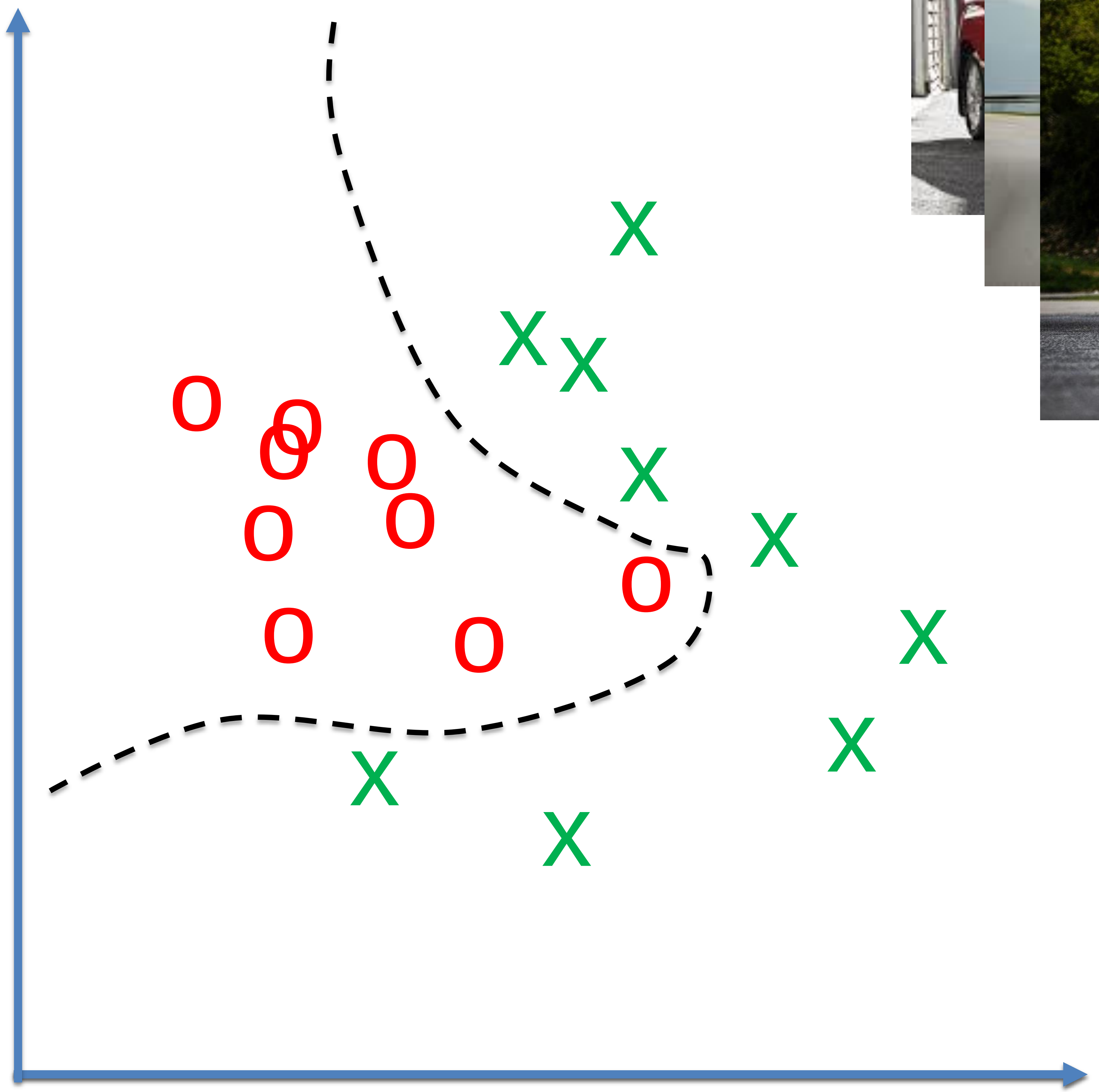
$\boldsymbol{x}^\mu$

Previous slide.

The comparison between the target output t^{μ} and the actual output $\hat{y}^{\mu}$ enables us to train the network.

The result of the comparison can be formulated as an ‘error function’ also called ‘loss function’.

Background: Classification as a geometric problem



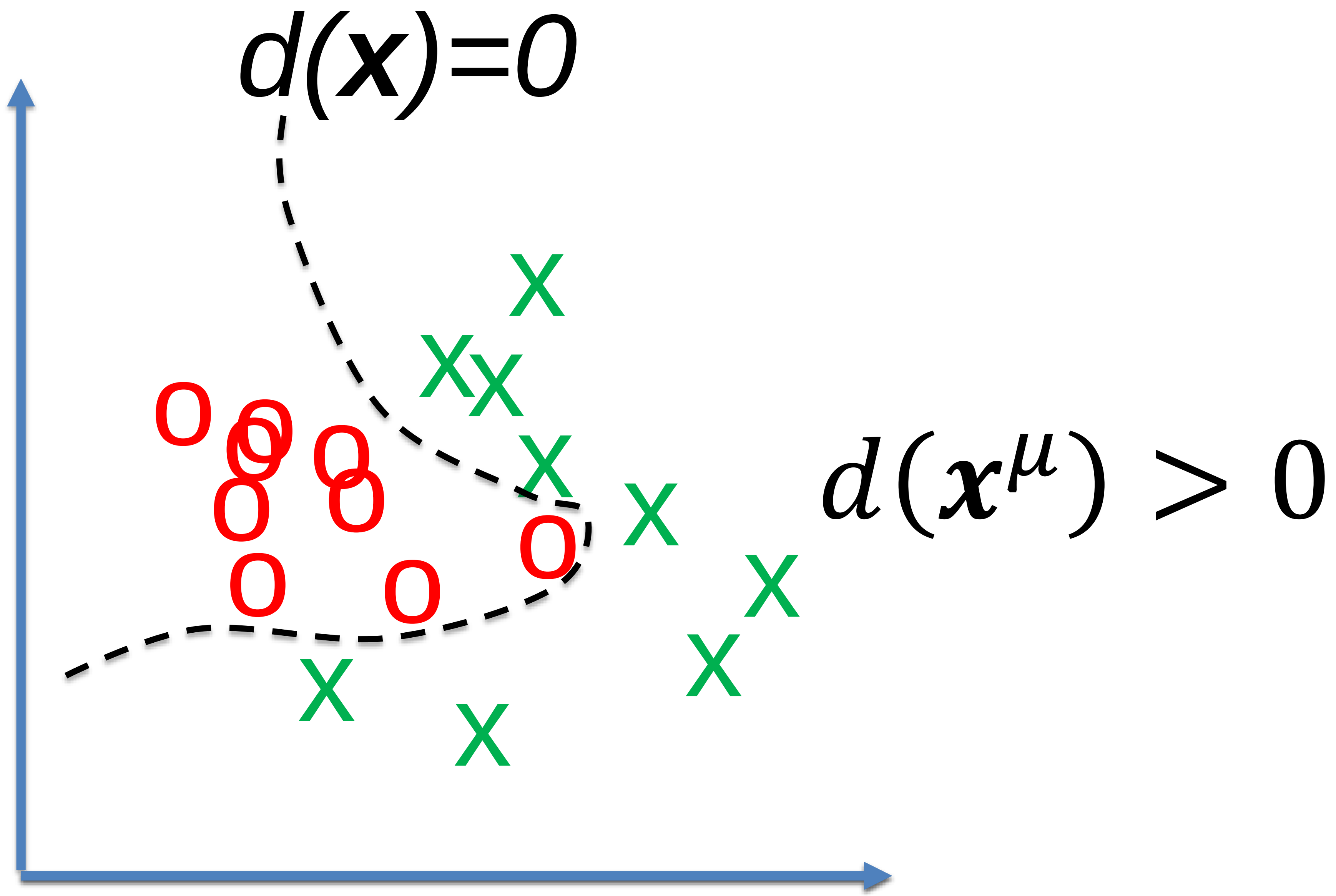
Previous slide.

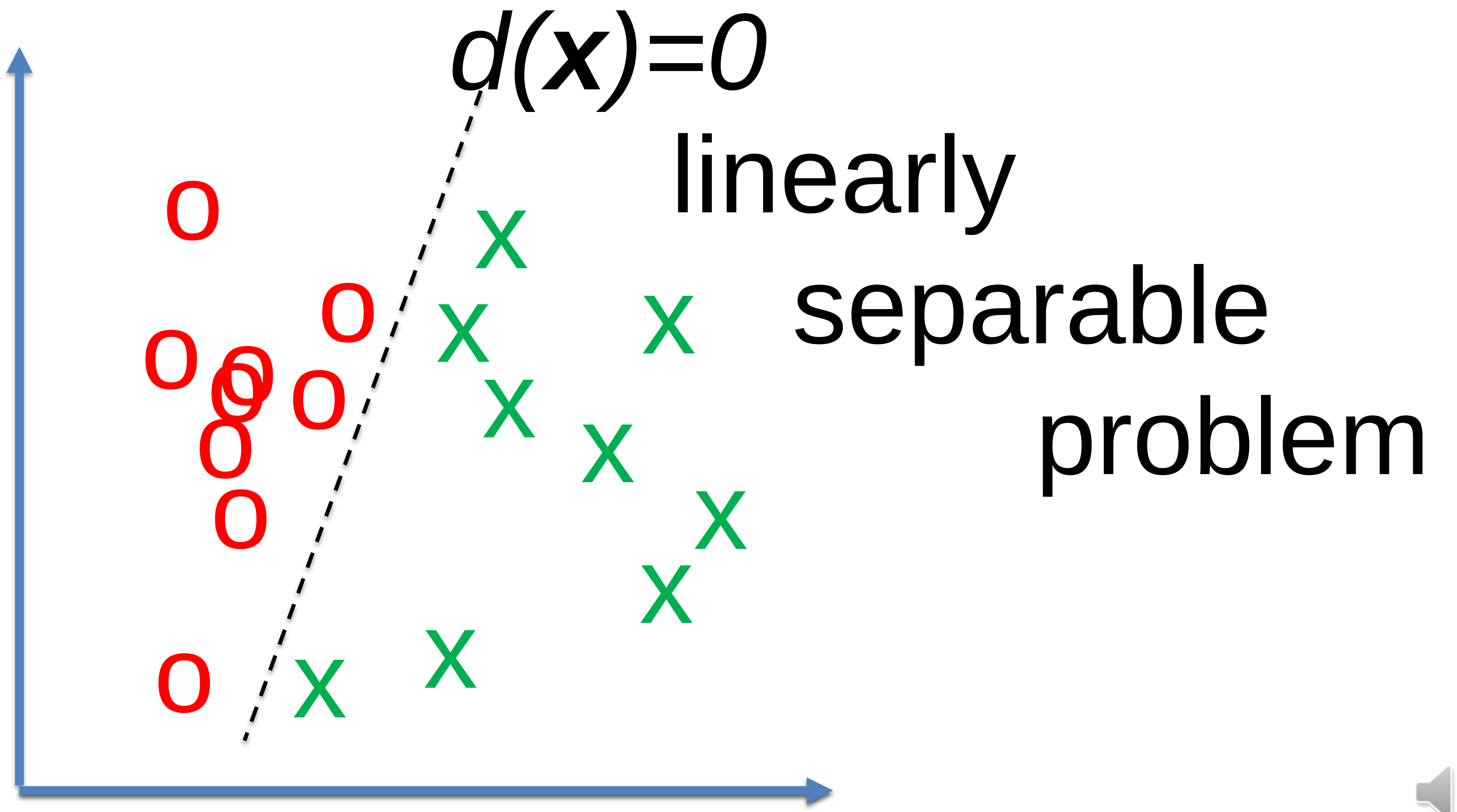
As an illustration we have used a classification of cars against all non-car images.

Background: Classification as a geometric problem

Task of Classification

= find a **separating surface** in the high-dimensional input space



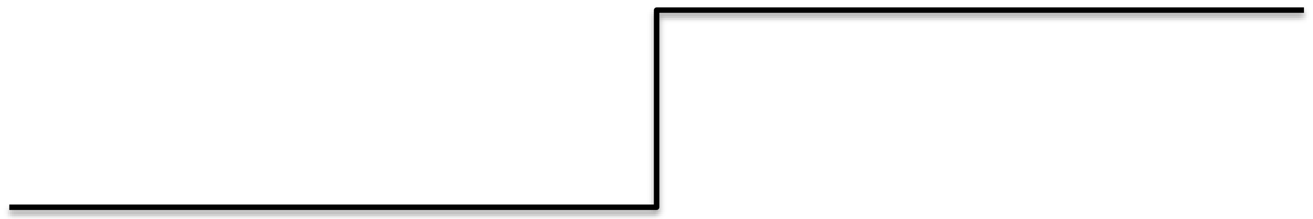


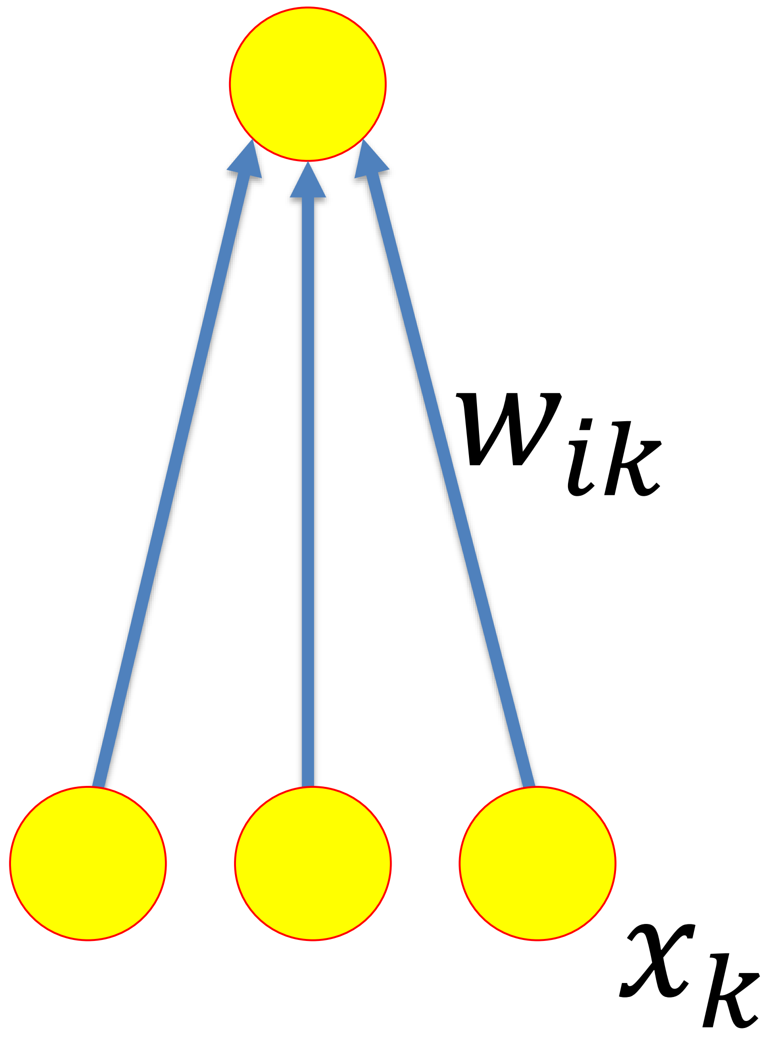
Previous slide.

After training, all positive examples t^{μ}=+1 (green crosses) should lie on the same side of a separating surface which is defined as the set of points where the discriminant function is zero. All negative examples t^{μ}=0 should lie on the other side.

Background: Single-Layer networks: simple perceptron

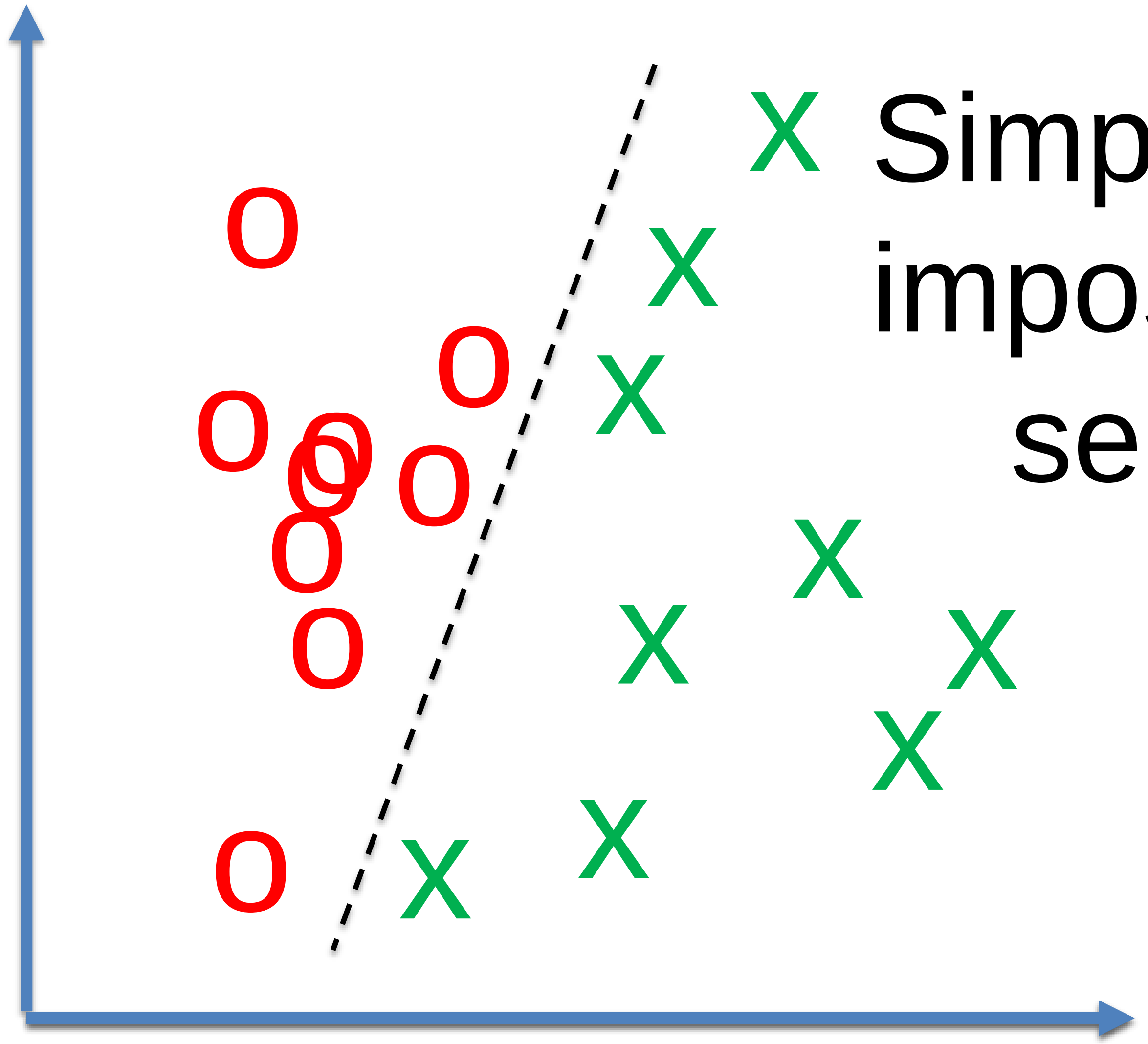
$$\hat{y} \quad = 0.5[1 + sgn(\textstyle\sum_k w_k\, x_k - \vartheta)]$$





vector ***x***

$$d(\boldsymbol{x}) = \sum_k w_k\, x_k - \vartheta = 0$$



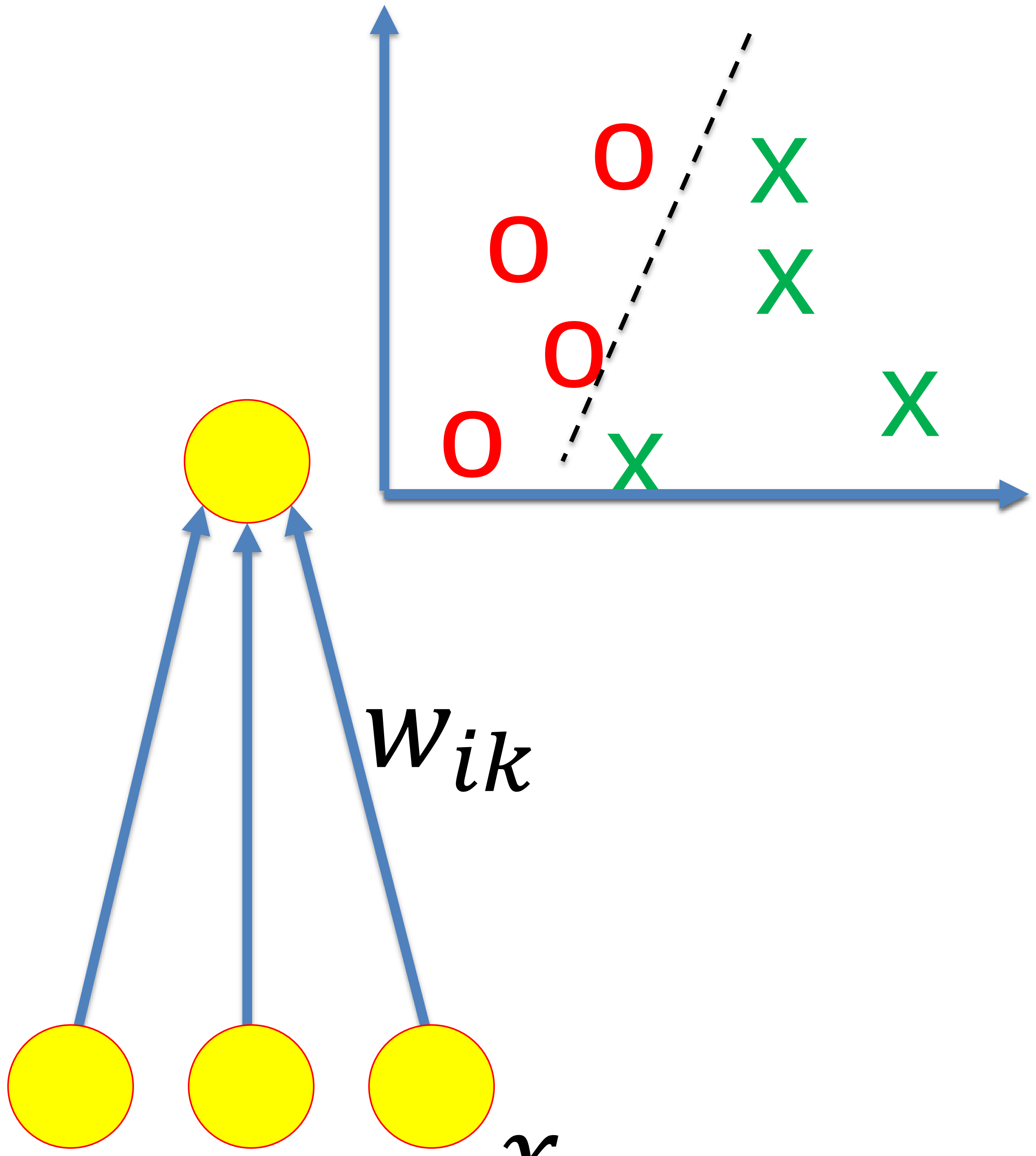
Simple perceptron imposes a linear separation

Previous slide.

For a network consisting of a single neuron, the discriminant function is a hyperplane.

Background: remove threshold: add extra input

$$d(\boldsymbol{x}) = \sum_{k=1}^{N} w_k\, x_k - \vartheta \;= 0$$

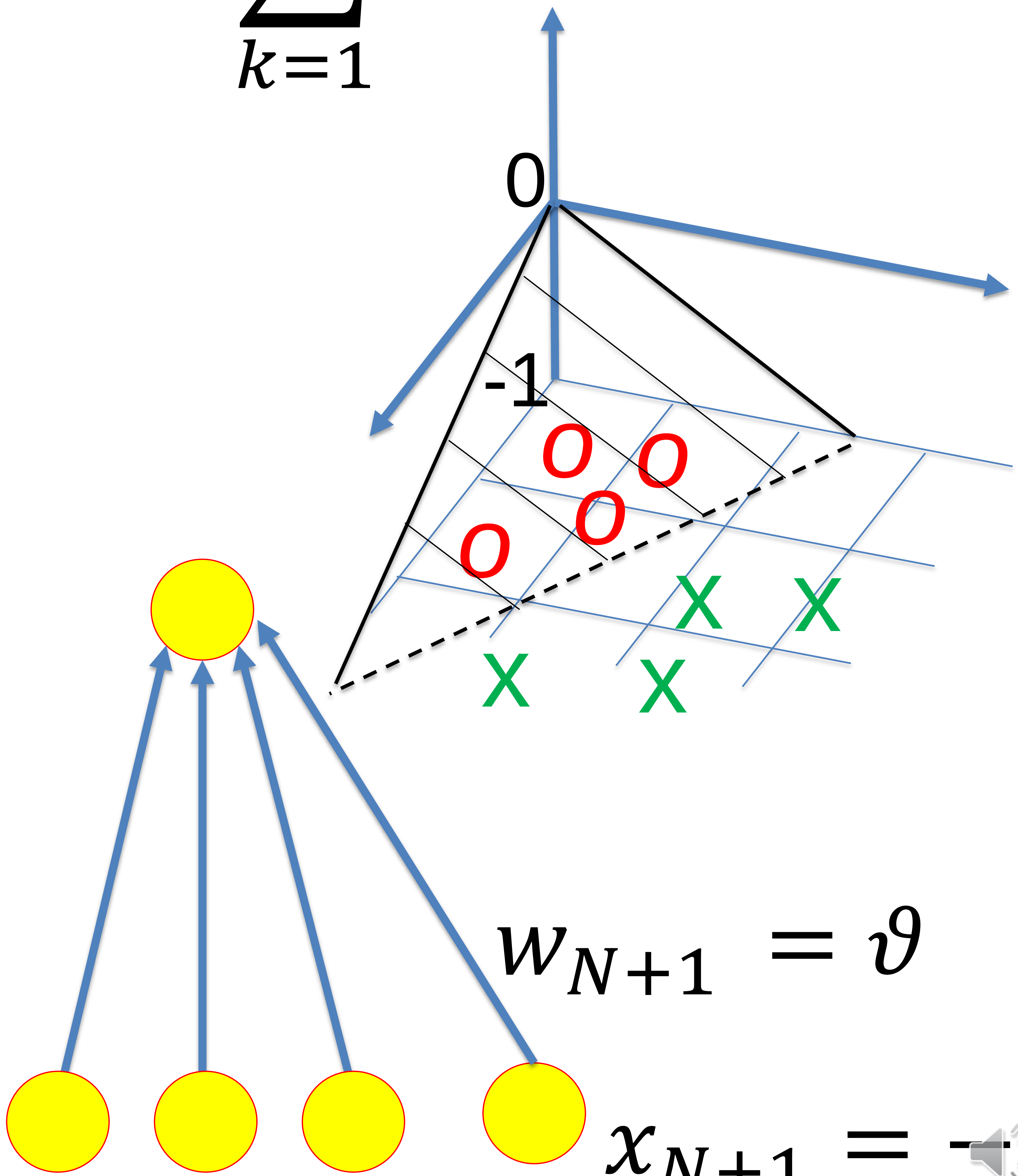


w_{ik}

$\boldsymbol{x} \in R^N$

x_k

$$d(\boldsymbol{x}) = \sum_{k=1}^{N+1} w_k\, x_k \;= 0$$



$w_{N+1} \;= \vartheta$

$\boldsymbol{x} \in R^{N+1}$

$x_{N+1} = -1$

Previous slide.

After a switch from input dimension N to dimension N+1, this hyperplane runs through the origin.

Background: Single-Layer networks

a simple perceptron

- can only solve linearly separable problems
- imposes a separating hyperplane
- in ***N+1*** dimensions hyperplane always goes through origin
- Adapt weights by gradient descent (perceptron algo and other algos)

Previous slide.

As we have seen in the first week, a simple perceptron can only solve linearly separable problems. In N+1 dimensions each (non-vanishing) update step of an iterative algorithm corresponds to a rotation of the separating hyperplane.

Background and Objectives of Lecture on Deep Nets 1

- How can we train a multi-layer network
  - → Gradient descent algorithms
- How can a multilayer network solve problems that are not linearly separable?
  - → XOR problem
- How does the BackProp algorithm work?
  - → Derivation and Implementation
- Why have we to be careful when training deep networks?
  - → The Problem of overfitting and generalization
  - → The prediction must work for **future** data

Previous slide.
This lecture covers several topics that form the foundations of deep networks.

Some of the material will be known to you from Introductory courses on Machine Learning.

The core of the lecture is the derivation and interpretation of the famous BackProp algorithm.

Artificial Neural Networks

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Deep Nets 1: Backprop and multilayer networks

1. Gradient Descent Methods

Previous slide.
Almost all learning algorithms approach a minimum of the error function iteratively, typically by gradient descent or variants thereof.
In a batch rule, a single update step is implemented after **all** patterns have been used once, whereas in an online rule a single update step is implemented after **each** pattern.

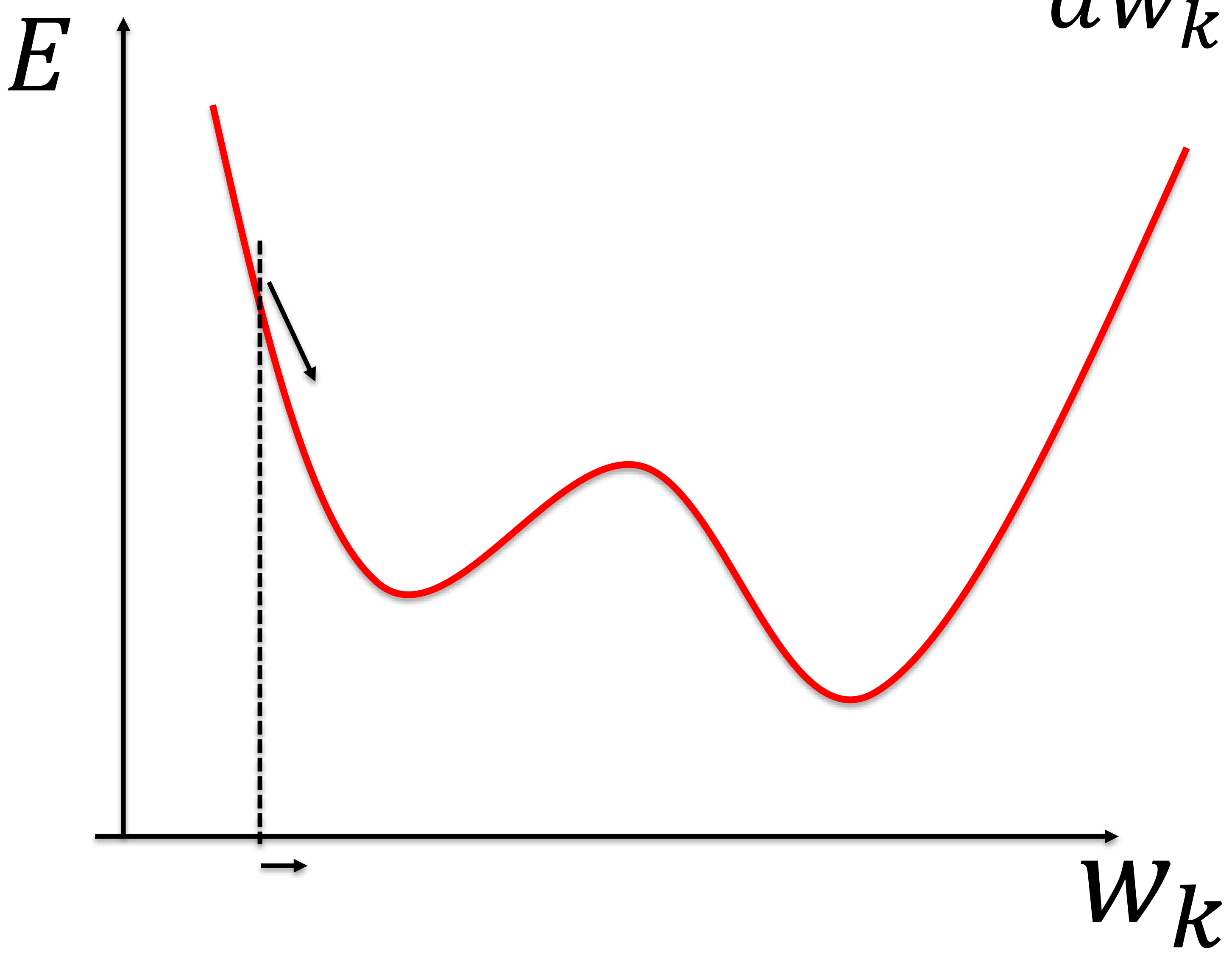
Review: gradient descent

First week: Quadratic **error**

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{\mu=1}^{P}\left[t^{\mu} - \hat{y}^{\mu}\right]^2$$

gradient descent

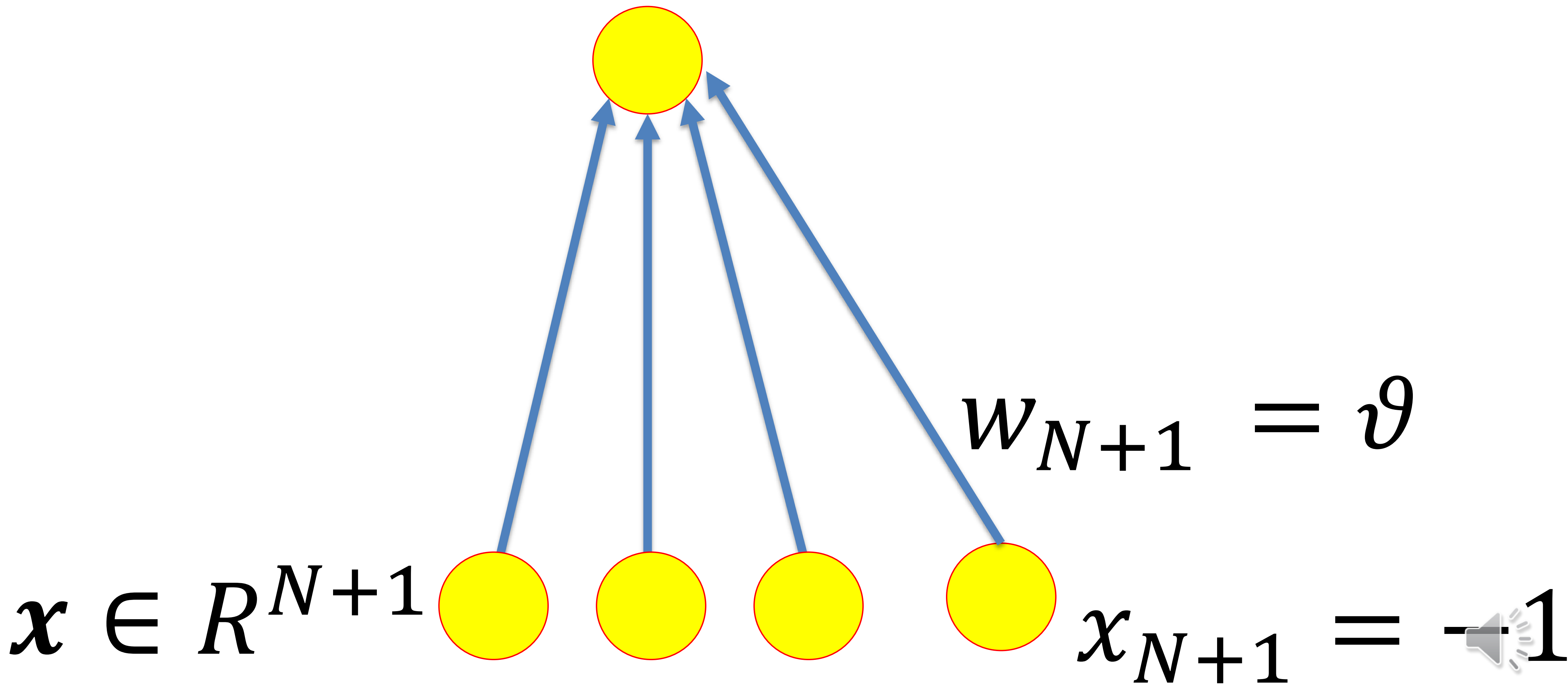
$$\Delta w_k = -\gamma \frac{dE}{dw_k}$$



Batch rule:
one update after all patterns
(normal gradient descent)

Online rule:
one update after one pattern
(stochastic gradient descent)

$$\hat{y}^{\mu} = g(\boldsymbol{w}^T \boldsymbol{x}^{\mu})$$



Previous slide.

For the special case of a quadratic error function in a simple perceptron, we have already derived last week gradient descent in its batch and online version. The true gradient yields the batch rule; the online version is called stochastic gradient descent.

Modern gradient descent methods no longer make this strict separation between online or batch and often use minibatches.

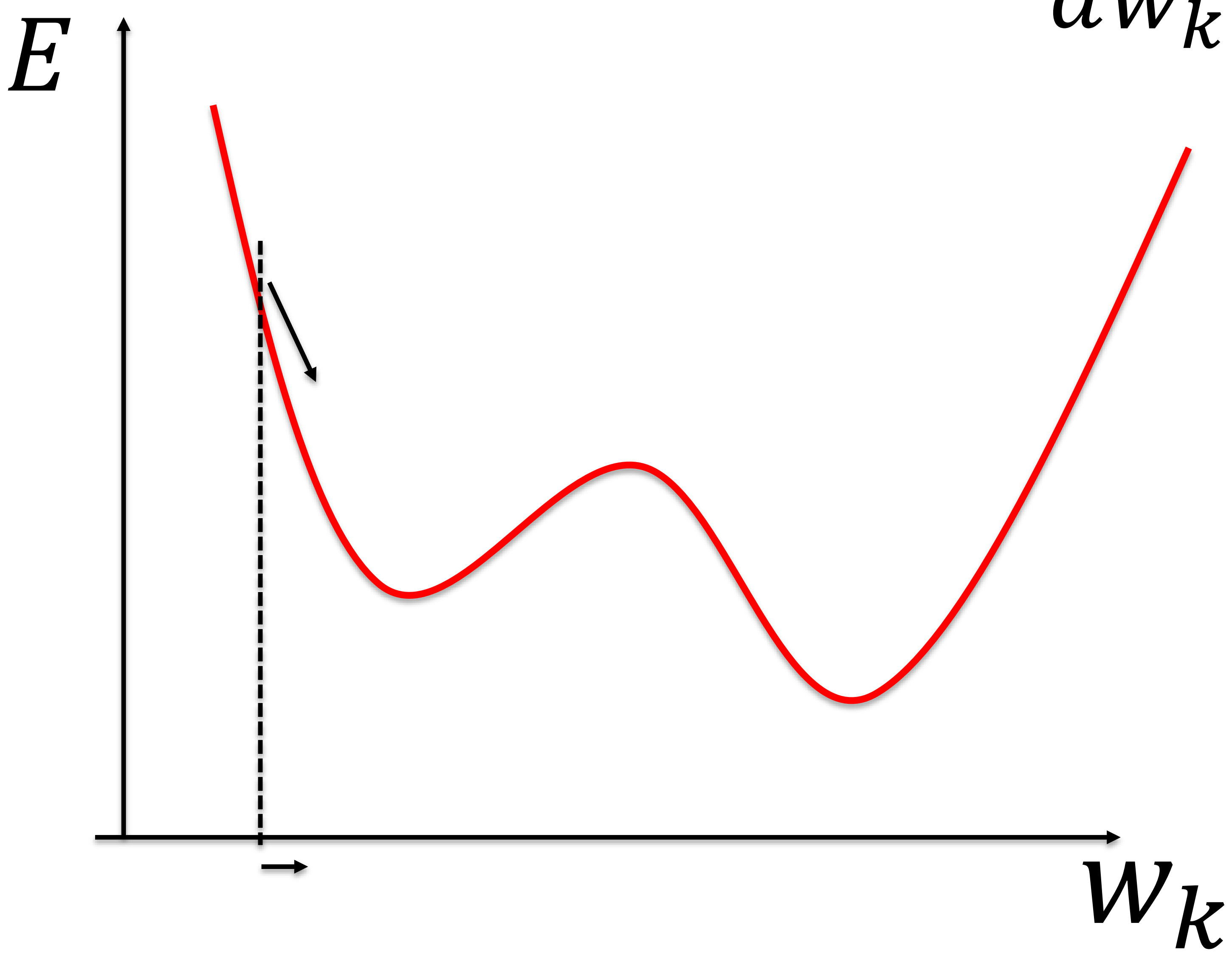
Modern gradient descent

Take some **error function,** also called **loss function**

$E(\boldsymbol{w})$

gradient descent

$$\Delta w_k = -\gamma \frac{dE}{dw_k}$$



Batch rule:
one update after all ***P*** **patterns**
(normal gradient descent)

Online rule (stochastic gr. desc.):
one update after **one pattern**
(stochastic gradient descent)

Mini Batch rule:
one update after ***P'=P/K*** **patterns**
(minibatch update)

1 epoch = all patterns applied once.
(on average)
training over many epochs

Previous slide.
If there are P patterns in total, a minibatch is of size P’ = P/K. An update step is implemented after each minibatch.

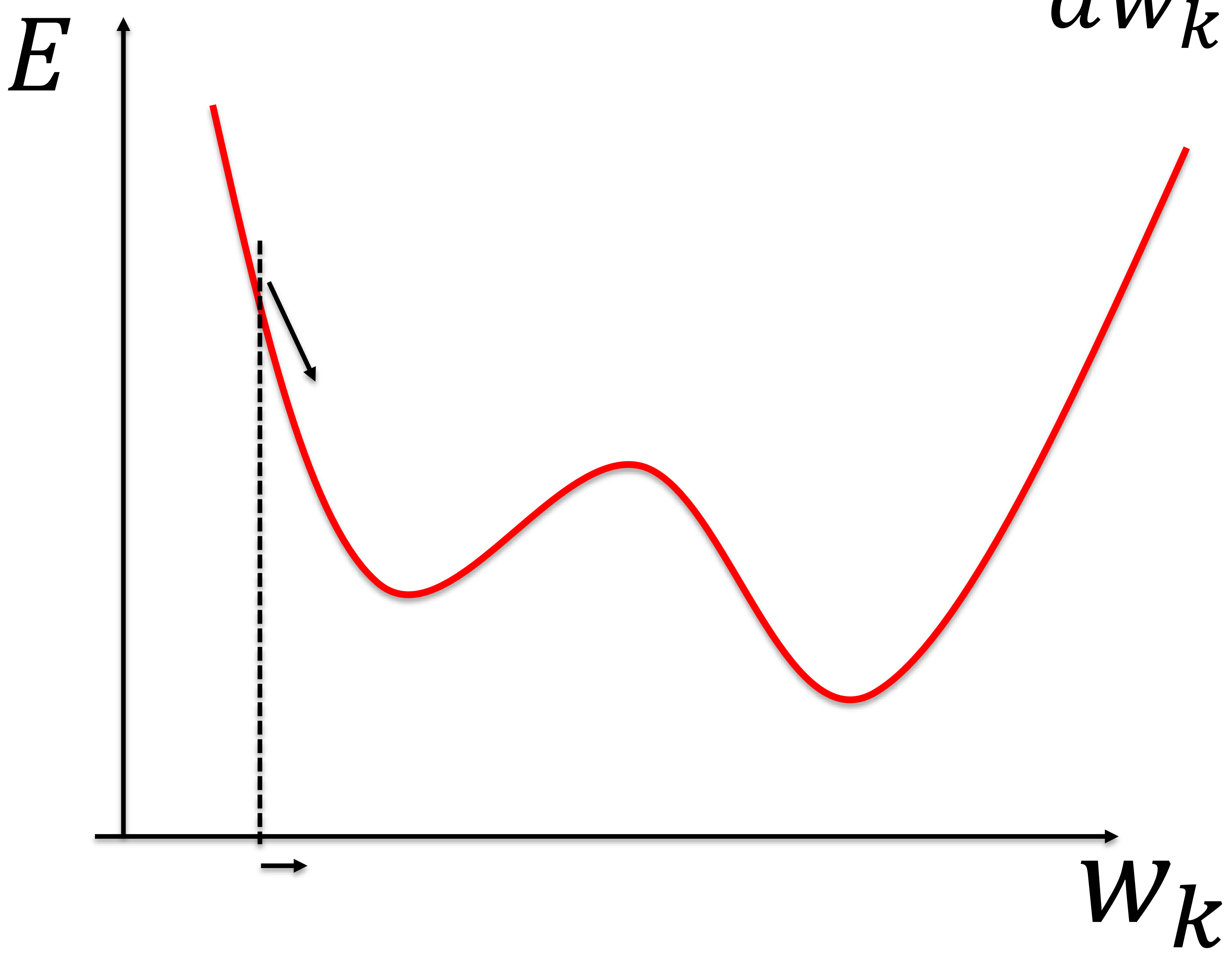
A minibatch is a useful practical compromise: it is closer to true gradient descent (‘batch’) than the online stochastic gradient descent algorithm, but is easier to implement than a regular batch algorithm for modern databases with millions of data points.

An ‘epoch’ is defined the number of iterations such that each pattern is used on average once. With a batch rule each update step is an epoch; with an online rule P update steps are one epoch. With a minibatch rule K steps are one epoch.

Properties of gradient descent

gradient descent

$$\Delta w_k = -\gamma \frac{dE}{dw_k}$$



Convergence

- To local minimum
- No guarantee to find global minimum
- Learning rate needs to be sufficiently small
- Learning rate can be further decreased once you are close to convergence

→ See course: *Machine Learning* (Jaggi-Urbanke)

Previous slide.
None of the gradient descent algorithm comes with a guarantee that it finds the global minimum: if there are many local minima, it typically converges to one of these.

Because of the finite step size γ (learning rate), the algorithm will always show a bit of jitter around the minimum. Decreasing the learning rate γ over the course of many iterations helps to reduce the jitter

1. Review of Modern Gradient Descent Methods: Quiz

Your friend claims the following. Do you agree?

[ ] We can perform gradient descent on all sorts of differentiable loss functions, not just the quadratic error function

[ ] If we have P patterns, then P steps of stochastic gradient descent are **identical** to a single step of batch gradient descent.

[ ] During convergence to a **local minimum** with a **batch gradient descent,** parameters **can jitter** a little bit around the minimum (for fixed learning rate)

[ ] After convergence to a **global minimum**, a **stochastic gradient descent** method with fixed learning rate **always jitters** a little bit around the minimum.

Previous slide (Hints).

For the first question: think of how you would train a policy gradient RL agent.

For the second question: think of how you choose patterns, and think of what an update step does.

For the third question: it is important that it is a local (=non-global) minimum (see last question).

For the last question: think of the Exercises in Week 1 where we derived the Perceptron Algorithm as stochastic gradient descent on a specific loss function. If the problem can be solved, the algorithm finds the solution and stops at the minimum of zero-loss without further jitter.

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Artificial Neural Networks

Deep Nets 1: Backprop and multilayer networks

Part 2: The XOR Problem

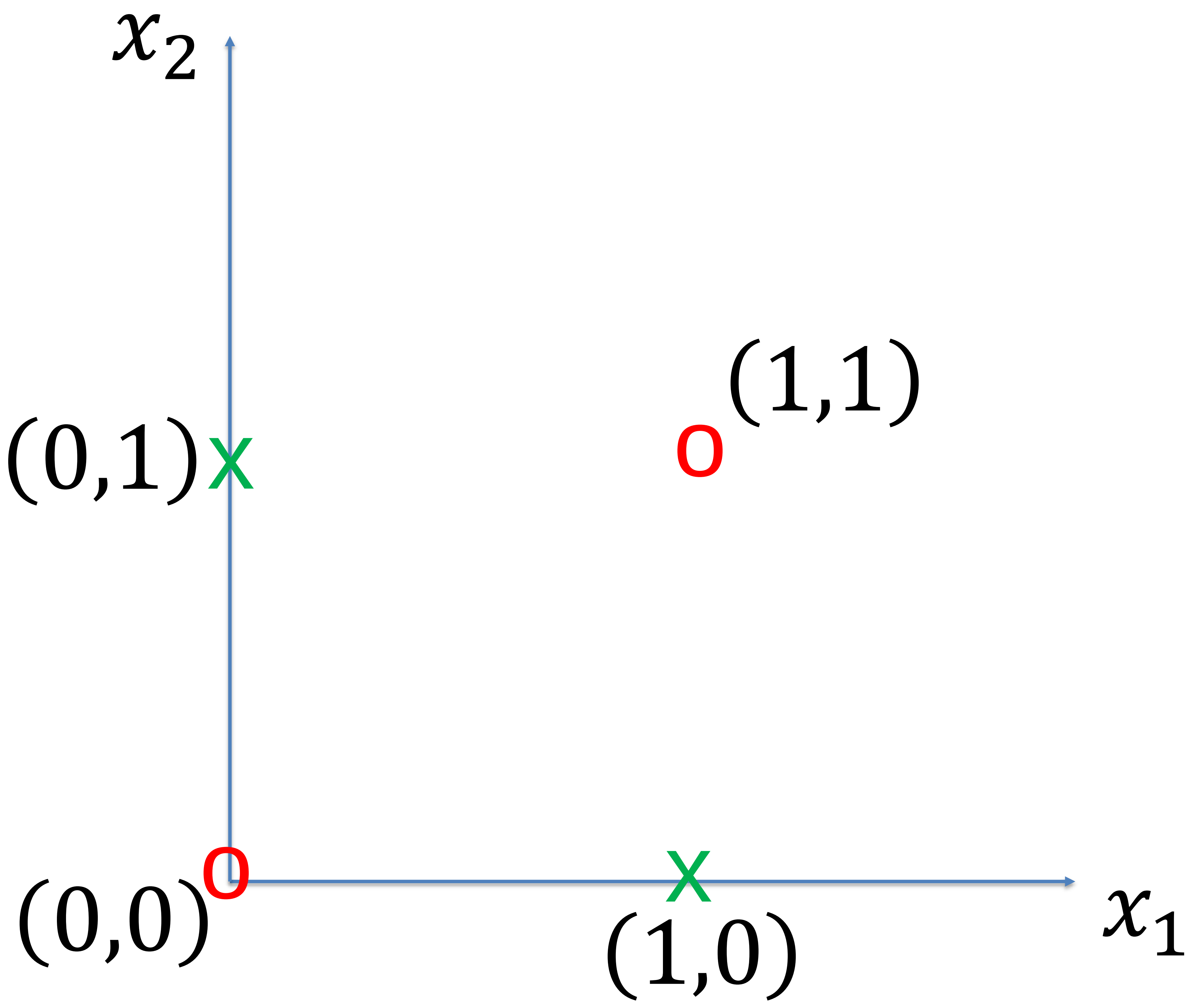
1. Gradient Descent Methods
2. **XOR problem**

Previous slide.

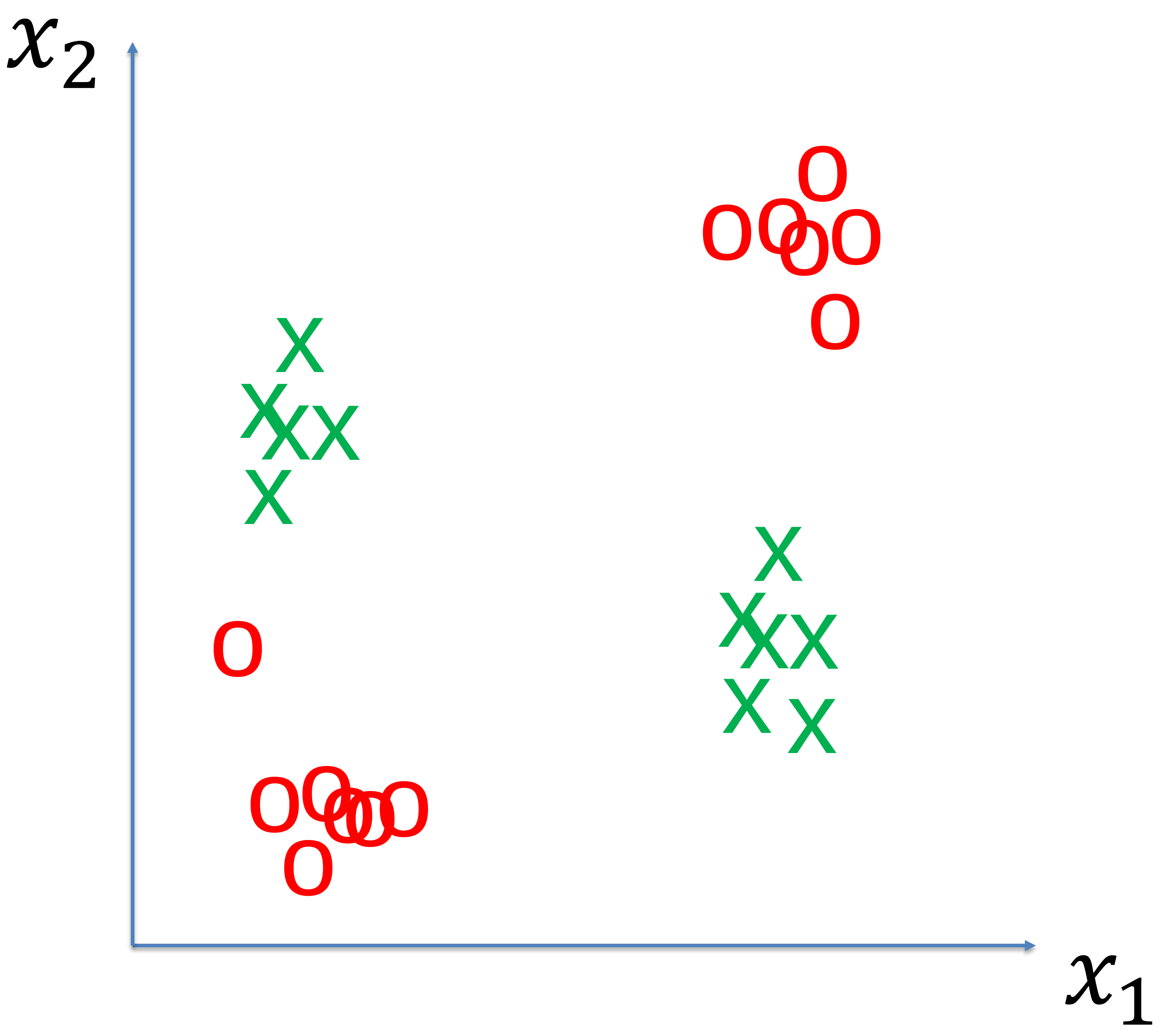
A famous example of a task that is not linearly separable is the XOR problem

2. The XOR problem

just 4 data points



(or many)

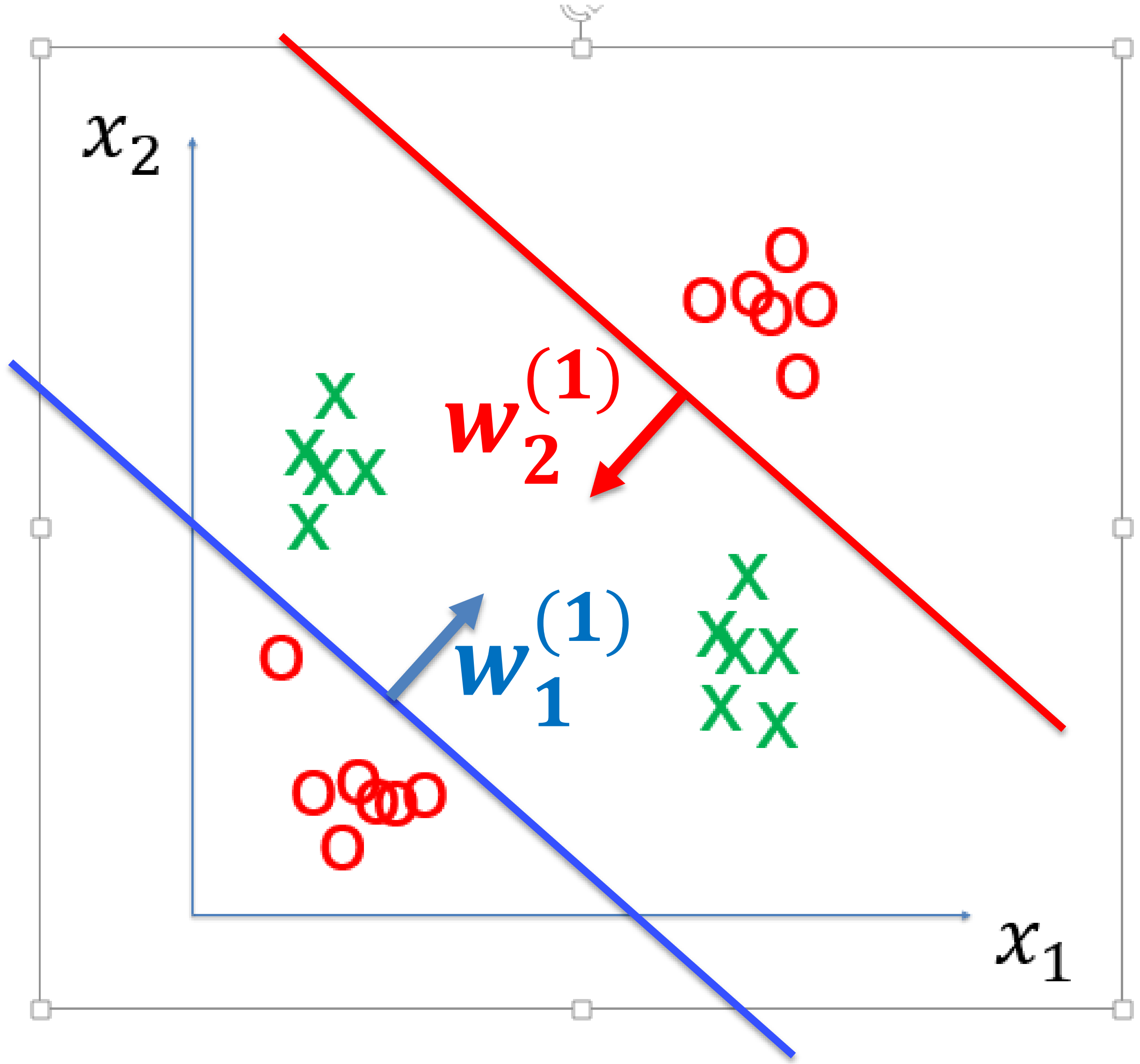


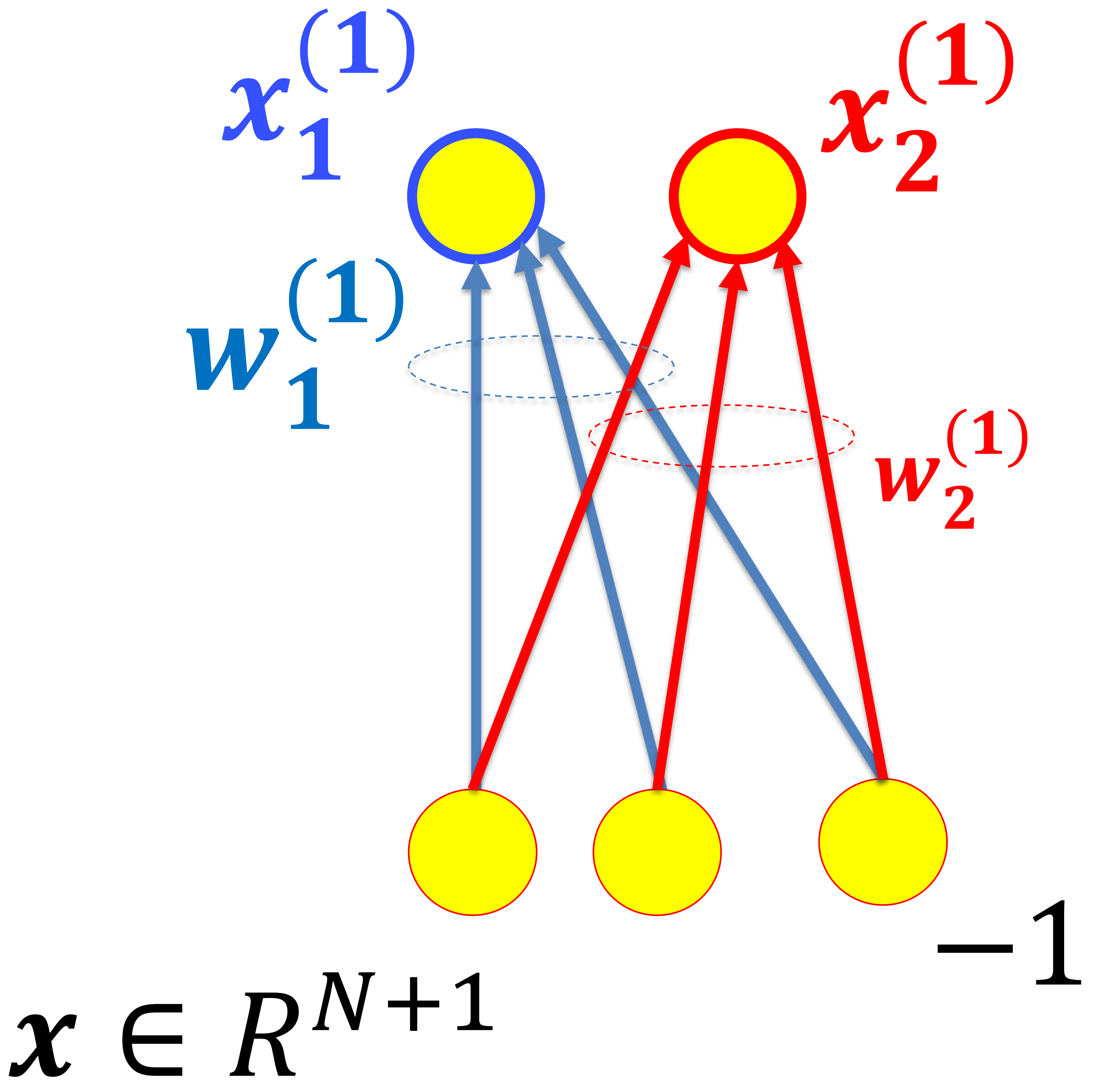
Previous slide.
The XOR problem derives its name from the logical operator XOR (left) with only four patterns, but the term is also used for groups of patterns that show an XOR-like configuration (right).

The configuration of data points in an XOR problem is not linearly separable. Hence it cannot be solved by a simple perceptron.

2. The XOR problem

Take two neurons



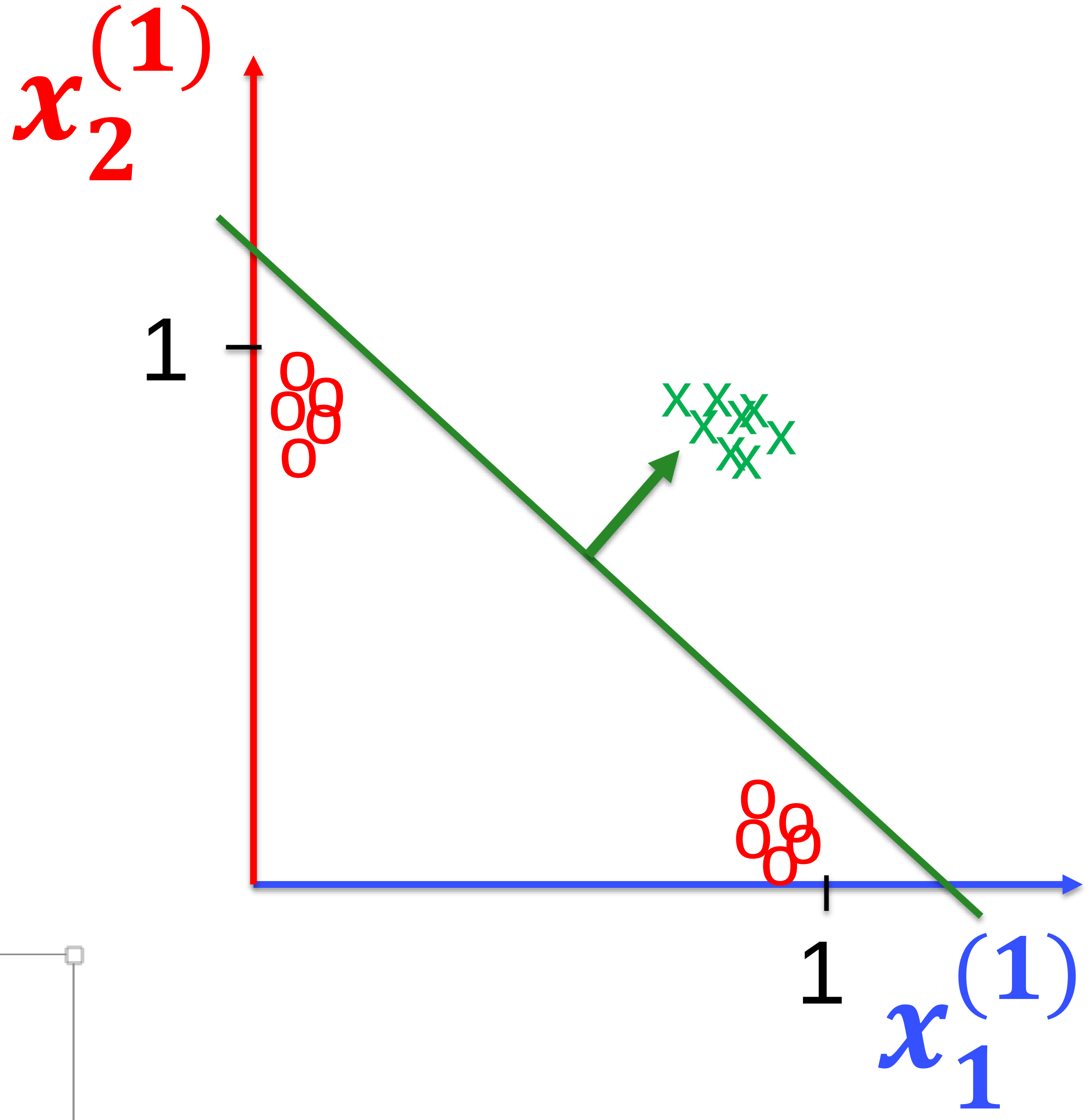


Previous slide:
We now construct a solution that involves two layers of neurons.

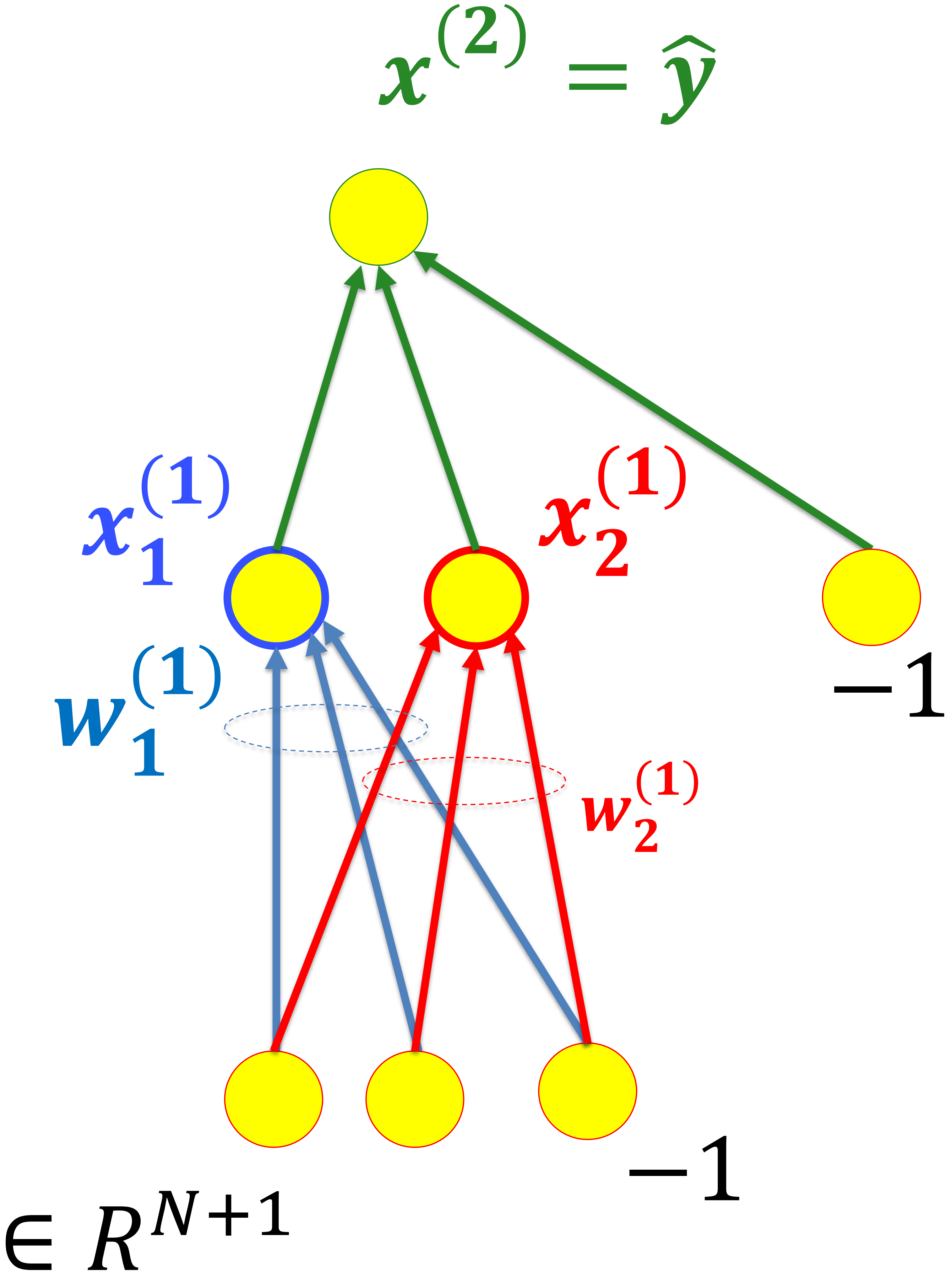
We first construct two hyperplanes in the input space. Each of these solves part of the problem, but neither of them separates all positive examples (green crosses) from the negative ones (red circles)

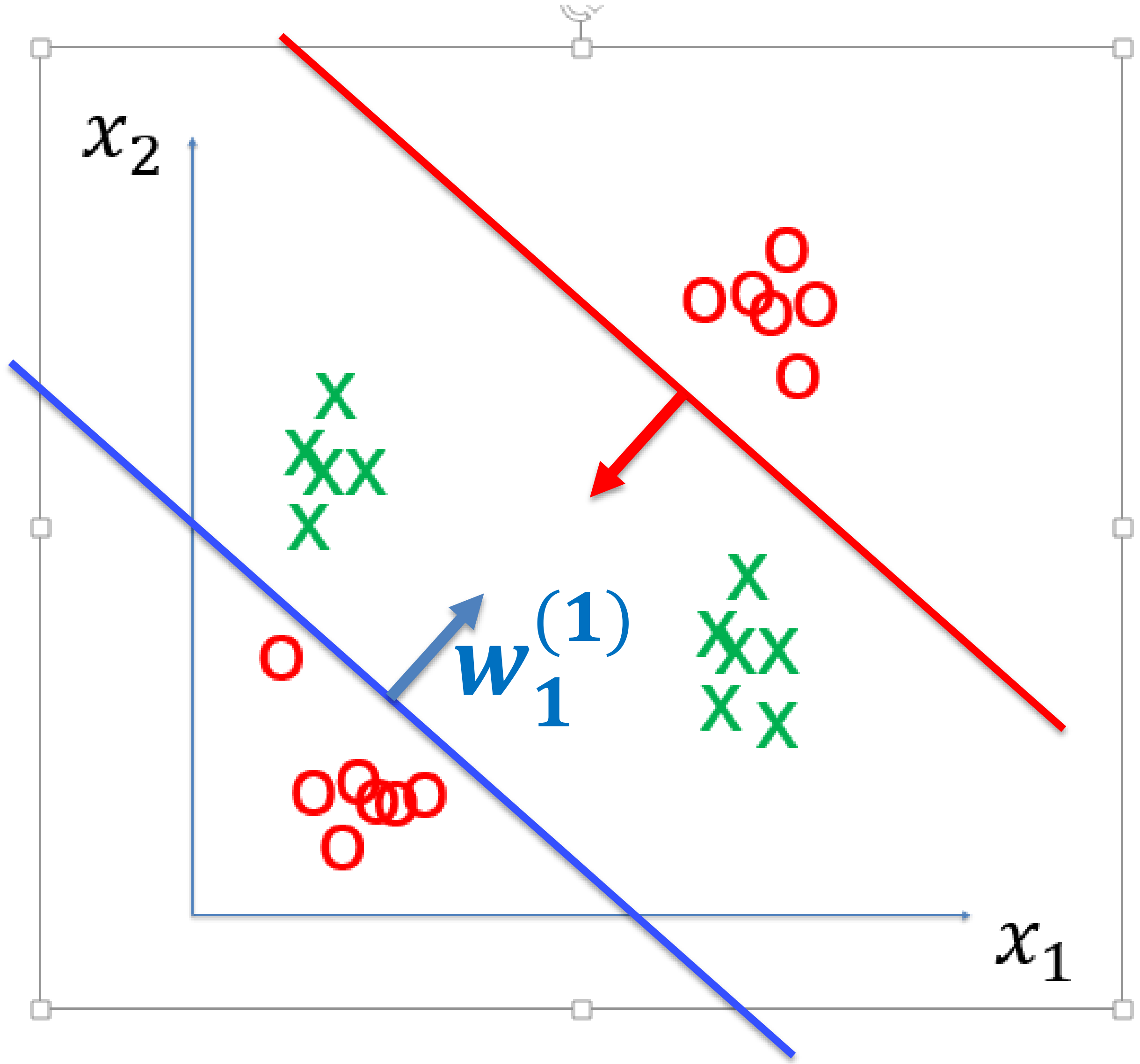
2. The XOR problem

Add output neuron



output solves XOR





$x \in R^{N+1}$

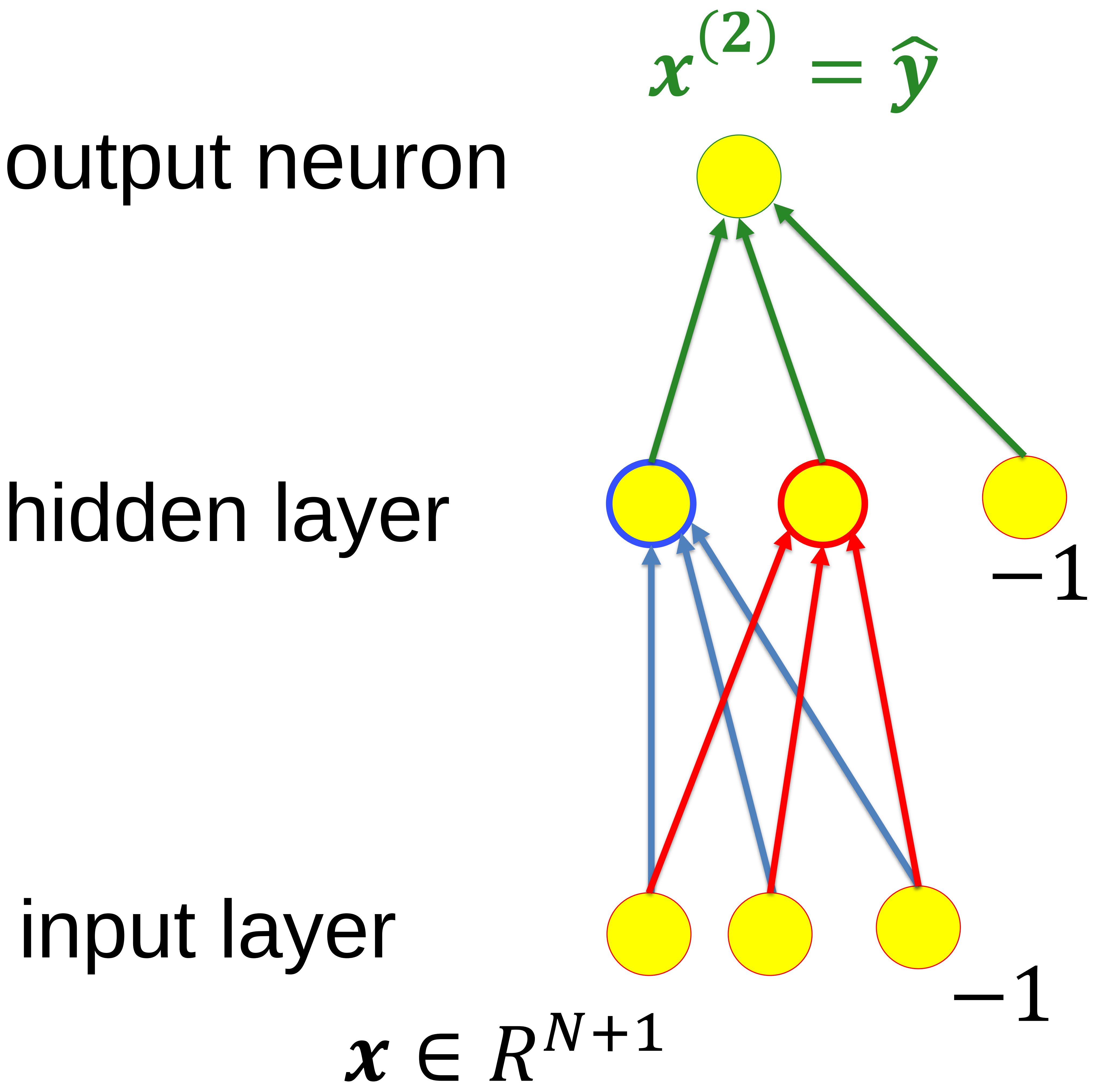
Previous slide:
We then construct a second layer which takes the output of the blue and red neurons as input.

Interestingly, ALL the positive examples are now mapped to a small region in the input space of the green output neuron; if the neurons implement hard threshold functions (as opposed to sigmoidal functions), then they are even mapped all to the same point (1,1). It is hence easy to separate them from the rest with a further hyperplane. This gives the output

2. Solution of XOR problem

Conclusion:

1. A two-layer network can solve the XOR problem
2. A two-layer network can solve problems that are not linearly separable.
3. Here we constructed the solution → we need to develop automatic methods to find solution



$x^{(2)} = \hat{y}$

output neuron

hidden layer

-1

input layer

-1

$x \in R^{N+1}$

Previous slide.
For the blue and the red neurons in the hidden layer, we construct, separating hyperplanes in input space.
We then construct, for the green neuron, a separating hyperplane in the space of the hidden neurons.

Conclusion: a neural network with one hidden layer can solve the XOR problem

Artificial Neural Networks

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Deep Nets 1: Backprop and multilayer networks

Part 3: Multilayer Networks - First Steps

1. Gradient Descent Methods
2. XOR problem
3. **Multilayer Networks: First Steps**

    → notation

    → derivation of BackProp

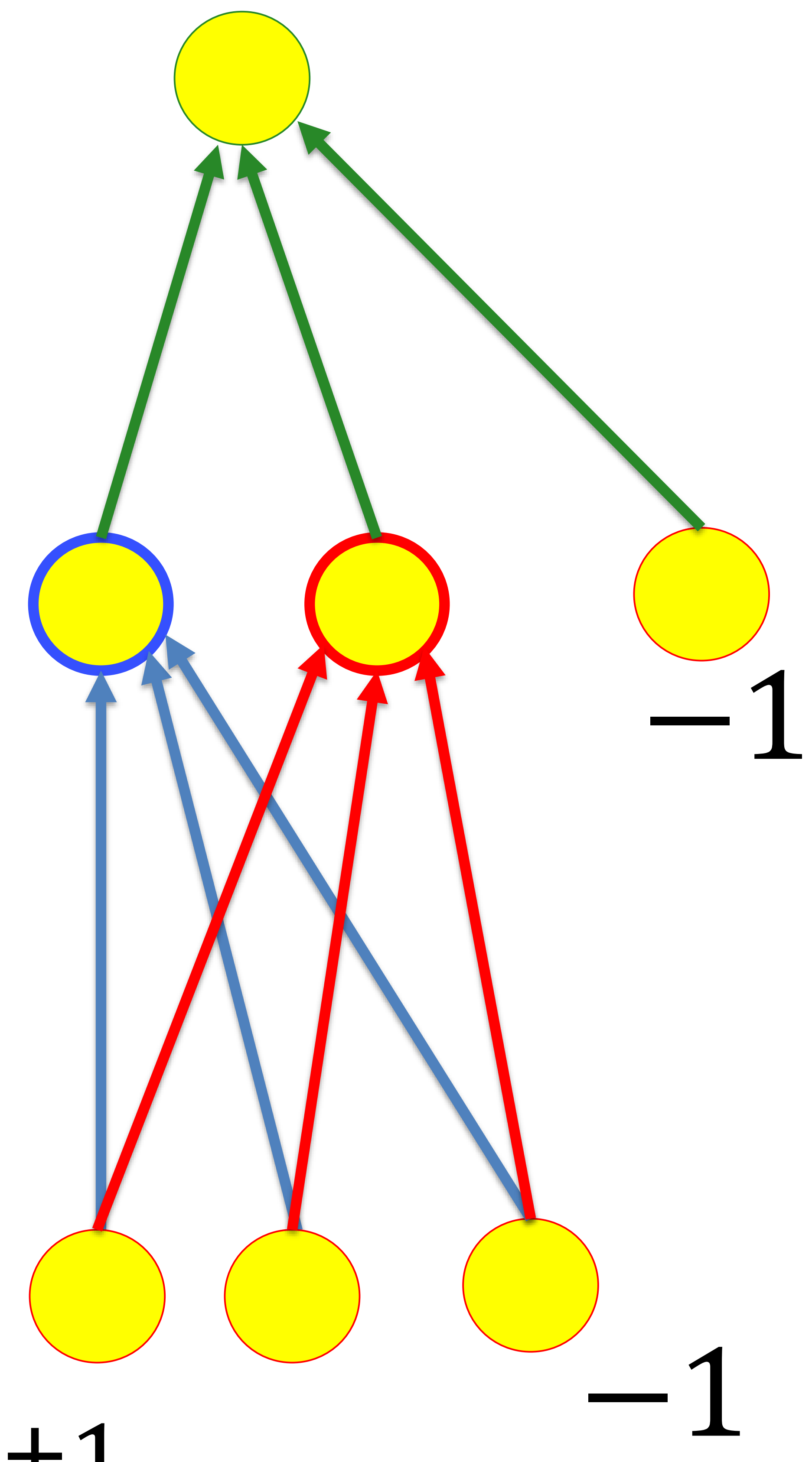
Previous slide.

A multilayer perceptron (or multi-layer network) has one or several hidden layers between input layer and output layer.

3. Multi-layer perceptron

- OK, can solve the XOR problem (by construction)

- But is there an **algorithm to find the weights** in more complicated cases?



output neuron

hidden layer

-1

input layer

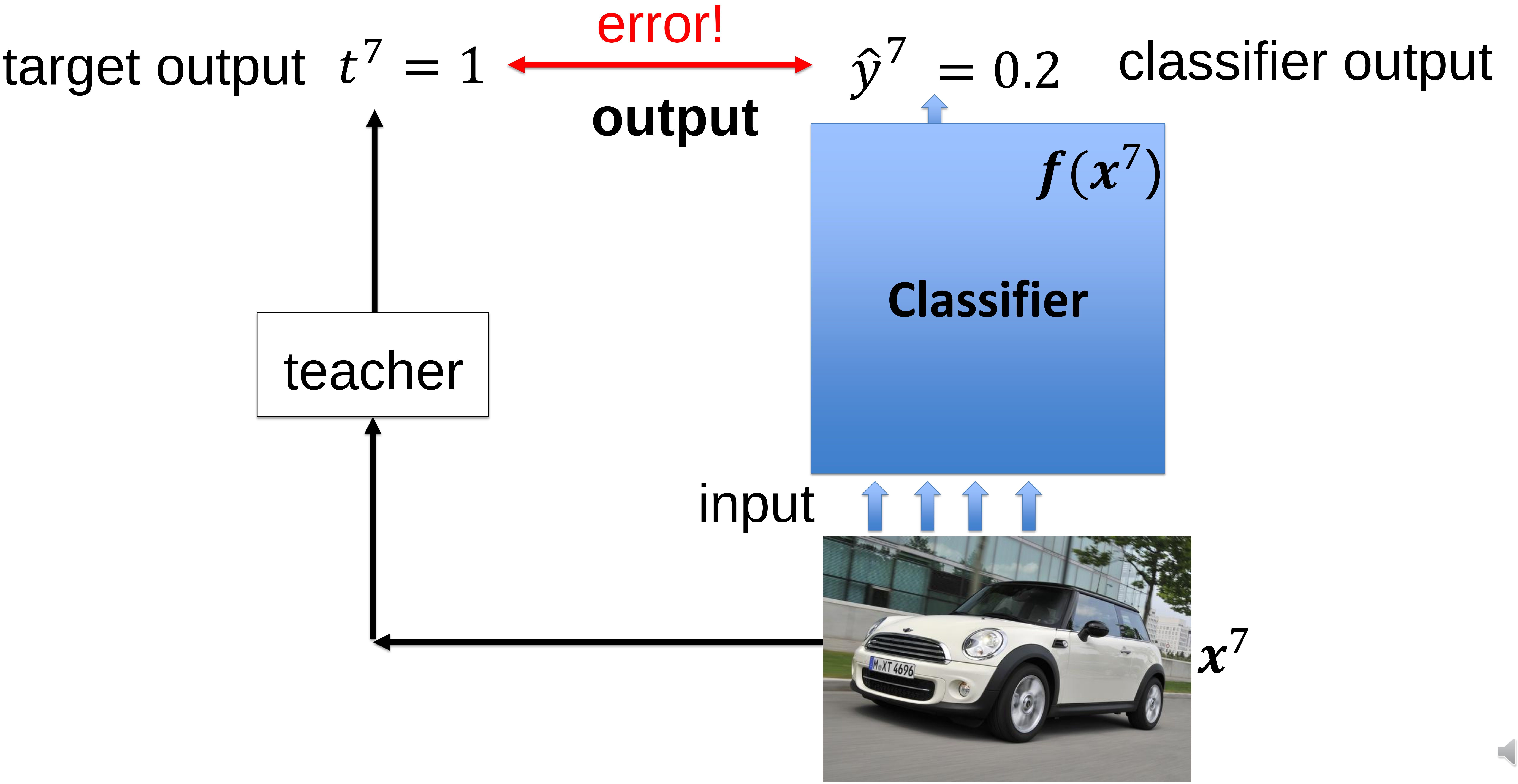
-1

$\boldsymbol{x} \in R^{N+1}$

Previous slide.
Neural networks with hidden layers are much more powerful, because they can solve problems that are not linearly separable.

However, we need to answer the question of how we find a solution in cases where we cannot construct the solution geometrically.

3. Supervised learning with sigmoidal output



target output $t^7 = 1$

error!

output

$\hat{y}^7 = 0.2$ classifier output

$\boldsymbol{f}(\boldsymbol{x}^7)$

Classifier

teacher

input

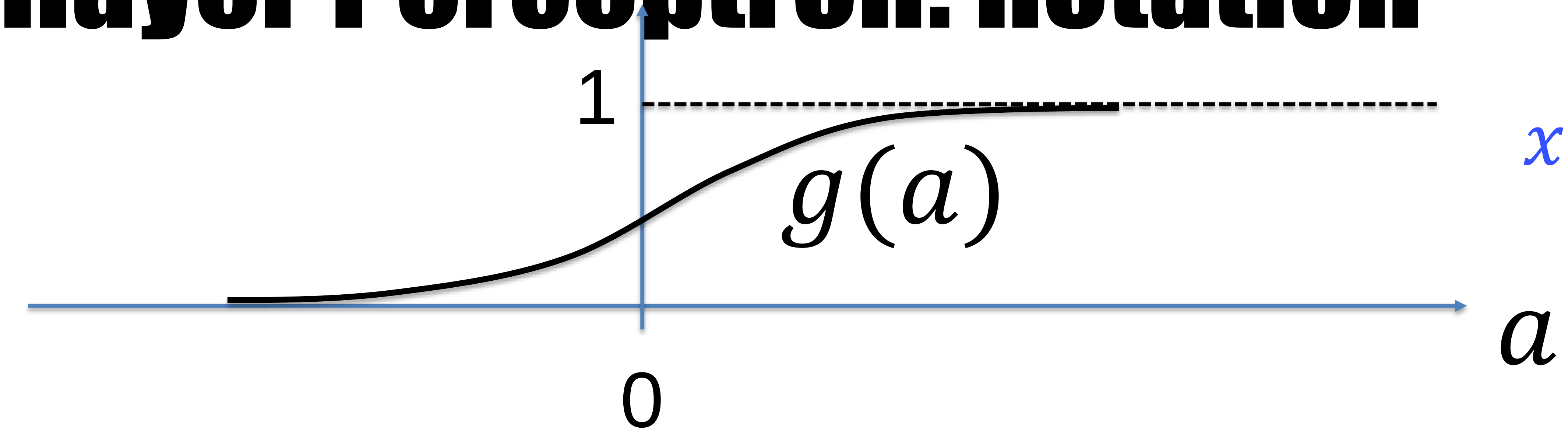
$\boldsymbol{x}^7$

Previous slide.
To this end we start with an error function defined via the comparison of the actual output with the target outpt.

3. Multilayer Perceptron: notation

$a_i^{(n)}$ = activation/drive of neuron

$x_i^{(n)}$ = neuron output



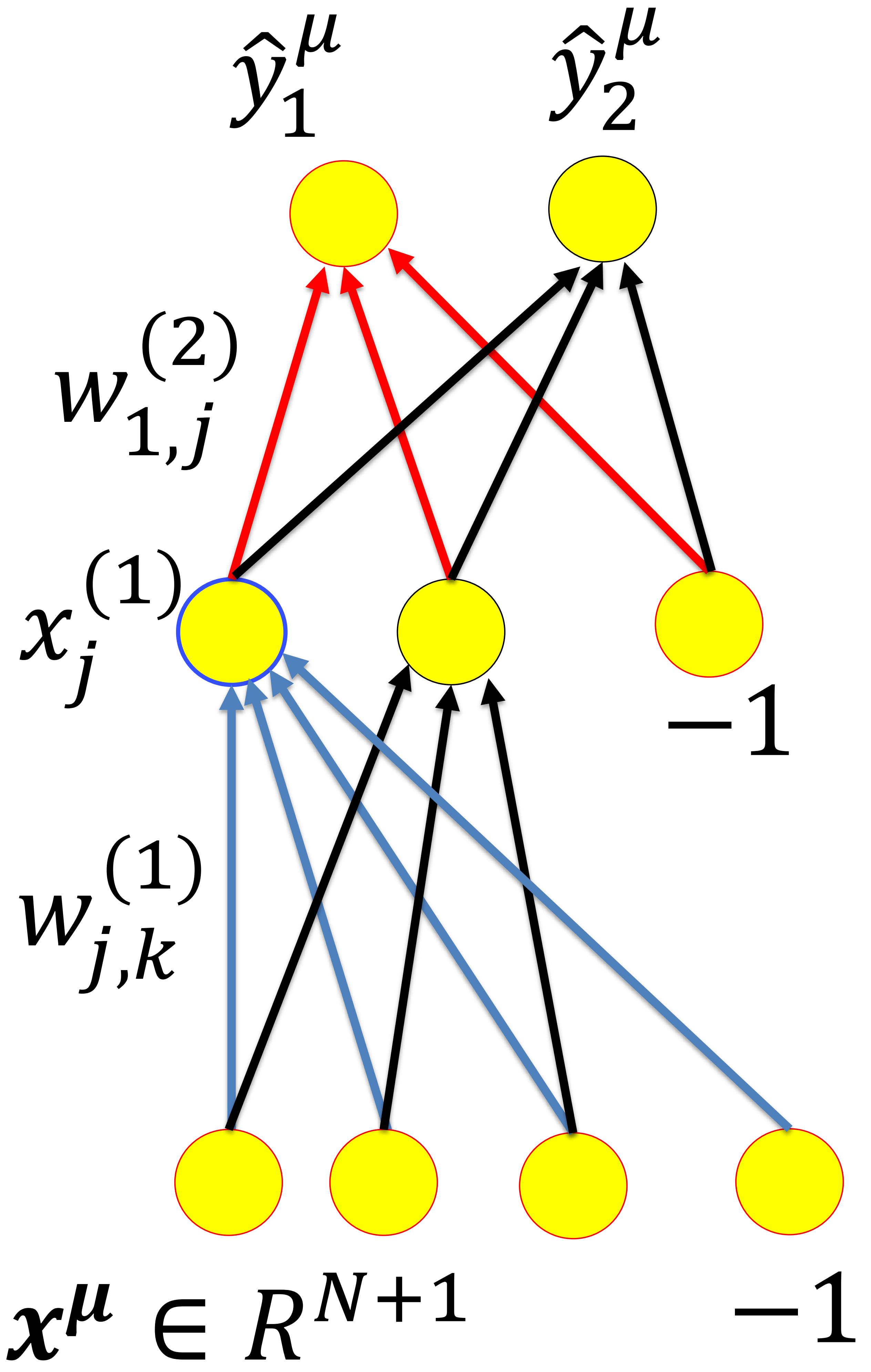
$$\hat{y}_i^\mu = x_i^{(2)} \qquad (1)$$

$$= g^{(2)}[a_i^{(2)}] \qquad (2)$$

$$= g^{(2)}[\sum_j w_{ij}^{(2)} x_j^{(1)}] \qquad (3)$$

$$= g^{(2)}[\sum_j w_{ij}^{(2)} g^{(1)}(a_j^{(1)})] \qquad (4)$$

$$= g^{(2)}[\sum_j w_{ij}^{(2)} g^{(1)}(\sum_k w_{jk}^{(1)} x_k^\mu)] \qquad (5)$$



Previous slide.
For the actual output we have an explicit formula.

Notation:

- upper index in parenthesis = layer of network
- Lower index = neuron in the layer
- $w_{1,j}^{(n)}$ = weight from neuron j in layer (n-1) to neuron 1 in layer n

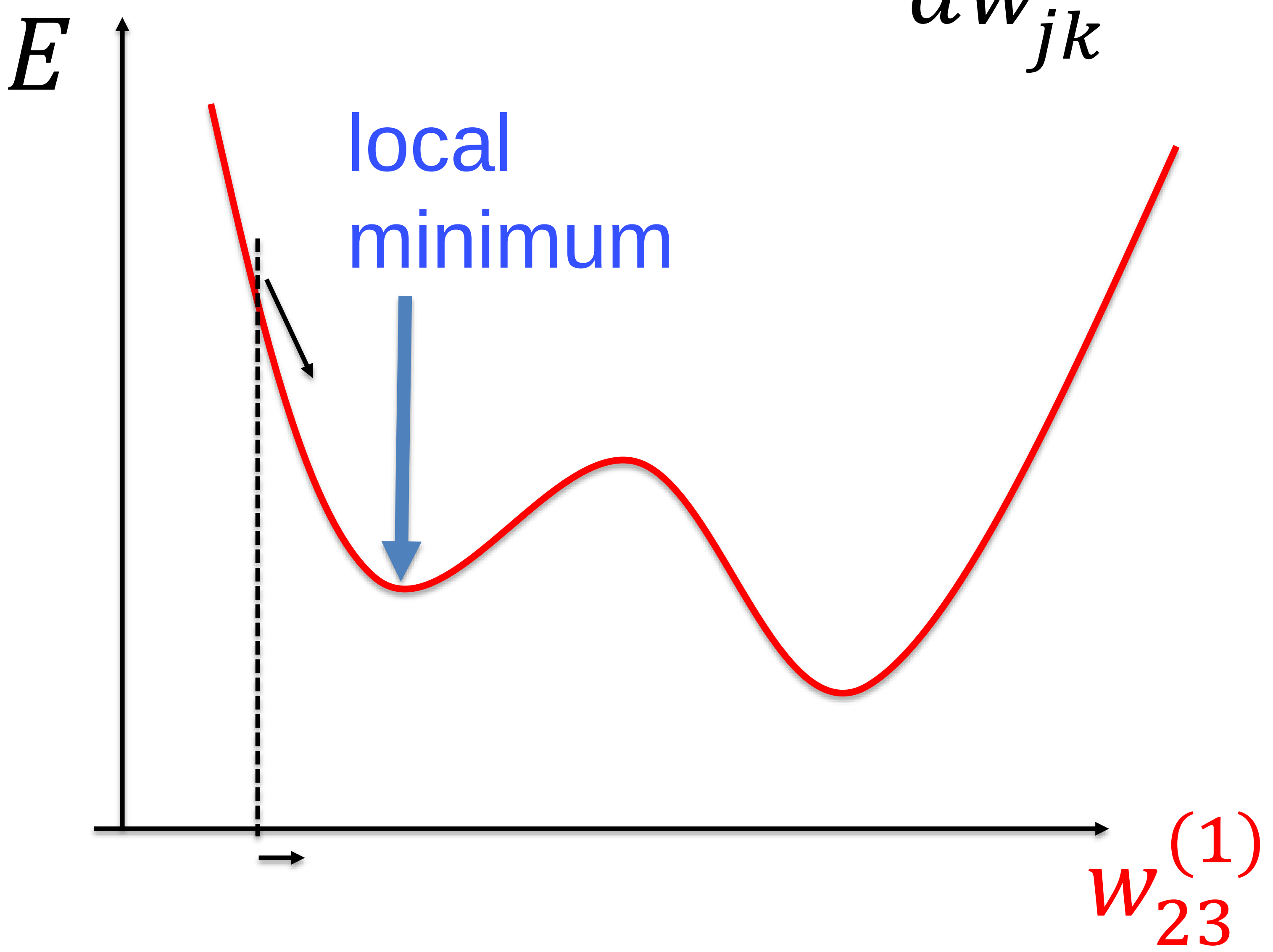
3. Multilayer Perceptron: gradient descent

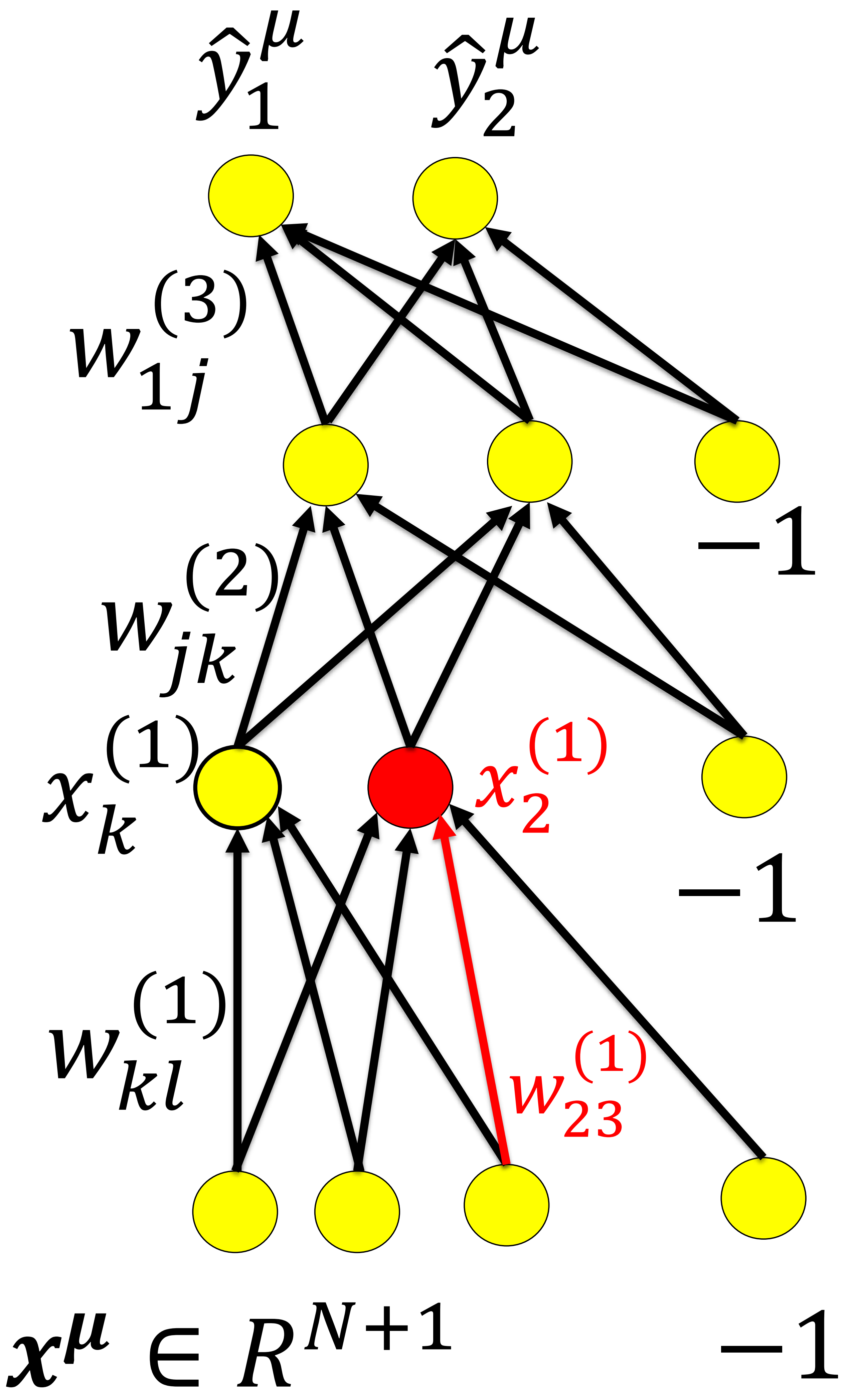
Quadratic **error**

$$E(\mathbf{w}) = \frac{1}{2}\sum_{\mu=1}^{P}\sum_{i}[t_i^{\mu} - \hat{y}_i^{\mu}]^2$$

gradient descent

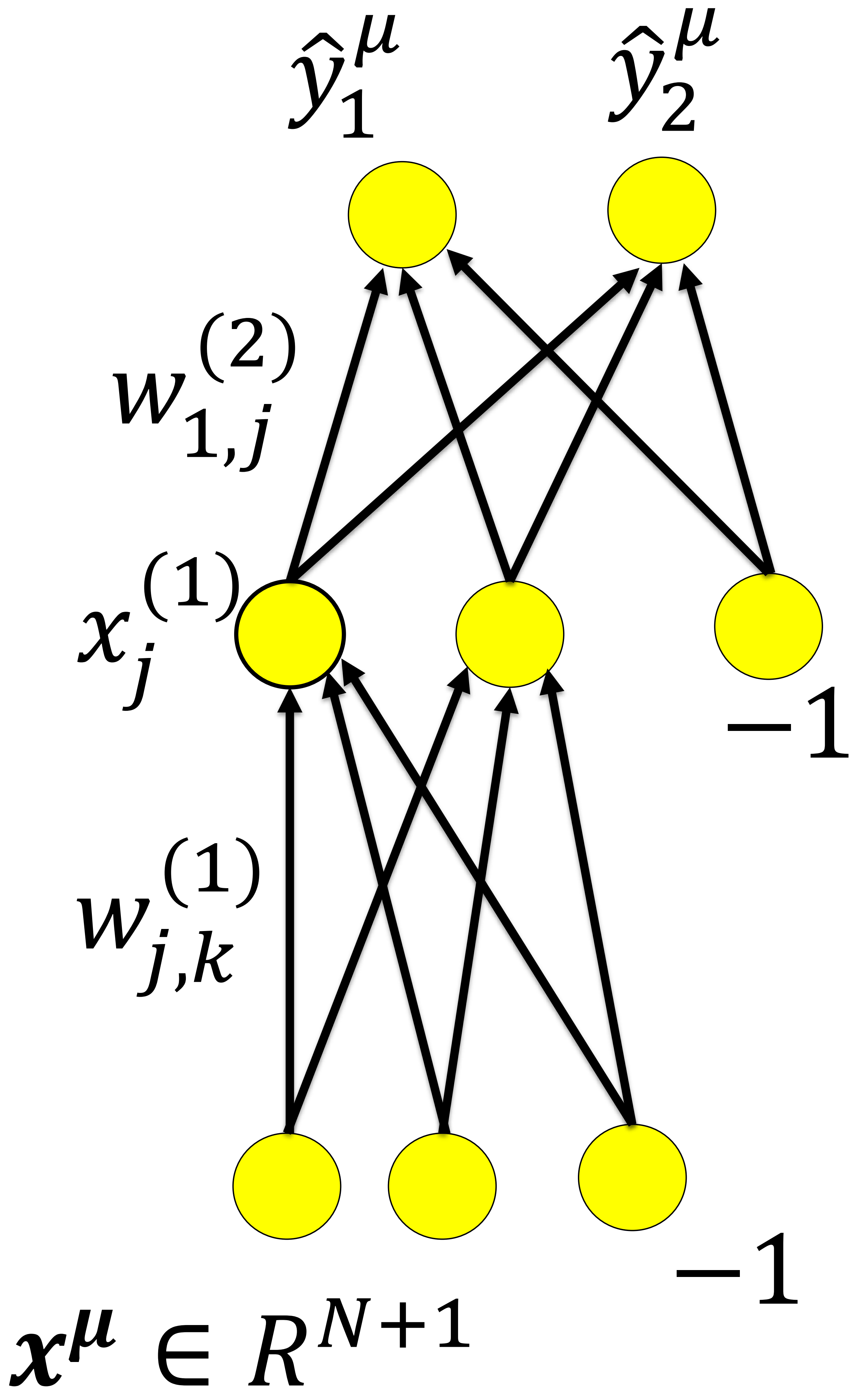
$$\Delta w_{jk}^{(1)} = -\gamma \frac{dE}{dw_{jk}^{(1)}}$$





Previous slide.
To reduce the error in the output we can use gradient descent.

Error depends on all weights.
We plot dependence with respect to one specific weight.



Pause the video, take a few minutes,
Apply chain rule to calculate gradient

$$E(\mathbf{w}) = \frac{1}{2}\sum_{\mu=1}^{P}\sum_{i}[t_i^{\mu} - \hat{y}_i^{\mu}]^2$$

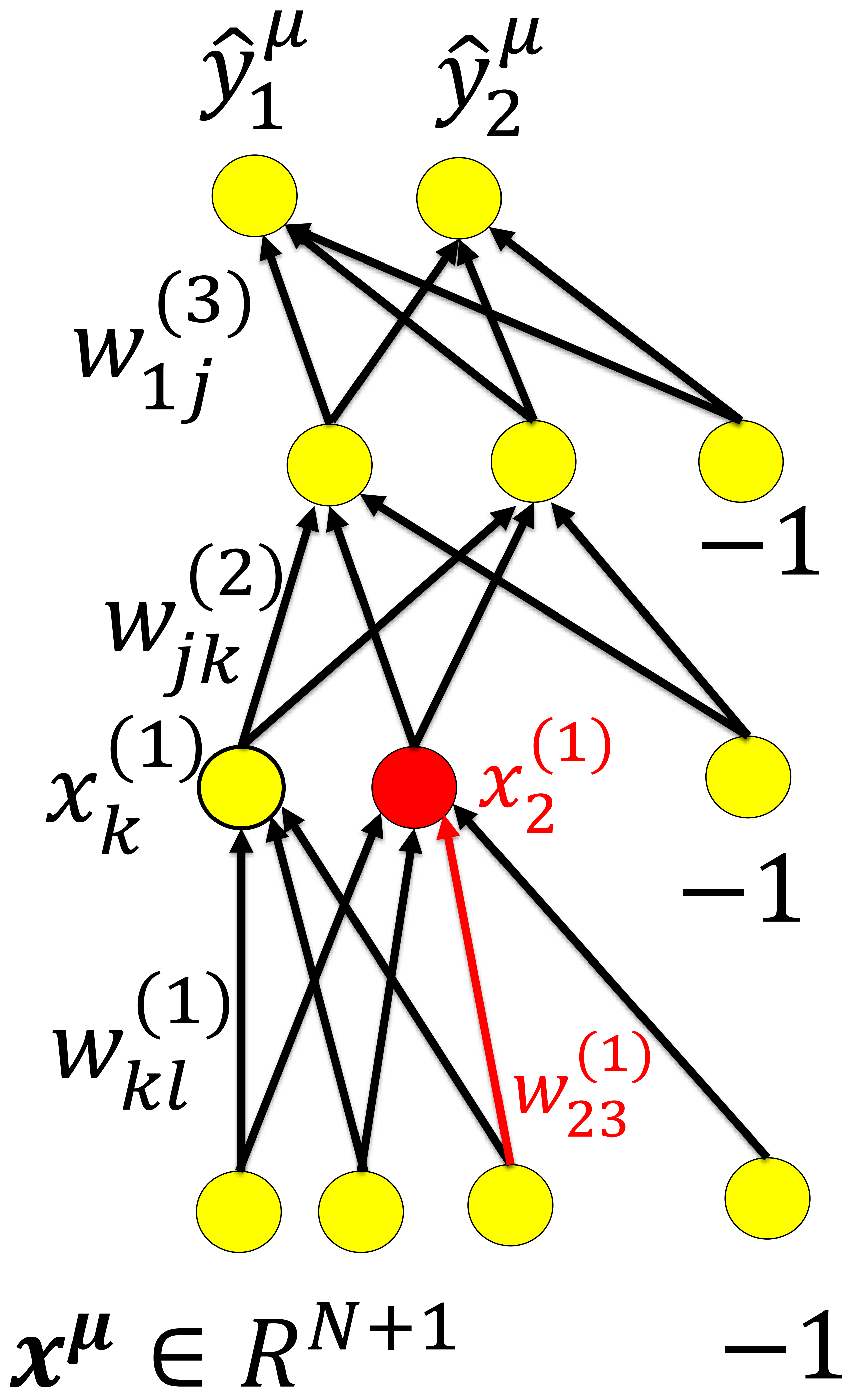
$$\Delta w_{23}^{(1)} = -\gamma \frac{dE}{dw_{23}^{(1)}}$$

with

$$\hat{y}_i^{\mu} = g^{(3)}\Big(\sum_j w_{ij}^{(3)}\, x_j^{(2)}\Big)$$

$$x_j^{(2)} = g^{(2)}\Big(\sum_k w_{jk}^{(2)}\, x_k^{(1)}\Big)$$

$$x_2^{(1)} = g^{(1)}(a_2^{(1)}) = g^{(1)}(\textstyle\sum_l w_{2l}^{(1)}\, x_l^{(0)})$$



Your notes.

In the image $w_{1j}^{(3)} = w_{1,j}^{(3)}$

refers to the j’th component of the weight vector converging onto the first neuron in layer 3. You can put a comma or not.

3. Calculate gradient: step 1

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{\mu=1}^{P}\sum_{i}[t_i^{\mu} - \hat{y}_i^{\mu}]^2$$

$$\hat{y}_i^{\mu} = g^{(3)}(\, a_i^{(3)})$$

$$a_i^{(3)} = \sum_j w_{ij}^{(3)}\, x_j^{(2)}$$

$$x_j^{(2)} = g^{(2)}(\, a_j^{(2)})$$

$$a_j^{(2)} = \sum_k w_{jk}^{(2)}\, x_k^{(1)}$$

$$x_k^{(1)} = g^{(1)}(\, a_k^{(1)})$$

$$a_k^{(1)} = \sum_l w_{kl}^{(1)}\, x_l^{(0)}$$

Input: layer 0

Step 1: identify intermediate variables

$a_i^{(n)}$ = activation/drive of neuron

$x_i^{(n)}$ = neuron output

3. Calculate gradient: step 1

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{\mu=1}^{P}\sum_{i}[t_i^{\mu} - \hat{y}_i^{\mu}]^2$$

$$\hat{y}_i^{\mu} = g^{(3)}(a_i^{(3)})$$

$$a_i^{(3)} = \sum_j w_{ij}^{(3)} x_j^{(2)}$$

$$x_j^{(2)} = g^{(2)}(a_j^{(2)})$$

$$a_j^{(2)} = \sum_k w_{jk}^{(2)} x_k^{(1)}$$

$$x_k^{(1)} = g^{(1)}(a_k^{(1)})$$

$$a_k^{(1)} = \sum_l w_{2l}^{(1)} x_l^{(0)}$$

Step 1: identify intermediate variables

$a_i^{(n)}$ = activation/drive of neuron

$x_i^{(n)}$ = neuron output

$$\delta_k^{(n)} = \frac{\partial E}{\partial a_k^{(n)}}$$

3. Calculate gradient: step 2

Step 1: identify intermediate variables

$a_i^{(n)}$= activation/drive of neuron

$x_i^{(n)}$= neuron output

$$\delta_k^{(n)} = \frac{\partial E}{\partial a_k^{(n)}}$$

Step 2: write weight update with these variables

$$\Delta w_{23}^{(1)} = -\gamma \frac{dE}{dw_{23}^{(1)}} = -\gamma \frac{dE}{da_2^{(1)}} \frac{da_2^{(1)}}{dw_{23}^{(1)}}$$

$$= -\gamma\, \delta_2^{(1)}\, x_3^{(0)}$$

$$a_k^{(1)} = \sum_l w_{2l}^{(1)}\, x_l^{(0)}$$

Analogous for all weights in all layers

3. Calculate gradient: step 3

Step 2: write weight update

$$\Delta w_{23}^{(1)} = -\gamma\, \delta_2^{(1)}\, x_3^{(0)}$$

Step 3: Analyze dependency graph

$$\delta_2^{(1)} = \frac{\partial E}{\partial a_2^{(1)}}$$

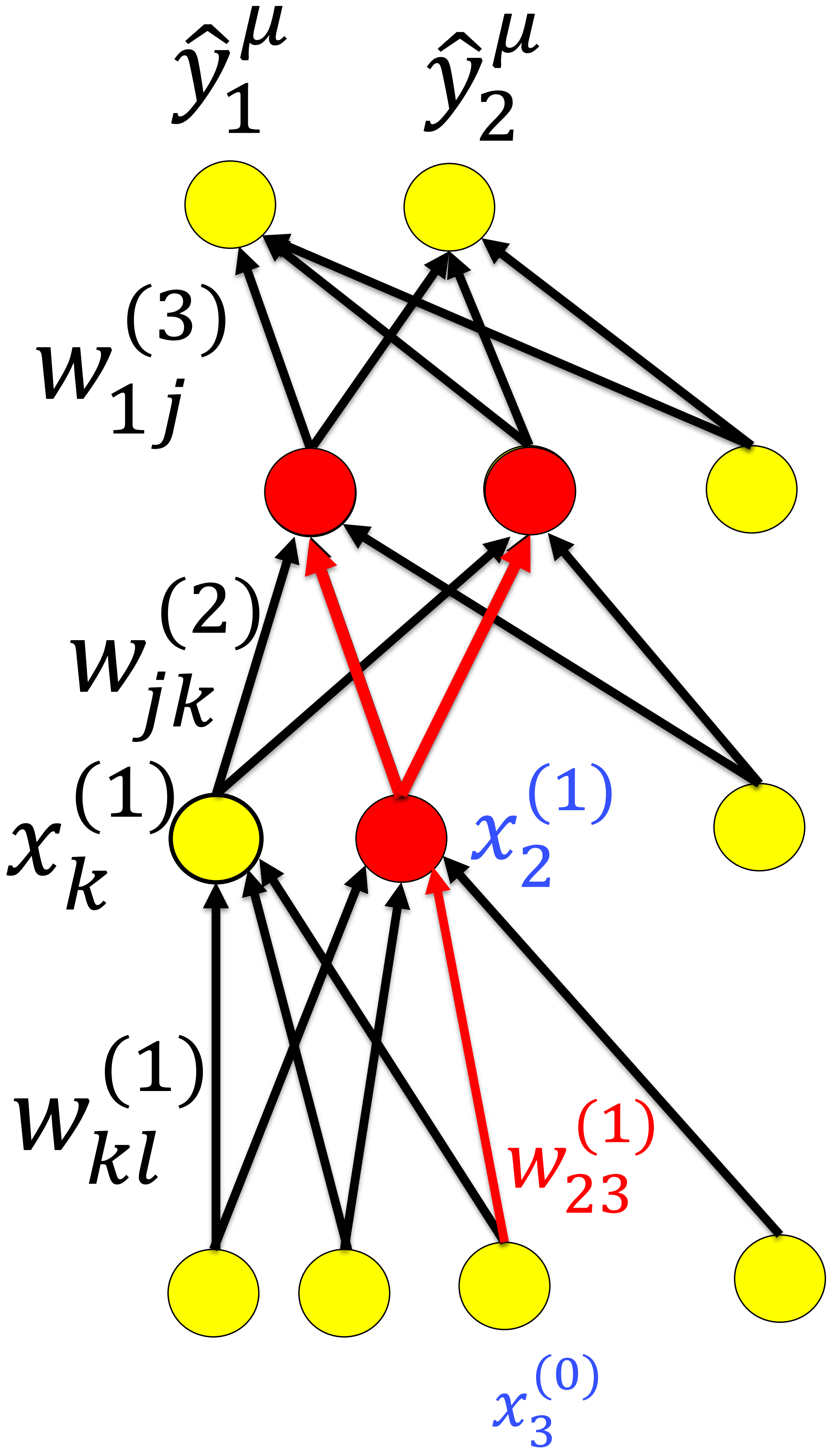
Chain rule again

$$\delta_2^{(1)} = \frac{\partial E}{\partial a_2^{(1)}} = \sum_j \frac{\partial E}{\partial a_j^{(2)}} \frac{\partial a_j^{(2)}}{\partial a_2^{(1)}}$$

$$\delta_2^{(1)} = \sum_j \delta_j^{(2)}\, w_{j2}^{(2)} g'$$

$$a_j^{(2)} = \sum_k w_{jk}^{(2)}\, x_k^{(1)}$$

$$a_j^{(2)} = \sum_k w_{jk}^{(2)} g(a_k^{(1)})$$



Your notes.

3. Calculate gradient by chain rule (summary)

Step 1: identify intermediate variables

$$a_k^{(n)} = \sum_l w_{kl}^{(n)} \, x_l^{(n-1)}$$

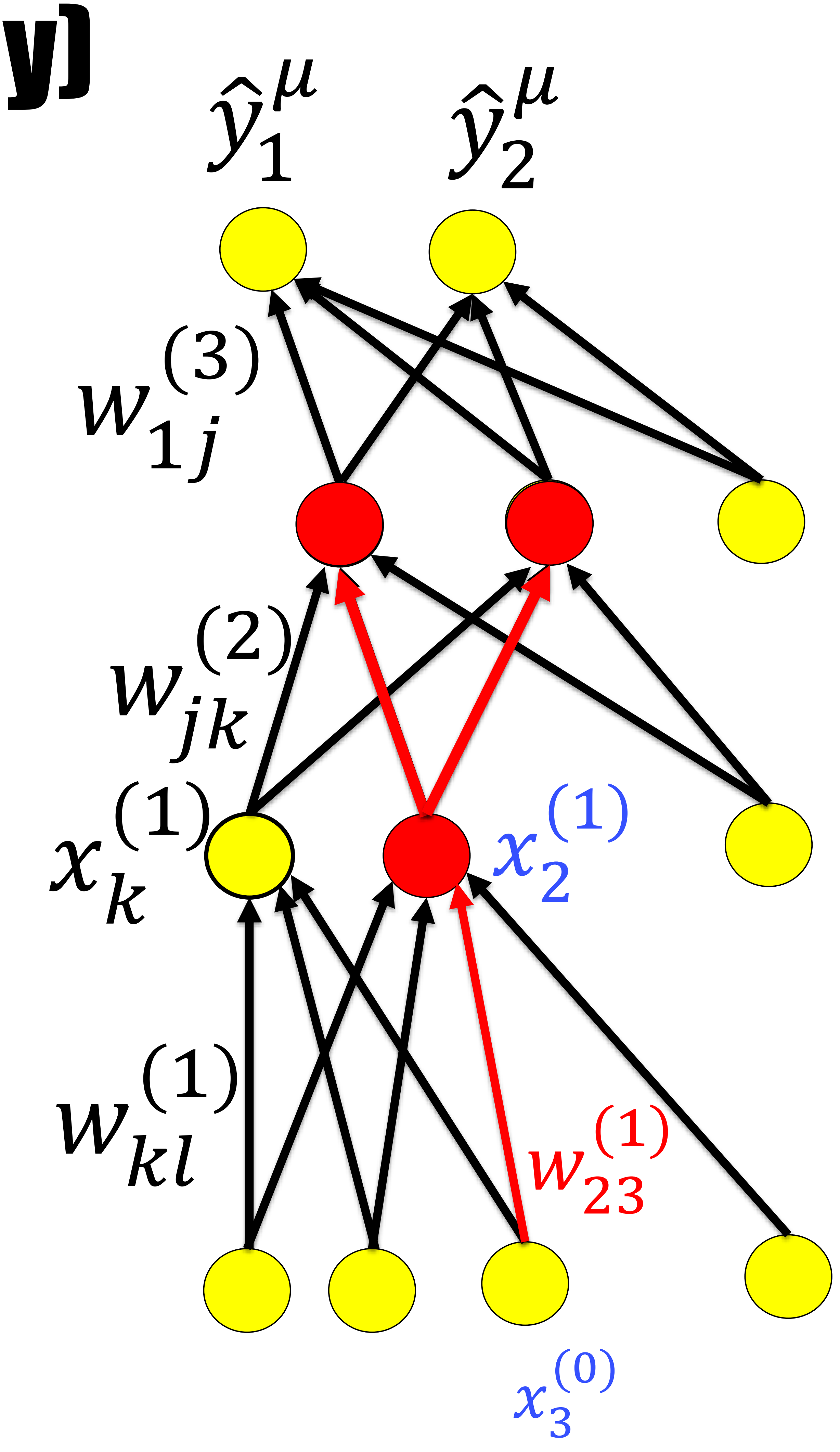
$$\delta_k^{(n)} = \frac{\partial E}{\partial a_k^{(n)}}$$

Step 2: write weight update

$$\Delta w_{23}^{(n)} = -\gamma \, \delta_2^{(n)} \, x_3^{(n-1)}$$

Step 3: Analyze dependency graph

$$\delta_k^{(n-1)} = \sum_j \delta_j^{(n)} \, w_{jk}^{(2)} g'$$



3. Multilayer Perceptron: gradient descent

Calculating a gradient in multi-layer networks:

- write down chain rule
- analyze dependency graph
- store intermediate results
- update intermediate results while proceeding through graph

compare with dynamic programming

- update all weights together at the end

Previous slide.
The chain rule in a multi-layer network gives rise to many, many terms.

Because of the nature of the chain rule, some terms depend on others. It is important to analyze these dependencies.
The dependency graph indicates which values or variables will be important to calculate other values or variables.
Just as in dynamic programming, the trick consists in storing those intermediate variables or values that can be reused.

The actual weight update is done only at the end.

Artificial Neural Networks

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Deep Nets 1: BackProp and Multilayer Networks

Part 4: BackProp Algorithm

1. Gradient Descent Methods
2. XOR problem
3. Multilayer Networks: First Steps
4. **BackProp Algorithm**

Previous slide.

The above ideas on implementing the chain rule with storage of intermediate values gives rise to the BackProp algorithm. In other words, BackProp is just an (efficient) implementation of the chain rule.

BackProp

0. Initialization of weights

1. Choose pattern $\mathbf{x}^{\mu}$

   input $x_k^{(0)} = x_k^{\mu}$

2. Forward propagation of signals $x_k^{(n-1)} \longrightarrow x_j^{(n)}$

$$x_j^{(n)} = g^{(n)}(a_j^{(n)}) = g^{(n)}(\sum w_{jk}^{(n)} x_k^{(n-1)}) \qquad (1)$$

   output $\hat{y}_i^{\mu} = x_i^{(n_{\max})}$

3. Computation of errors in output

$$\delta_i^{(n_{\max})} = g'(a_i^{(n_{\max})}) \; [\hat{y}_i^{\mu} - t_i^{\mu}] \qquad (2)$$

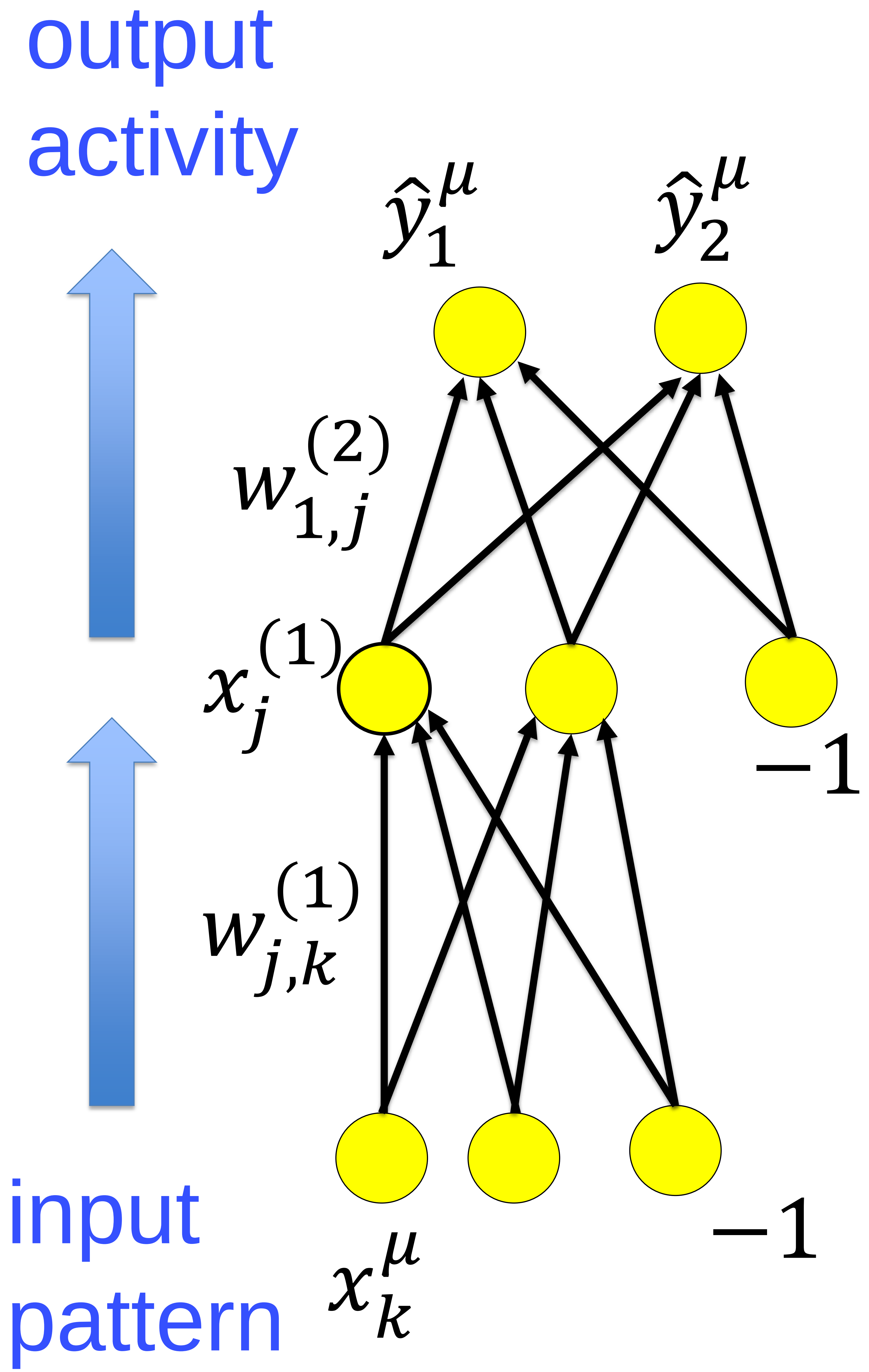
4. Backward propagation of errors $\delta_i^{(n)} \longrightarrow \delta_j^{(n-1)}$

$$\delta_j^{(n-1)} = g'^{(n-1)}(a^{(n-1)}) \sum_i w_{ij} \, \delta_i^{(n)} \qquad (3)$$

5. Update weights (for each (i,j) and all layers (n))

$$\Delta w_{ij}^{(n)} = -\gamma \; \delta_i^{(n)} \, x_j^{(n-1)} \qquad (4)$$

6. Return to step 1.



Previous slide.

Backprop (online rule/stochastic gradient descent)

1. We select a pattern and apply it as an input (activity in layer zero), and store the activity in layer zero.
2. Knowing the activity in layer n, we calculate the activity in layer n+1 and store the result (FORWARD pass)
3. We transform the mismatch for each neuron in the output layer into a delta-signal.
4. Knowing the delta signal for each neuron in layer n, we calculate the delta signal in for each neuron in layer n-1 and store the result (BACKWARD pass)
5. We update ALL weights.

BackProp

0. **Initialization of weights**

1. **Choose pattern $\mathbf{x}^{\mu}$**

   input $x_k^{(0)} = x_k^{\mu}$

2. **Forward propagation of signals** $x_k^{(n-1)} \longrightarrow x_j^{(n)}$

$$x_j^{(n)} = g^{(n)}(a_j^{(n)}) = g^{(n)}(\sum w_{jk}^{(n)} x_k^{(n-1)}) \qquad (1)$$

   output $\hat{y}_i^{\mu} = x_i^{(n_{\max})}$

3. **Computation of errors in output**

$$\delta_i^{(n_{\max})} = g'(a_i^{(n_{\max})}) \; [\hat{y}_i^{\mu} - t_i^{\mu}] \qquad (2)$$

4. **Backward propagation of errors** $\delta_i^{(n)} \longrightarrow \delta_j^{(n-1)}$

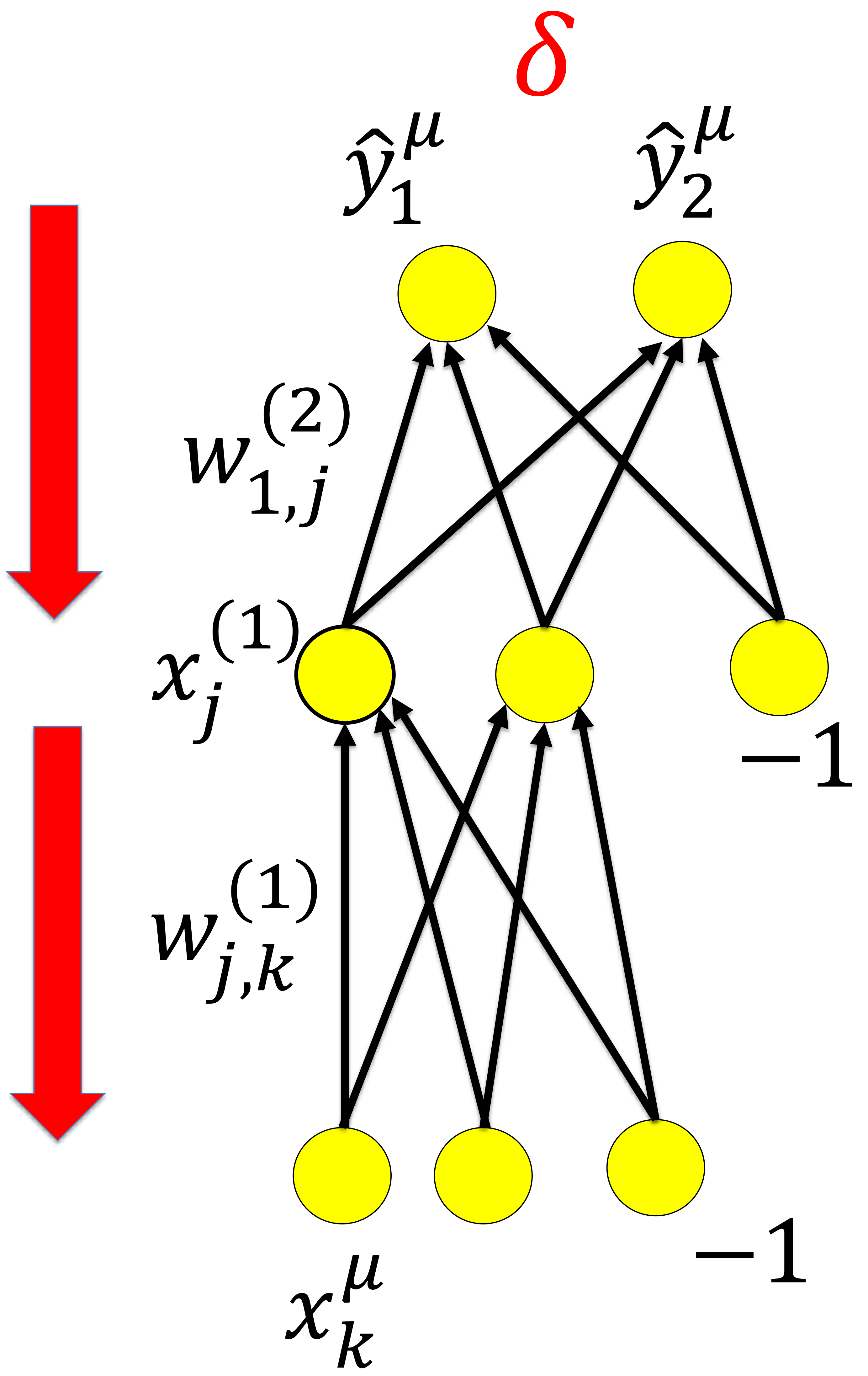
$$\delta_j^{(n-1)} = g'^{(n-1)}(a^{(n-1)}) \sum_i w_{ij} \, \delta_i^{(n)} \qquad (3)$$

5. **Update weights (for each (i,j) and all layers (n))**

$$\Delta w_{ij}^{(n)} = -\gamma \;\; \delta_i^{(n)} \, x_j^{(n-1)} \qquad (4)$$

6. **Return to step 1.**

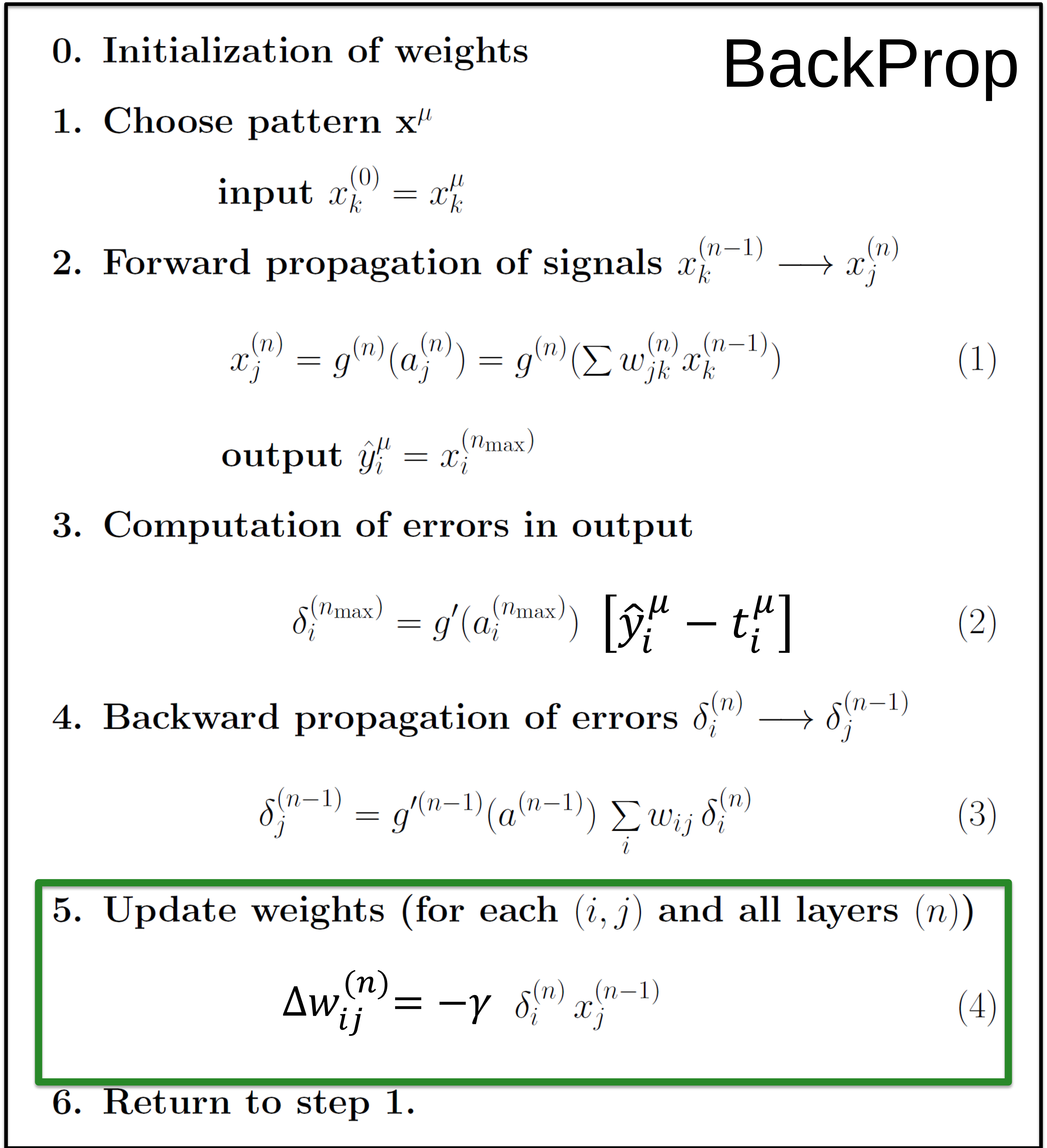
Calculate output error



Previous slide.

The name backpropagation of errors arises since in the BACKWARD PASS the information about delta-signals propagates backward.

update **all** weights



BackProp

0. Initialization of weights

1. Choose pattern $\mathbf{x}^{\mu}$

   input $x_k^{(0)} = x_k^{\mu}$

2. Forward propagation of signals $x_k^{(n-1)} \longrightarrow x_j^{(n)}$

$$x_j^{(n)} = g^{(n)}(a_j^{(n)}) = g^{(n)}(\sum w_{jk}^{(n)} x_k^{(n-1)}) \tag{1}$$

   output $\hat{y}_i^{\mu} = x_i^{(n_{\max})}$

3. Computation of errors in output

$$\delta_i^{(n_{\max})} = g'(a_i^{(n_{\max})}) \; [\hat{y}_i^{\mu} - t_i^{\mu}] \tag{2}$$

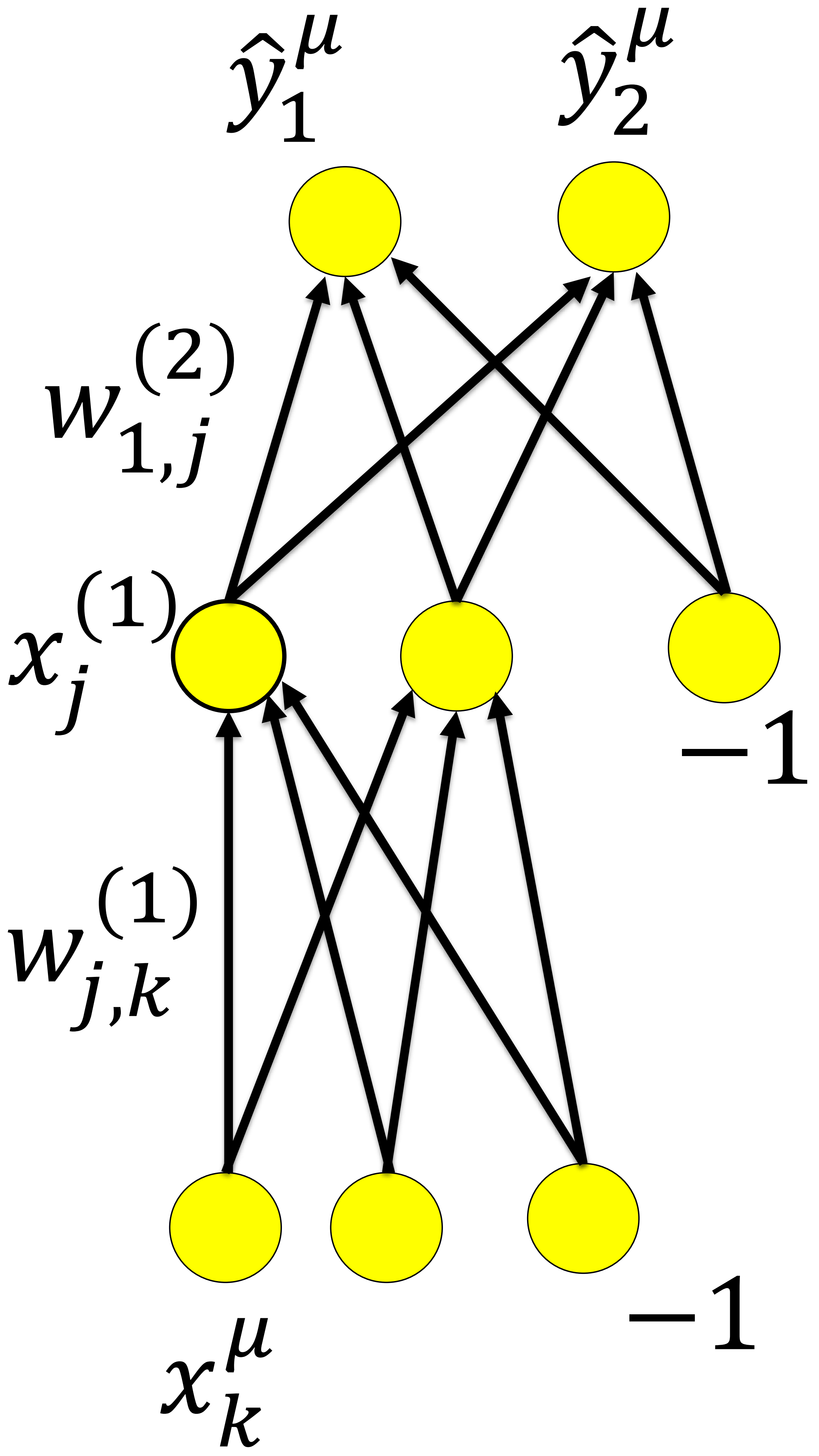
4. Backward propagation of errors $\delta_i^{(n)} \longrightarrow \delta_j^{(n-1)}$

$$\delta_j^{(n-1)} = g'^{(n-1)}(a^{(n-1)}) \sum_i w_{ij}\, \delta_i^{(n)} \tag{3}$$

5. Update weights (for each (i,j) and all layers (n))

$$\Delta w_{ij}^{(n)} = -\gamma \;\; \delta_i^{(n)} \, x_j^{(n-1)} \tag{4}$$

6. Return to step 1.



Previous slide.

The update of the weights needs the delta-signals AND the activation signals (outputs) of EACH NEURON. Important, ALL weights can be updated once we have calculated the intermediate variables $\delta_i^{(n)}$ and $x_j^{(n-1)}$ for all neurons in all layers.

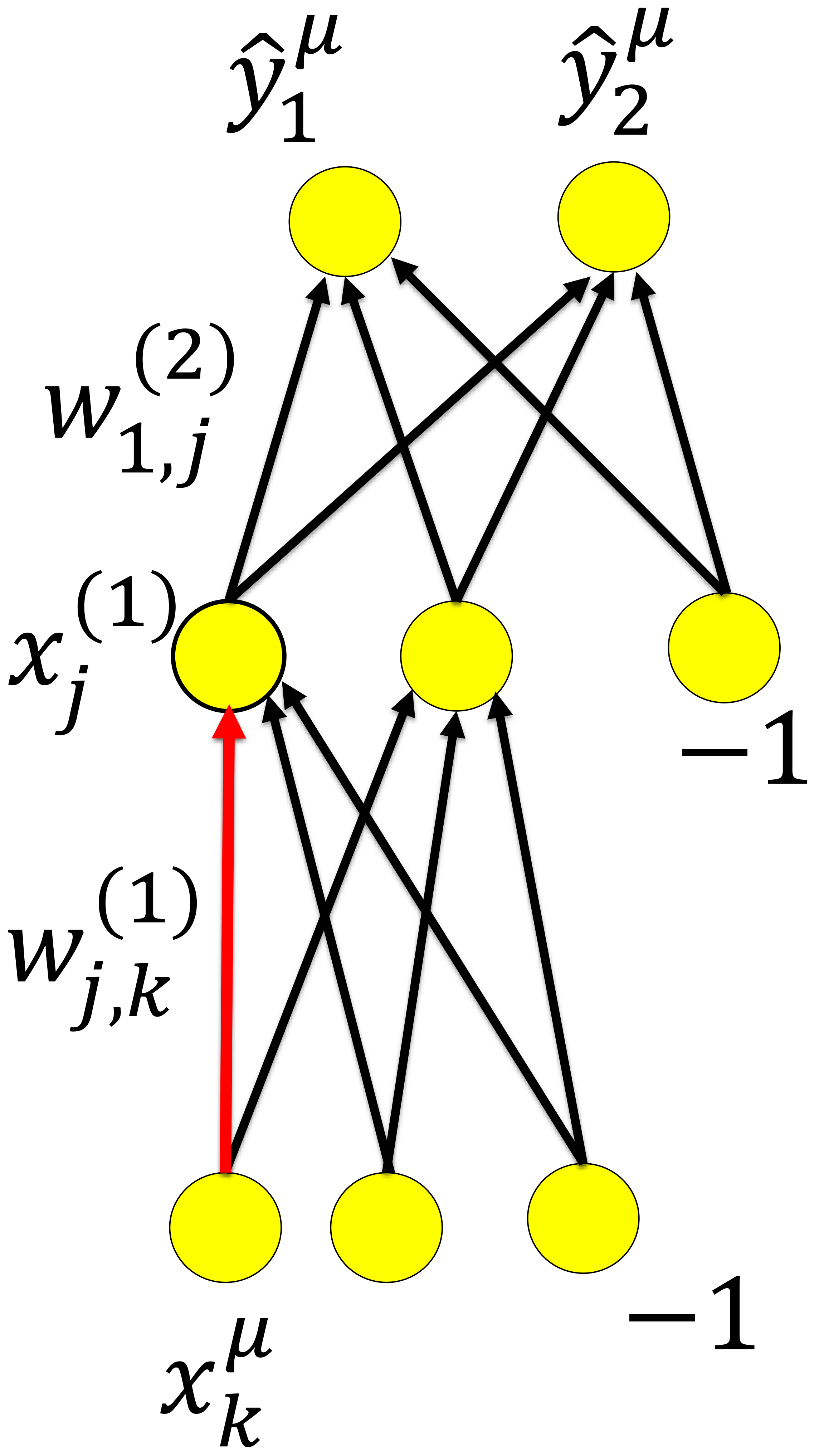
4. BackProp versus direct numerical evaluation of gradient

$$\Delta w_{jk}^{(1)} = -\gamma \frac{dE}{dw_{jk}^{(1)}}$$

$$= -\gamma \frac{E\left(w_{jk}^{(1)} + \varepsilon\right) - E\left(w_{jk}^{(1)} - \varepsilon\right)}{2\varepsilon}$$

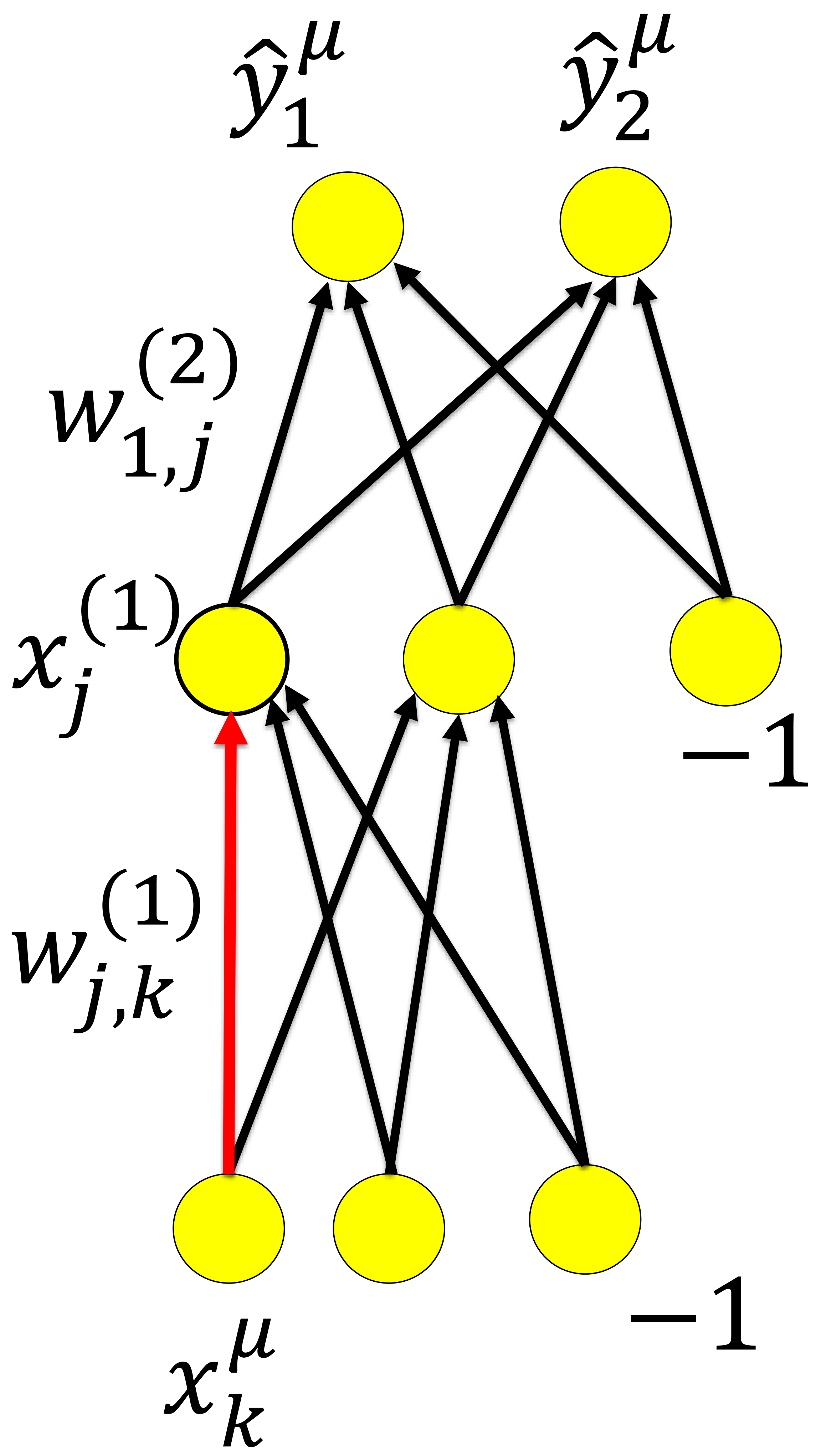
calculate $E(\boldsymbol{w}) = \frac{1}{2}\sum_{\mu=1}^{P}\sum_{i}\left[t_i^{\mu} - \hat{y}_i^{\mu}\right]^2$

→ calculate $\hat{y}_i^{\mu}$ for one pattern



Previous slide.

Backprop (exploitation of the chain rule) is very efficient compared to a direct numerical calculation of the gradient.



$$\frac{E\left(w_{jk}^{(1)}+\varepsilon\right)-E\left(w_{jk}^{(1)}-\varepsilon\right)}{2\varepsilon}$$

$$\boxed{E(\mathbf{w}) = \frac{1}{2}\sum_{\mu=1}^{P}\sum_{i}[t_i^{\mu} - \hat{y}_i^{\mu}]^2}$$

$$\begin{aligned}
\hat{y}_i^{\mu} &= x_i^{(2)} && (1)\\
&= g^{(2)}[a_i^{(2)}] && (2)\\
&= g^{(2)}[\sum_j w_{ij}^{(2)} x_j^{(1)}] && (3)\\
&= g^{(2)}[\sum_j w_{ij}^{(2)} g^{(1)}(a_j^{(1)})] && (4)\\
&= g^{(2)}[\sum_j w_{ij}^{(2)} g^{(1)}(\sum_k w_{jk}^{(1)} x_k^{\mu})] && (5)
\end{aligned}$$

Your notes.

4. Direct numerical differentiation: complexity

calculate $E(\mathbf{w}) = \frac{1}{2}\sum_{\mu=1}^{P}\sum_{i}[t_i^\mu - \hat{y}_i^\mu]^2$

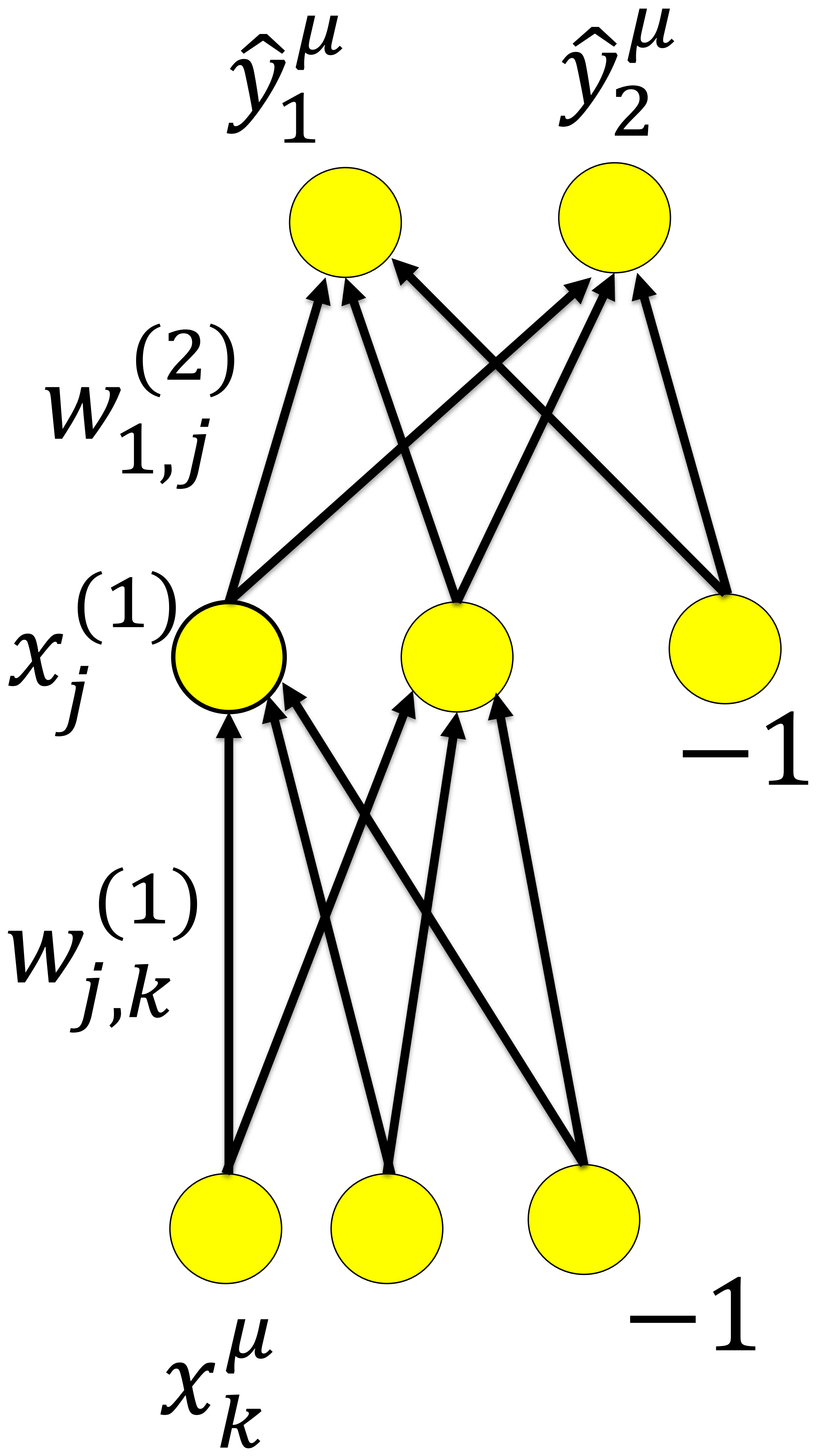
1) calculate $\hat{y}_i^\mu$ for one pattern

→ each weight is touched once

2) for each change of weight, evaluate E twice

$$\Delta w_{jk}^{(1)} = -\gamma \frac{dE\left(w_{jk}^{(1)} + \varepsilon\right) - dE\left(w_{jk}^{(1)} - \varepsilon\right)}{2\varepsilon}$$

3) For ***n*** weights, order ***n***-square!!!



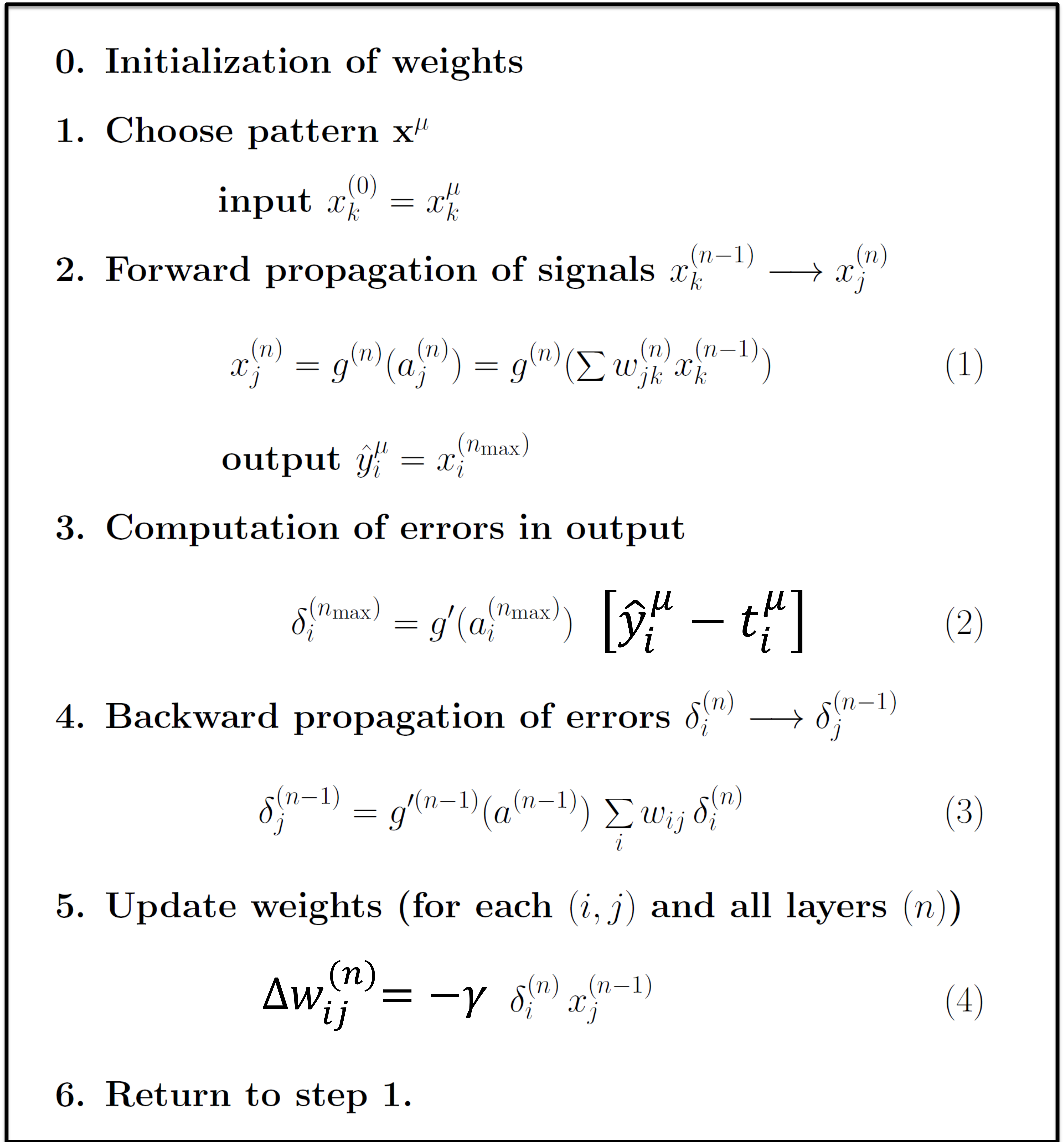
Previous slide.
The forward pass is the same as in Backprop, but we need to run the forward pass twice, once with the current weights PLUS a small correction and once with the current weights MINUS a small correction (or without).
Since each weight is touched once during the forward pass, the order of complexity is n, where n is the number of weights.

However, after n calculations we can just update a single one of the weights, the one to which we had applied the weight perturbation.

To update all the n weights, we therefore need n-square steps.

3. BackProp: complexity

Algorithm is of order ***n*** (# of weights)



0. Initialization of weights

1. Choose pattern $\mathbf{x}^{\mu}$

   input $x_k^{(0)} = x_k^{\mu}$

2. Forward propagation of signals $x_k^{(n-1)} \longrightarrow x_j^{(n)}$

$$x_j^{(n)} = g^{(n)}(a_j^{(n)}) = g^{(n)}(\sum w_{jk}^{(n)} x_k^{(n-1)}) \qquad (1)$$

   output $\hat{y}_i^{\mu} = x_i^{(n_{\max})}$

3. Computation of errors in output

$$\delta_i^{(n_{\max})} = g'(a_i^{(n_{\max})}) \; [\hat{y}_i^{\mu} - t_i^{\mu}] \qquad (2)$$

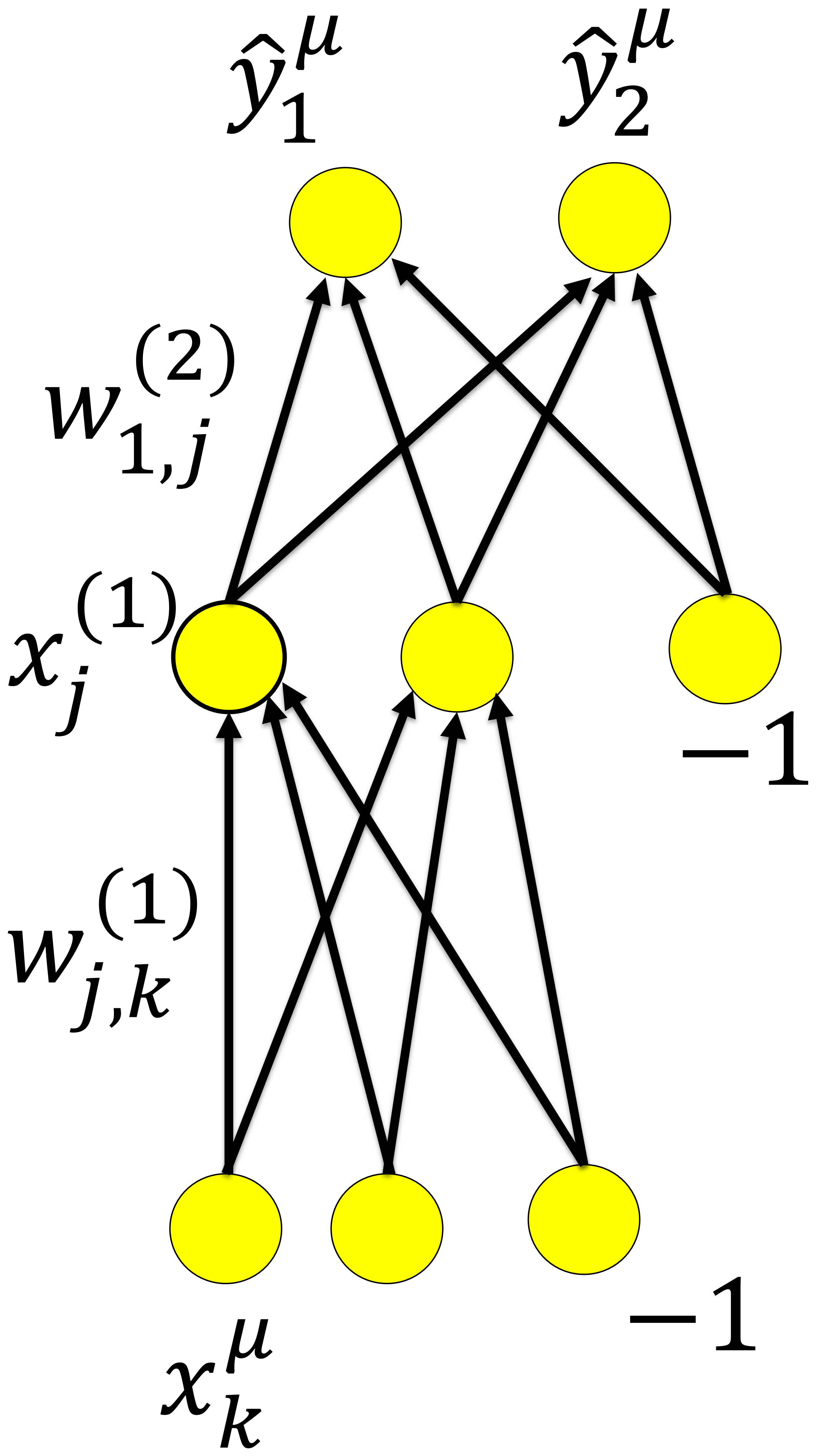
4. Backward propagation of errors $\delta_i^{(n)} \longrightarrow \delta_j^{(n-1)}$

$$\delta_j^{(n-1)} = g'^{(n-1)}(a^{(n-1)}) \sum_i w_{ij}\, \delta_i^{(n)} \qquad (3)$$

5. Update weights (for each (i,j) and all layers (n))

$$\Delta w_{ij}^{(n)} = -\gamma \;\; \delta_i^{(n)} \, x_j^{(n-1)} \qquad (4)$$

6. Return to step 1.



Previous slide.
However, with the Backprop algorithm the update of all the weights requires only n steps since the intermediate results for the deltas and the activations are stored and reused for all weights.

4. BackProp: Quiz

Your friend claims the following; do you agree?

[ ] BackProp is nothing else than the chain rule, handled well.

[ ] BackProp is just **numerical** differentiation

[ ] BackProp is a special case of automatic **algorithmic** differentiation

[ ] BackProp is an order of magnitude faster than numerical differentiation

Your notes.

4. Conclusions: Multilayer Networks and BackProp

- Hidden neurons increase the flexibility of the separating surface → beyond linearly separable problems
- Weights are the parameters of the separating surface
- Weights can be adapted by gradient descent
- Backprop is an implementation of gradient descent
- Gradient descent converges to a local minimum

→ **Big Multilayer networks are flexible and can be trained by BackProp to minimize classification error**

Previous slide.

Thus multi-layer perceptrons are more powerful than simple perceptrons and can be trained using backprop, a gradient descent algorithm.

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Artificial Neural Networks

Deep Nets 1: BackProp and Multilayer Networks

Part 5: The Problem of Overfitting

1. Gradient Descent Methods
2. XOR problem
3. Multilayer Networks: First Steps
4. BackProp Algorithm
5. **The Problem of Overfitting**

Previous slide.

The problem of overfitting is not specific to neural networks but occurs in all cases where a model if fitted to a finite amount of data.

5. The problem of overfitting

Deep networks with multiple layers are flexible and can be trained by BackProp to minimize classification error

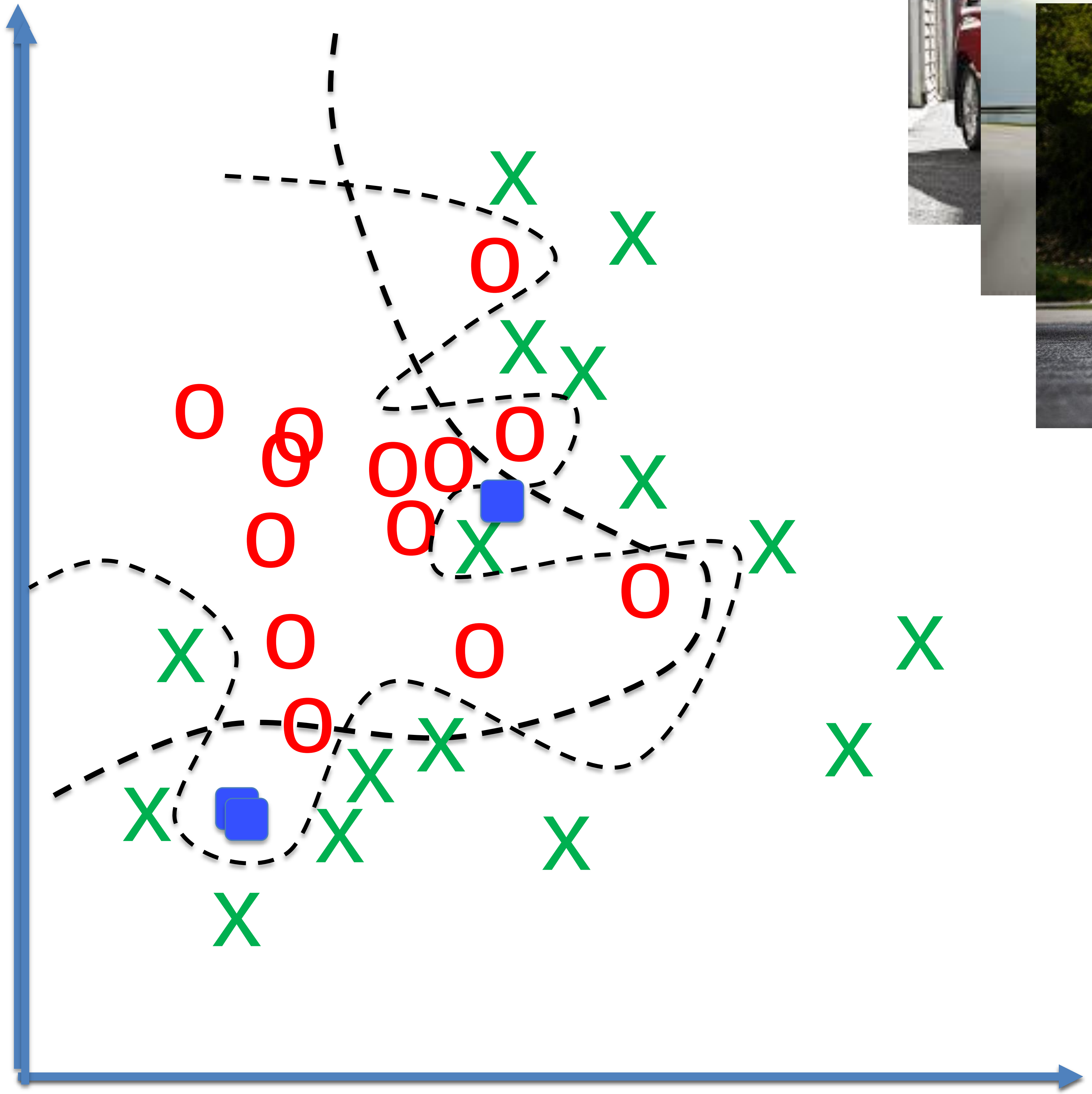
… but is flexibility always good?

Previous slide.
As we have seen, multilayer networks are more powerful than simple perceptrons. They can implement flexible separating surfaces – but is this always good?

The answer is negative – as is well known. In the following a quick repetition of the problem of generalization that is treated in any introductory class to machine learning or data science.

5. Classification of new inputs

→ Flexibility is not good for noisy data



X = ‘car’

o = ‘not car’

more data to learn from

Aim: predict classification for **new** inputs, not seen during training

■ = ‘new image’

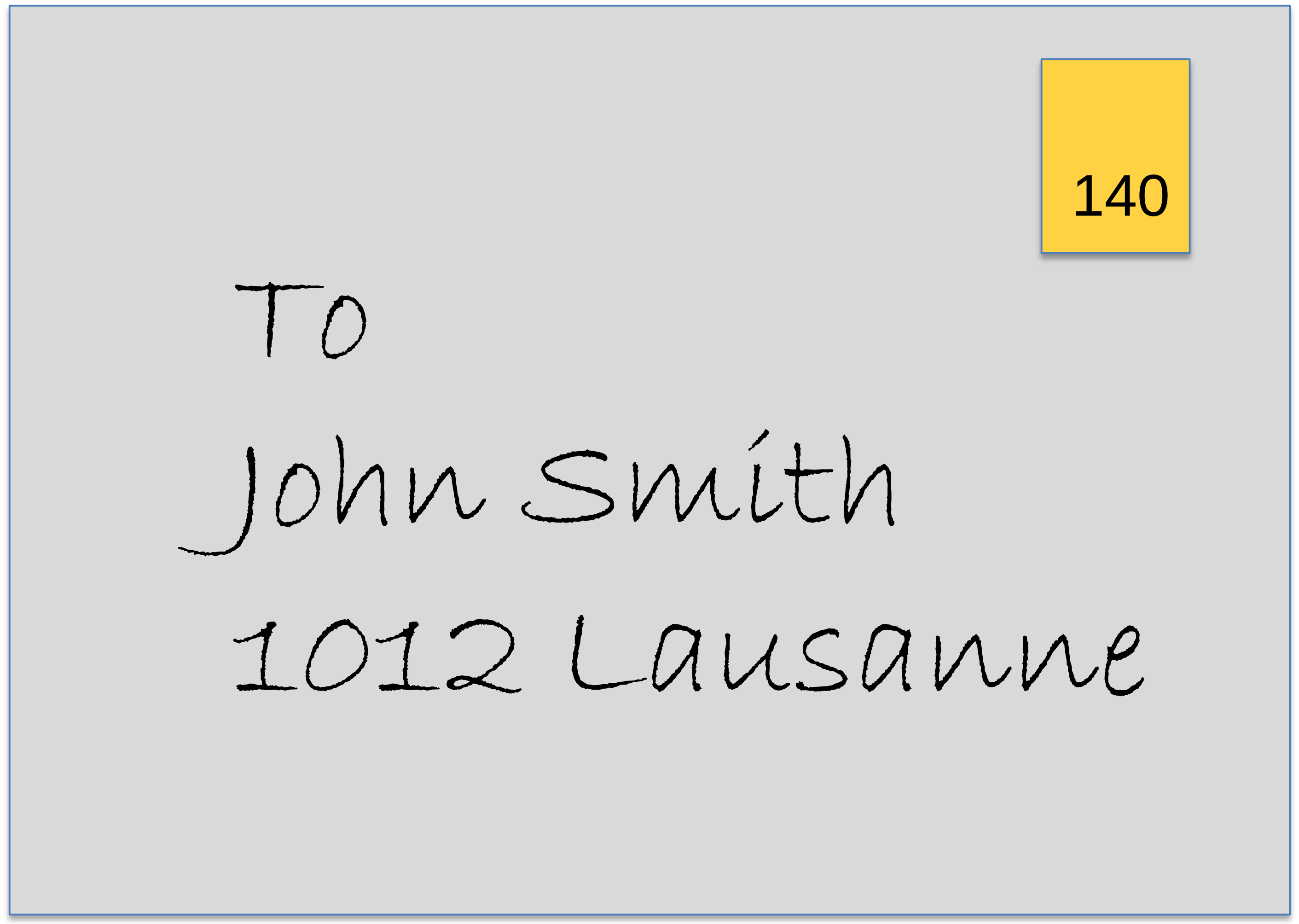
Previous slide.
The aim of training a neural network is always that in the end it should make correct predictions on NEW patterns.

BUT if we are perfect on the data base we may still ail on a new image.

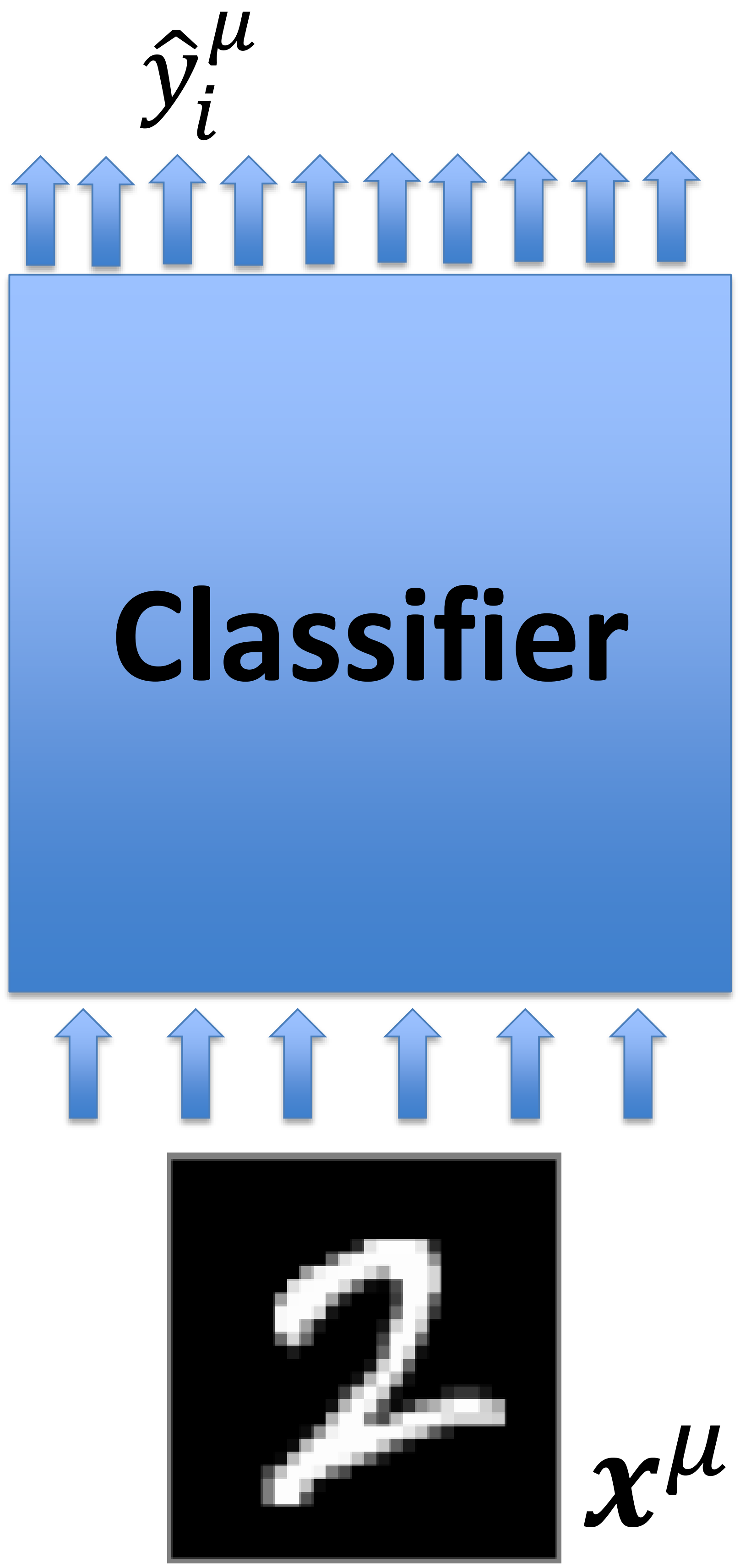
5. Classification of new inputs: Example

10 output units

Task: Read Postal Code



To
John Smith
1012 Lausanne

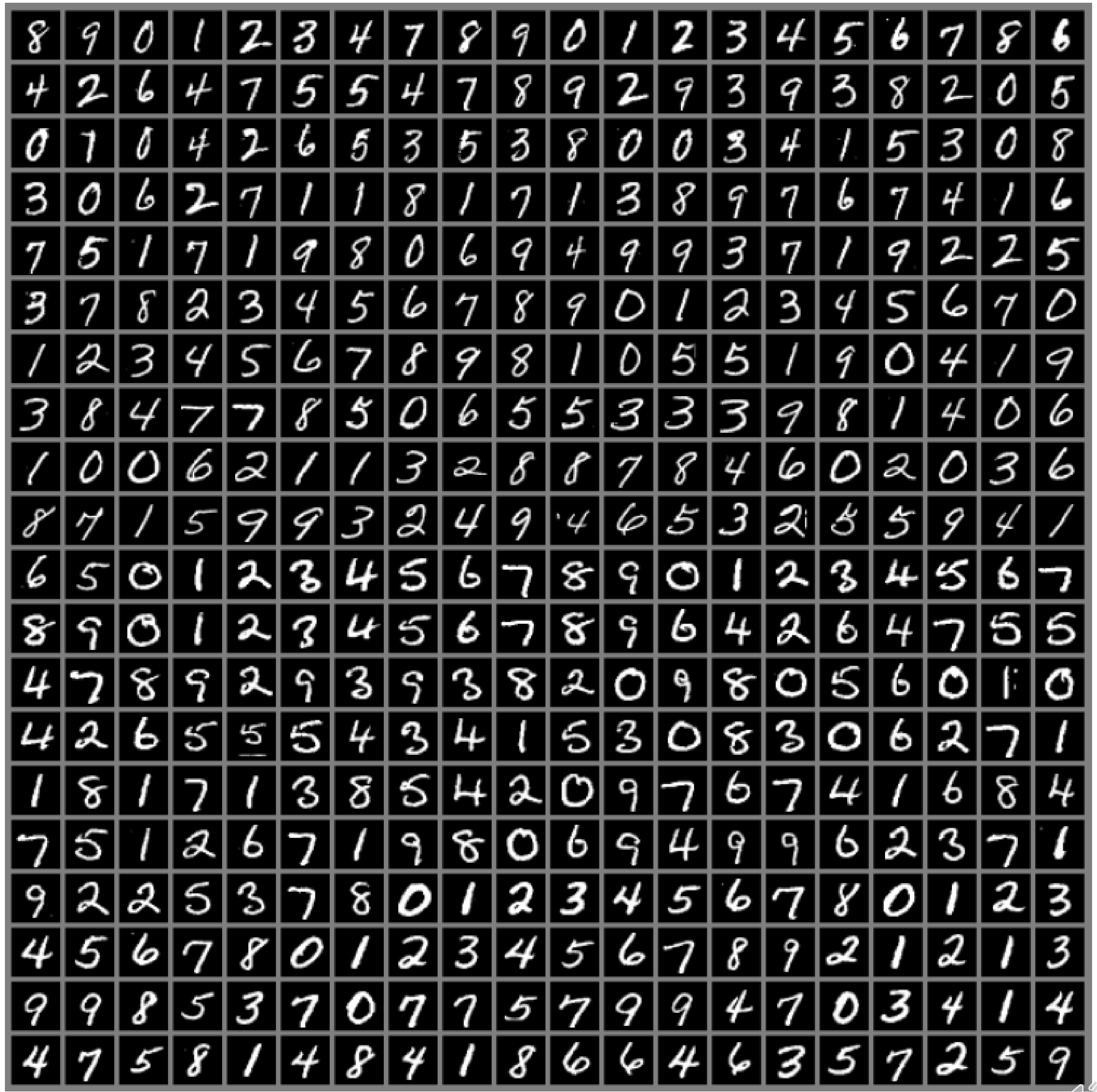


must work on future data!

Previous slide.
A famous example is the automatic recognition of addresses on letters.

5. Classification of new inputs: Example

MNIST data samples



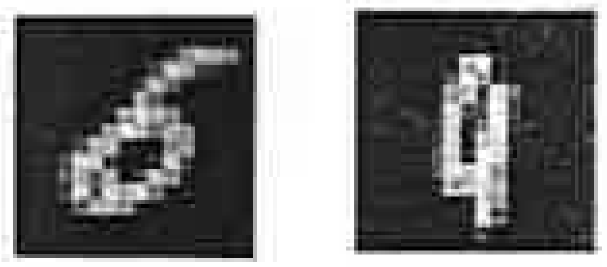
- images 28x28
- Labels: 0, …, 9
- 250 writers
- 60 000 images in training set

Picture: Goodfellow et al, 2016
Data base:
http://yann.lecun.com/exdb/mnist/

Previous slide.
The MNIST data base contains about 60 000 sample images of handwritten digits each one with the correct label.

5. Classification of new inputs: Example

"Is this a 0 or 5 or a 6?"
"Is this a 4 or a 9?"



- training data is always noisy
- the future data has different noise

- Classifier must extract the essence
  → **do not fit the noise!!**

Your notes.

5. The problem of overfitting

Big Multilayer perceptrons are flexible and can be trained by BackProp to minimize classification error

… but is flexibility always good?

→ Flexibility is not good for noisy data
→ Danger of overfitting!
→ Control of overfitting by ‘regularization’

Previous slide.
All data bases are noisy, even MNIST.
But for noisy data we have to be careful to avoid overfitting.

5. Detour: polynomial curve fitting

target data points

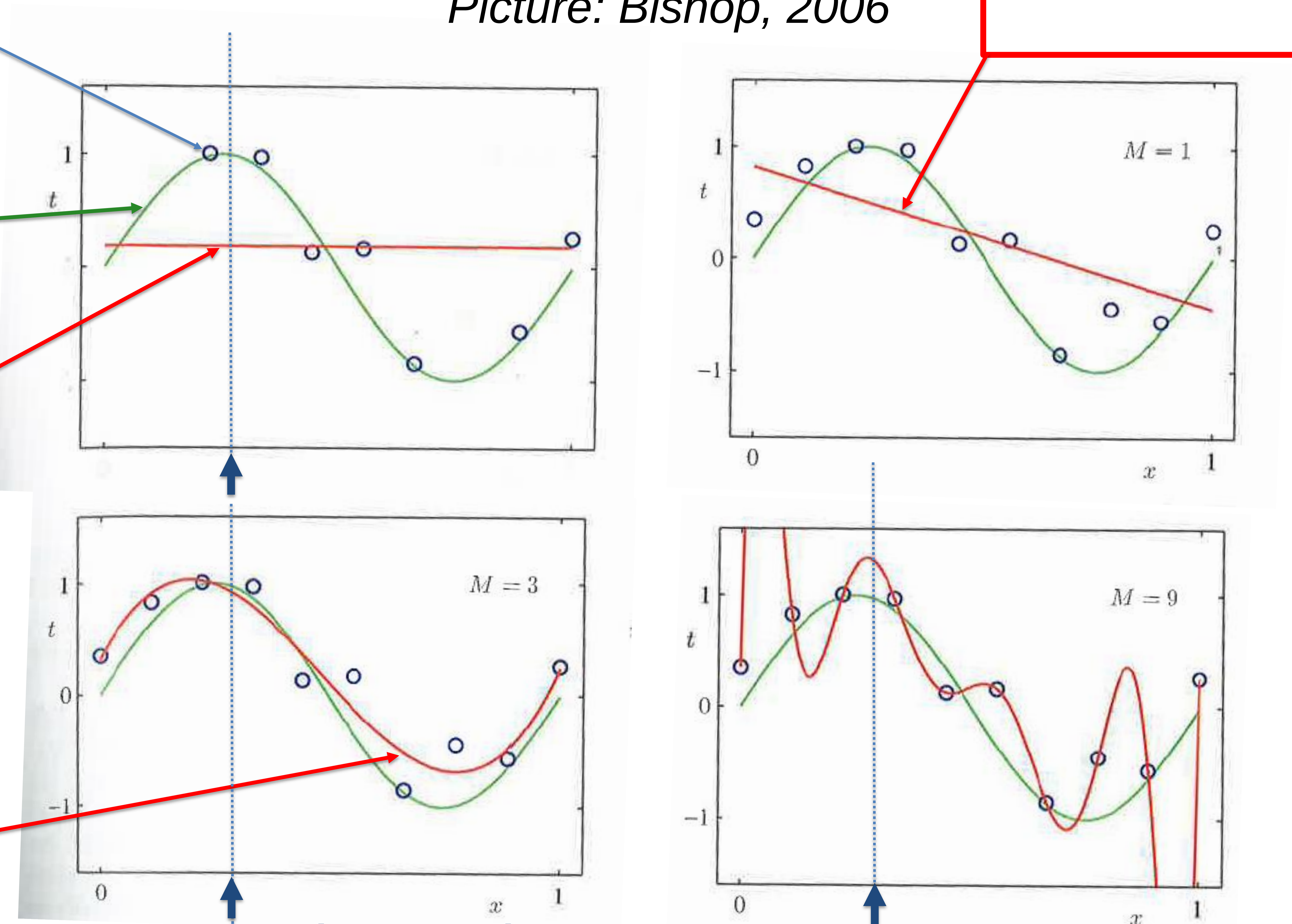
= f(x) + *noise*

f(x) = sin(x)

fit with $y = w_0$

Picture: Bishop, 2006

$y = w_0 + w_1 x$



$y = w_0 + w_1 x \quad + w_2\, x^2$
$+ w_3 x^3$

4 parameters

new data point

$y = w_0 + w_1 x^2 \ldots + w_9\, x^9$

10 parameters

Previous slide.
Polynomials are the standard example to illustrate the problem of overfitting.

10 data points were generated from a sinusoidal function with a small amount of added noise, but we do not know this.

Fitting with 4 free parameters gives a reasonable approximation, whereas a polynomial with 10 terms and 10 free parameters exhibits overfitting.

5. Curve fitting: Quiz

[ ] 20 data points can always be perfectly well fit by a polynomial with 20 parameters

[ ] The prediction for future data is best if the past data is perfectly fit

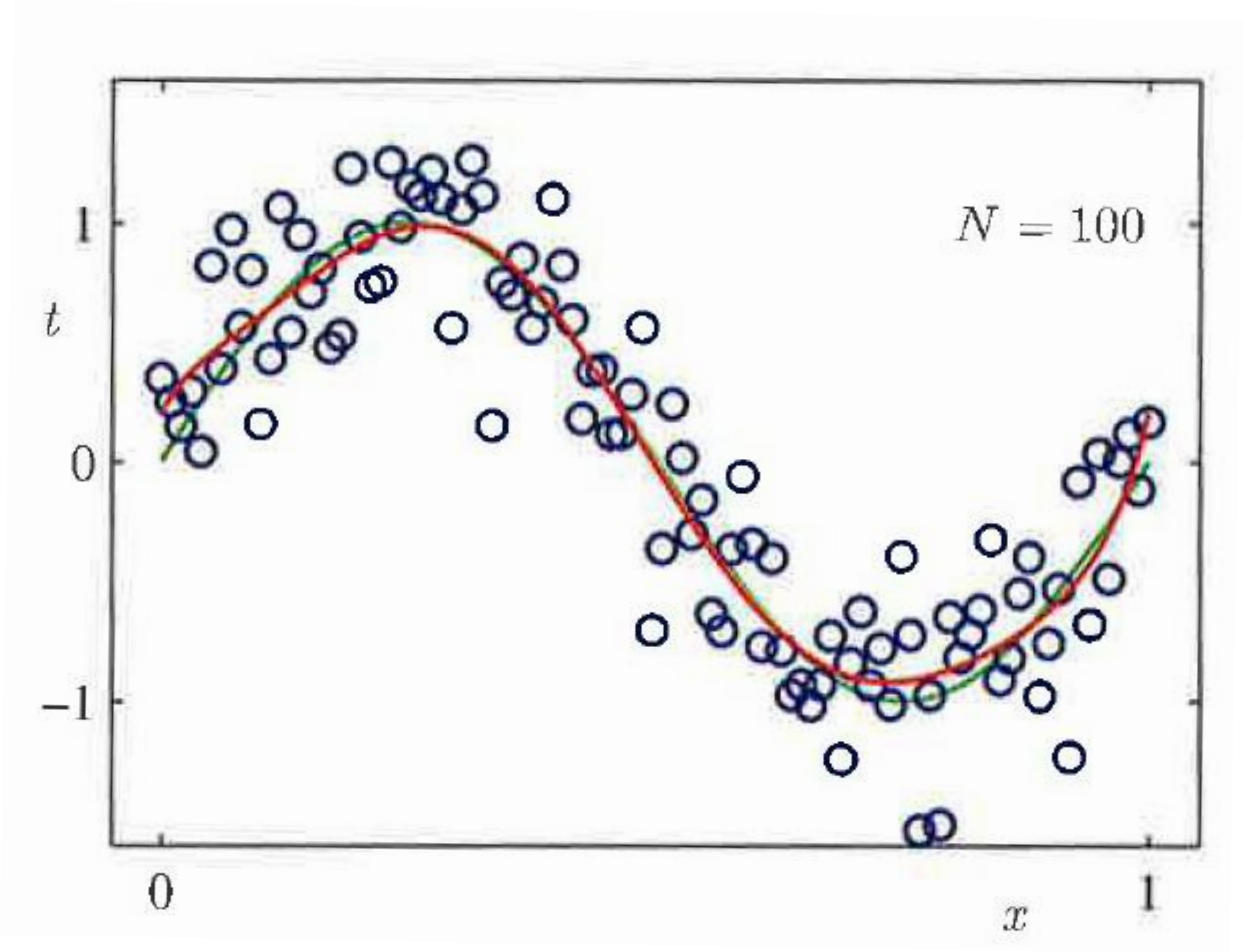
[ ] A sin-function on $[0,2\pi]$ can be well approximated by a polynomial with 10 parameters

Your notes.

5. Detour: polynomial curve fitting

Fit with P=100 data points

If we have enough data points,
10 parameters are not too much!



Picture: Bishop, 2006

$$y = w_0 + w_1 x^2 \ldots + w_9\, x^9$$

10 parameters

Previous slide.

Fitting the sinusoidal with a polynomial with 10 free parameters does work very well, if we have 100 data points.

5. Detour: curve fitting

- flexibility increases with number of parameter
- flexibility is bad for noisy data
- flexibility OK if we have LARGE amounts of data
- for finite amounts of data, we need to control flexibility!

→ See course:
Machine Learning

Previous slide.
Summary:
The flexibility depends on the number of parameters. Flexibility if bad if we have small number of noisy data points but is OK if we have a really large amount of data.
The flexibility can be controlled by the number of parameters or by methods of regularization.

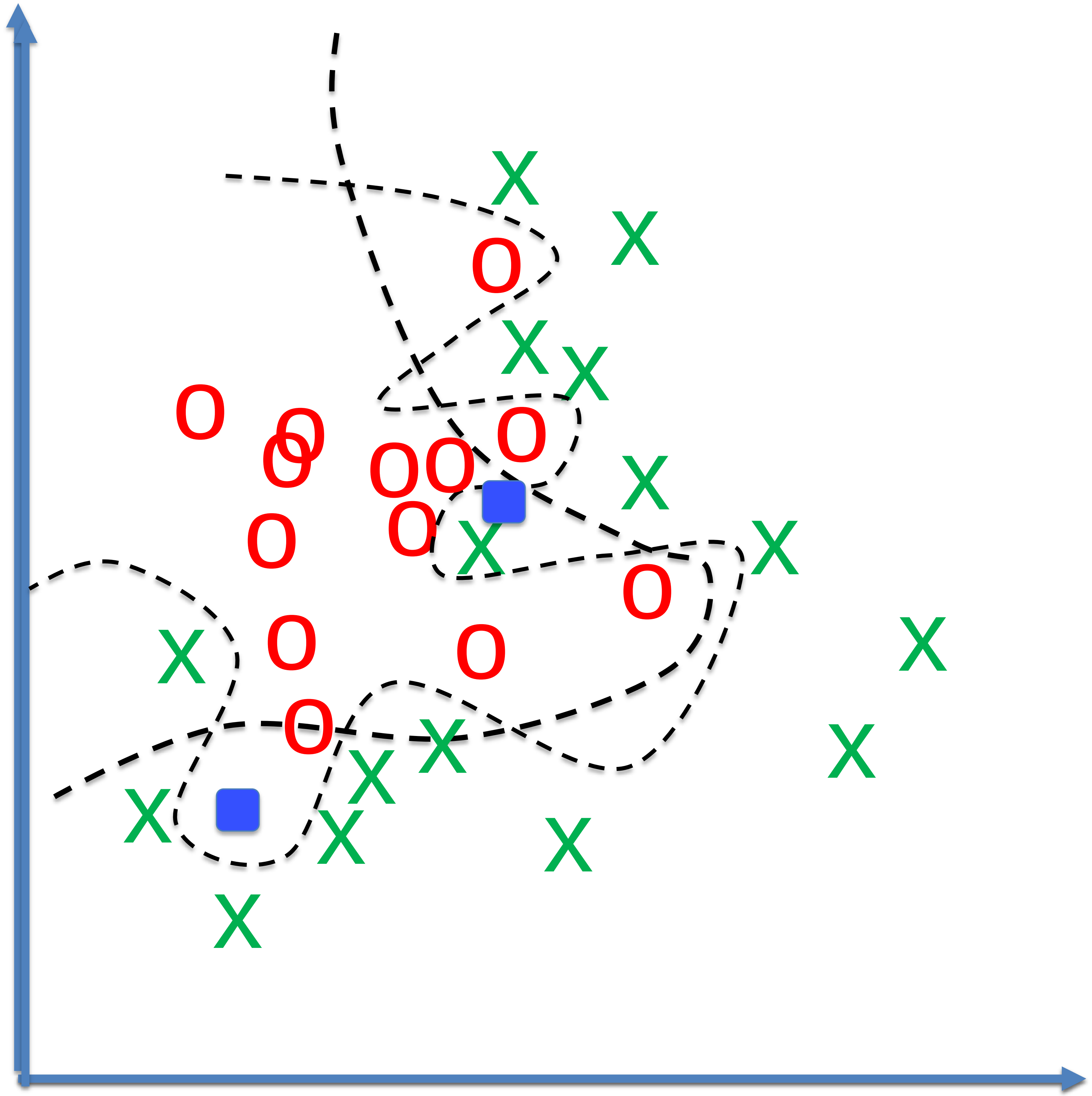
→ See course: *Machine Learning* (Jaggi-Urbanke)

5. The problem of overfitting

Big Multilayer perceptrons are flexible and can be trained by BackProp to minimize classification error

… but is flexibility always good?

→ Flexibility is bad for noisy data
→ Danger of overfitting!
→ Control flexibility!
  ‘Regularization Methods’



Previous slide.

How to control the flexibility is the topic of the next section.

Artificial Neural Networks

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Deep Nets 1: BackProp and Multilayer Networks

Part 6/7: Regularization Methods: Modified Loss Function,
Training Base and Validation Base

1. Gradient Descent Methods
2. XOR problem
3. Multilayer Networks: First Steps
4. BackProp Algorithm
5. The Problem of Overfitting
6. **Simple Regularization Methods:**
   Training Base and Validation Base
7. **Modified Loss Function (Weight Decay)**

Previous slide.

In practice the control of flexibility always requires a split of the data base in two or three subgroups. We start with a split in two groups: training base and validation base.

6. Training base and validation base

Our data base contains

P data points $\{(\boldsymbol{x}^{\mu}, t^{\mu}), \quad 1 \leq \mu \leq P\};$

input target output

Split data base $P = P1 + P2$

$\{(\boldsymbol{x}^{\mu}, t^{\mu}), \quad 1 \leq \mu \leq P1\};$

Training base, used
to optimize parameters

$\{(\boldsymbol{x}^{\mu}, t^{\mu}), \quad P1 + 1 \leq \mu \leq P\};$

Validation base, used
to mimic ‘future data’

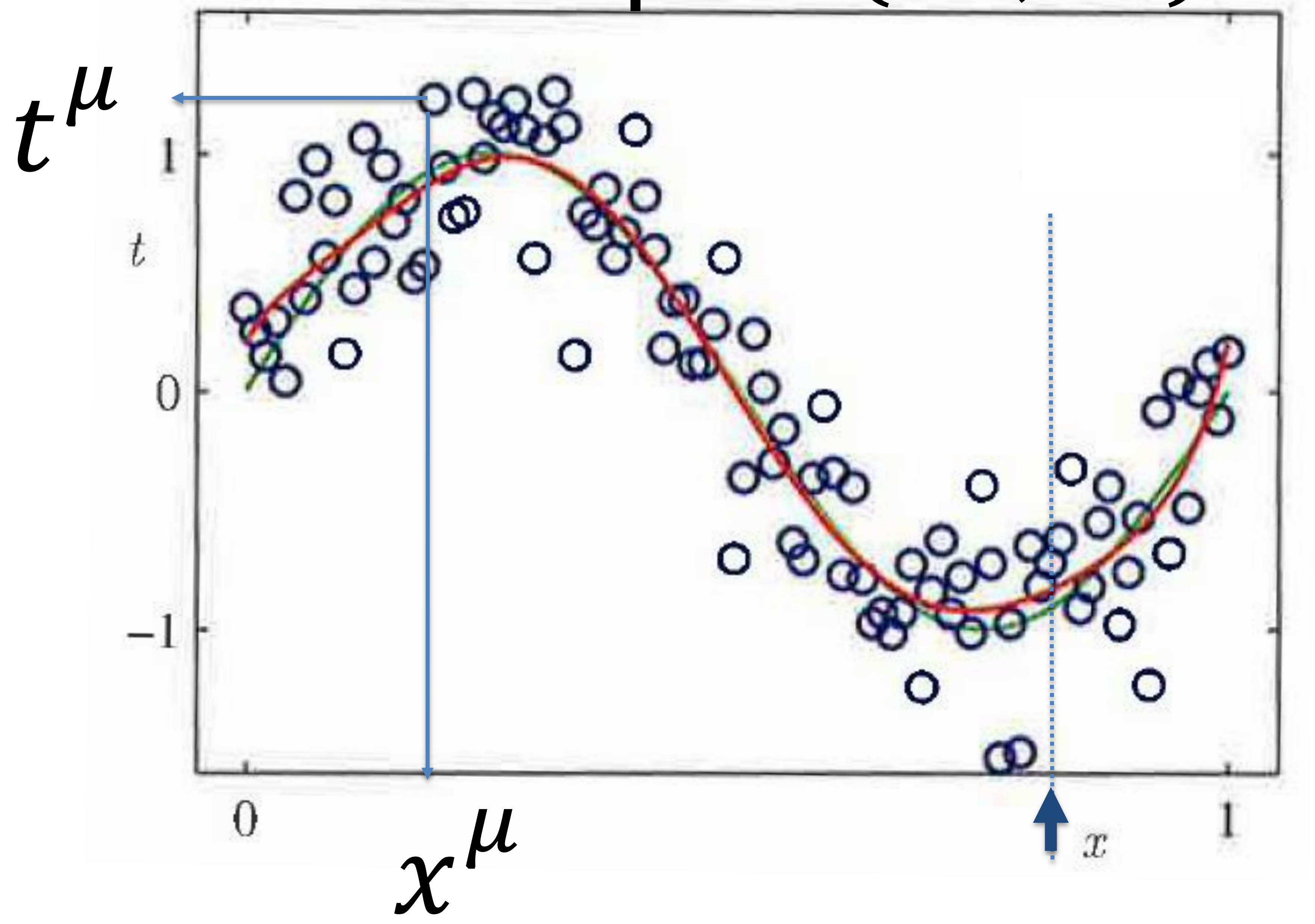
Previous slide.
P1 data points are randomly selected and put into the training base. This data is used to optimize parameters, for example by training the weights via gradient descent.

P2 data points are set apart and put into the validation base. This data plays the role of ‘data in the future’.

The validation set is used to check the performance once training is finished.

6. Error function on training data and validation data

Definition of pair $(\boldsymbol{x}^{\mu}, t^{\mu})$



Minimize error on **training set**

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{\mu=1}^{P1} \left[t^{\mu} - \hat{y}^{\mu}\right]^{2}$$

Evaluate validation error on new data (**validation set**)

$$\overset{\text{val}}{E}(\boldsymbol{w}) = \frac{1}{2}\sum_{\mu=P1+1}^{P} \left[t^{\mu} - \hat{y}^{\mu}\right]^{2}$$

Picture: Bishop, Pattern Recognition and ML, 2006

Previous slide.

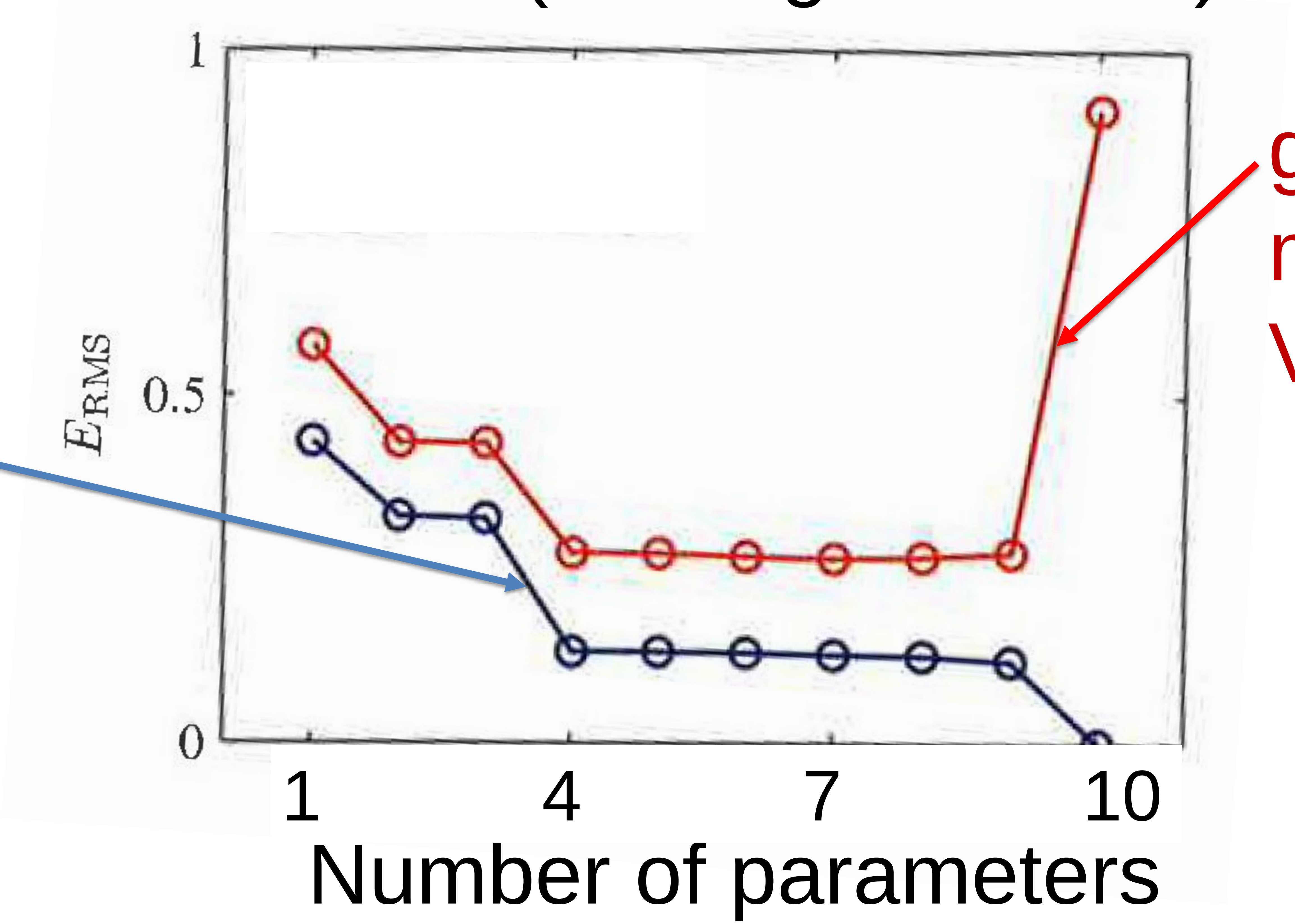
Data in the validation base is used to test the performance (but not to change the weights).
An error on the validation base that is much larger than the error on the training base is a signature of overfitting.

Figure on right-hand-side:
We mainly look at classification tasks, but for regression tasks (continuous outputs) we define target values analogously. Each data point yields a pair $(\boldsymbol{x}^{\mu}, t^{\mu})$.

6. Error function on training data and validation data

Example: polynomial curve fitting with $P1$=10 (training data size)

parameters optimized to minimize training Error E



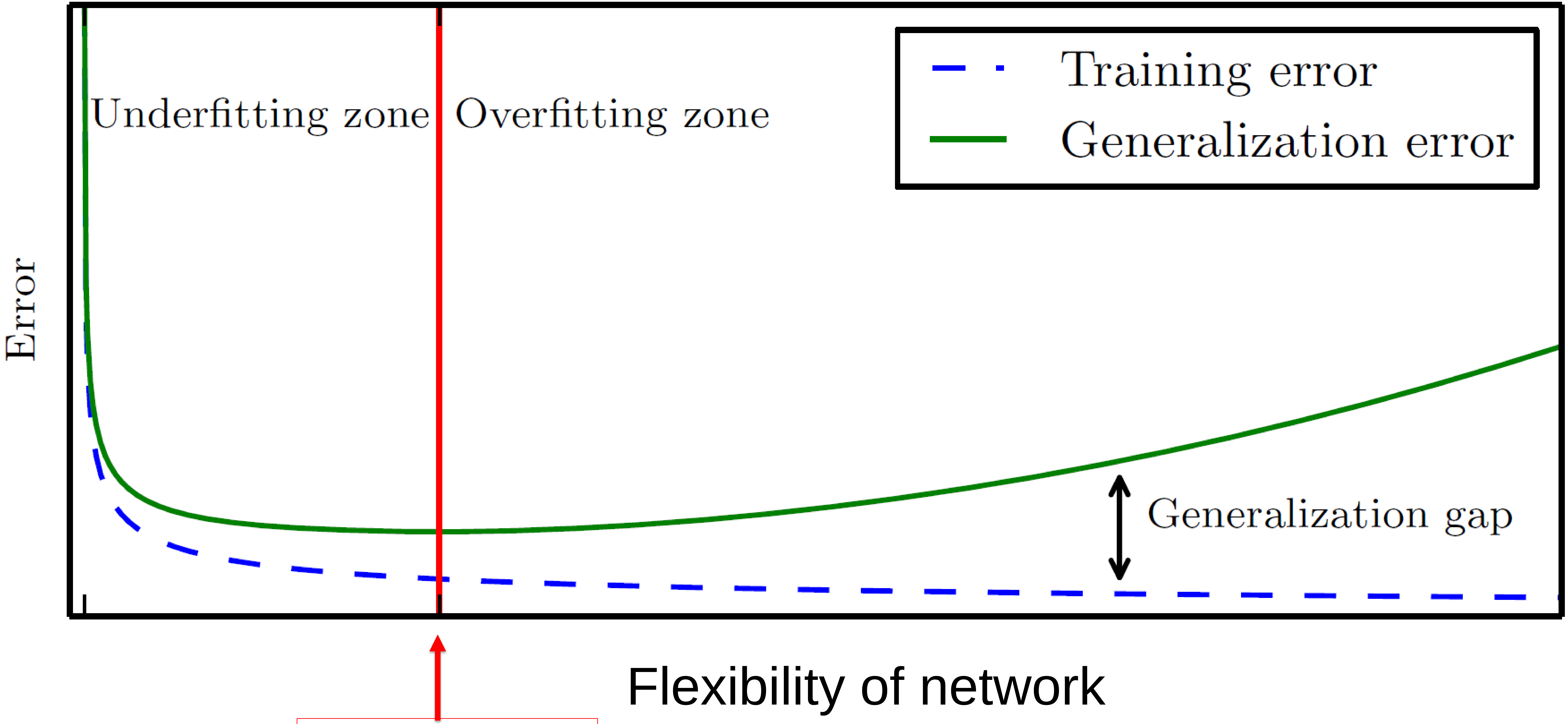
generalization measured as Val. error E^{val}

Picture: Bishop, Pattern Recognition and ML, 2006

Previous slide.

In the example of the 10 data points, polynomials with 3-8 parameters show an error on the validation base that it only slightly larger than the one of the training base; however, a polynomial with 10 parameters clearly exhibits overfitting

6. Error function on training data and validation data



best choice

Picture: Goodfellow et al., Deep Learning, 2016

Previous slide.
More generally, the correct flexibility of the network is the one where the error on the validation set is minimal.

Here flexibility is used as a generic term because flexibility can be controlled not only be the explicit number of parameters but also by early stopping or penalty terms, to be discussed below.

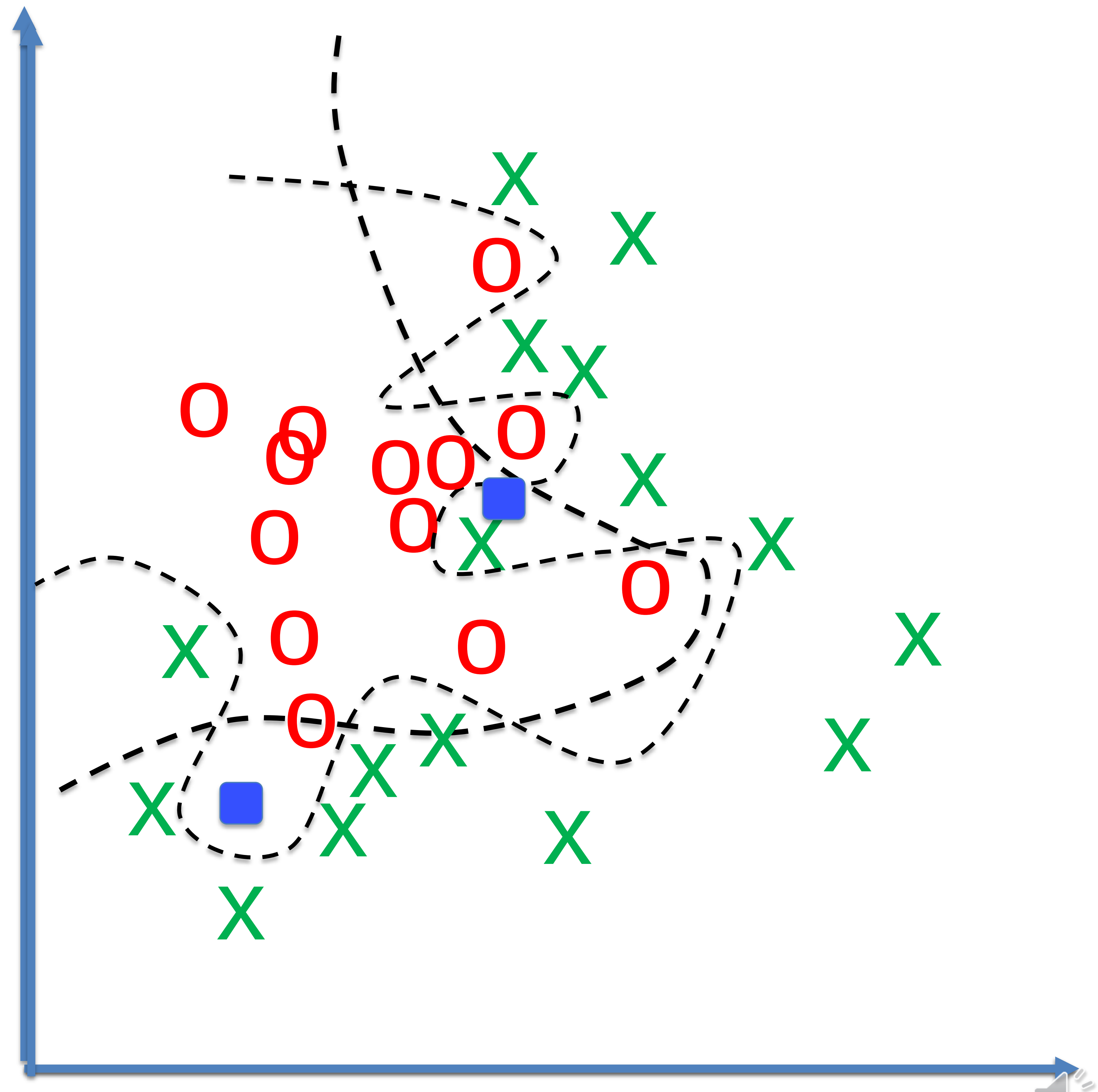
6. The problem of overfitting (revisited)

Deep Multilayer Networks are flexible and can be trained by BackProp to minimize classification error

… but is flexibility always good?

→ Flexibility is bad for noisy data
→ Danger of overfitting!
→ Control flexibility!

We can control overfitting by splitting into training base and validation base



Previous slide.
Always split the data base into training base and validation base.

This holds for curve fitting as well as for generalization as well as for all problems of learning from data.

See also course of Jaggi and Urbanke.

6. Control of flexibility with Artificial Neural networks

```
1 Change flexibility (several times)
  Choose number of hidden neurons and number of layers
    2 Split data base into training base and validation base
        3 Optimize parameters (several times):
          Initialize weights
              4 Iterate until convergence
                Gradient descent on training error


          Report training error and validation error
      Report mean training and validation error and standard dev.
  Plot mean training and validation error
Pick optimal number of layers and hidden neurons
```

Previous slide.
The most obvious way of controlling flexibility is via a control of the number of neurons. Each additional neuron brings several new parameters (the incoming weights).

Important for gradient descent:

- Since the algorithm can get stuck in local minima, you need to start several times with different initial conditions.
- Always run until strict convergence. Sometimes the algorithm seems to improve only very slightly for a long time, but 10 000 steps later there is a further big improvement.

You decide after many runs with different network architecture which one is the best. For future applications you pick the one with the lowest validation error.

Note: indirect methods of controlling the flexibility will be discussed further below in the next section.

Objectives for today:

- XOR problem and the need for multiple layers
  - → hidden layer provide flexibility
- understand backprop as a smart algorithmic implementation of the chain rule
  - → algorithmic differentiation is better than numeric differentiation
- hidden neurons add flexibility, but flexibility is not always good
  - → control flexibility: use validation data

Today exercises: XOR with Keras simulator/tutorial

Your notes.

Artificial Neural Networks

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Deep Nets 1: BackProp and Multilayer Networks

Part 7: Simple Regularization: Modified Loss Function

1. Gradient Descent Methods
2. XOR problem
3. Multilayer Networks: First Steps
4. BackProp Algorithm
5. The Problem of Overfitting
6. Training Base and Validation base
7. **Simple Regularization: Modified Loss**

Previous slide.

Regularization is an alternative of explicit changes of the number of neurons in the network.

7. Controling Flexibility

Flexibility = number of free parameters

→ Change flexibility = change network structure or number of hidden neurons

Flexibility = ‘**effective**’ number of free parameters

→ Change flexibility = regularization of network

Previous slide.

Regularization controls the flexibility without changing the explicit number of free parameters.

7. Regularization by a penalty term: Modified Loss Function

Minimize on **training set a modified Error function**

$$\tilde{E}(\boldsymbol{w}) = \frac{1}{2}\sum_{\mu=1}^{P1} \left[t^{\mu} - \hat{y}^{\mu}\right]^2 + \lambda \text{ penalty}$$

assigns a 'cost'
to flexible solutions

check 'normal' error on separate data (**validation set**)

$$E^{\text{val}}(\boldsymbol{w}) = \frac{1}{2}\sum_{\mu=P1+1}^{P} \left[t^{\mu} - \hat{y}^{\mu}\right]^2$$

Previous slide.

Important:

While validation on the validation set is performed using the NORMAL error function, training is done on the training set using an error function that includes a penalty term. The penalty term penalizes ‘flexible’ solutions.

7. Regularization by a weight decay (L2 regularization)

Specific example: L2-regularization

Minimize on **training set a modified Error function**

$$\tilde{E}(\boldsymbol{w}) = \frac{1}{2}\sum_{\mu=1}^{P1} \left[t^{\mu} - \hat{y}^{\mu}\right]^2 \quad + \lambda \sum_{k} \underset{\uparrow}{(w_k)^2}$$

assigns an 'error' to solutions with large pos. or neg. weights

check 'normal' error on separate data (**validation set**)

$$E^{\text{val}}(\boldsymbol{w}) = \frac{1}{2}\sum_{\mu=P1+1}^{P} \left[t^{\mu} - \hat{y}^{\mu}\right]^2$$

Previous slide.
A simple example is to assign a penalty to networks that have many weights with a large absolute value (L1 regularization) or absolute values squared (L2 regularization). The logic is that ‘curvy’ separating surface requires big positive and negative weights, whereas zero weights or tiny weights enable no or very litte curvature only.

The sum in the penalty term runs over all weights, but not the thresholds!!!.

The terminology ‘weight decay’ arises from the update rule of stochastic gradient descent. Just take the derivative!

7. Regularization: Quiz

If we **increase the penalty parameter** λ

[ ] the flexibility of the fitting procedure **decreases**

[ ] the ‘**effective**’ number of free parameters **decreases**

[ ] the **‘explicit’** number of parameters **remains the same**

Your notes.

7. Regularization by a weight decay: curve fitting

Curve fitting, 10 data points, 10 parameters (as before)

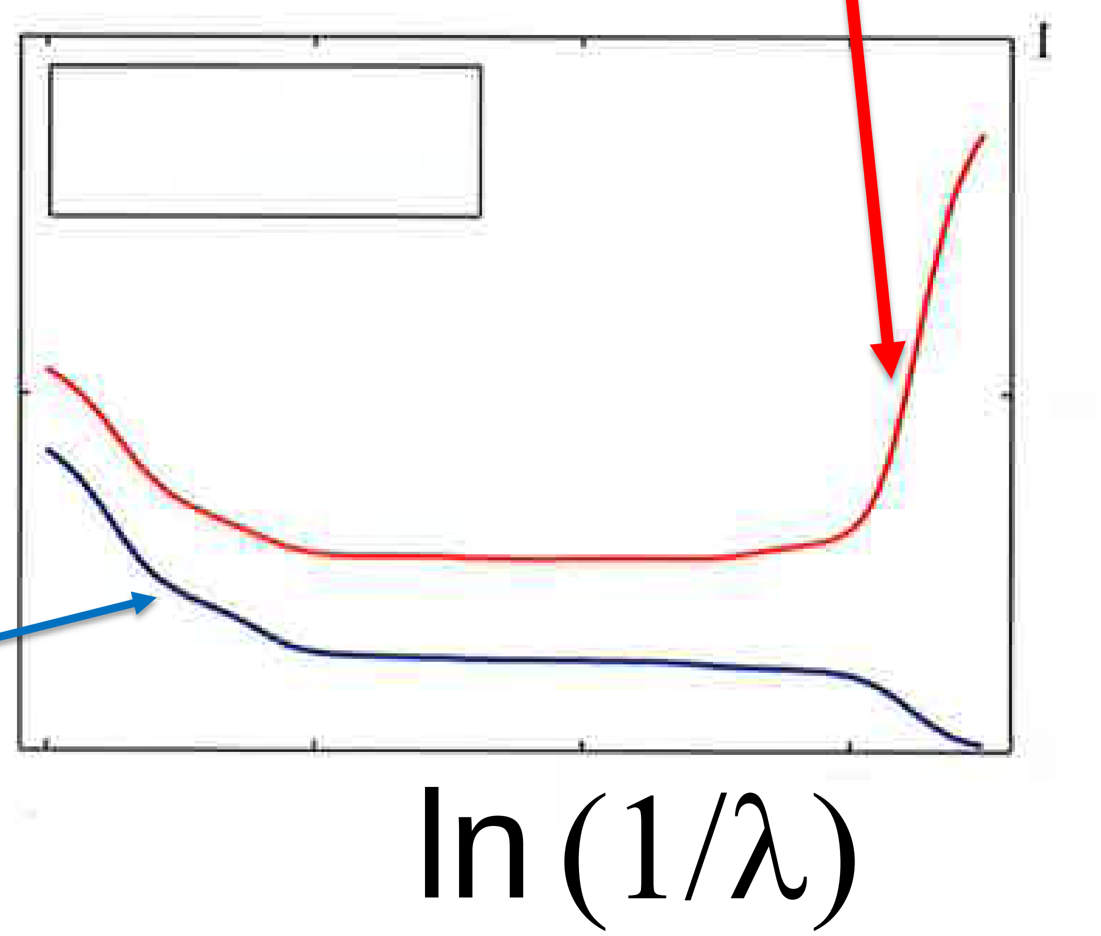
Minimize on **training set a modified Error function**

$$\tilde{E}(\boldsymbol{w}) = \frac{1}{2}\sum_{\mu=1}^{P1} \left[t^{\mu} - \hat{y}^{\mu}\right]^{2} + \lambda \sum_{k} (w_{k})^{2}$$

plot 'normal' error for both data sets

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{\mu=P1+1}^{P} \left[t^{\mu} - \hat{y}^{\mu}\right]^{2}$$

If we decrease λ , validation error increases (overfitting)



Training error

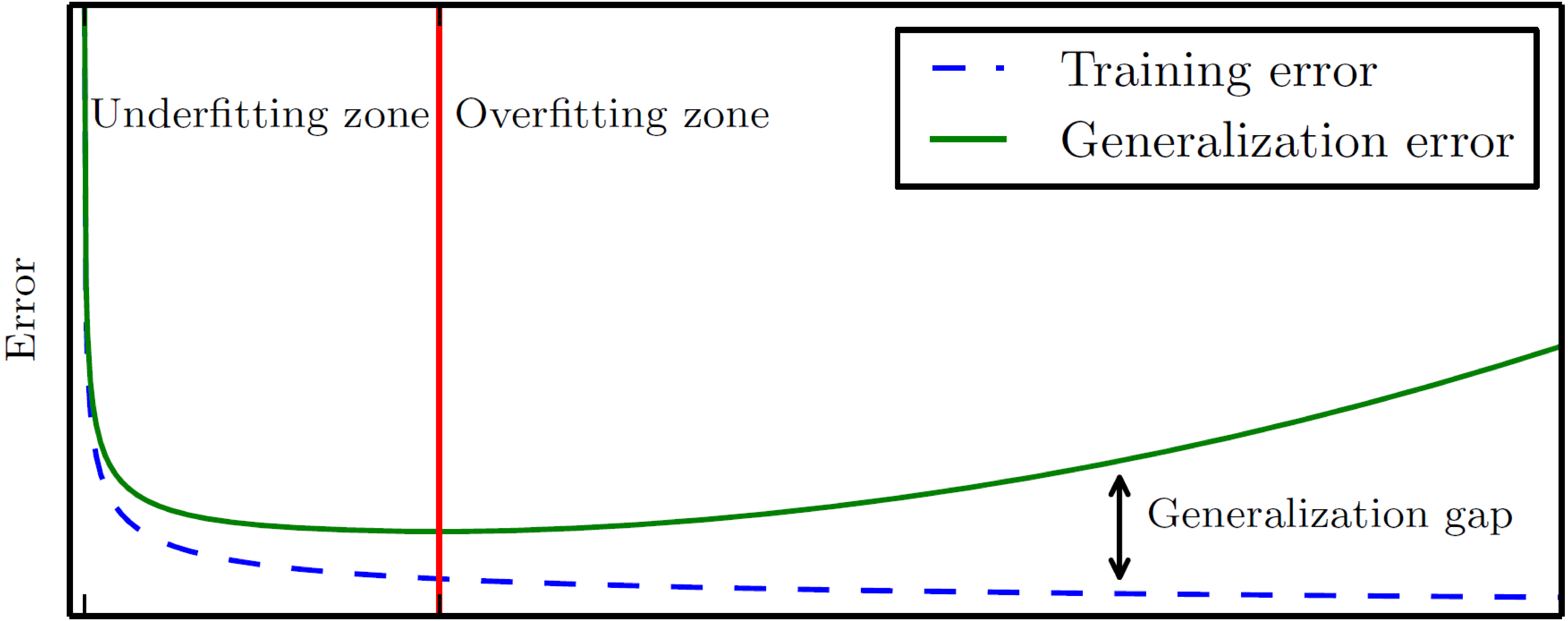
decreasing λ → increasing flexibility →

Picture: Bishop, Pattern Classification and ML, 2006

Previous slide.
In this example there are always 10 free parameters; however, the flexibility is controlled by the penalty term. A big penalty term (toward the left of the graph) implies a unflexible network.
A penalty term close to zero implies a very flexible network prone to overfitting.

Note: We use the neural network terminology and refer to the parameters of the polynomial now as ‘weights’.

6. (repeated): Error function on training data and validation



Flexibility of network

best choice

Picture: Goodfellow et al., 2016

Previous slide.

Same slide as earlier, but now flexibility would be controlled by the penalty term. Big penalty/small flexibility is on the left; small penalty/large flexibility on the right.

7. Regularization

Conclusion:

- we can keep the real number of parameters fixed and large, and still change flexibility via λ

  → Standard Method in all fields of Machine Learning

Application to Artificial Neural Networks:

- we can work with fixed (large) number of hidden neurons and fixed (deep) network structure and control flexibility via regularization

Previous slide.

The advantage of working with a penalty term is that we can always work with a big network of fixed size and then control the flexibility by a single parameter lambda.

→ See course: *Machine Learning*

7. Control of flexibility by regularizer

```
1 Change flexibility (several times)
  Choose λ
    2 Split data base into training set and validation set
        3 Optimize parameters (several times):
          Initialize weights
            4 Iterate until convergence
              Gradient descent on modified
               training error $\tilde{E}(\boldsymbol{w})$
          Report training error $\boldsymbol{E}$ and test error $\boldsymbol{E}^{\text{val}}$ on validation set
      Report mean training and test error and SD
  Plot mean training and test error
Pick weights for results with optimal λ (lowest validation error)
```

Previous slide.
Schema analogous to the earlier one. However, we no longer need to change the network architecture or number of neurons but just a single parameter lambda.

7. Control of flexibility by regularizer

- weights are parameters
- λ is also a parameter (hyperparameter)

BUT ATTENTION:

→ Test set/validation set is no longer ‘future data’ because we have used it to optimize λ

If you have enough data, and many hyperparameters, use a double-split and keep some data aside to mimic ‘future data’.

Previous slide.
There could be a potential problem: we added lambda as one of the parameters (sometimes called a hyper-parameter). To optimize lambda we use the validation base. But this logically implies that we can not consider the validation based as ‘future data’!

In fact, the same logic also applies to the earlier scheme where we changed the explicit number of neurons - the neuron number is also a hyperparameter which is optimized by exploiting the validation base.

Artificial Neural Networks

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Deep Nets 1: Backprop and multilayer networks

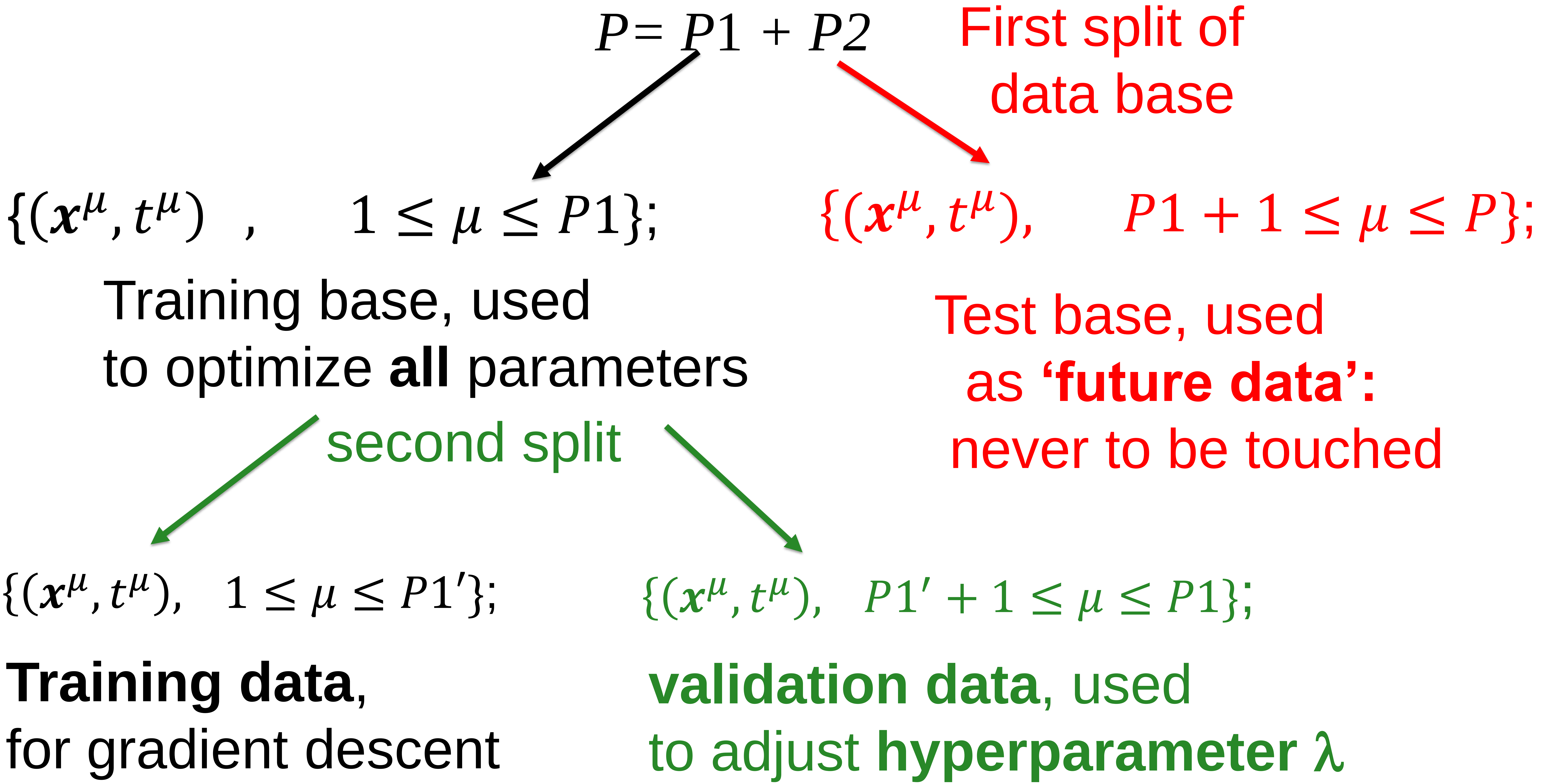
Part 8/9: Careful Cross-validation and Early Stopping

1. Gradient Descent Methods
2. XOR problem
3. Multilayer Networks: First Steps
4. BackProp Algorithm
5. The Problem of Overfitting
6. Training Base and Validation base
7. Simple Regularization
8. **Careful Cross-validation**
9. **Regularization by Early Stopping**

Previous slide.

To solve the problem of the optimization of hyperparameters some researchers suggest a more careful method of cross-validation.

8. Training base, validation base, test base



$P = P1 + P2$

First split of data base

$\{(\boldsymbol{x}^{\mu}, t^{\mu}) \; , \quad 1 \leq \mu \leq P1\};$

Training base, used to optimize **all** parameters

$\{(\boldsymbol{x}^{\mu}, t^{\mu}), \quad P1 + 1 \leq \mu \leq P\};$

Test base, used as ‘**future data**’: never to be touched

second split

$\{(\boldsymbol{x}^{\mu}, t^{\mu}), \;\; 1 \leq \mu \leq P1'\};$

Training data, for gradient descent

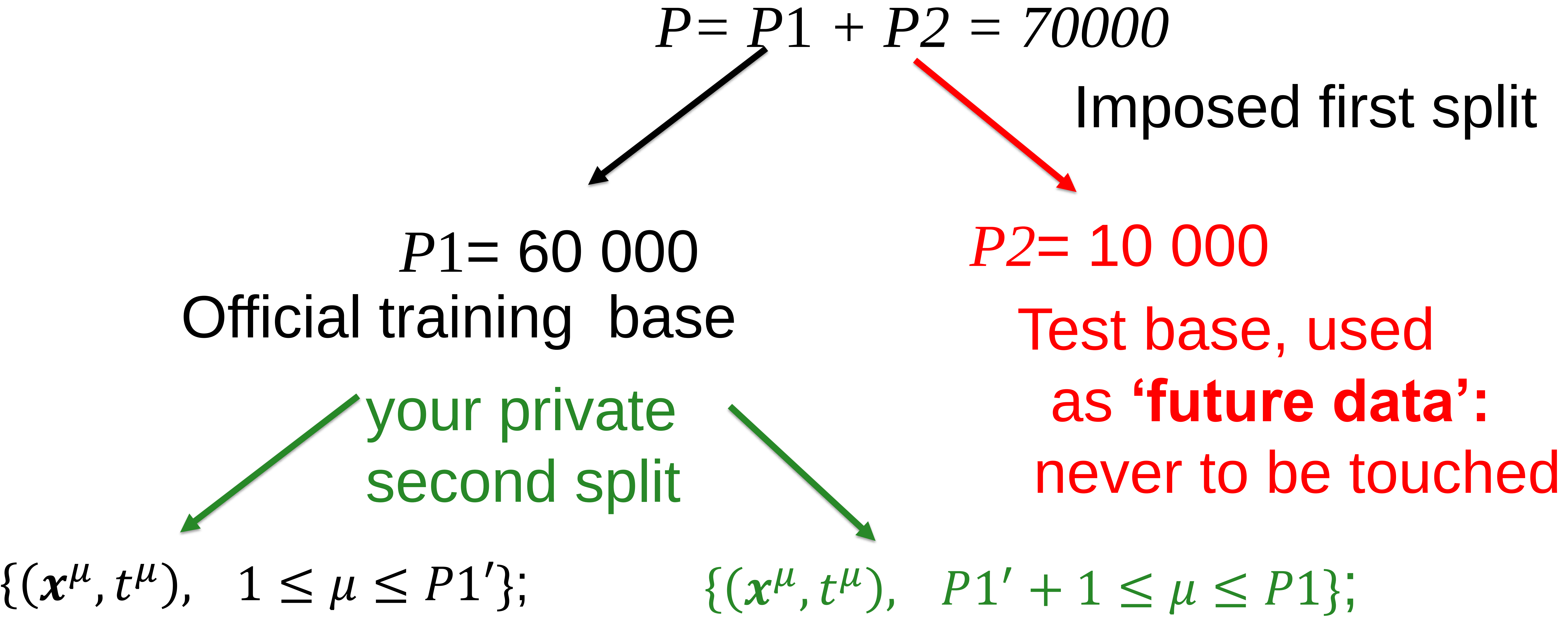
$\{(\boldsymbol{x}^{\mu}, t^{\mu}), \;\; P1' + 1 \leq \mu \leq P1\};$

validation data, used to adjust **hyperparameter** λ

Previous slide.
We first split the data base into two parts:
A training base used to optimize ALL parameters
And a test base that is NEVER to be touched.

The Training base is then further split into the training data (in the narrow sense) and the validation data. The validation data is used to adjust the hyperparameter lambda (searching for the lowest error on the validation set) while the training data is used for gradient descent (on the augmented error function including the penalty term).

8. Example: MNIST Training base, validation base, test base



$P = P1 + P2 = 70000$

Imposed first split

$P1$= 60 000

Official training base

$P2$= 10 000

Test base, used as ‘**future data**’: never to be touched

your private second split

$\{(\boldsymbol{x}^{\mu}, t^{\mu}), \quad 1 \le \mu \le P1'\};$

$\{(\boldsymbol{x}^{\mu}, t^{\mu}), \quad P1' + 1 \le \mu \le P1\};$

e.g., P1’=50 000

Training data,
for gradient descent

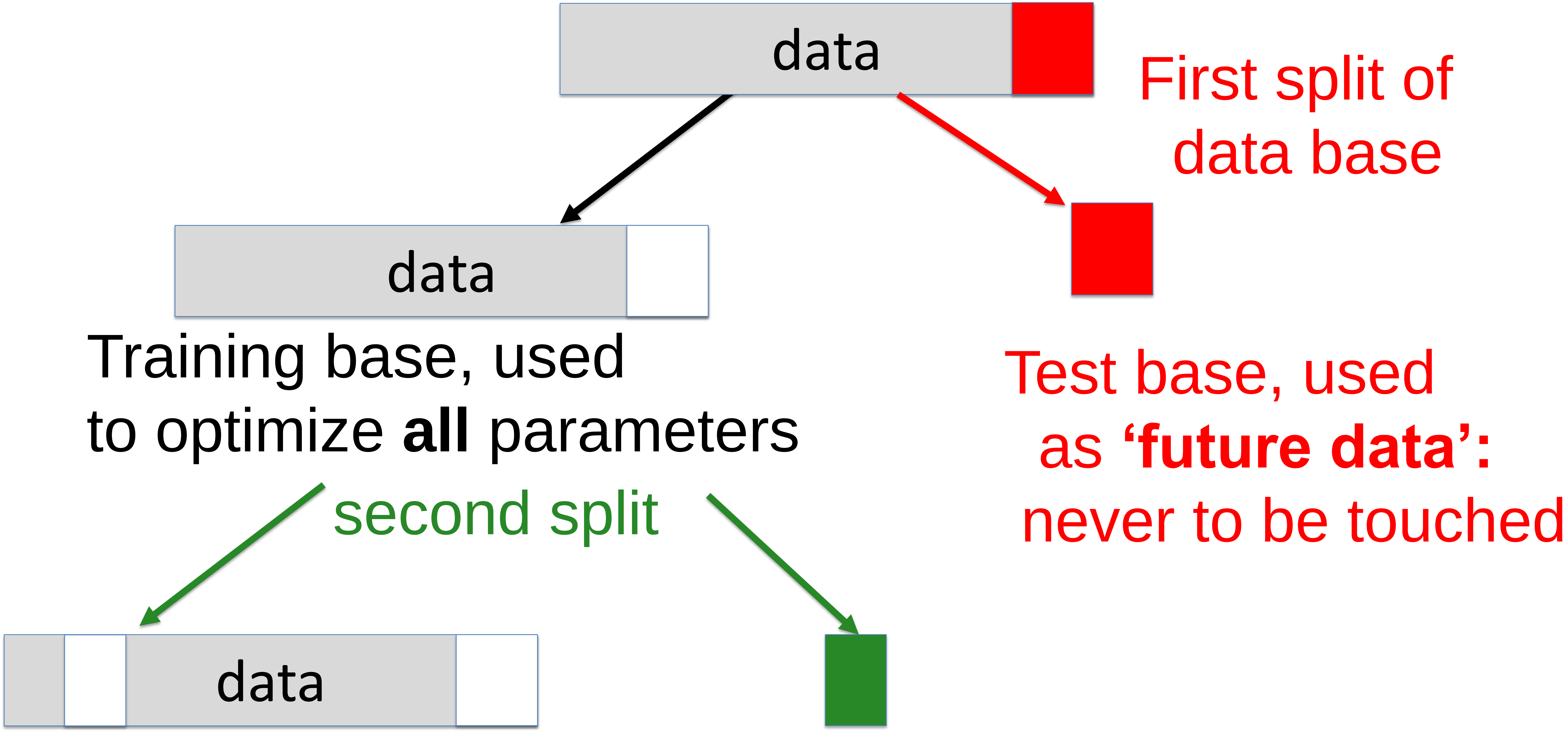
validation data, used
to adjust **hyperparameter** λ

http://yann.lecun.com/exdb/mnist/

Previous slide.
For example, for MNIST a first split is suggested by the designer of the data set who put some data apart to play the role as ‘future data’.

The second split is done by the user.

8. Training base, validation base, test base



First split of data base

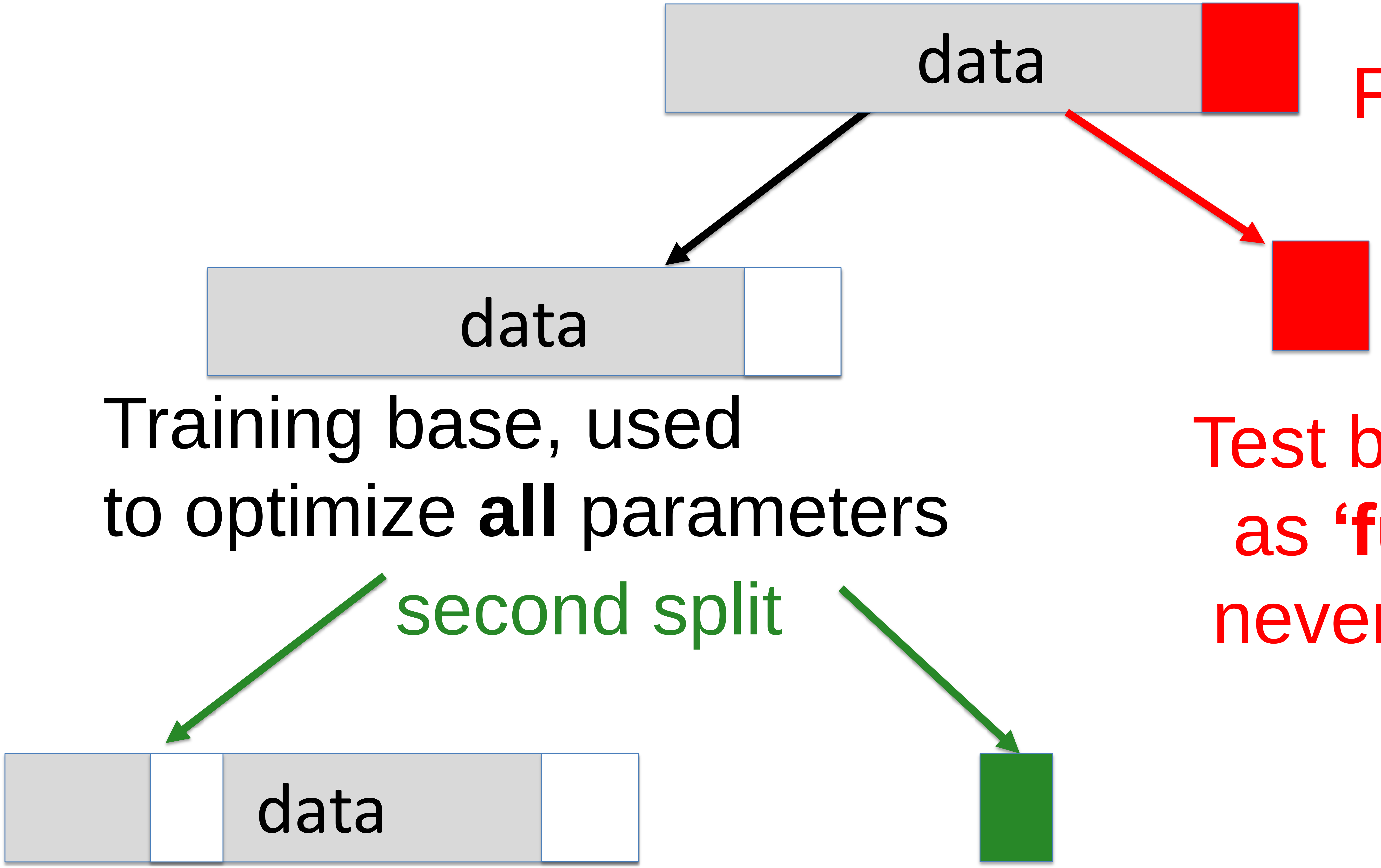
Training base, used to optimize **all** parameters

Test base, used as ‘**future data**’: never to be touched

second split

Previous slide.
For the second split it is recommended to use k-fold cross validation:
You only put a fraction 1/k into the validation set.
You train and validate the usual way.
You write down the result for training and validation.

8. k-fold cross-validation



First split of data base

Training base, used to optimize **all** parameters

Test base, used as ‘**future data**’: never to be touched

second split

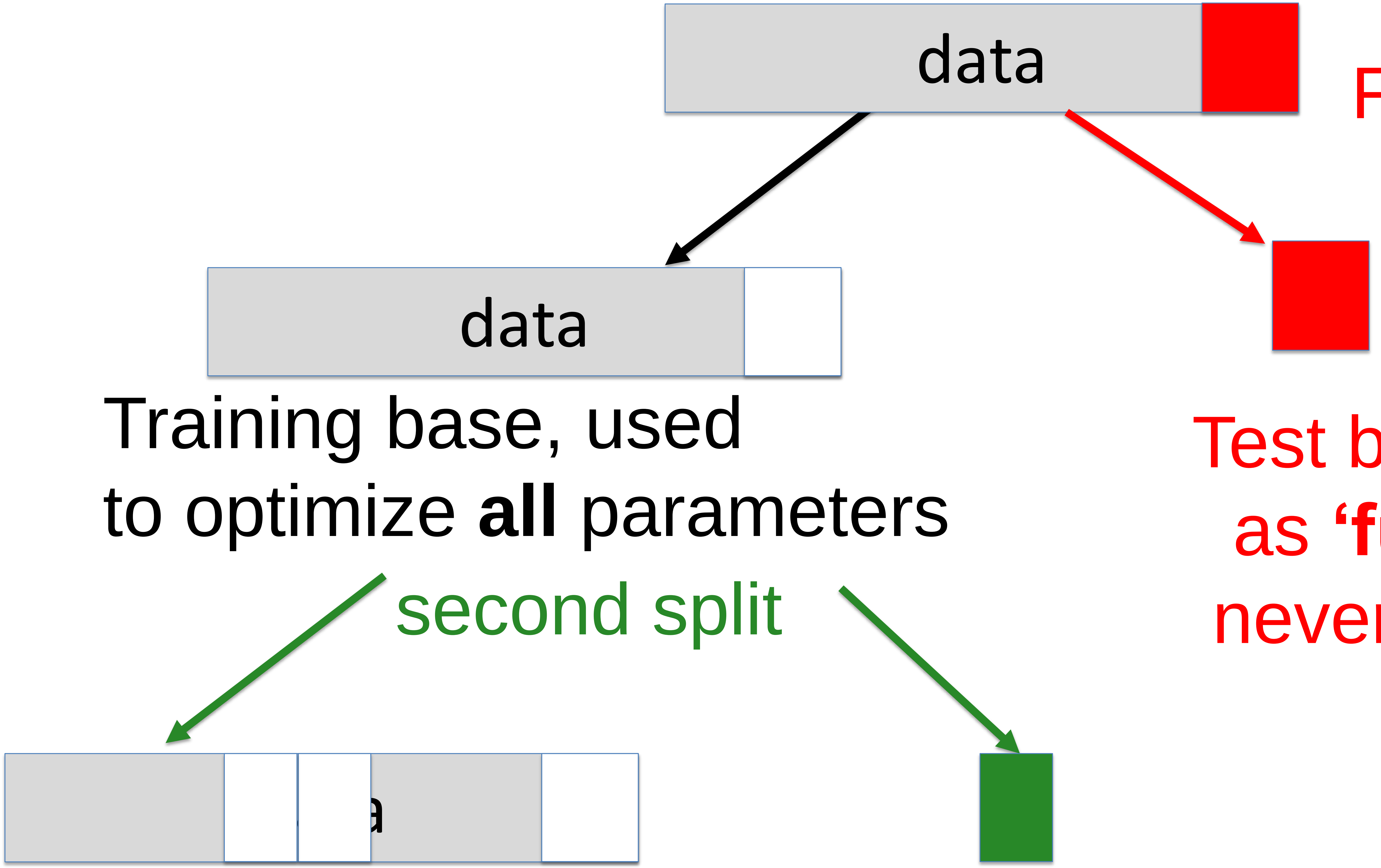
Repeat this second split k times

Previous slide.

Then you repeat this with a second split (another 1/k of the data as validation set).
You re-initialize, retrain, and validate, and not the results.

You repeat this k times.

8. k-fold cross-validation



First split of data base

Training base, used to optimize **all** parameters

Test base, used as ‘**future data**’: never to be touched

second split

Previous slide.
In the end, you average over all k ‘folds’.
Repeat for different values of the hyperparameter lambda.

Pick lambda which minimizes the validation error.

If you want you can now retrain with this specific lambda on the full training set.
Since you have ‘regularized’ with the appropriate lambda, overtraining should not happen.

The final value to report is the performance on the test base. Importantly, once you have touched the test base the game is over. You are not allowed to retrain.

In practice, people rarely put aside a test base. Careful k-fold cross-validation is accepted as a performance measure.

Artificial Neural Networks

Wulfram Gerstner

EPFL, Lausanne, Switzerland

Deep Nets 1: Backprop and multilayer networks

Part 9: Regularization by Early Stopping

1. Gradient Descent Methods
2. XOR problem
3. Multilayer Networks: First Steps
4. BackProp Algorithm
5. The Problem of Overfitting
6. Training Base and Validation base
7. Simple Regularization
8. Careful Cross-validation
9. **Regularization by Early Stopping**

Previous slide.

Early stopping is a surprisingly simple regularization method. It works well, is easy to implement, and avoids a formal hyperparameter.

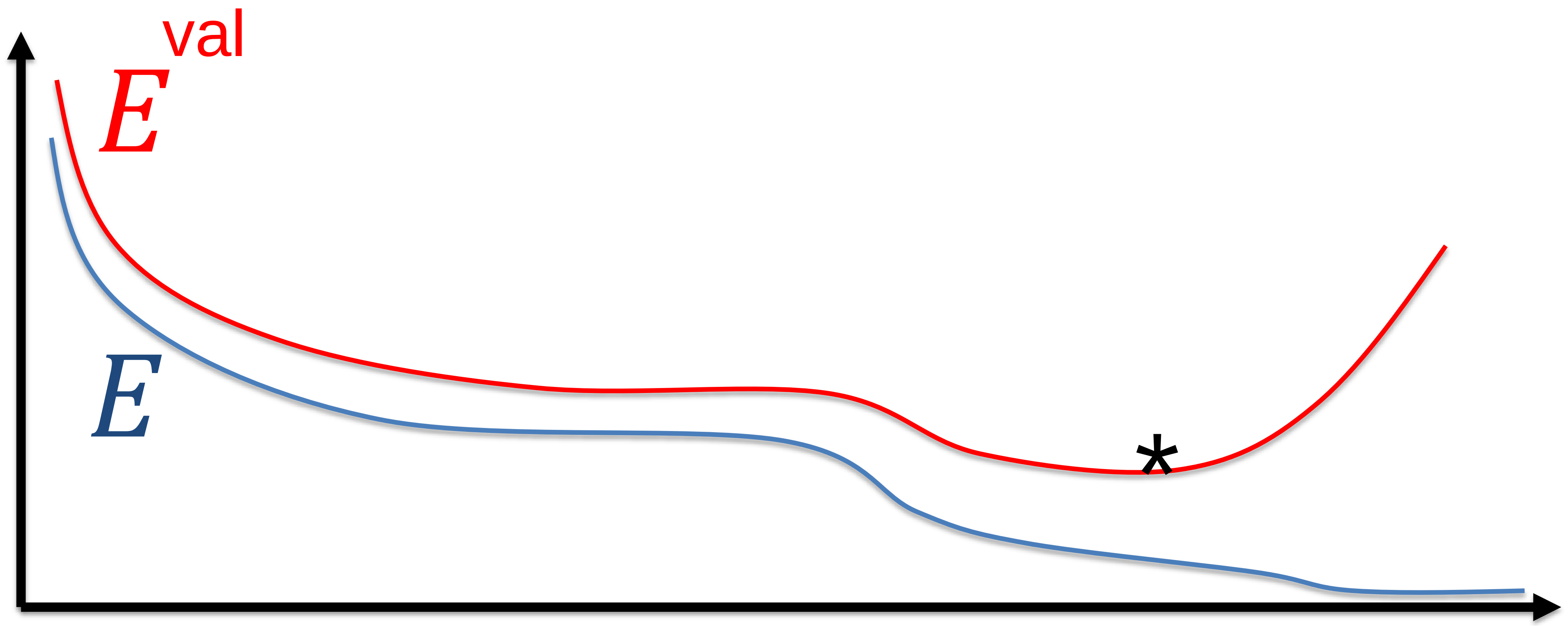
9. Regularization by early stopping

Minimize **training error** **stepwise** by gradient descent

$$E(\mathbf{w}) = \frac{1}{2}\sum_{\mu=1}^{P1} \left[t^{\mu} - \hat{y}^{\mu}\right]^2$$

Every k steps plot error for both data sets

$$E^{\text{val}}(\mathbf{w}) = \frac{1}{2}\sum_{\mu=P1+1}^{P} \left[t^{\mu} - \hat{y}^{\mu}\right]^2$$



epochs, training time

* Keep the weights for minimal validation error

Previous slide.
At the end of every epoch, or after every 1000 steps of stochastic gradient descent, you simply measure the performance on the validation set.
You continue for a long time: sometimes the measured validation error is stable, sometimes it goes down, sometimes it slightly decreases. At the end it always strongly increases if the network is overall flexible enough.

Whenever you go through a minimum you record the momentary values of all weights – but you continue training. Once you have finished, you go back to the values which had the minimal validation error.

This method can also be combined with k-fold cross-validation.

9. Regularization by early stopping

-very easy to implement

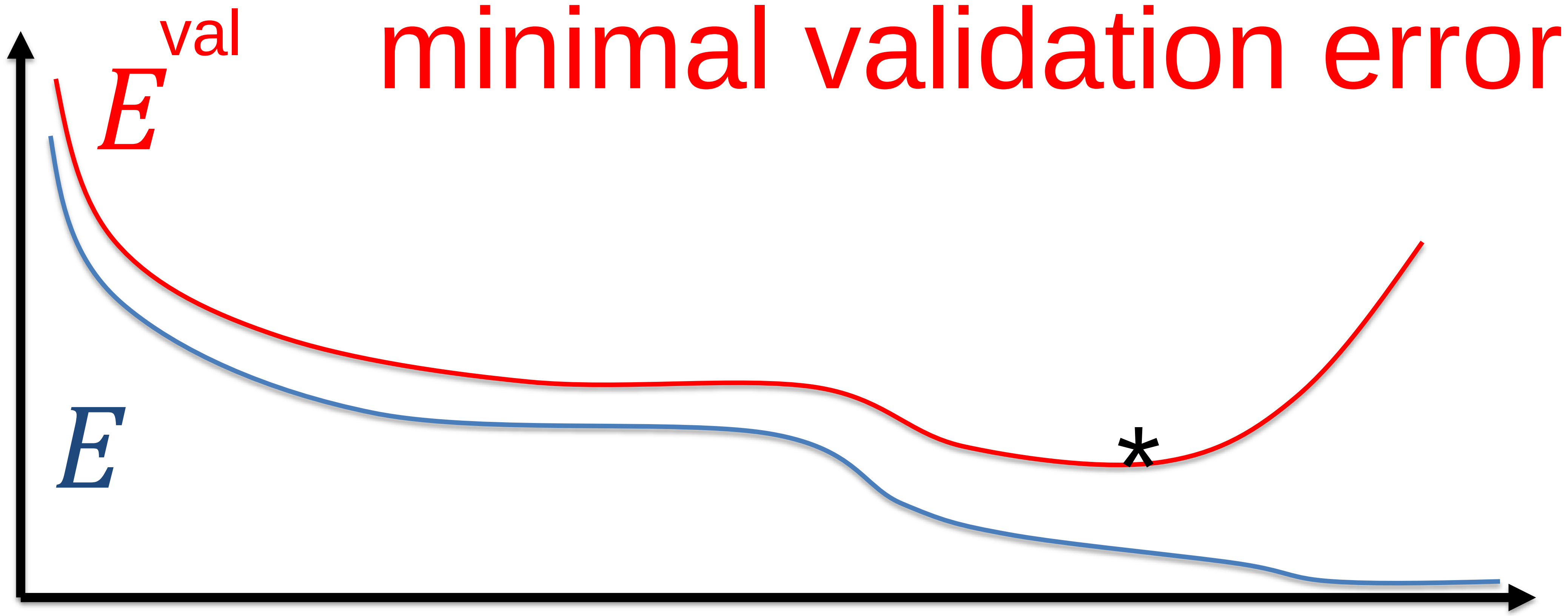
-control of flexibility via learning time

-network ‘uses’ its total flexibility only after lengthy optimization

→ go back to ‘earlier’ solution

→ maximal flexibility not exploited

*Keep the weights of minimal validation error



epochs, training time

see also: week 3 and 4

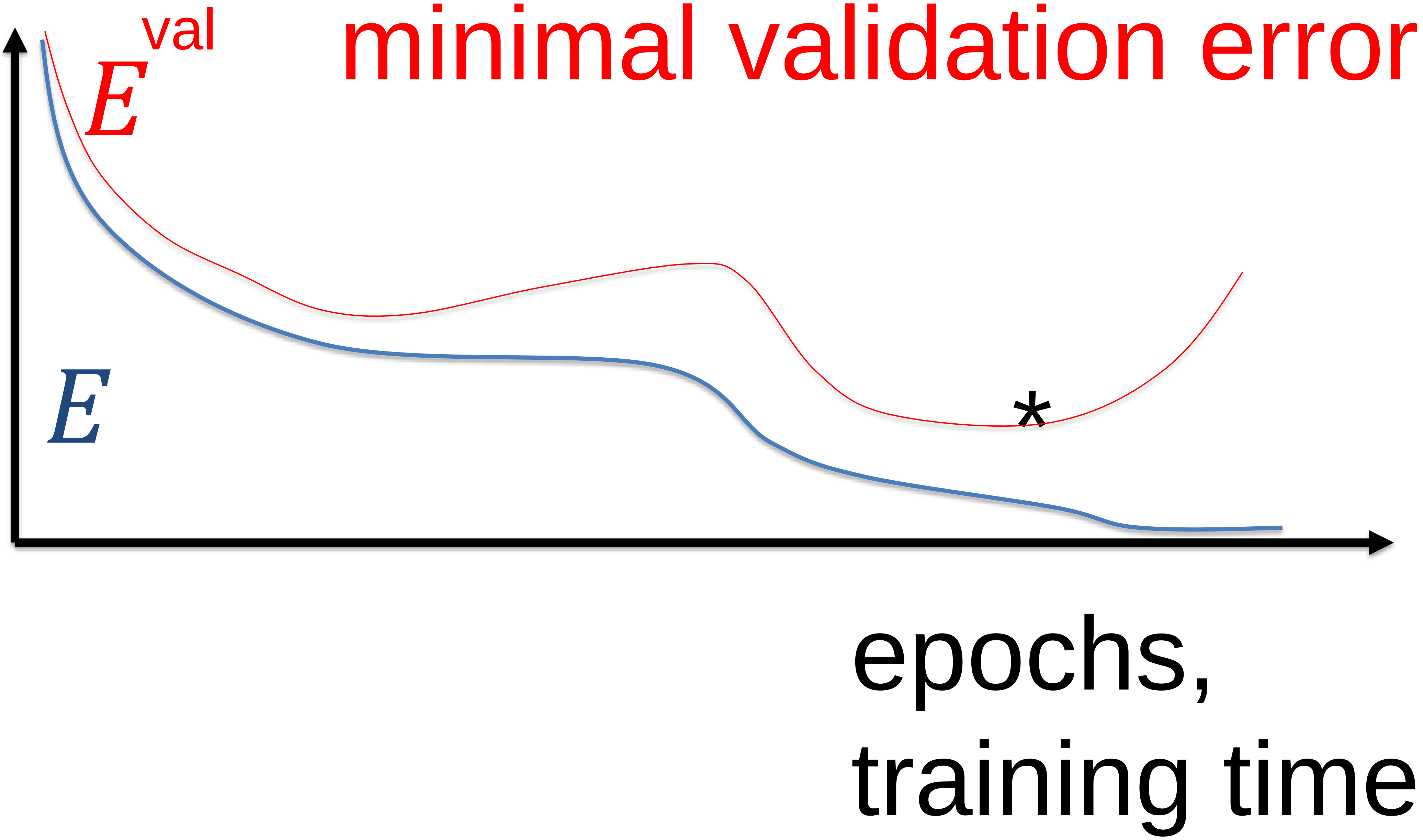
Previous slide.

The basic idea of early stopping is that weights are initialized close to zero and move only slowly to large absolute values. In order to implement a flexible surface with high curvature, big weights are needed. Therefore the flexibility of a network increases over training.
Early stopping means stopping before the maximal flexibility is exploited.

9. Regularization by early stopping

- control of flexibility via learning time

*Keep the weights of minimal validation error



- store weights
  of previous best solution

- continue to convergence

→ **go back to ‘earlier’ solution**:
keep weights of minimal validation error

→ maximal flexibility not exploited

‘Not early stopping, but going back’

see also: week 3 and 4

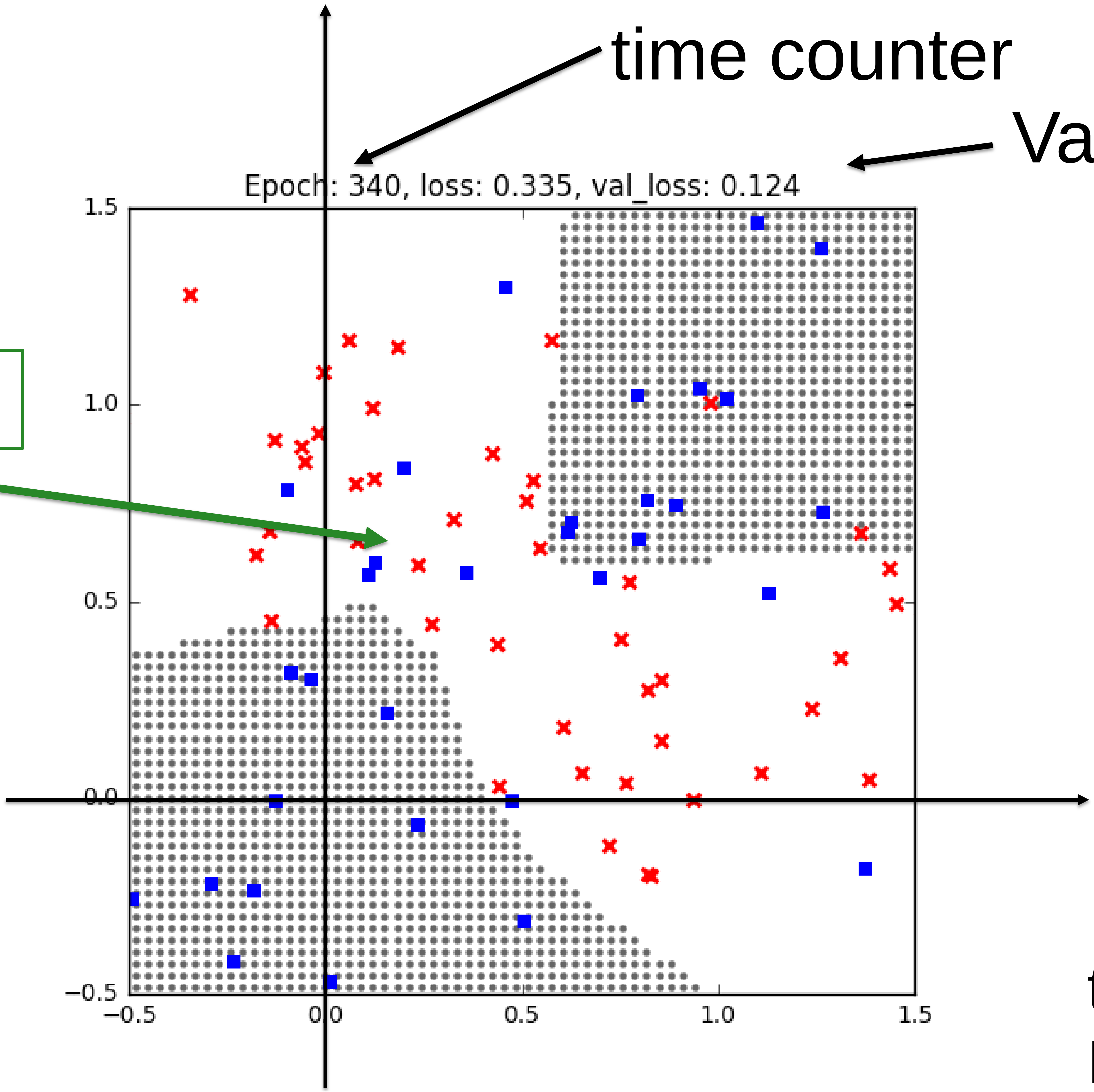
Previous slide.
Note that early stopping does not mean actual stopping at the first minimum of the validation error – because there could be a much better minimum later.
Rater it means: go back later to the ‘earlier’ solution.

9. Example: Noisy XOR problem, as a function of training time

100 data points

low noise
high noise

interesting region



time counter

Validation error

thanks to
Florian Colombo

Previous slide.
Noisy XOR means: the perfect split for an infinite amount of data (or for future data) would be along horizontal and vertical axis at value 0.5. However, the algorithm optimizes the separating surface with a data base of only 100 noisy data points.

The time counter indicates how far we are in the gradient optimization.
The validation error the performance on the infinite amount of data.

The separation into positive and negative parts (=the discriminant function=separating surface, visualized by the shading) looks quite good from 200-500 epochs. After 800 epochs clear signs of overfitting are visible in the discriminant function (new patches appear).

The value of the validation error is lowest at time point around 500 epochs.

Note that the shading (zero crossings of discriminant function) would correspond to a hard classification in the output layer; however the validation error works with the real output which is continuous (and therefore gives a small error even for points that are ‘correct’ but close to the boundary)

Objectives for today:

- XOR problem and the need for multiple layers
  → hidden layer provides flexibility
- understand backprop as a smart algorithmic implementation of the chain rule
  → algorithmic differentiation is better than numeric differentiation
- hidden neurons add flexibility, but flexibility is not always good
  → control flexibility by hyperparameters or early stopping: use validation data
- training base and validation and test base: the need to predict well for future data
  →test Error

Reading for this lecture:

Bishop 2006, Ch. 1.1 and 5.3 of

Pattern recognition and Machine Learning

or

Bishop 1995, Ch. 1 and 4.8 of

Neural networks for pattern recognition

or

Goodfellow et al.,2016 Ch. 5.1-5.3 and 6.5 of

Deep Learning

Now exercises (+ XOR with Keras simulator/tutorial)