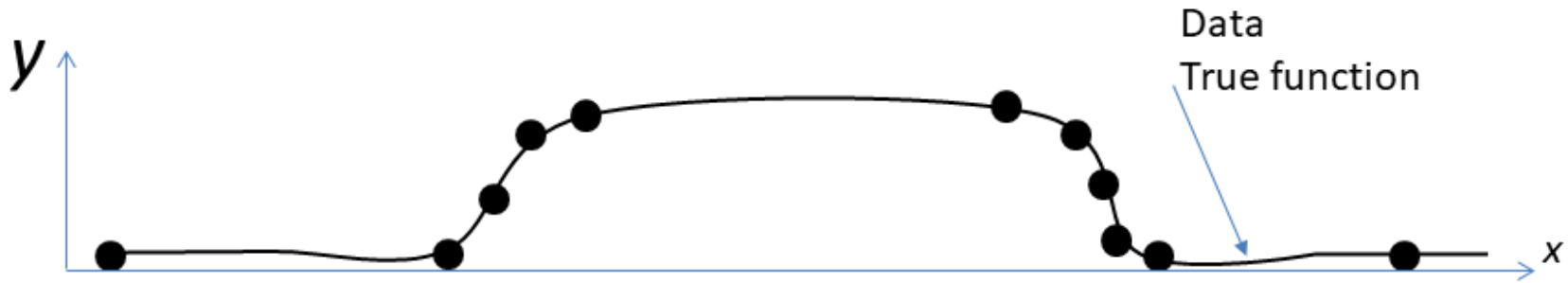
# Neural Networks

Volker Tresp

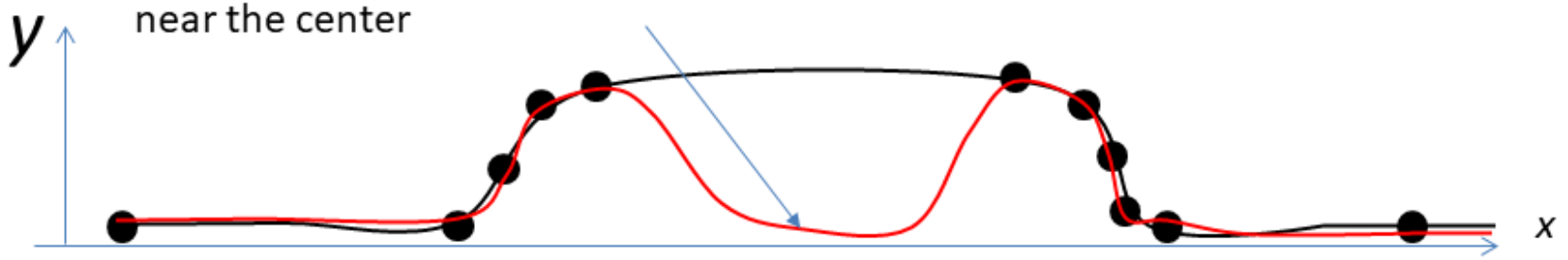
Summer 2019

## Introduction

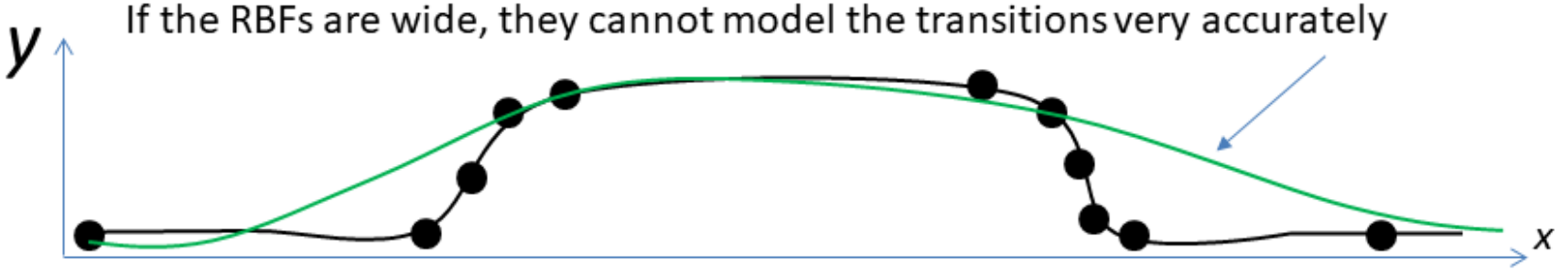
- In many applications, data might be uniformly distributed in input space, but complexity in $y$-space might be nonuniform
- In the next slide the function has two areas of high complexity; RBF approaches, with a notion of uniform complexity, have problems




If the RBFs are narrow, I might need a lot of them and the prediction is bad near the center




If the RBFs are wide, they cannot model the transitions very accurately



# Sigmoidal Basis Functions

- A sigmoidal basis function has the form

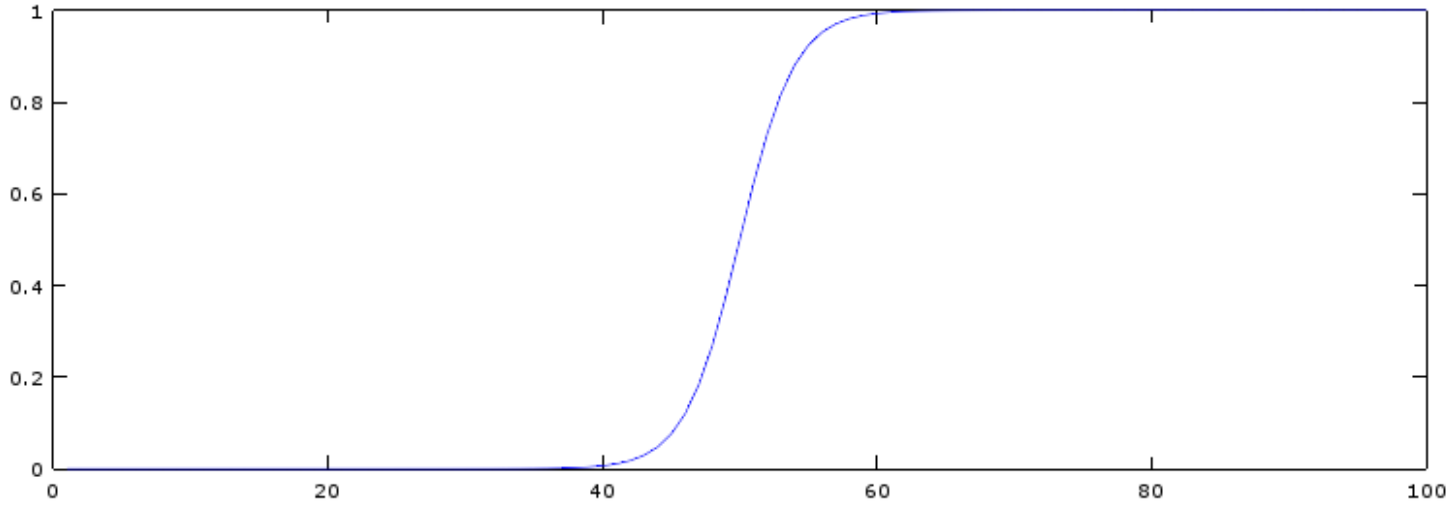
$$f(x) = \text{sig}(vx + v_0)$$

where

$$\text{sig}(arg) = \frac{1}{1 + \exp(-arg)}$$

- The function is only complex near its center where $arg \approx 0$
- Important: **To get the location and the slope at the center, we need to adapt the inner parameters** $v_0, v$. There is no closed-form solution for the estimate of those parameters: we need to use gradient-based approaches like SGD

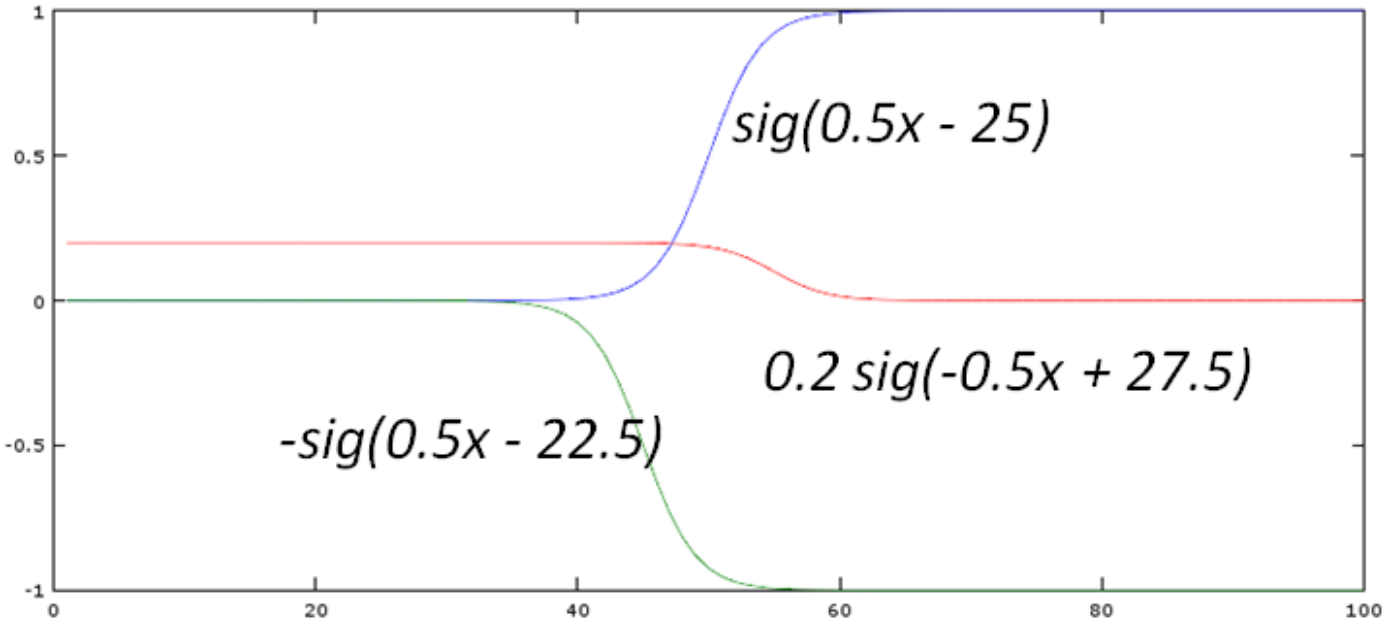
*sig(0.5x - 25)*

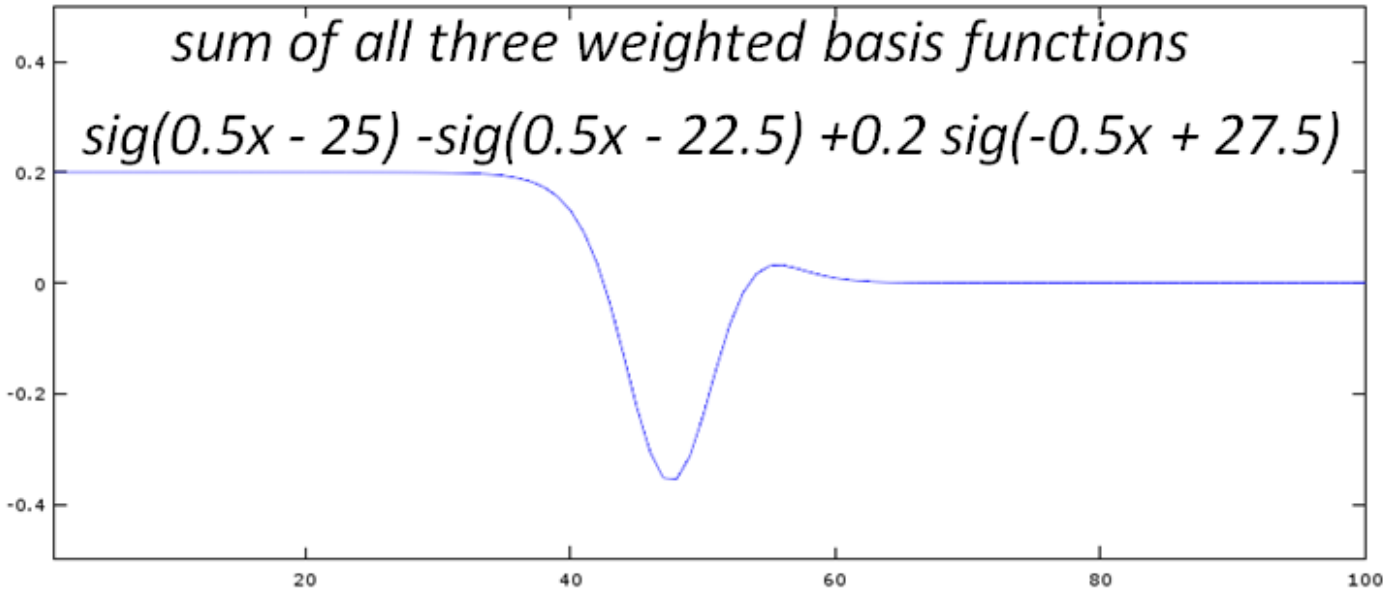


## Several Sigmoidal Basis Functions

- With several weighted sigmoidal basis functions, we are able to model a variety of functions with local complexity

$$f(x) = \sum_{h=1}^{H} w_h \, \mathsf{sig}(v_h x + v_{h,0})$$


sig(0.5x - 25)
0.2 sig(-0.5x + 27.5)
-sig(0.5x - 22.5)
1
0.5
0
-0.5
-1
0
20
40
60
80
100


sum of all three weighted basis functions
sig(0.5x - 25) -sig(0.5x - 22.5) +0.2 sig(-0.5x + 27.5)
0.4
0.2
0
-0.2
-0.4
20
40
60
80
100

## Overcomplete Basis

- Another approch would be to select a sparse subset in an overcomplete basis
- This is the approach used in Wavelets, Sparse coding, ...

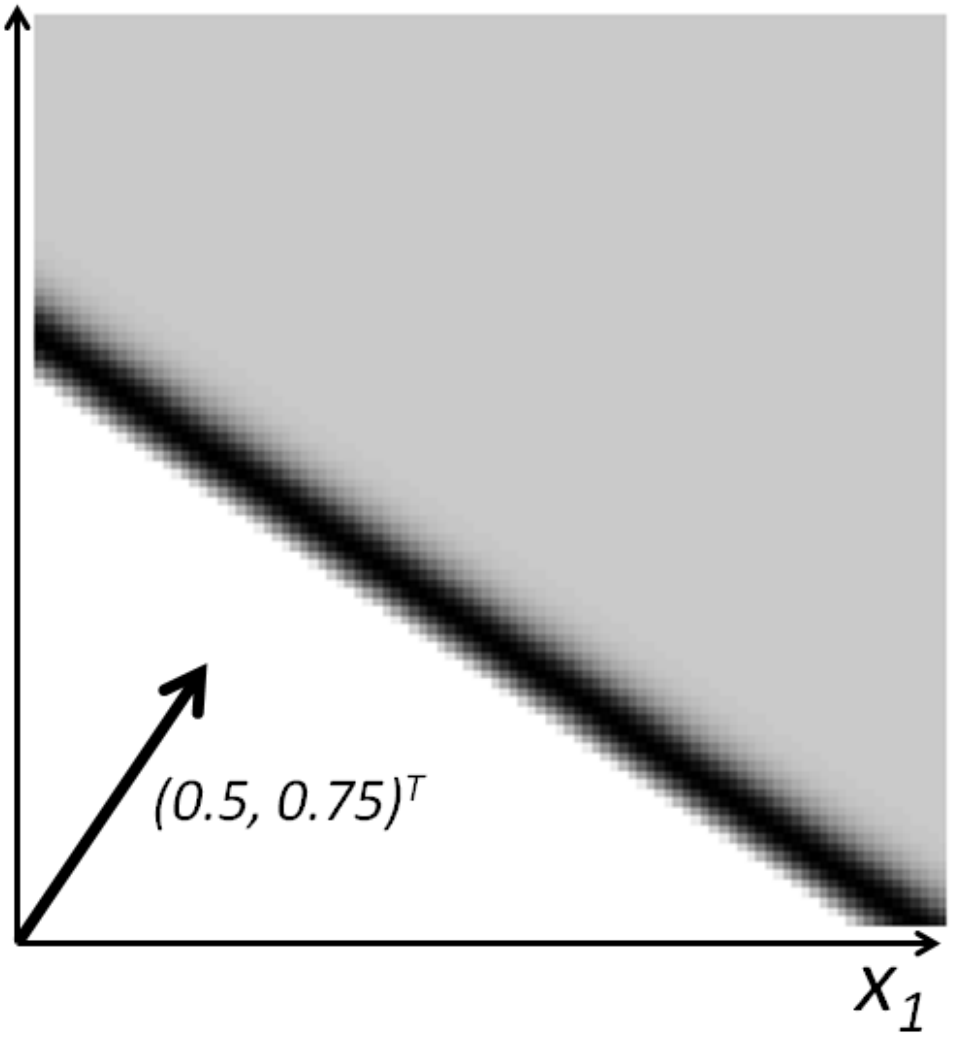
## Multidimensional Input Space

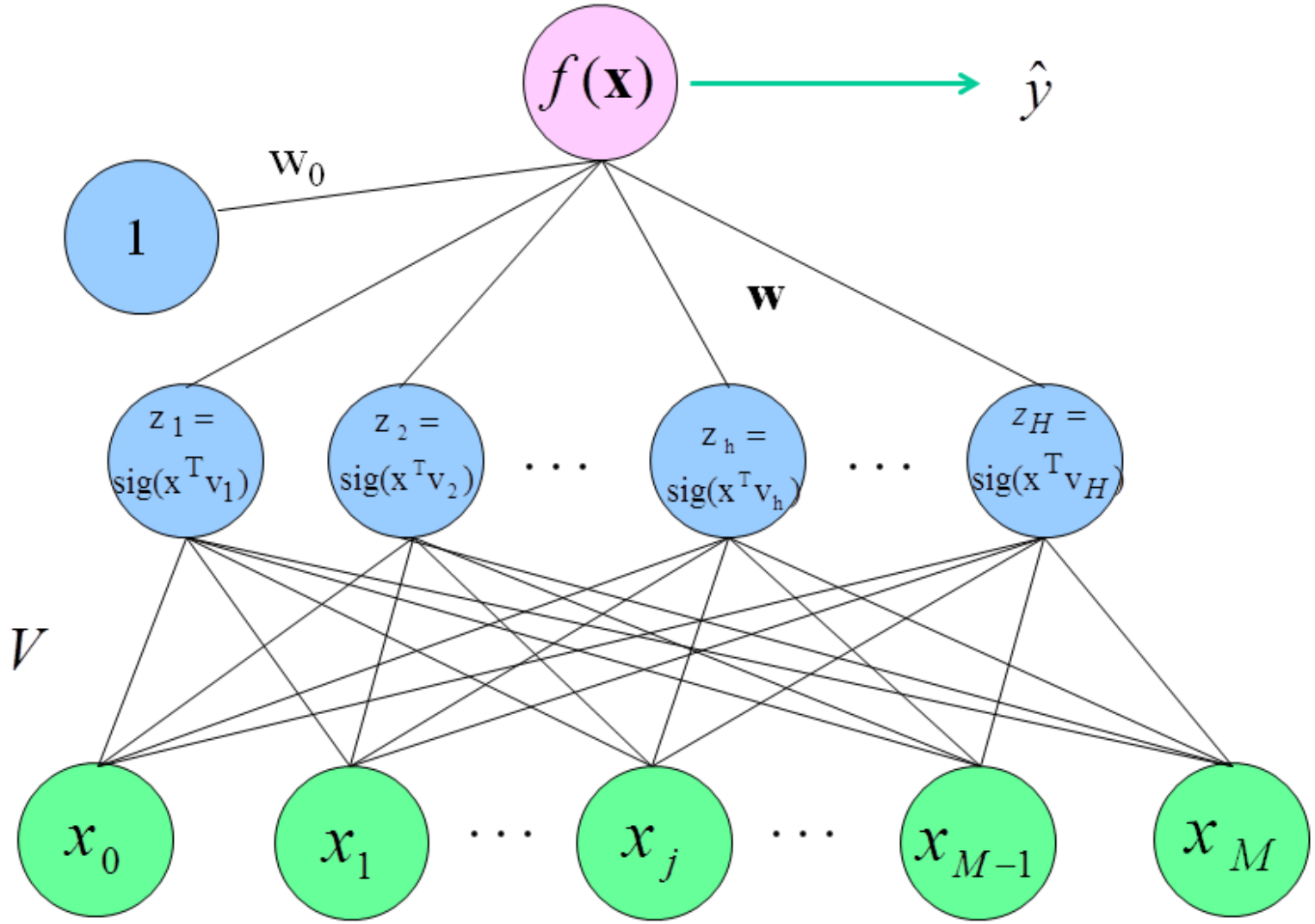
- With several sigmoidal basis functions, we are able to model a variety of functions with local complexity *in a high dimensional input space*

$$f(\mathbf{x}) = \sum_{h=1}^{H} w_h \, \mathrm{sig}\left(\sum_{j=1}^{M} v_{h,j} x_j + v_{h,0}\right)$$

- This equation describes neural network, more specifically, a Multilayer Perceptron with one hidden layer

$sig(0.5x_1+0.75x_2 - 25)$
$-sig(0.5x_1 +0.75x_2 - 22.5)$
$+0.2\ sig(-0.5x_1 -0.75x_2 + 27.5)$




$f(\mathbf{x})$
$\hat{y}$
$w_0$
1
$\mathbf{w}$
$z_1 = \text{sig}(x^T v_1)$
$z_2 = \text{sig}(x^T v_2)$
$z_h = \text{sig}(x^T v_h)$
$z_H = \text{sig}(x^T v_H)$
$V$
$x_0$
$x_1$
$x_j$
$x_{M-1}$
$x_M$

## Dimensionality Reduction

- Note that the neural network can also performs dimensionality reduction: in the figure, any components orthogonal to $(0.5, 0.75)^T$ are ignored
- So neural networks are well suited for large $M$, large $\nu$ and noisy features, if the function has high complexity in the projection of some low-dimensional subspaces

## Neural Networks: Essential Advantages

- Neural Networks are universal approximators: any continuous function can be approximated arbitrarily well (with a sufficient number of neural basis functions)
- Naturally, they can solve the XOR problem and at the time (mid 1980's) were considered the response to the criticism by Minsky and Papert with respect to the limited power of the single Perceptron
- Important advantage of Neural Networks: a good function fit can often (for a large class of important function classes) be achieved with a **small number of** neural basis functions
- Neural Networks scale well with input dimensions

## Flexible Models: Neural Networks

- For regression, the output of a neural network is the weighted sum of basis functions

$$\widehat{y} = f(\mathbf{x}) = w_0 + \sum_{h=1}^{H} w_h \text{sig}(\mathbf{x}^T \mathbf{v}_h)$$

- Note, that in addition to the output weights $\mathbf{w}$, the neural network also has inner weights $\mathbf{v}_h$

## Notation

- $x_1, \ldots, x_j, \ldots, x_M$: inputs
- $z_1, z_2, \ldots, z_h, \ldots, z_H$: Outputs of the $H$ hidden units (Thus $M_\phi \rightarrow H$)
- $y$: single neural network output, or $y_1, \ldots, y_k, \ldots, y_K$: $K$ neural network outputs

# Neural Basis Functions

- Special form of the basis functions

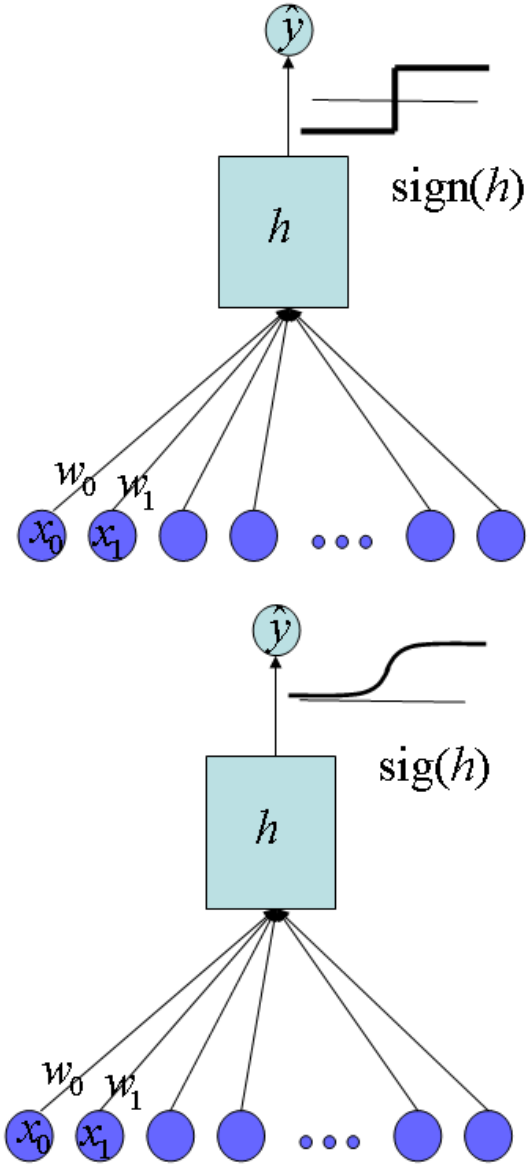
$$z_h = \text{sig}(\mathbf{x}^T \mathbf{v}_h) = v_{h,0} + \text{sig}\left(\sum_{j=0}^{M} v_{h,j} x_j\right)$$

  using the *logistic function*

$$\text{sig}(arg) = \frac{1}{1 + \exp(-arg)}$$

- Adaption of the inner parameters $v_{h,j}$ of the basis functions!

# Hard and Soft (sigmoid) Transfer Functions




- First, the activation function of the neurons in the hidden layer are calculated as the weighted sum of the inputs as

$$h(\mathbf{x}) = \sum_{j=0}^{M} w_j x_j$$

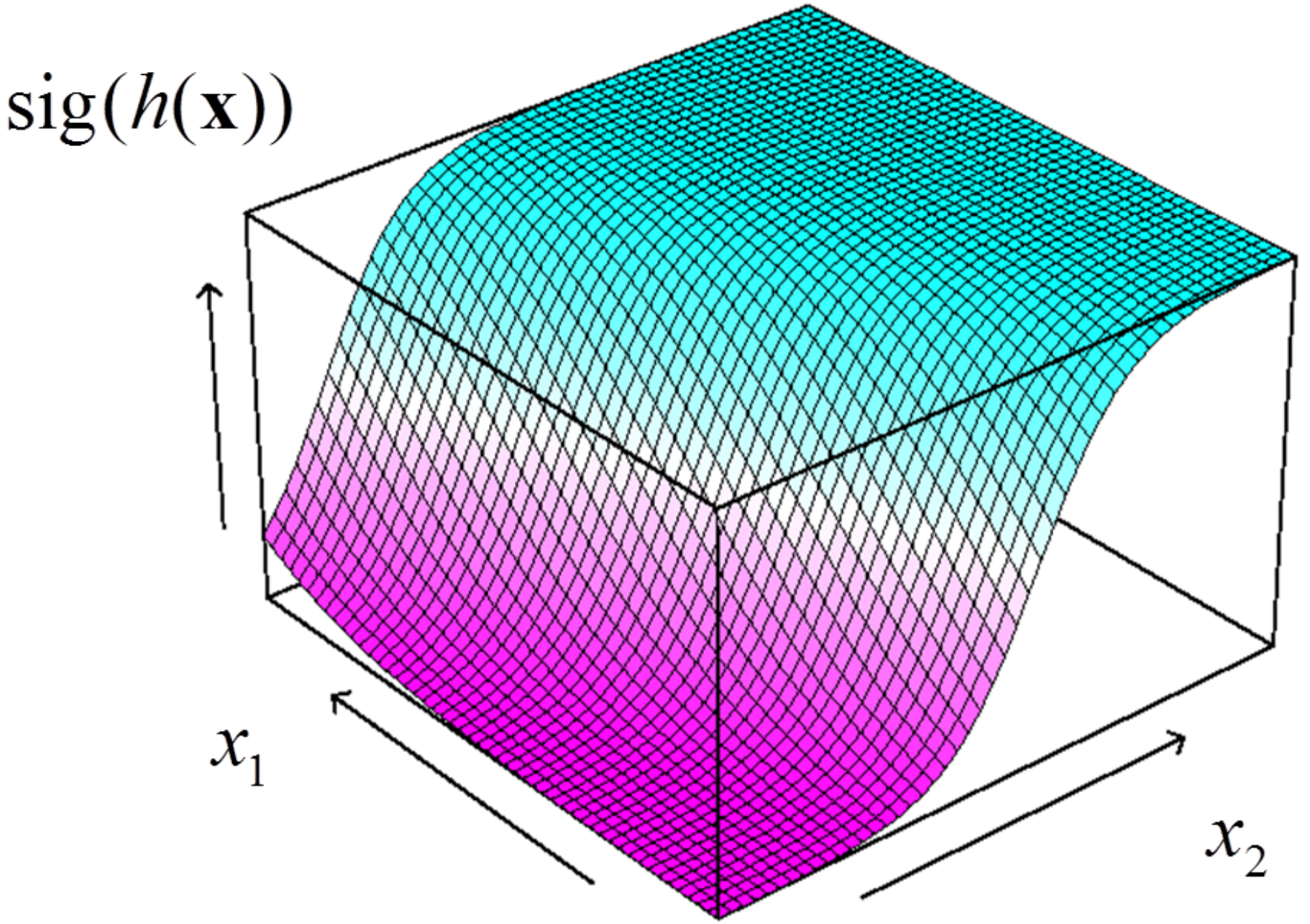
(note: $x_0 = 1$ is a constant input, so that $w_0$ corresponds to the bias)

- The sigmoid neuron has a soft (sigmoid) transfer function

$$\textbf{Perceptron}: \quad \hat{y} = \text{sign}(h(\mathbf{x}))$$

**Sigmoidal neuron:** $\quad \hat{y} = \text{sig}(h(\mathbf{x}))$

# Transfer Function


sig(h(x))
x1
x2

# Separating Hyperplane

- Definition of the hyperplane

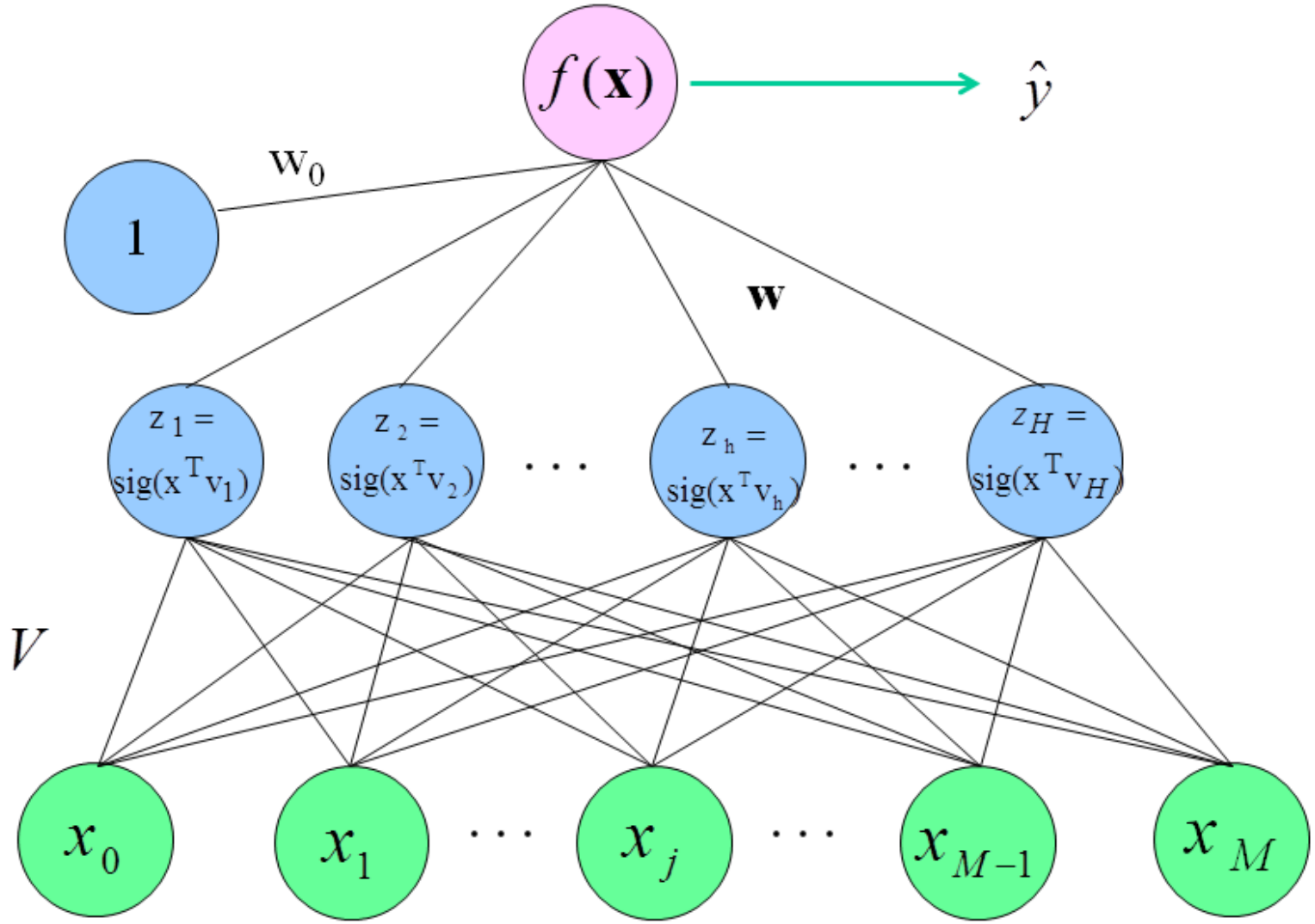
$$\text{sig}\left(\sum_{j=0}^{M} v_{h,j} x_j\right) = 0.5$$

  which means that:

$$\sum_{j=0}^{M} v_{h,j} x_j = 0$$

- "carpet over a step"

# Architecture of a Neural Network


$f(\mathbf{x})$
$\hat{y}$
$w_0$
1
$\mathbf{w}$
$z_1 = \text{sig}(x^T v_1)$
$z_2 = \text{sig}(x^T v_2)$
$z_h = \text{sig}(x^T v_h)$
$z_H = \text{sig}(x^T v_H)$
$V$
$x_0$
$x_1$
$x_j$
$x_{M-1}$
$x_M$

# Variants

- For a **2-class neural network classifier** apply the sigmoid transfer function to the output neuron, and calculate

$$\widehat{y} = \text{sig}(f(\mathbf{x})) = \text{sig}(\mathbf{z}^T\mathbf{w})$$

- For **multi-class tasks** (e.g., recognizing digits $0, 1, \ldots, 9$), one uses several output neurons. For example, to classify $K$ digits

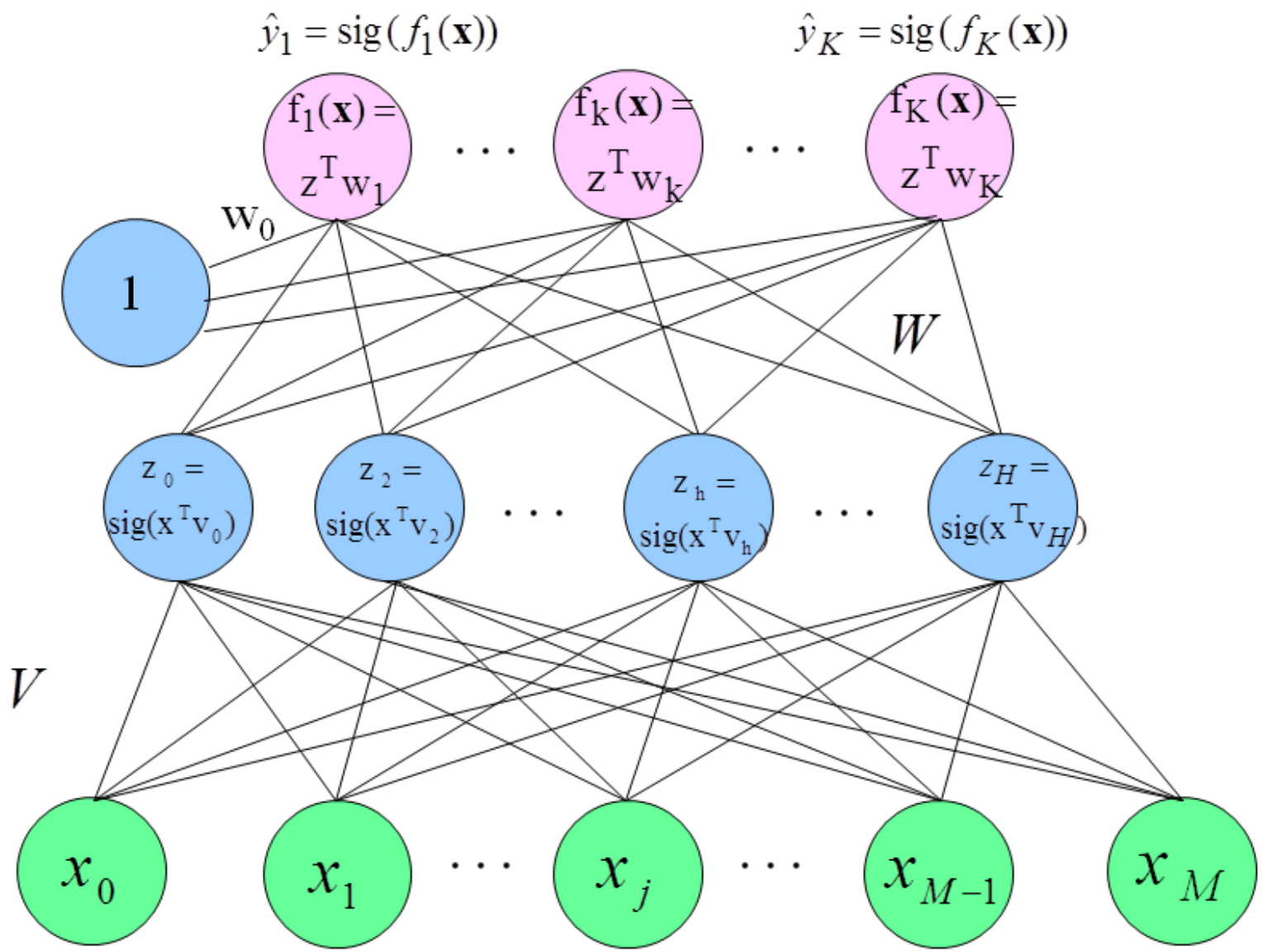
$$\widehat{y}_k = \text{sig}(f_k(\mathbf{x})) = \text{sig}(\mathbf{z}^T\mathbf{w}_k) \quad k = 1, 2, \ldots, K$$

  and one decides for class $l$, with $l = \arg\max_k(\widehat{y}_k)$

- (Nowadays on typically uses the softmax function: $\widehat{y}_k = \exp f_k/Z$ with $Z = \sum_k \exp f_k$, since then the outputs are nonnegative and sum to one)

- A Neural Network with at least one hidden layer is called a Multilayer Perceptron (MLP)

# Architecture of a Neural Network for Several Classes


$\hat{y}_1 = \mathrm{sig}(f_1(\mathbf{x}))$
$\hat{y}_K = \mathrm{sig}(f_K(\mathbf{x}))$
$f_1(\mathbf{x}) = z^T w_1$
$f_k(\mathbf{x}) = z^T w_k$
$f_K(\mathbf{x}) = z^T w_K$
$w_0$
1
$W$
$z_0 = \mathrm{sig}(x^T v_0)$
$z_2 = \mathrm{sig}(x^T v_2)$
$z_h = \mathrm{sig}(x^T v_h)$
$z_H = \mathrm{sig}(x^T v_H)$
$V$
$x_0$
$x_1$
$x_j$
$x_{M-1}$
$x_M$

## Learning Multiple-Class Classifiers

- The goal again is the minimization of the squared error calculated over all training patterns and all outputs

$$\text{cost}(W, V) = \sum_{i=1}^{N} \text{cost}(\mathbf{x}_i, W, V)$$

  with $\text{cost}(\mathbf{x}_i, W, V) = \sum_{k=1}^{K} (y_{i,k} - \hat{y}_{i,k})^2$

- The least squares solution for $V$ cannot be calculated in closed-form
- Typically both $W$ and $V$ are trained via (stochastic) gradient descent

# Adaption of the Output Weights

- The gradient of the cost function for an output weight for pattern $i$ becomes

$$\frac{\partial \text{cost}(\mathbf{x}_i, W, V)}{\partial w_{k,h}} = -2\delta_{i,k} z_{i,h}$$

  where

$$\delta_{i,k} = \text{sig}'(\mathbf{z}_i^T \mathbf{w}_k)[y_{i,k} - \hat{y}_{i,k}]$$

  is the back propagated error signal (error back propagation). Note, that $\delta_{i,k}$ is attached to an output node $k$.

- The pattern based gradient descent learning becomes (pattern: *i*, output: *k*, hidden: *h*):

$$w_{k,h} \leftarrow w_{k,h} + \eta \delta_{i,k} z_{i,h}$$

## The Derivative of the Sigmoid Transfer Function with Respect to the Argument

... can be written elegantly as

$$\mathrm{sig}'(in) = \frac{\exp(-in)}{(1+\exp(-in))^2} = \mathrm{sig}(in)(1-\mathrm{sig}(in))$$

Thus

$$\delta_{i,k} = \widehat{y}_{i,k}(1-\widehat{y}_{i,k})\ (y_{i,k}-\widehat{y}_{i,k})$$

# Adaption of the Input Weights

- The gradient of an input weight with respect to the cost function for pattern $i$ becomes

$$\frac{\partial \text{cost}(\mathbf{x}_i, W, V)}{\partial v_{h,j}} = -2\delta_{i,h} x_{i,j}$$

  with the back propagated error

$$\delta_{i,h} = \text{sig}'(\mathbf{x}_i^T \mathbf{v}_h) \sum_{k=1}^{K} w_{k,h}\delta_{i,k} = z_{i,h}(1 - z_{i,h}) \sum_{k=1}^{K} w_{k,h}\delta_{i,k}$$

- Note, that $\delta_{i,h}$ is attached to hidden node node $h$.
- For the pattern based gradient descent, we get (pattern: *i*, hidden: *h*, input: *j*):

$$v_{h,j} \leftarrow v_{h,j} + \eta\delta_{i,h} x_{i,j}$$

# Pattern-based Learning

- Iterate over all training patterns
- Let $\mathbf{x}_i$ be the training data point at iteration $t$
  - Apply $\mathbf{x}_i$ and calculate $\mathbf{z}_i, \mathbf{y}_i$ (*forward propagation*)
  - Via *error backpropagation* calculate the $\delta_{i,h}, \delta_{i,k}$
  - Adapt

$$w_{k,h} \leftarrow w_{k,h} + \eta \delta_{i,k} z_{i,h}$$

$$v_{h,j} \leftarrow v_{h,j} + \eta \delta_{i,h} x_{i,j}$$

- All operations are "local": biologically plausible

## Complexity Analysis

- Neural networks work well in all situations covered in the discussion on basis function, except for Case I. (curse of dimensionality)
- In particular, they offer an excellent solution for Case Ia (sparse basis).

## Neural Networks and Overfitting

- In comparison to conventional statistical models, a Neural Network has a huge number of free parameters, which might easily lead to over fitting
- The two most common ways to fight over fitting are regularization and stopped-training
- Let's first discuss regularization

# Neural Networks: Regularisation

- We introduce regularization terms and get

$$\text{cost}^{pen}(W,V) = \sum_{i=1}^{N} \text{cost}(\mathbf{x}_i, W, V) + \lambda_1 \sum_{k=1}^{K}\sum_{h=0}^{H} w_{k,h}^2 + \lambda_2 \sum_{h=1}^{H}\sum_{j=0}^{M} v_{h,j}^2$$

- The learning rules change to (with *weight decay term*, the constant bias is typically not regularized)

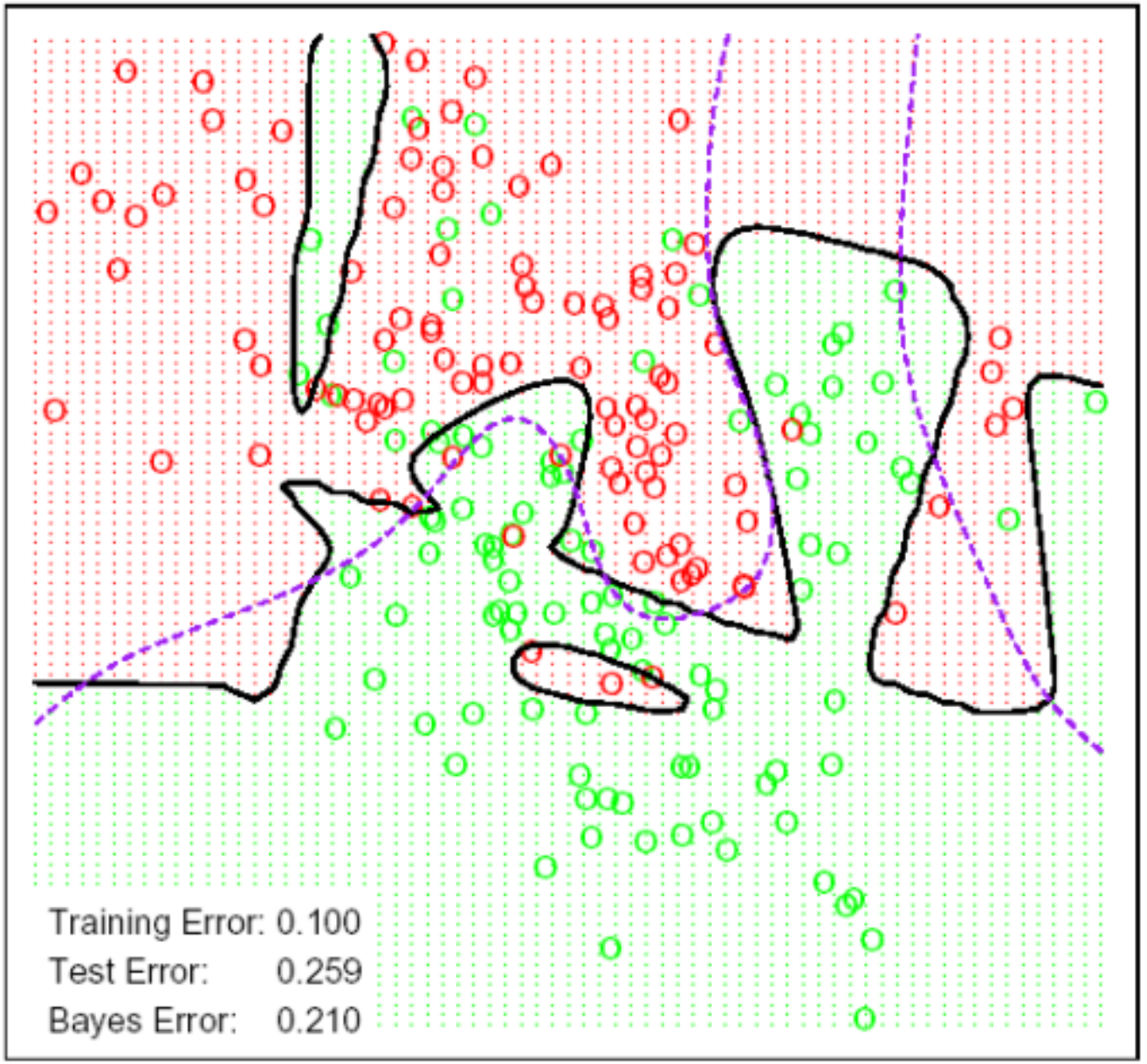
$$w_{k,h} \leftarrow w_{k,h} + \eta \left( \delta_{i,k} z_{i,h} - \frac{\lambda_1}{N} w_{k,h} \right)$$

$$v_{h,j} \leftarrow v_{h,j} + \eta \left( \delta_{i,h} x_{i,j} - \frac{\lambda_2}{N} v_{h,j} \right)$$

## Artificial Example

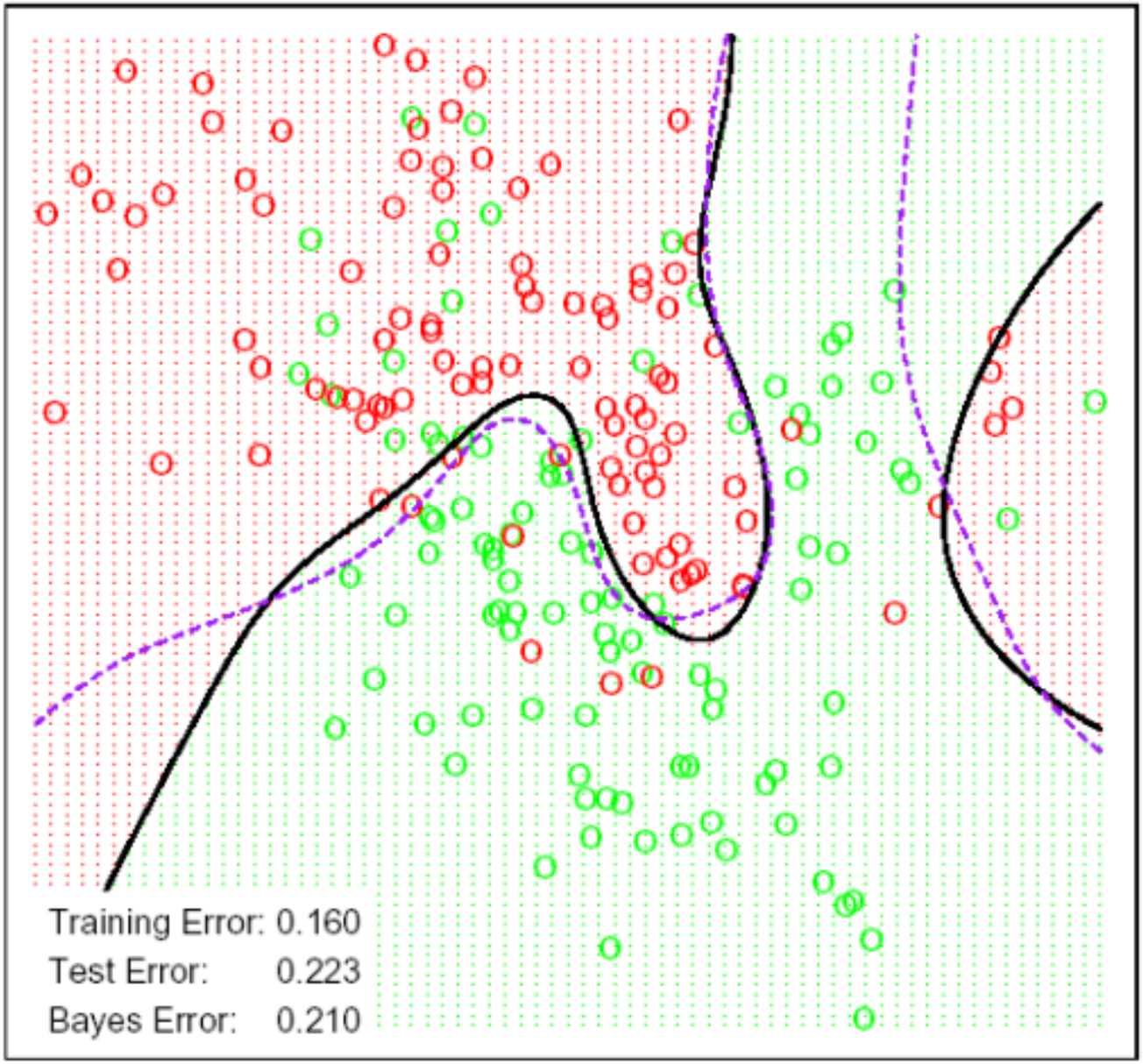
- Data for two classes (red/green circles) are generated
- Classes overlap
- The optimal separating boundary is shown dashed
- A neural network without regularization shows over fitting (continuous line)

Neural Network - 10 Units, No Weight Decay



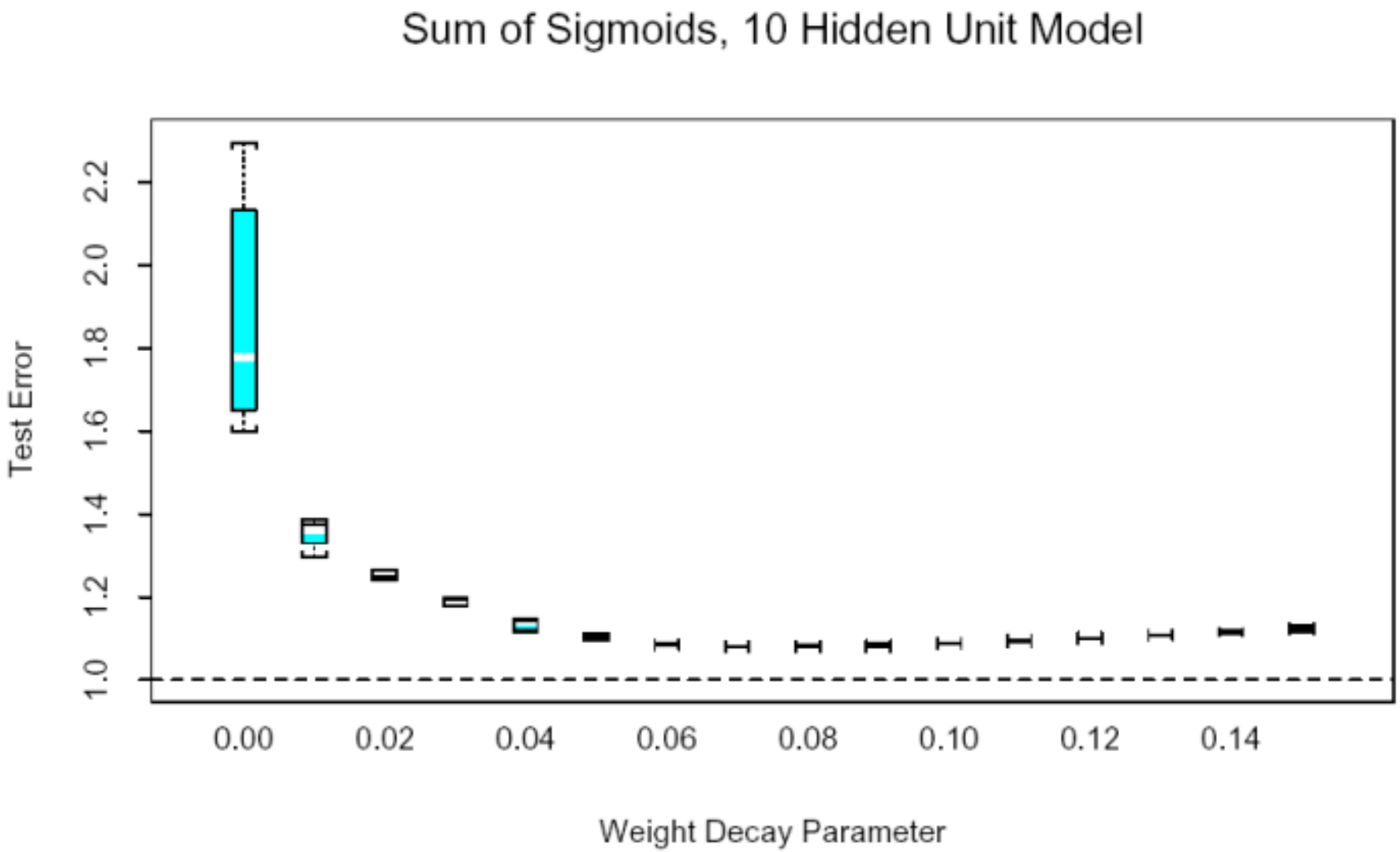
## Same Example with Regularization

- With regularization ($\lambda_1 = \lambda_2 = 0.2$) the separating plane is closer to the true class boundaries
- The training error is smaller with the unregularized network, the test error is smaller with the regularized network


Neural Network - 10 Units, Weight Decay=0.02
Training Error: 0.160
Test Error: 0.223
Bayes Error: 0.210

## Optimized Regularization Parameters

- The regularization parameter is varied between 0 and 0.15
- The vertical axis shows the test error for many independent experiments
- The best test error is achieved with regularization parameter 0.07
- The test error varies a lot with no regularization


Sum of Sigmoids, 10 Hidden Unit Model
Test Error
1.0
1.2
1.4
1.6
1.8
2.0
2.2
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
Weight Decay Parameter

## Variations

- Use more than one hidden layer (see deep learning)
- Use $\tanh(arg) \in (-1, 1)$ instead of $\text{sig}(arg) \in (0, 1)$
- For the $\tanh(arg)$, use targets $y \in \{-1, 1\}$, instead of $y \in \{0, 1\}$
- Often: Use $\tanh(arg)$ in the hidden layer and $\text{sig}(arg)$ in the output layer

# Cross-entropy Cost Function

- Instead of the sum-squared-error cost function, use the cross-entropy cost function
- With $y_k \in \{0, 1\}$

$$\text{cost}(W, V) = \sum_{i=1}^{N} \text{cost}(\mathbf{x}_i, W, V)$$

and

$$\text{cost}(\mathbf{x}_i, W, V) = -\left[\sum_{k=1}^{K} y_{i,k} \log \hat{y}_{i,k} + (1 - y_{i,k}) \log(1 - \hat{y}_{i,k})\right]$$

- Recall, that the gradient w.r.t output weights is

$$\frac{\partial \text{cost}(\mathbf{x}_i, W, V)}{\partial w_{k,h}} = -2\delta_{i,k} z_{i,h}$$

- For cross entropy we get:

$$\delta_{i,k} = (y_{i,k} - \hat{y}_{i,k})$$

(Derivation of this equation in the lecture on linear classifiers)

- For the squared loss, this term was:

$$\delta_{i,k} = [\widehat{y}_{i,k}(1 - \widehat{y}_{i,k})](y_{i,k} - \widehat{y}_{i,k})$$

The differences are the terms in the squared bracket. Thus the squared error has the problem that the gradient becomes zero when the output gets saturated $\widehat{y}_{i,k} \to 0/1$, thus also when the output is completely wrong! Today cross entropy is preferred!

# Softmax Cost Function

- Often the outputs are mutual exclusive: a handwritten digit is exactly one out off 10 digit
- As activation, one uses softmax

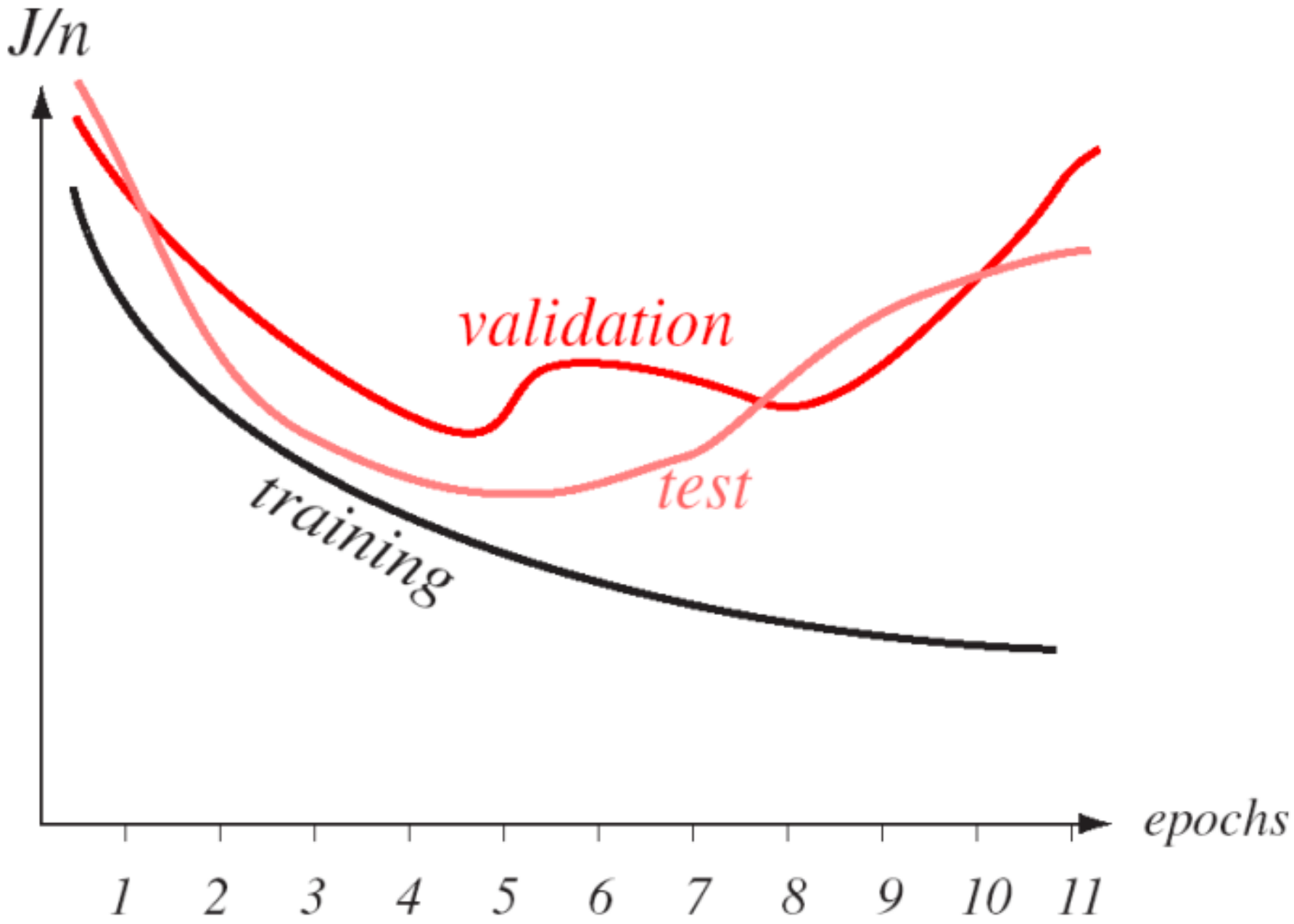
$$\widehat{y}_k = \frac{\exp f_k}{\sum_k \exp f_k}$$

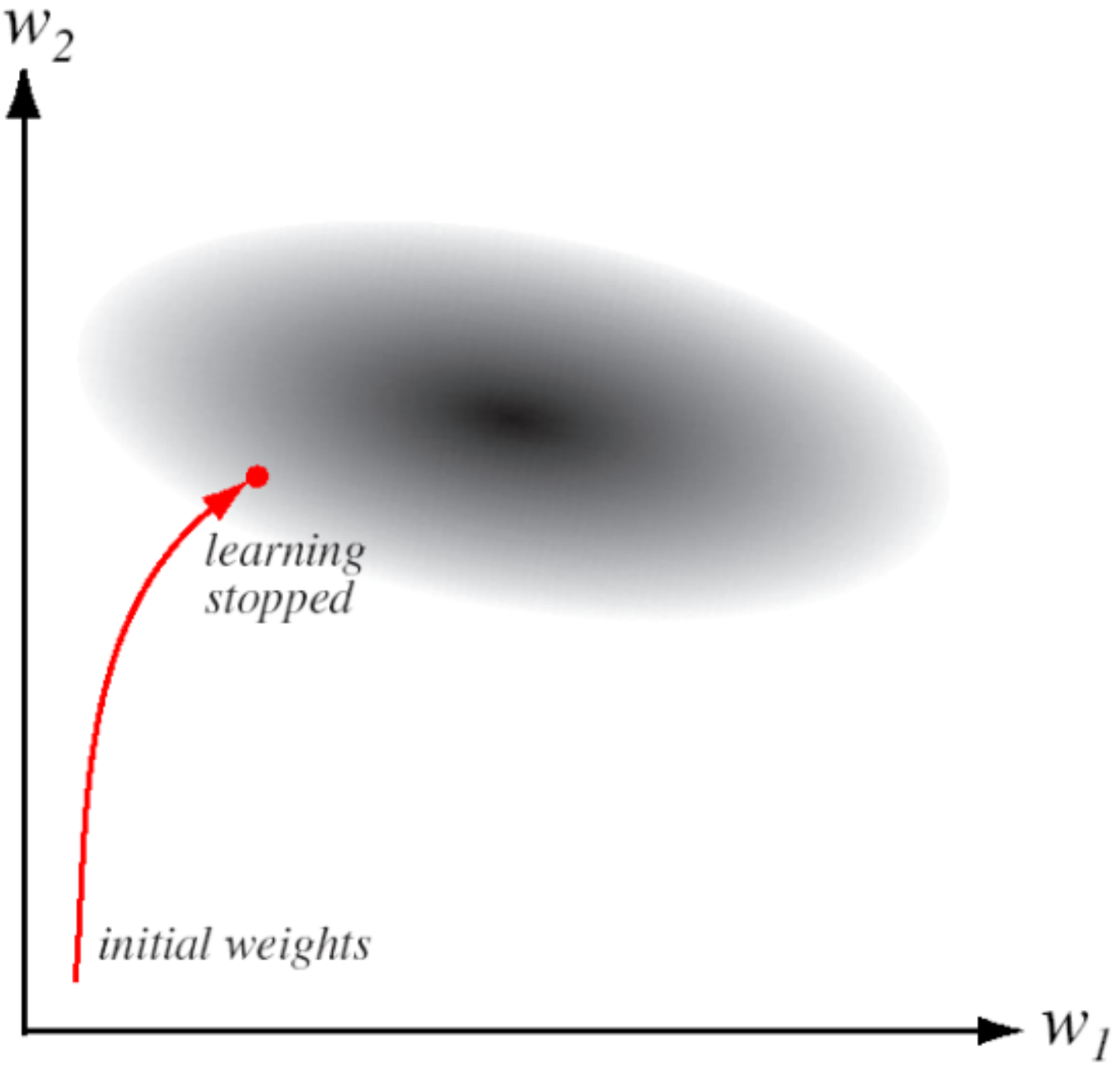
The cost term is (one hot encoding with $y_{i,k} \in \{0, 1\}$)

$$\text{cost}(\mathbf{x}_i, W, V) = \left( \log \sum_k \exp f_{i,k} \right) - \sum_k y_{i,k} f_{i,k}$$

## Regularization with Stopped-Training

- In the next picture you can see typical behavior of training error and test error as a function of training time (an epoch is one pass through all data during learning)
- As expected the training error steadily decreases with epochs
- As expected, the test error first decreases as well; maybe surprisingly there is a minimum, after which the test error increases
- Explanation: During training, the degrees of freedom in the neural network slowly increase; with too many degrees of freedom, overfitting occurs
- It is possible to regularize a neural network by simply stopping the adaptation at the right moment (regularization by stopped-training)


J/n
validation
test
training
epochs
1
2
3
4
5
6
7
8
9
10
11


$w_2$
learning
stopped
initial weights
$w_1$

## Optimizing the Learning Rate $\eta$

- Convergence can be influenced by the learning rate $\eta$
- Next figure: if the learning rate is too small, convergence can be very slow, if too large the iterations can oscillate and even diverge
- The learning rate can be adapted to the learning process ("Adaptive Learning Rate Control"); a popular variant is called Adaptive Moment Estimation (Adam) (see deep learning lecture)

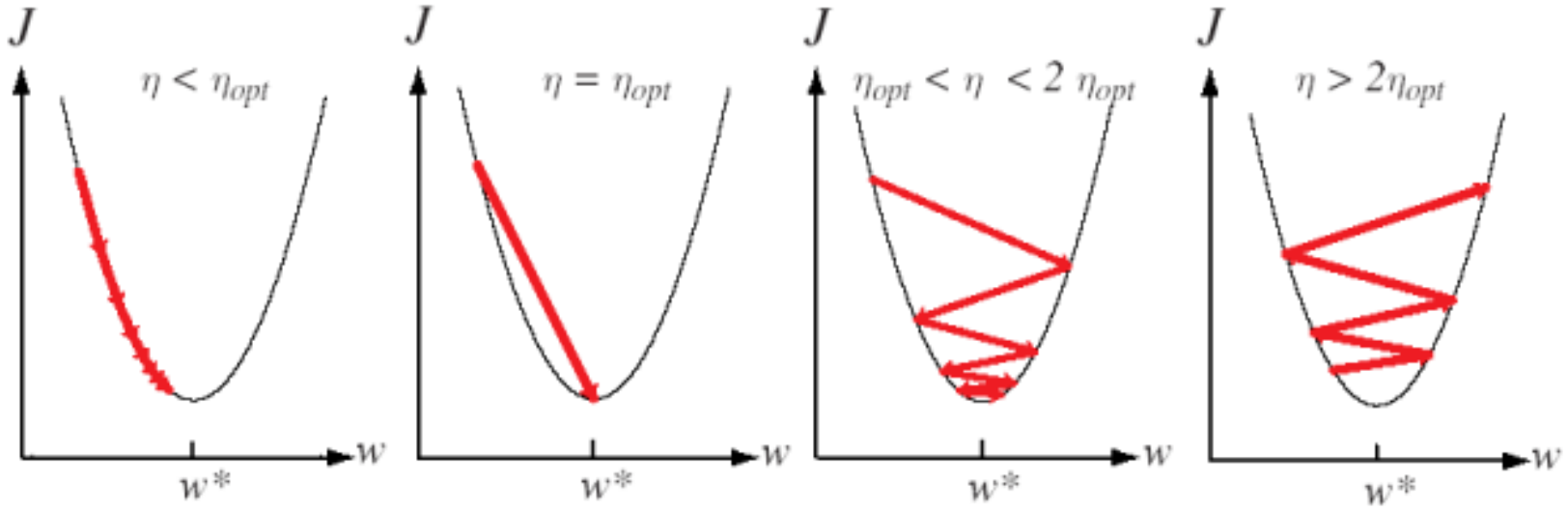
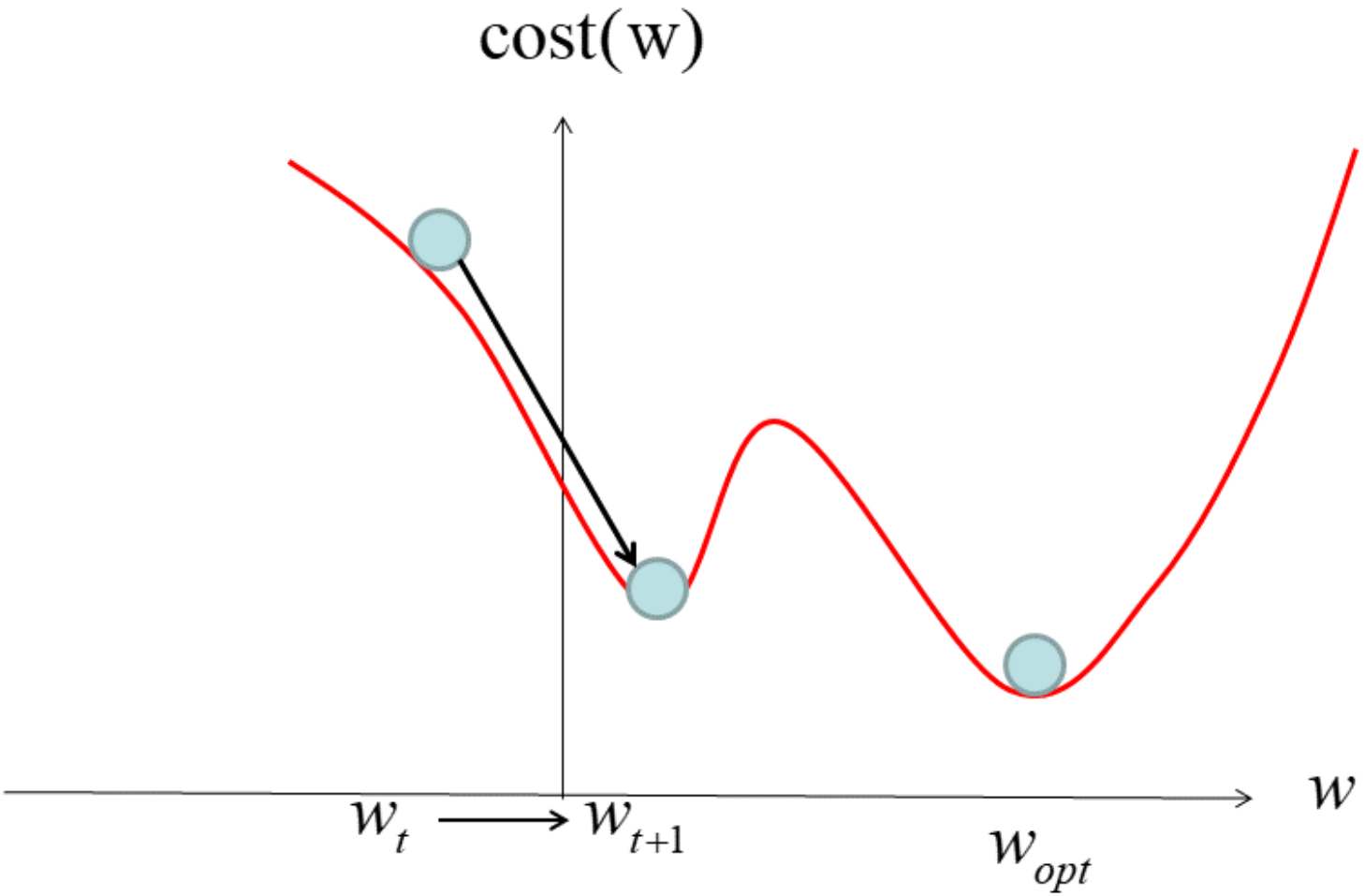
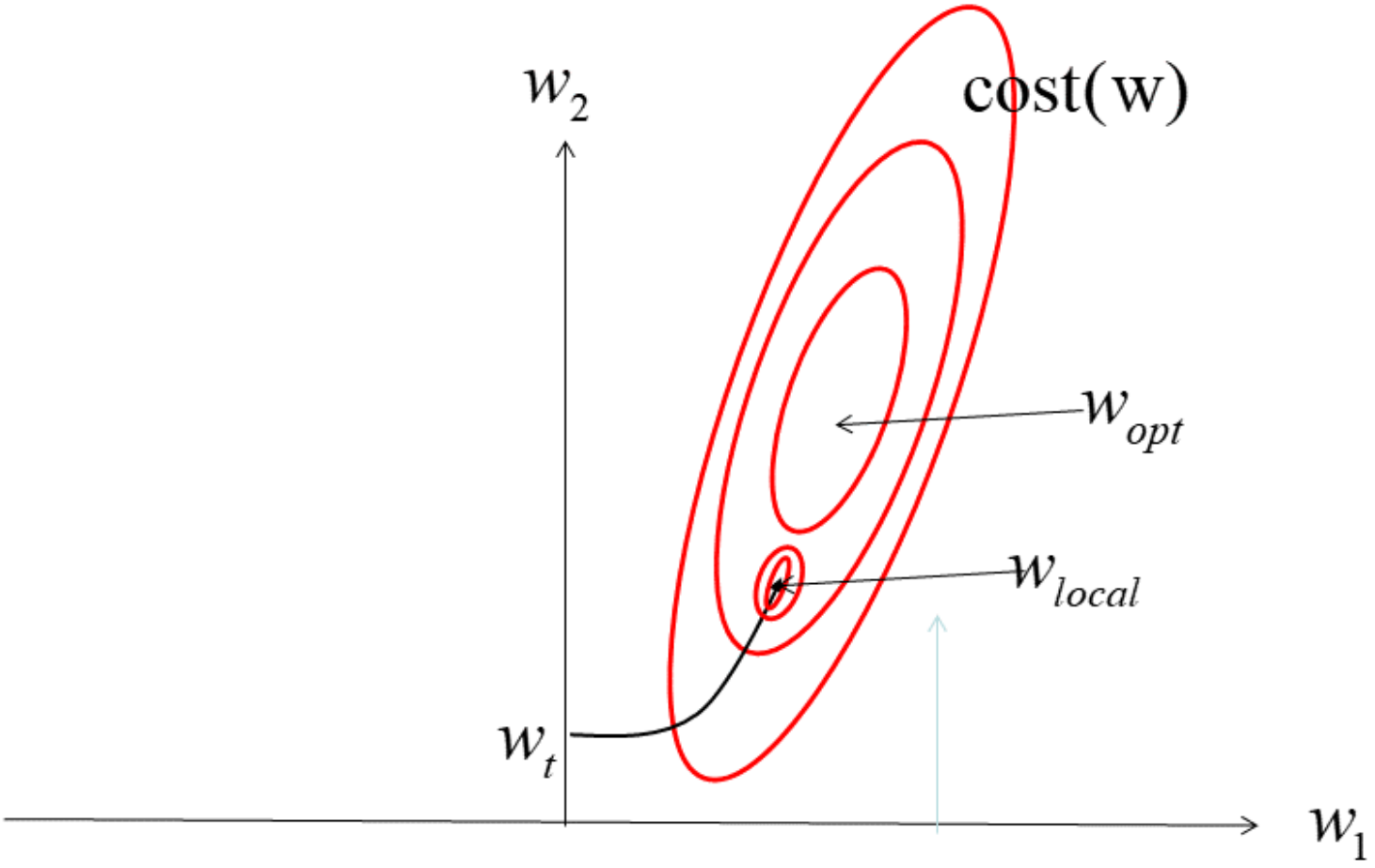



**FIGURE 6.16.** Gradient descent in a one-dimensional quadratic criterion with different learning rates. If $\eta < \eta_{opt}$, convergence is assured, but training can be needlessly slow. If $\eta = \eta_{opt}$, a single learning step suffices to find the error minimum. If $\eta_{opt} < \eta < 2\eta_{opt}$, the system will oscillate but nevertheless converge, but training is needlessly slow. If $\eta > 2\eta_{opt}$, the system diverges. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

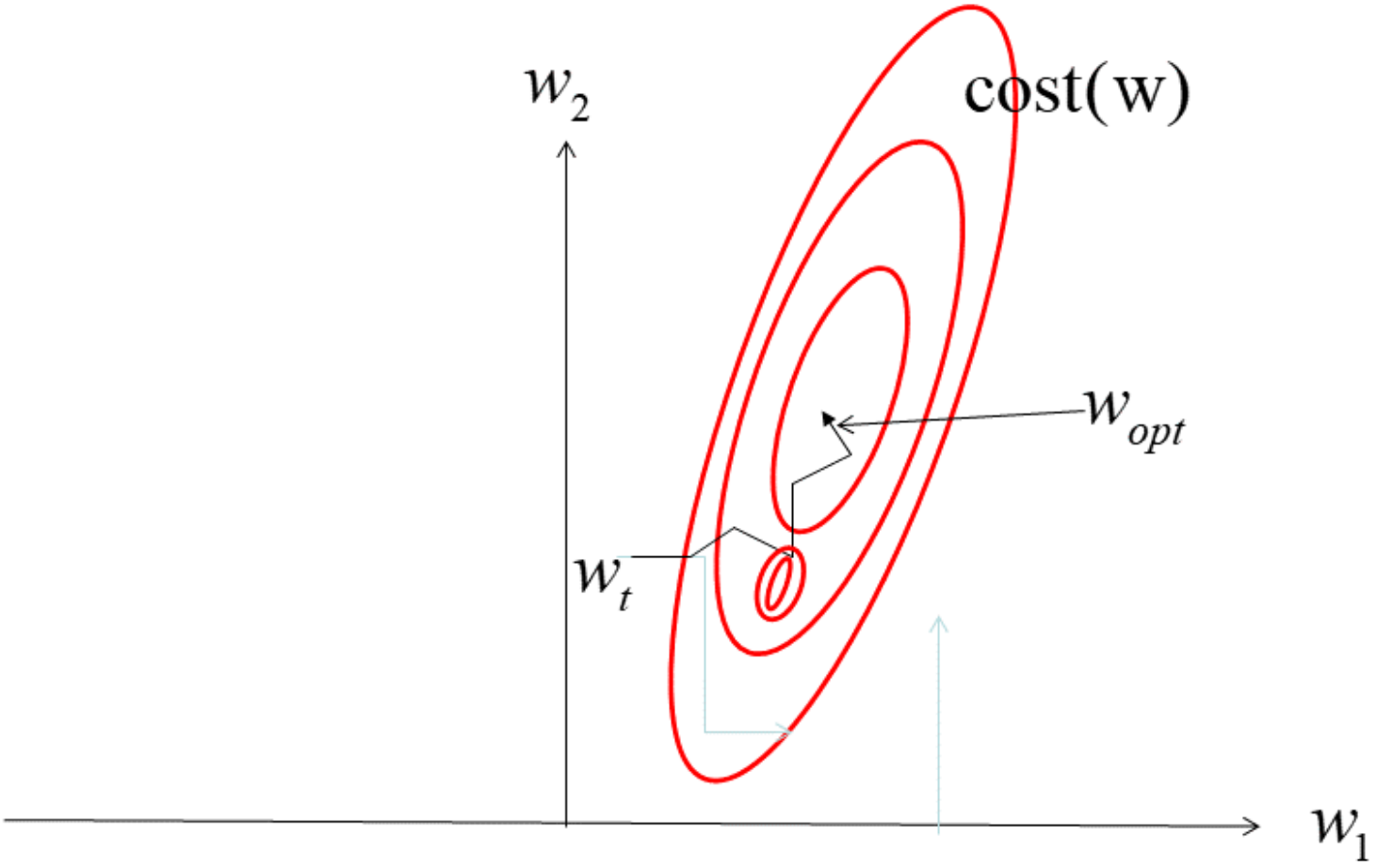
# Local Solutions


cost(w)
$w_t$
$w_{t+1}$
$w_{opt}$
$w$

# Local Solutions


$w_2$
cost(w)
$w_{opt}$
$w_{local}$
$w_t$
$w_1$

## SGD has Fewer Problems with Local Optima

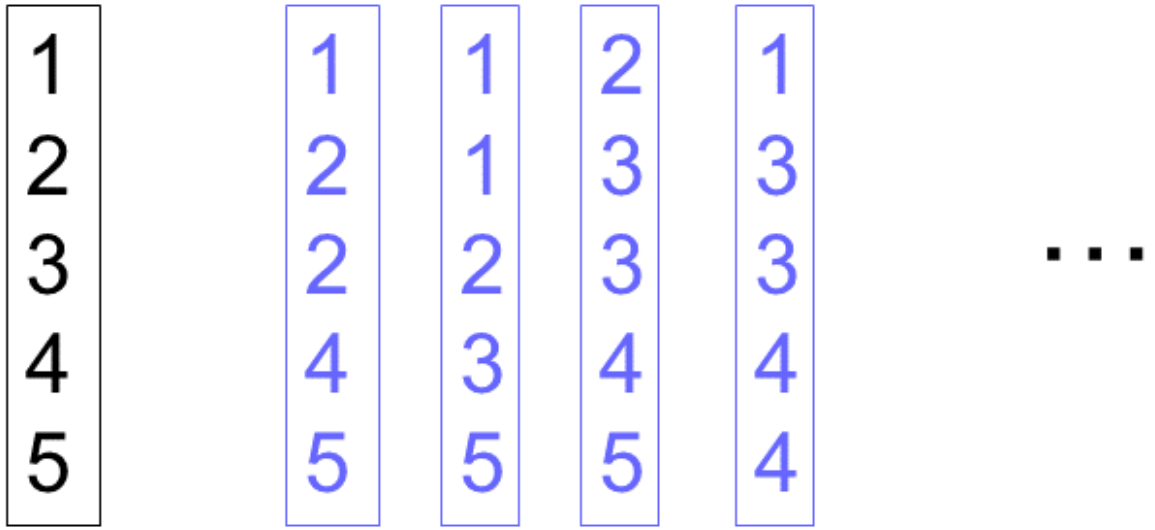

$w_2$
cost(w)
$w_{opt}$
$w_t$
$w_1$

## Dealing with Local Optima

- Restart: Simply repeat training with different initial values and take the best one
- Committee: Repeat training with different initial values and take all of them: for regression, simply average the responses, for classification, take a majority vote

# Bagging

- Bagging: Bootstrap AGGregatING
- Committee as before, but each neural network is trained on a different bootstrap sample of the training data
- Bootstrap sample: From $N$ training data, randomly select $N$ data points *with replacement*. This means one generates a new training data set with again $N$ data points but where some data points of the original set occur more than once and some not at all
- If you apply this committee idea to decision trees you get Random Forests (used to win many Kaggle competitions; now often beaten by deep neural networks)


1
2
3
4
5
1 2 2 4 5
1 1 2 3 5
2 3 3 4 5
1 3 3 4 4
...
Original data
Bootstrap sample

# Conclusion

- Neural Networks are very powerful and show excellent performance
- Training can be complex and slow, but one might say with some justification, that a neural network really learns something: the optimal representation of the data in the hidden layer
- Predictions are fast!
- Neural Networks are universal approximators and have excellent approximation properties
- Key: Basis functions are not predifined by some more or less smart procedure (as in fixed basis function approaches) but the learning algorithm attempts to find the "optimal", problem specific basis functions
- Disadvantage: training a neural network is something of an art; a number of hyper parameters have to be tuned (number of hidden neurons, learning rate, regularization parameters, ...)

- Not all problems can be formulated as a neural network learning problem (but surprisingly many real world problems)
- Disadvantage: A trained neural network finds a local optimum. The solution is not unique, e.g. depends on the initialization of the parameters. Solutions: multiple runs, committee machines
- Note added in 2016; Computing libraries like Theano, TensorFlow, and Keras use symbolic differentiation; you never have to programm backprop: calculating the gradient manually is error prone and tedious for complex structured models

# APPENDIX: Approximation Accuracy of Neural Networks

# Complexity Measure

- How many hidden neurons are required for a certain approximation accuracy?
- Define the complexity measure $C_f$ as

$$\int |\omega||\tilde{f}(\omega)|\, d\omega = C_f,$$

  where $\tilde{f}(\omega)$ is the Fourier transform of $f(\mathbf{x})$. $C_f$ penalizes (assigns a high value to) non smooth functions containing high frequency components!
- The task is to approximate $f(\mathbf{x})$ with a given $C_f$ with a model $f_{\mathbf{w}}(\mathbf{x})$
- The input vector is $\mathbf{x} \in \mathbb{R}^M$, the neural network has $H$ hidden units
- The approximation error $\epsilon$ is the mean squared distance between a target function $f(\mathbf{x})$ and the model $f_{\mathbf{w}}(\mathbf{x})$

$$\epsilon_{AAE} = \int_{B_r} (f(\mathbf{x}) - f_{\mathbf{w}}(\mathbf{x}))^2 \mu(d\mathbf{x}) \qquad (1)$$

$\mu$ is an arbitrary probability distribution on the sphere $B_r = \{\mathbf{x} : |\mathbf{x}| \leq r\}$ with radius $r > 0$

- Barron showed that for each $f(\mathbf{x})$, for which $C_f$ is finite, there is a neural network with one hidden layer, such that for neural networks

$$\epsilon_{AAE} \leq \frac{(2rC_f)^2}{H} \tag{2}$$

- Thus for a good approximation, we might need many hidden units $H$, but the bound does NOT contain the number of inputs $M$!

- Note that for approximations with fixed basis functions, one obtains with $M_\phi$ basis functions

$$\epsilon_{AAE} \propto \frac{1}{M_\phi}^{\frac{1}{M}}$$

- Comment: another way of writing this is that $M_\phi$ must be on the order of $\epsilon_{AAE}^{-M} = (1/\epsilon_{AAE})^M$ (this result was already derived in the lecture on basis functions)

- For important function classes it could be shown that $C_f$ only increases weakly (e.g., proportional) with $M$
- Comment: This is the case when the functions become very smooth in high dimensions (Case III (smooth)), or in Case Ia (sparse basis)
- Comment: But remember, in Case I (curse of dimensionality) also neural networks would suffer from the exponential increase in $H$!
- Quellen: Tresp, V. (1995). Die besonderen Eigenschaften Neuraler Netze bei der Approximation von Funktionen. *Künstliche Intelligenz,* Nr. 4.

  A. Barron. Universal Approximation Bounds for Superpositions of a Sigmoidal Function. *IEEE Trans. Information Theory,* Vol. 39, Nr. 3, 1993.

# The Power of SGD

- The fact that $M_\phi$ is on the order of $\epsilon_{AAE}^{-M}$ indicates that one needs exponentially many fixed basis functions. This might be true both for RBFs and sigmoidal basis functions
- Assume the function class is such that it can be approximated by $H <<< \epsilon_{AAE}^{-M}$ basis functions. In other words: almost all basis functions get zero weights. If we work with fixed basis functions, this does not really help since we still have to initialize with $\mathcal{O}(1/\epsilon_{AAE}^{-M})$ basis functions
- On the other hand, if we know $H$, we can design a model with only $H$ RBFs or sigmoidal basis functions. We make the assumption that SGD finds the globally best solution for the all parameters (in the case of the sigmoidal basis functions, this would include the weights from the input layer to the hidden layer; in the case of RBFs, this would involve an adaptation of basis function centers $\mathbf{c}$ and widths $s$). The number of basis functions is then independent of $M$ and only depends on $H$.
- So the sigmoidal shape might not be as important as the use of SGD