Neural Networks

Volker Tresp

Summer 2015

Introduction

- The performance of a classifier or a regression model critically depends on the choice of appropriate basis functions
- The problem with generic basis functions such as polynomials or RBFs is, that the number of basis functions required for a good approximation increases rapidly with dimensions (“curse of dimensionality”)
- It should be possible to “learn” the appropriate basis functions
- This is the basic idea behind Neural Networks
- Neural Networks use particular forms of basis functions: sigmoidal (neural) basis functions (or similar ones)

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

- As before, the output of a neural network (respectively the activation function $h(x)$, in the case of a linear classifier) is the weighted sum of basis functions

$$\hat{y} = f_{\mathbf{w},V}(\mathbf{x}) = \sum_{h=0}^{H-1} w_h \text{sig}(\mathbf{x}^T \mathbf{v}_h)$$

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

Neural Basis Functions

- Special form of the basis functions

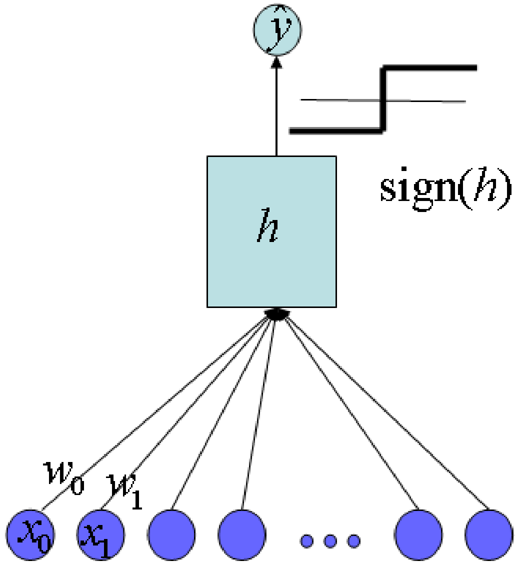
$$z_h = \text{sig}(\mathbf{x}^T \mathbf{v}_h) = \text{sig}\left(\sum_{j=0}^{M-1} v_{h,j} x_j\right)$$

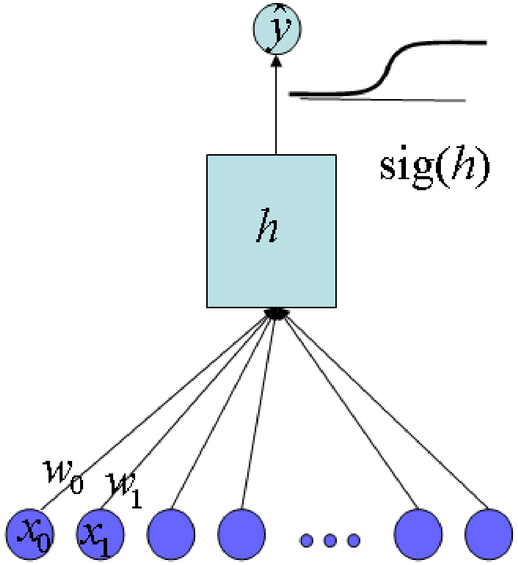
using the *logistic function*

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

- 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 x_i as

$$h(\mathbf{x}) = \sum_{j=0}^{M-1} w_j x_j$$

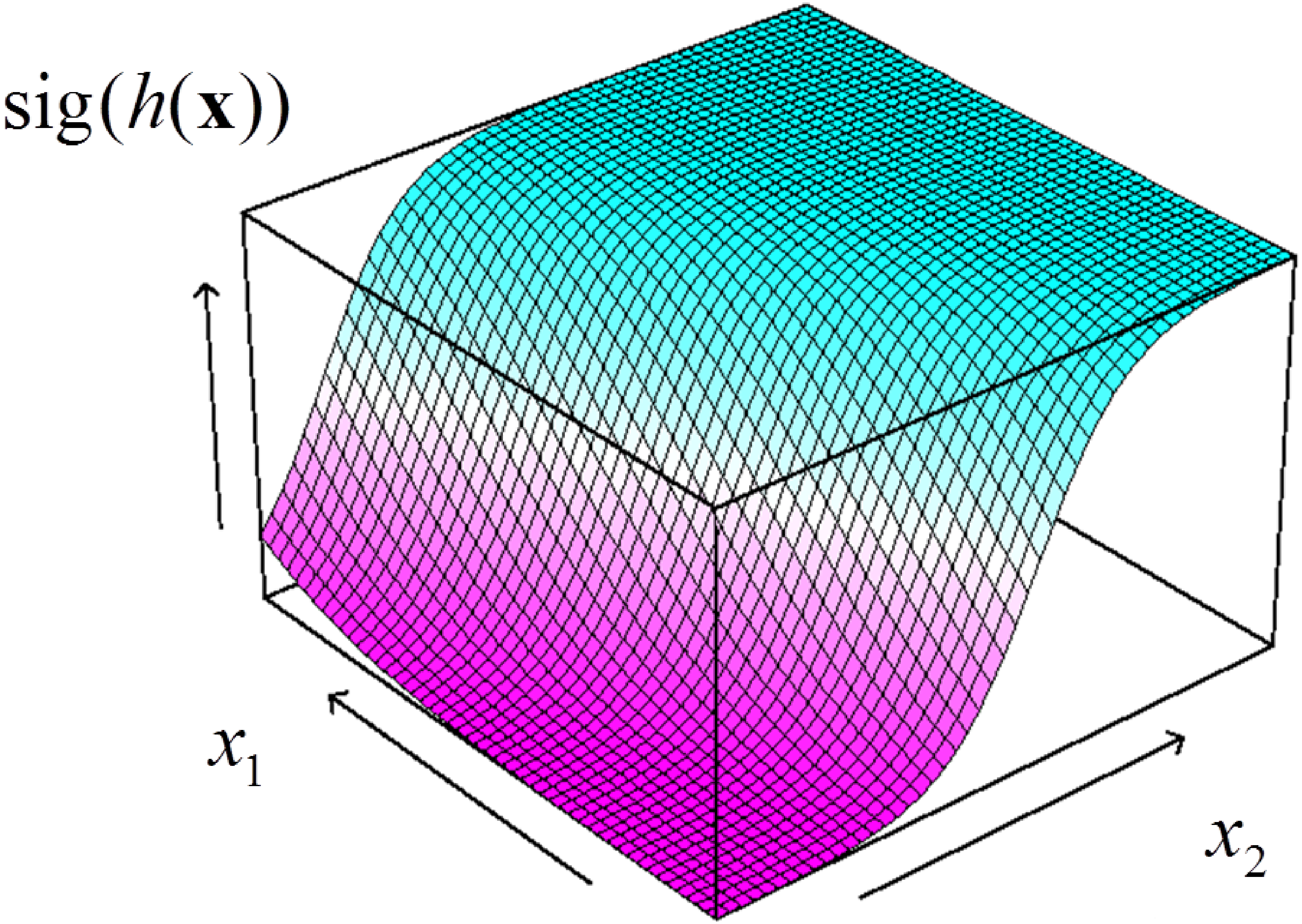
(note: $x_0 = 1$ is a constant input, so that w_0 corresponds to the bias)

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

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

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

Transfer Function



Separating Hyperplane

- Definition of the hyperplane

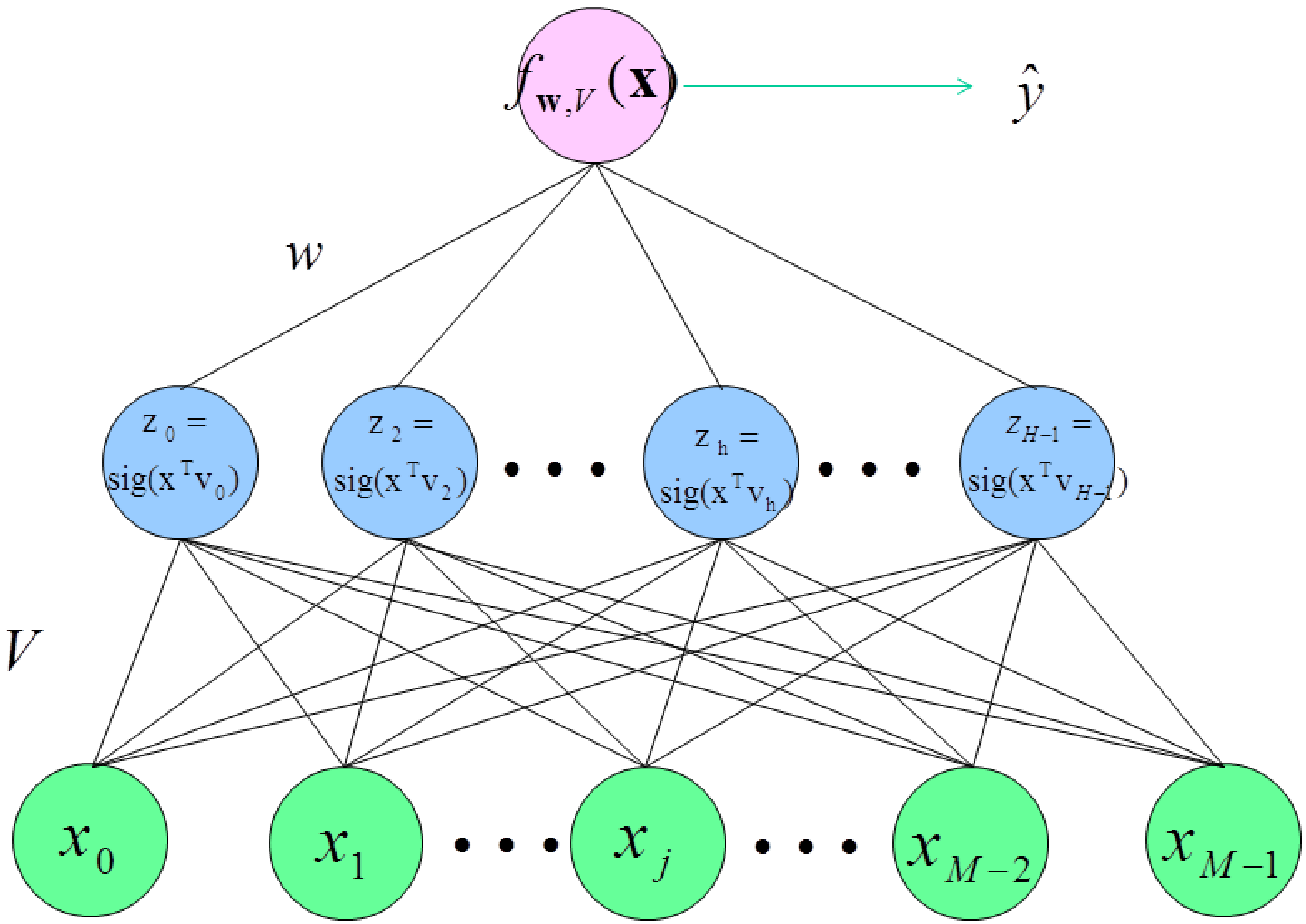
$$\mathrm{sig}\left(\sum_{j=0}^{M-1} v_{h,j}x_j\right) = 0.5$$

which means that:

$$\sum_{j=0}^{M-1} v_{h,j}x_j = 0$$

- “carpet over a step”

Architecture of a Neural Network



Variants

- For a neural network classifier (binary) apply the sigmoid transfer function to the output neuron, and calculate

$$\widehat{y} = \mathsf{sig}(f_{\mathbf{w},V}(\mathbf{x})) = \mathsf{sig}(\mathbf{z}^T\mathbf{w})$$

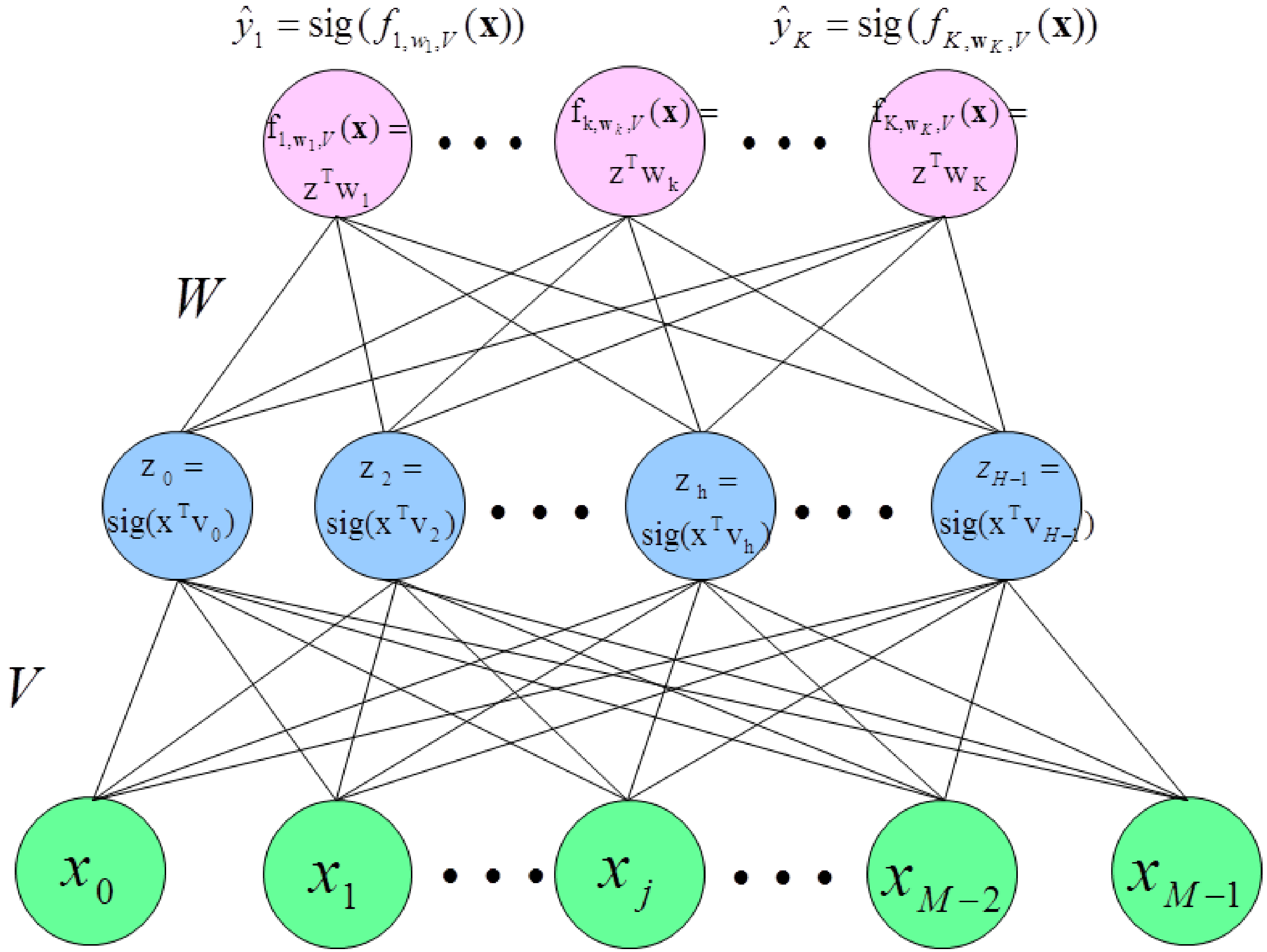
- For multi class tasks, one uses several output neurons. For example, to classify K digits

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

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

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

Architecture of a Neural Network for Several Classes


$\hat{y}_1 = \text{sig}(f_{1,w_1,V}(\mathbf{x}))$
$\hat{y}_K = \text{sig}(f_{K,\mathbf{w}_K,V}(\mathbf{x}))$
$f_{1,\mathbf{w}_1,V}(\mathbf{x}) = z^T w_1$
$f_{k,\mathbf{w}_k,V}(\mathbf{x}) = z^T w_k$
$f_{K,\mathbf{w}_K,V}(\mathbf{x}) = z^T w_K$
W
$z_0 = \text{sig}(x^T v_0)$
$z_2 = \text{sig}(x^T v_2)$
$z_h = \text{sig}(x^T v_h)$
$z_{H-1} = \text{sig}(x^T v_{H-1})$
V
x_0
x_1
x_j
x_{M-2}
x_{M-1}

Learning with Several Outputs

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

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

with $\mathsf{cost}(\mathbf{x}_i, W, V) = \sum_{k=1}^{K} (y_{i,k} - \mathsf{sig}(f_{k,\mathbf{w}_k,V}(\mathbf{x}_i))^2$

- 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

$$\delta_{i,k} = \text{sig}'(\mathbf{z}_i^T \mathbf{w}_k)(y_{i,k} - \text{sig}(f_{k,\mathbf{w}_k,V}(\mathbf{x}_i))))$$

  is the back propagated error signal (error back propagation).

- 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

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

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

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

- 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”: biological plausible

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

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

- The learning rules change to (with *weight decay term*)

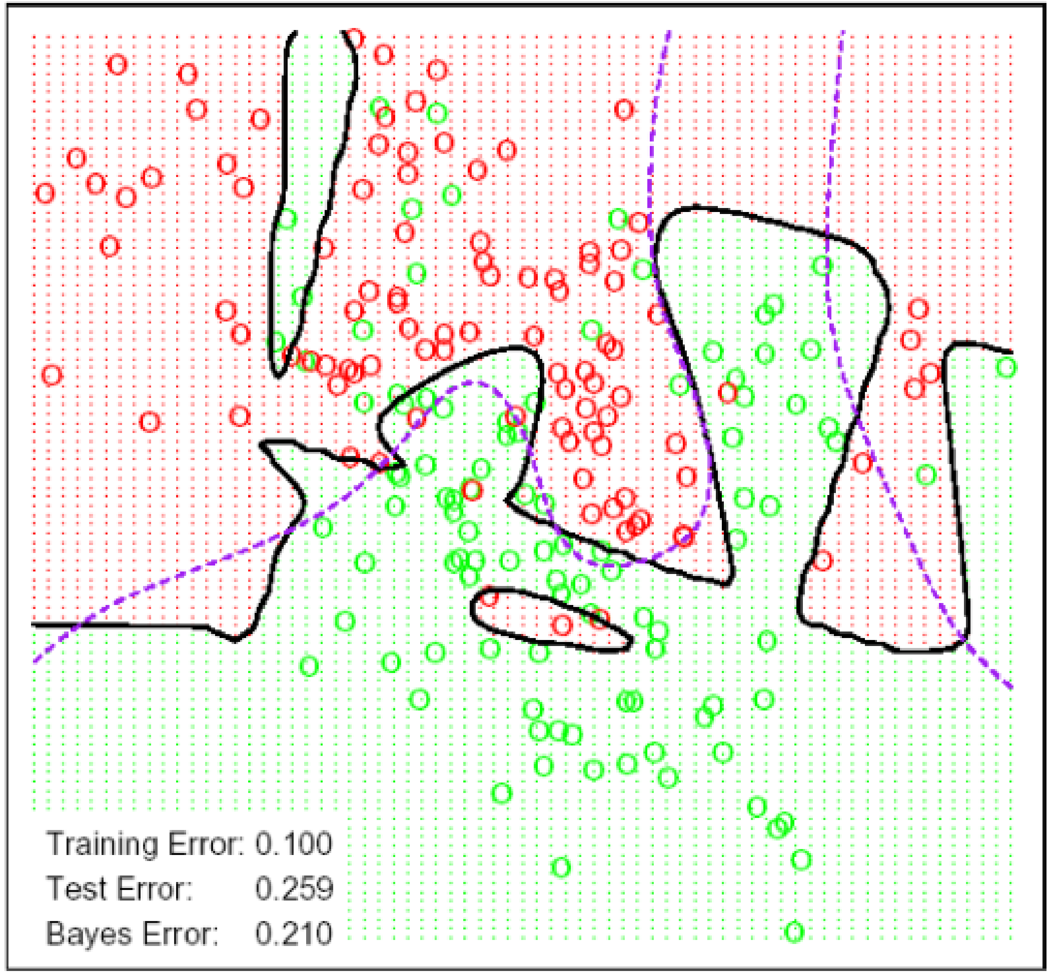
$$w_{k,h} \leftarrow w_{k,h} + \eta \left(\delta_{i,k} z_{i,h} - \lambda_1 w_{k,h}\right)$$

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

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

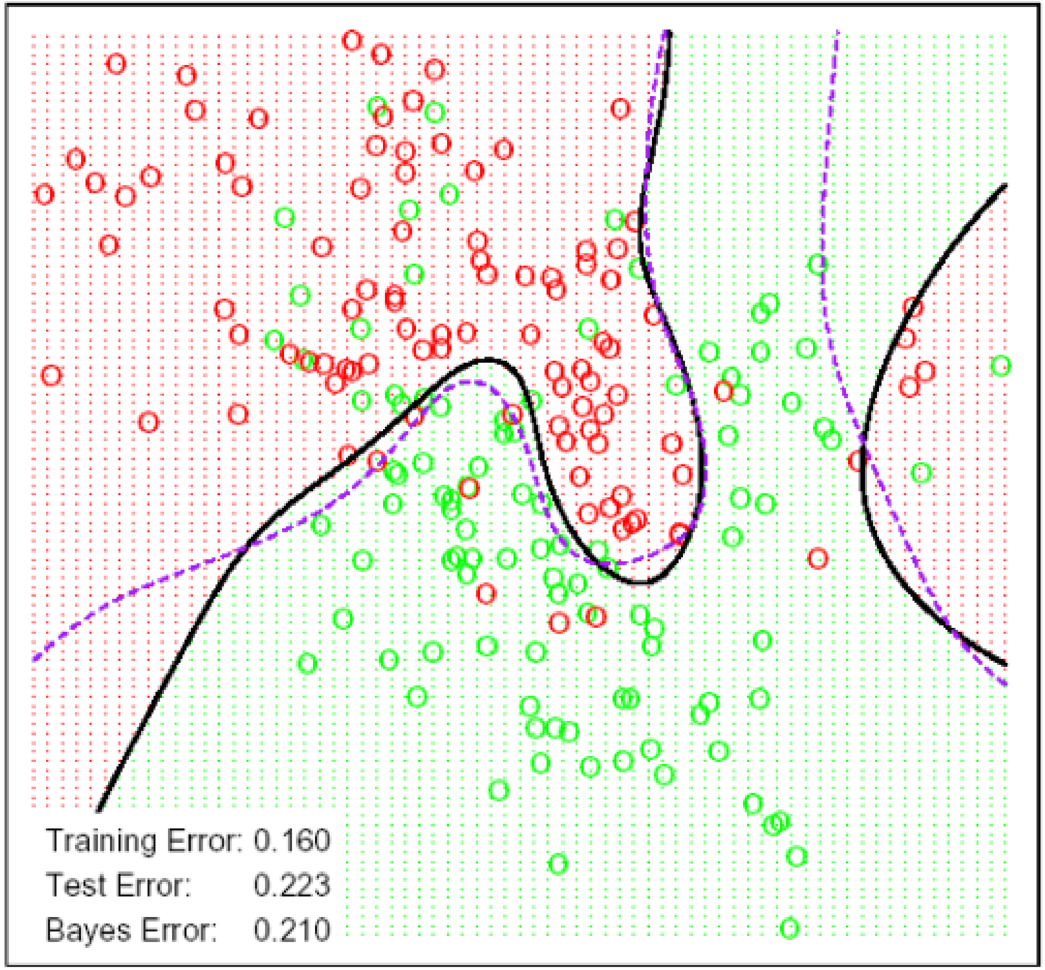


Same Example with Regularization

- With regularization ($\lambda_1 = \lambda_2 = 0.2$) the separating plane is closer to the true class boundaries

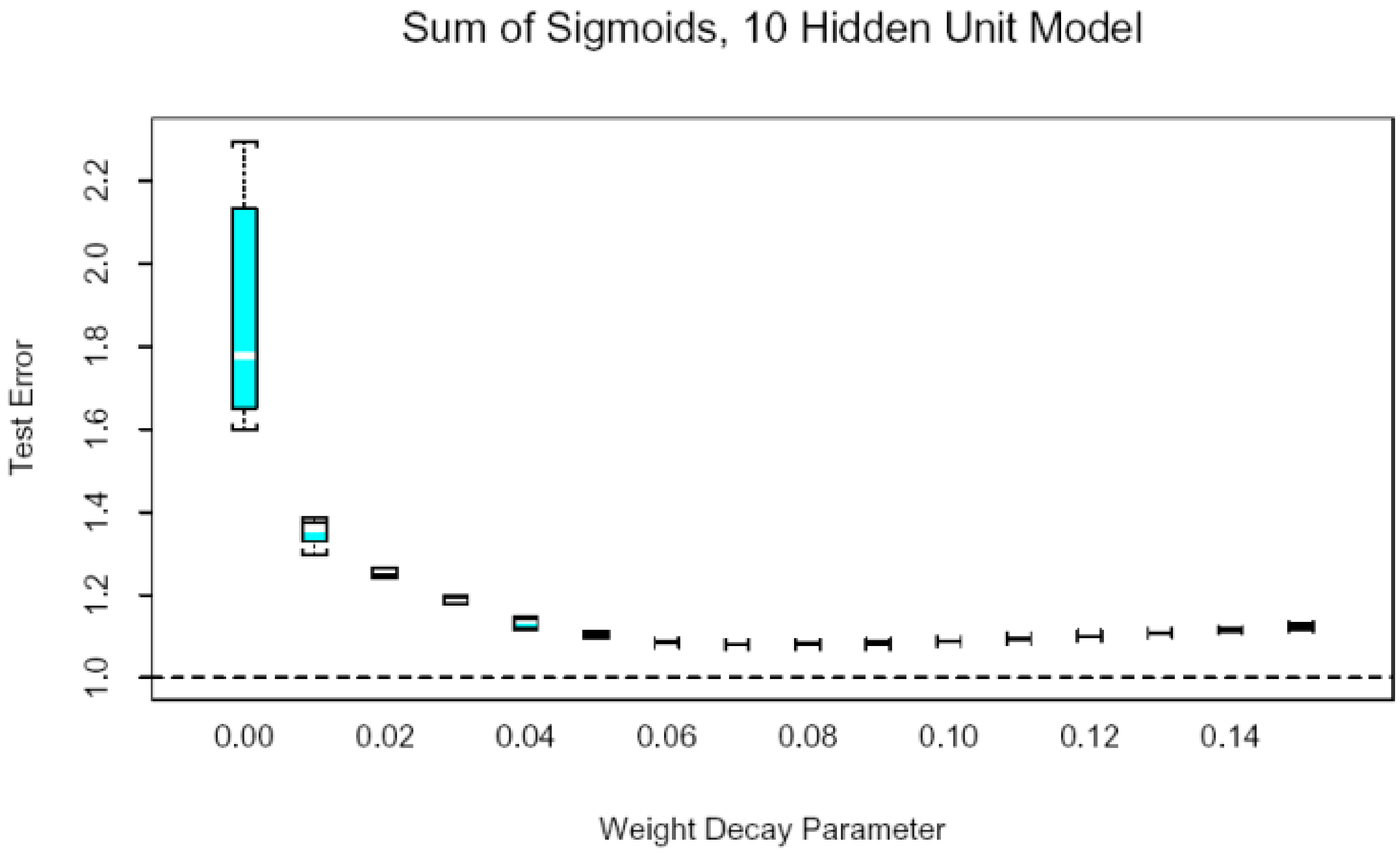
- The training error is smaller with the unregularized network, the test error is smaller with the regularized network

Neural Network - 10 Units, Weight Decay=0.02



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 variation in the test error decreases with increasing regularization parameter



Sum of Sigmoids, 10 Hidden Unit Model

Variations

- Use more than one hidden layer (see deep learning)
- Use $\mathsf{tanh}(arg) \in (-1, 1)$ instead of sig$(arg) \in (0, 1)$
- For the $\mathsf{tanh}(arg)$, use targets $y \in \{-1, 1\}$, instead of $y \in \{0, 1\}$
- Instead of the sum-squared-error cost function, use the cross-entropy cost function (logistic cost function) with $y_k \in \{0, 1\}$

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

with

$$\mathsf{cost}(\mathbf{x}_i, W, V) = \sum_{k=1}^{K} -y_{i,k} \log \widehat{y}_{i,k} - (1 - y_{i,k}) \log(1 - \widehat{y}_{i,k})$$

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

with

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

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

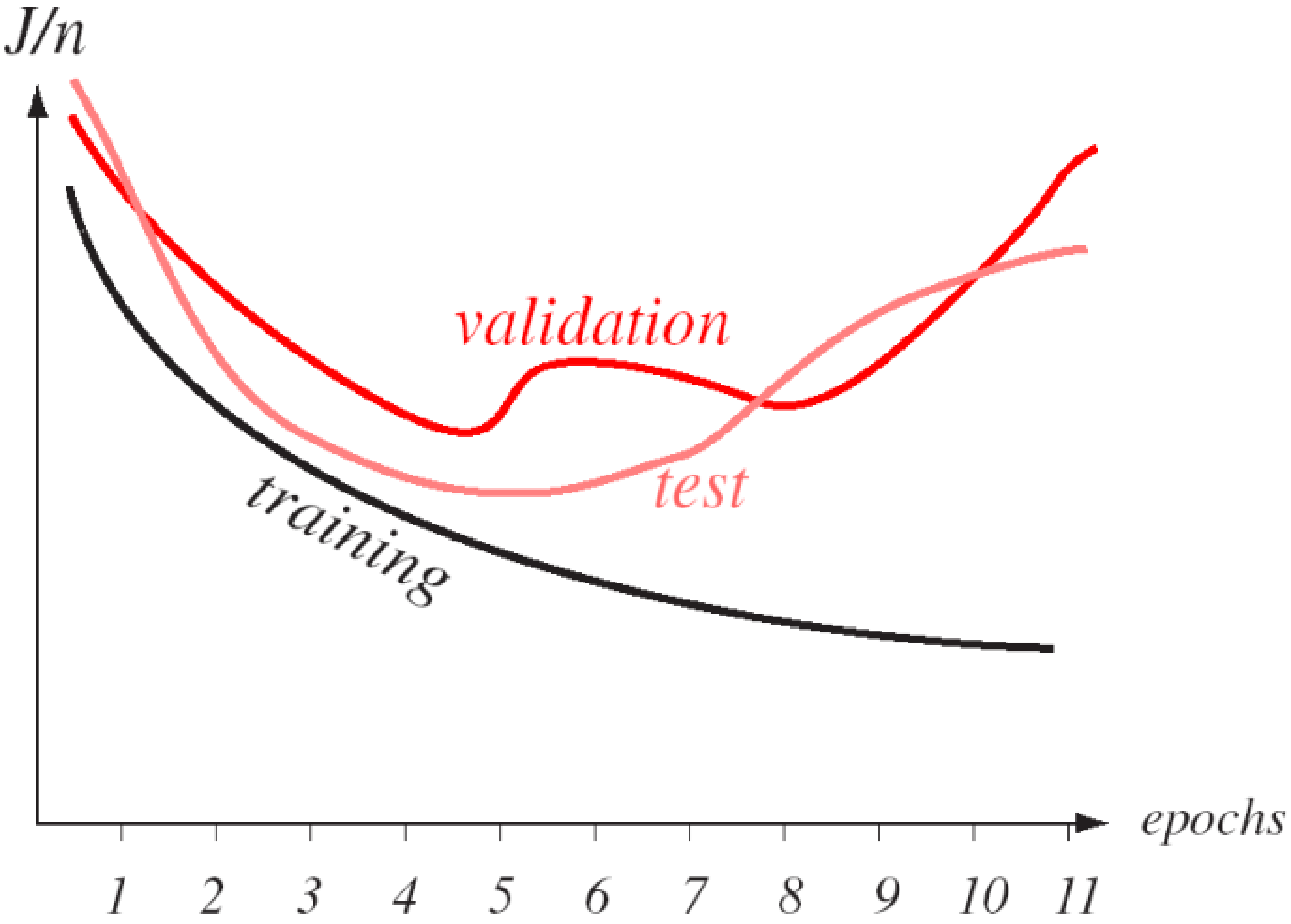
- Compare with the squared loss:

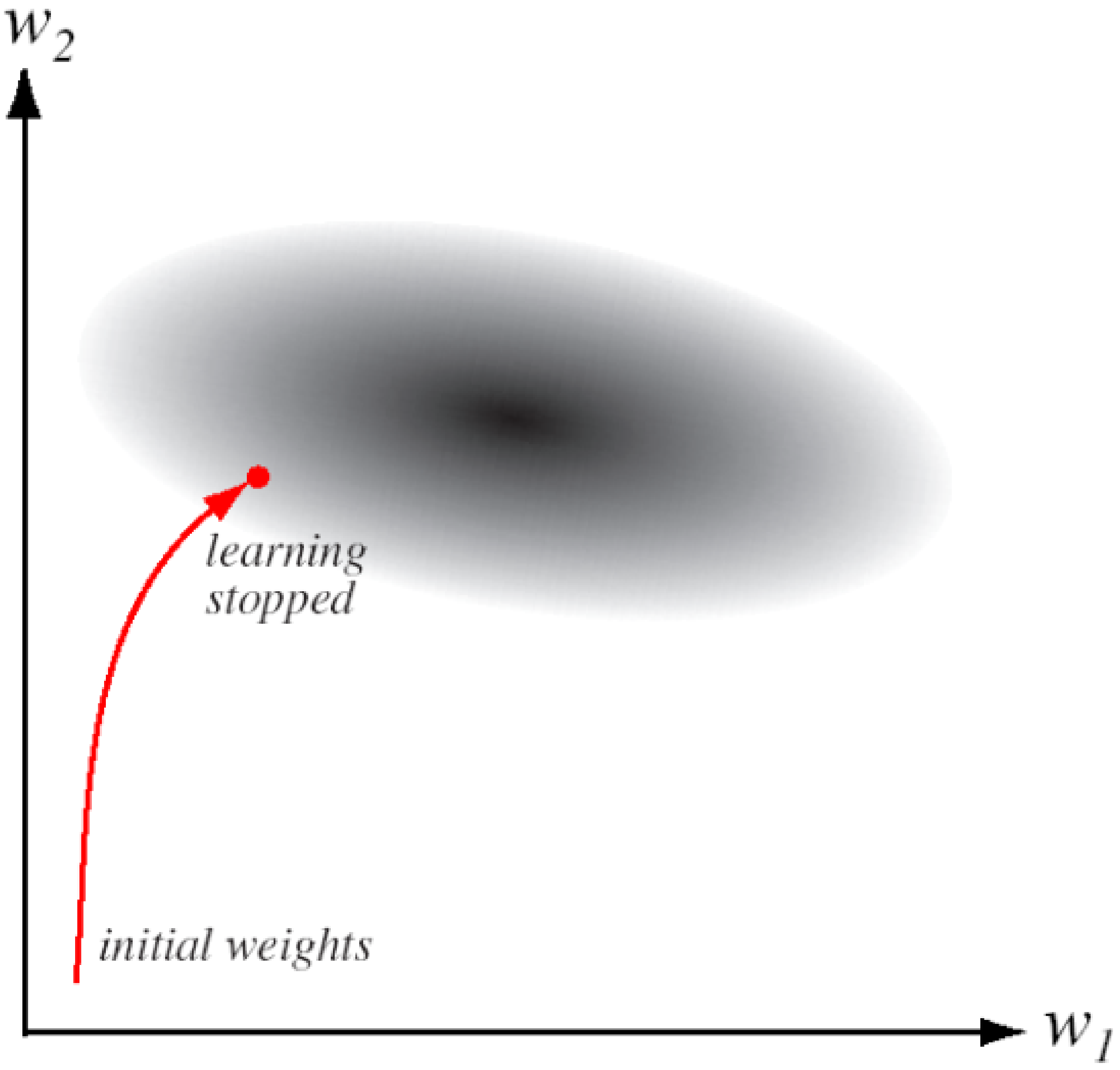
$$\delta_{i,k} = \widehat{y}_{i,k}(1 - \widehat{y}_{i,k})(y_{i,k} - \widehat{y}_{i,k})$$

Thus with cross-entropy, the gradient does not become zero of the prediction completely agrees or disagrees with the target

Regularization with Stopped-Training

- In the next picture you can see typical behavior of training error and test error as a function of training iterations
- As expected the training error steadily decreases with the training time
- 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 η

- Convergence can be influenced by the learning rate η
- Next figure: if the learning rate is too small, convergence can be vary 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”)

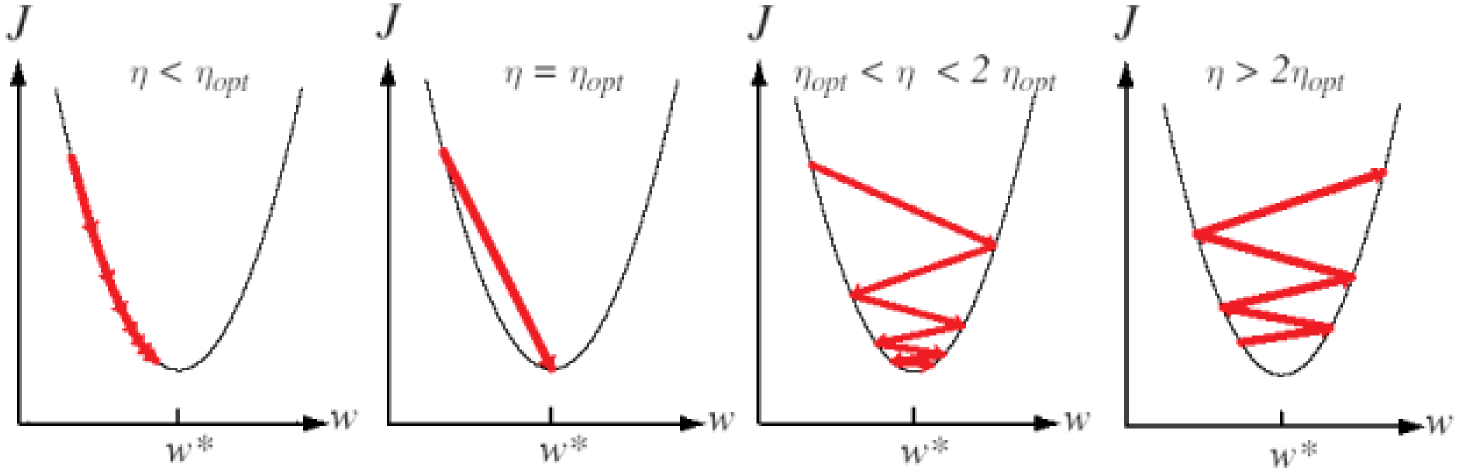
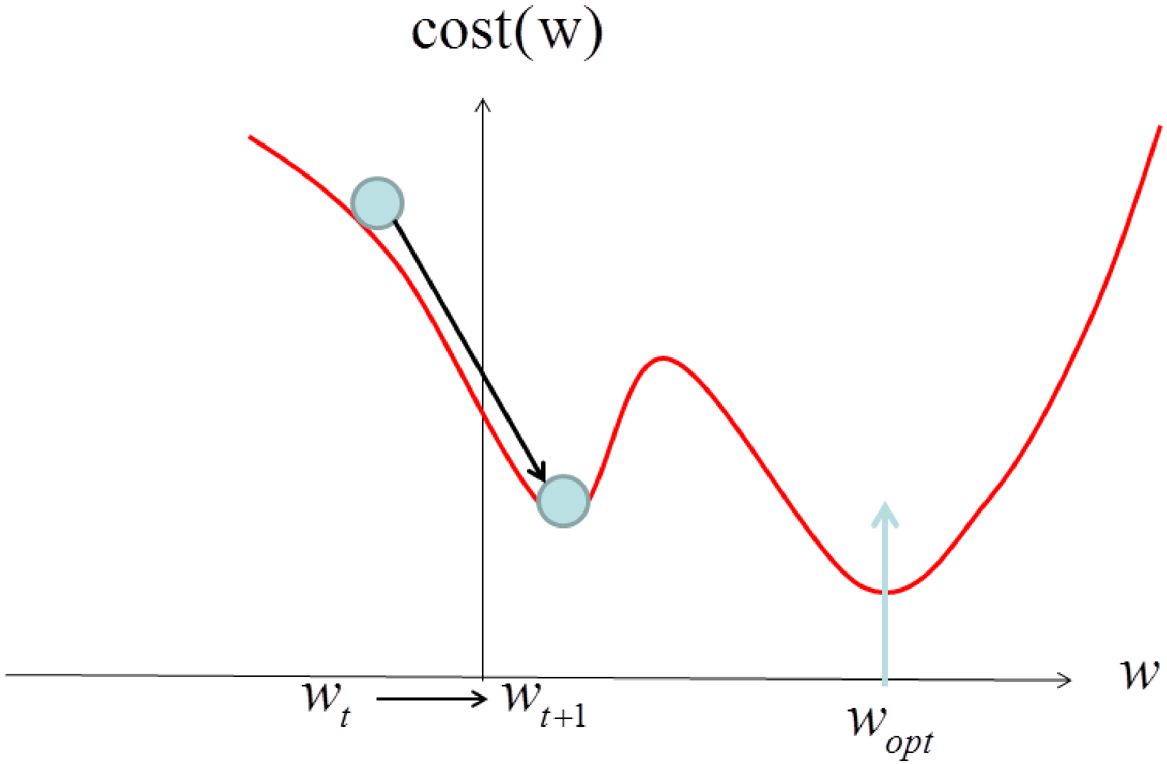
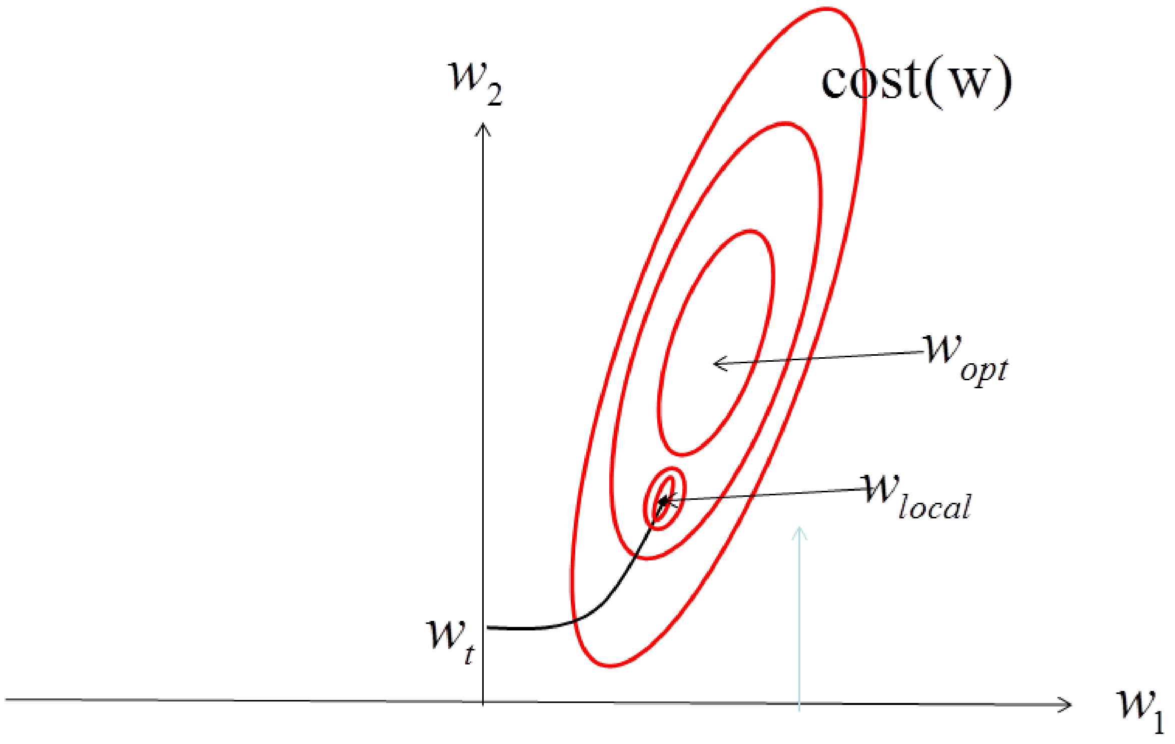


FIGURE 6.16. Gradient descent in a one-dimensional quadratic criterion with different learning rates. If $\eta < \eta_{opt}$, convergence is assured, but training can be needlessly slow. If $\eta = \eta_{opt}$, a single learning step suffices to find the error minimum. If $\eta_{opt} < \eta < 2\eta_{opt}$, 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.

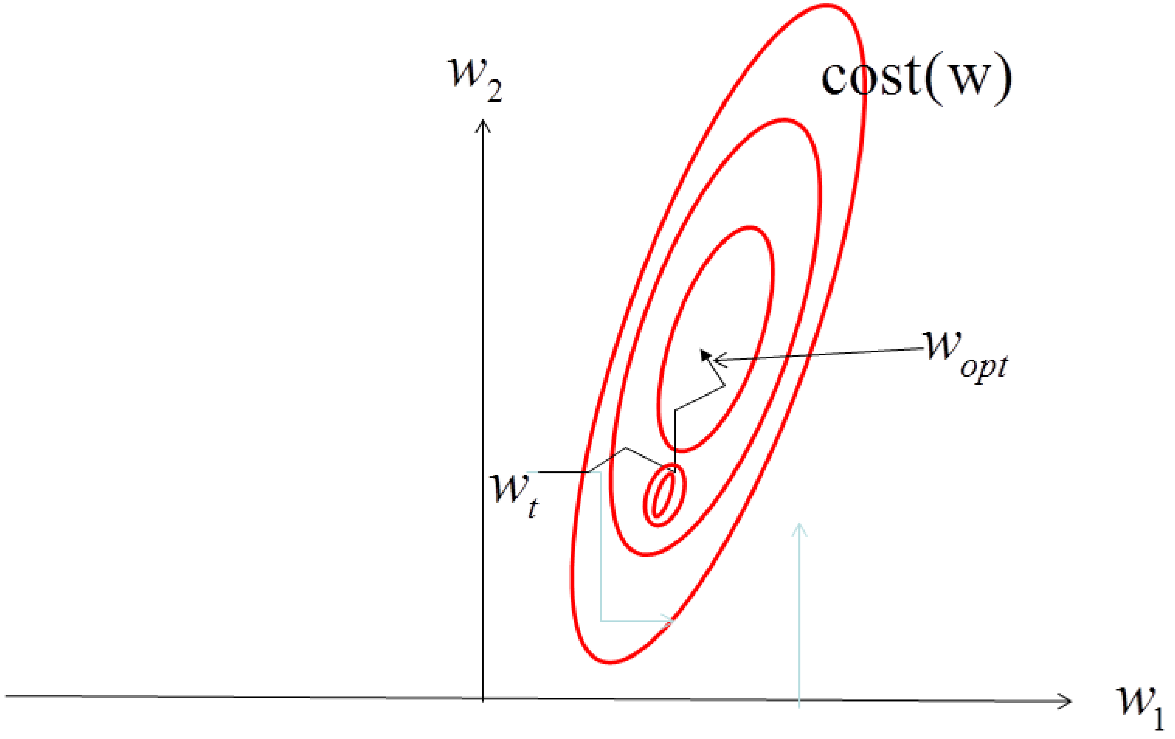
Local Solutions



Local Solutions



SGD has Fewer Problems with Local Optima

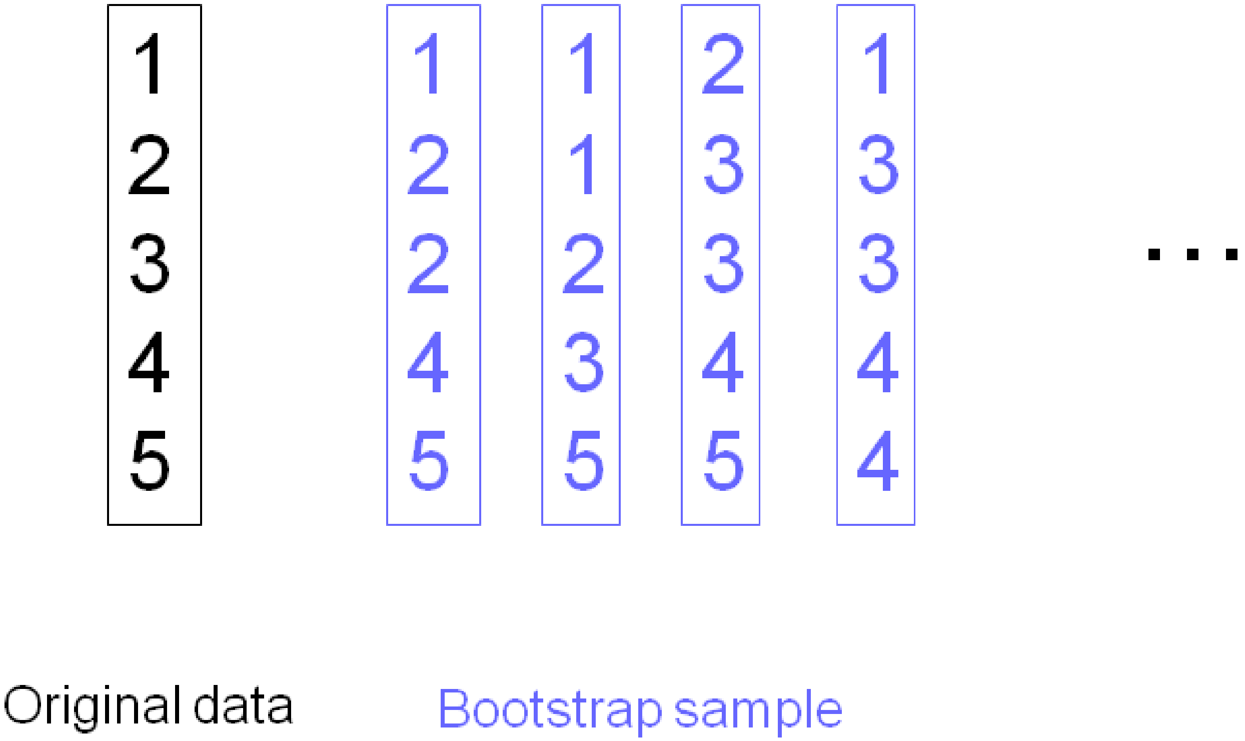


Dealing with Local Optima

- 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 the 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 (wins many Kaggle competitions; now deep neural networks seem to work better)



1	1	1	2	1	
2	2	1	3	3	
3	2	2	3	3	. . .
4	4	3	4	4	
5	5	5	5	4	

Original data

Bootstrap sample

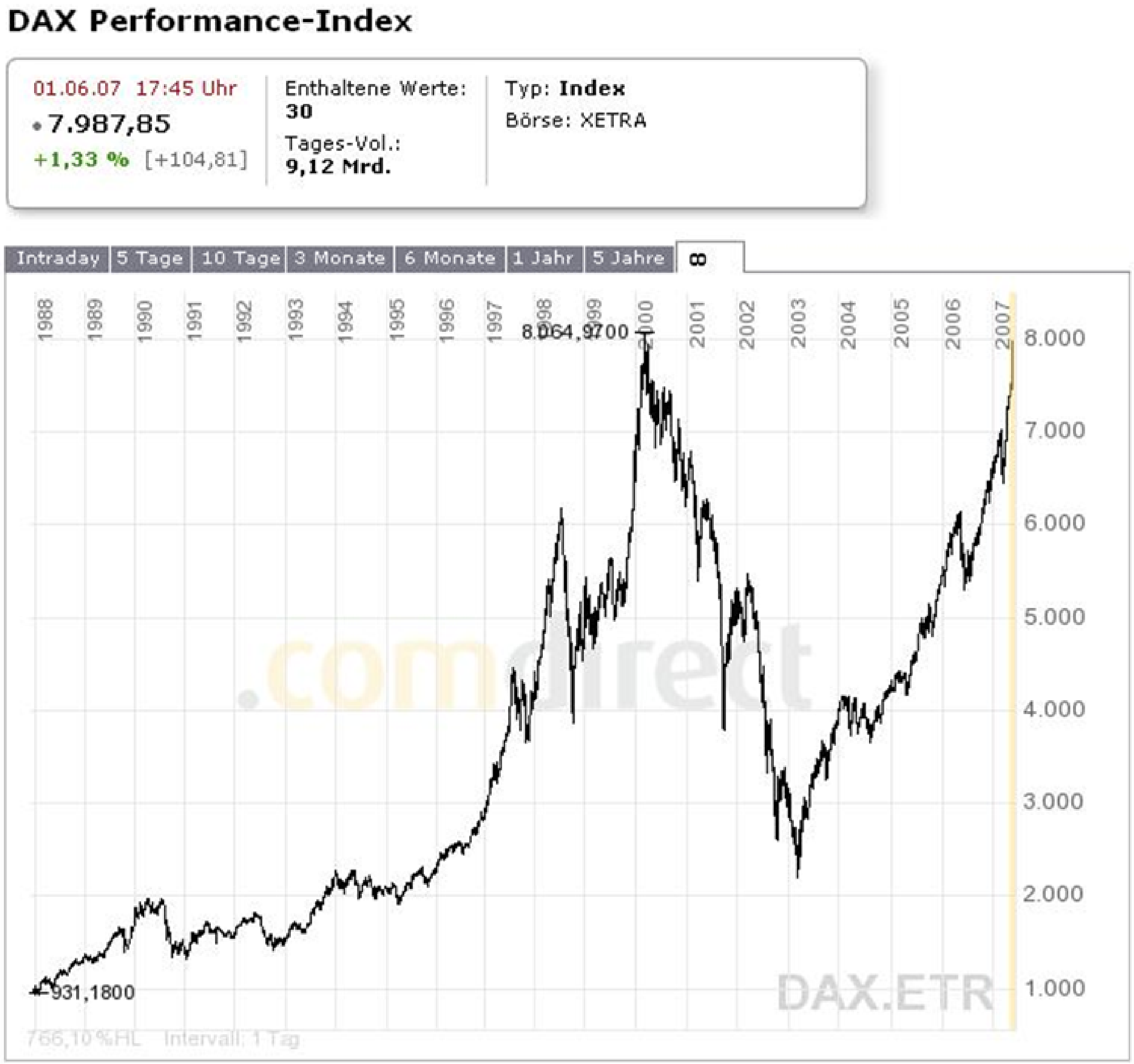
Conclusion

- 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
- 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

Modelling of Time Series

- The next figure shows a tim series (DAX)
- Other interesting time-series: energy prize, energy consumption, gas consumption, copper prize, ...



DAX Performance-Index

01.06.07 17:45 Uhr

• 7.987,85

+1,33 % [+104,81]

Enthaltene Werte: **30**

Tages-Vol.: **9,12 Mrd.**

Typ: **Index**

Börse: XETRA

Neural Networks for Time-Series Modelling

- Let $z_t, t = 1, 2, \ldots$ be the time-discrete time-series of interest (example: DAX)
- Let $x_t, t = 1, 2, \ldots$ denote a second time-series, that contains information on z_t (Example: Dow Jones)
- For simplicity, we assume that both z_t and x_t are scalar. The goal is the prediction of the next value of the time-series
- We assume a system of the form

$$z_t = f(z_{t-1}, \ldots, z_{t-T}, x_{t-1}, \ldots, x_{t-T}) + \epsilon_t$$

with i.i.d. random numbers $\epsilon_t, t = 1, 2, \ldots$ which model unknown disturbances.

Neural Networks for Time-Series Modelling (cont'd)

- We approximate, using a neural network,

$$f(z_{t-1}, \dots, z_{t-T}, x_{t-1}, \dots, x_{t-T})$$

$$\approx f_{\mathbf{w},V}(z_{t-1}, \dots, z_{t-T}, x_{t-1}, \dots, x_{t-T})$$

  and obtain the cost function

$$\mathrm{cost}(\mathbf{w}, V) = \sum_{t=1}^{N} (z_t - f_{\mathbf{w},V}(z_{t-1}, \dots, z_{t-T}, x_{t-1}, \dots, x_{t-T}))^2$$

- The neural network can be trained as before with simple back propagation *if in training all z_t and all x_t are known*!

- This is a NARX model: **N**onlinear **A**uto **R**egressive Model with e**x**ternal inputs. Another name: TDNN (time-delay neural network)

Recurrent Neural Network

- If z_t cannot be measure directly (e.g., if there is noise on the measurements) we can model

$$z_t = f(z_{t-1}, \ldots, z_{t-T}, x_{t-1}, \ldots, x_{t-T}) + \epsilon_t$$

$$y_t = z_t + \delta_t$$

- Only the y_t are measured
- Now the inputs to the neural network are not really known in training (they are hidden, latent) and an estimate $\widehat{y}_t$ is influenced by measurements x_t from far in the past
- Recurrent Neural Network: in prediction the neural network uses its own past predictions as inputs

Generic Recurrent Neural Network Architecture

- Consider a feedforward neural network where there are connections between the hidden units

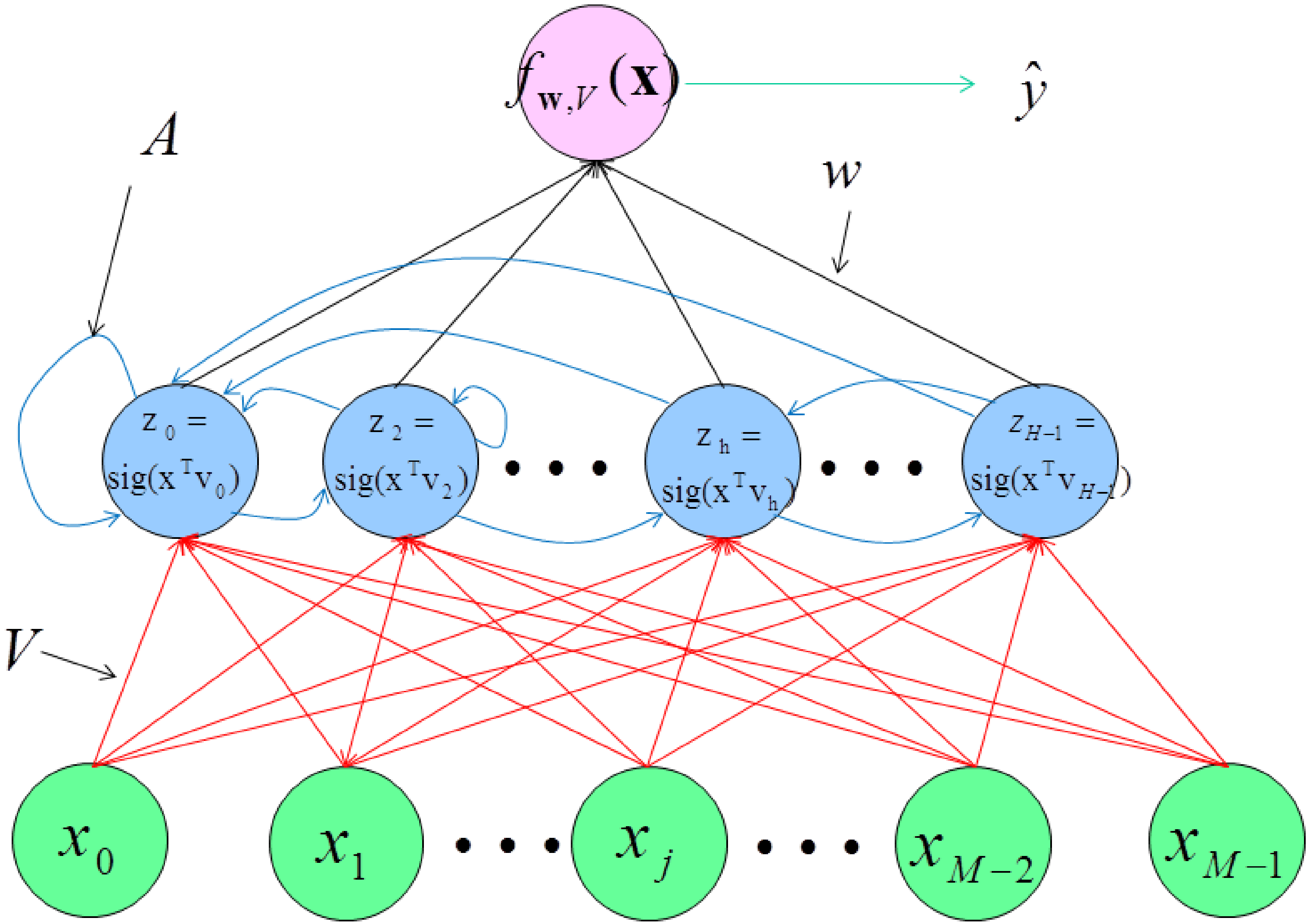
$$z_{t,h} = \mathrm{sig}(\mathbf{z}_{t-1}^T \mathbf{a}_h + \mathbf{x}_{t-1}^T \mathbf{v}_h)$$

  and, as before,

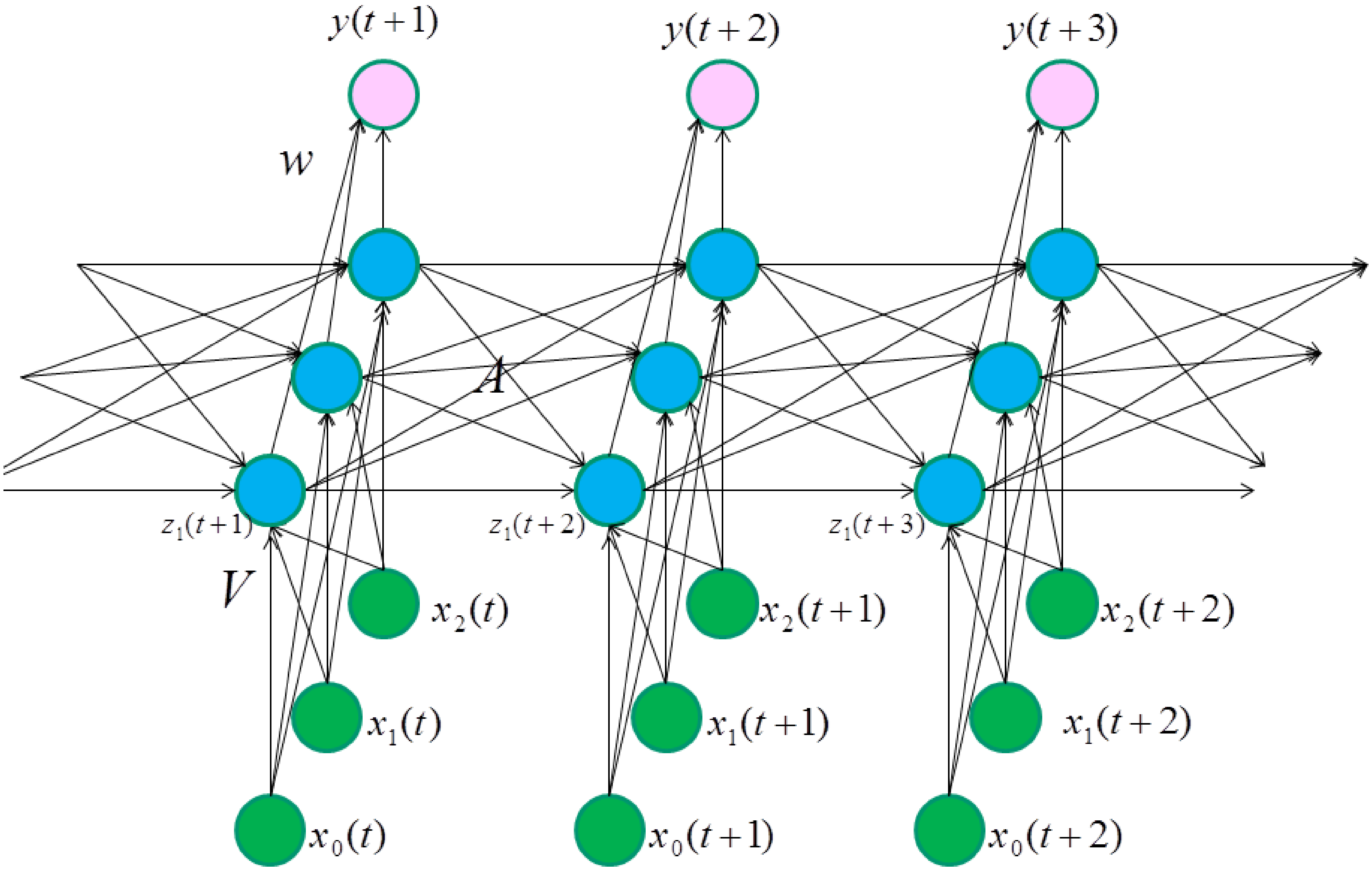
$$\hat{y}_t = \mathrm{sig}(\mathbf{z}_t^T \mathbf{w})$$

- Here, $\mathbf{z}_t = (z_{t,1}, z_{t,2}, \ldots, z_{t,H})^T$, $\mathbf{x}_t = (x_{t,0}, x_{t,1}, \ldots, x_{t,M-1})^T$
- In Recurrent Neural Networks (RNNs) the next state of the neurons in the hidden layer depends on their last state and both are not directly measured
- $\mathbf{a}_h, \mathbf{w}, \mathbf{v}_h$ are weight vectors
- The next figure shows an example. Only some of the recurrent connections are shown (blue). The blue and red connection also model a time lag. Without recurrent connections ($\mathbf{a}_h = 0, \forall h$) and without time lag, we obtain a regular feedforward network

- Note, that a recurrent neural network has an internal memory


$f_{\mathbf{w},V}(\mathbf{x})$
$\hat{y}$
A
w
$z_0 = \text{sig}(x^T v_0)$
$z_2 = \text{sig}(x^T v_2)$
$z_h = \text{sig}(x^T v_h)$
$z_{H-1} = \text{sig}(x^T v_{H-1})$
V
x_0
x_1
x_j
x_{M-2}
x_{M-1}

A Recurrent Neural Network Architecture unfolded in Time



$y(t+1)$
$y(t+2)$
$y(t+3)$
w
A
$z_1(t+1)$
$z_1(t+2)$
$z_1(t+3)$
V
$x_2(t)$
$x_1(t)$
$x_0(t)$
$x_2(t+1)$
$x_1(t+1)$
$x_0(t+1)$
$x_2(t+2)$
$x_1(t+2)$
$x_0(t+2)$

Training of Recurrent Neural Network Architecture

- Backpropagation through time (BPTT): essentially backpropagation applied to the unfolded network; note that all that happened before time t influences $\widehat{y}_t$, so the error needs to be backpropagated backwards in time, in principle until the beginning of the experiment! In reality, one typically truncates the gradient calculation (review in: Werbos (1990))
- Real-Time Recurrent Learning (RTRL) (Williams and Zipser (1989))
- Time-Dependent Recurrent Back-Propagation: learning with continuous time (Lagrangian approach) (Pearlmutter 1998)

Echo-State Network

- Recurrent Neural Networks are notoriously difficult to train
- A simple alternative is to initialize A and V randomly (according to some recipe) and only train $\mathbf{w}$, e.g., with the ADALINE learning rule
- This works surprisingly well and is done in the Echo-State Network (ESN)

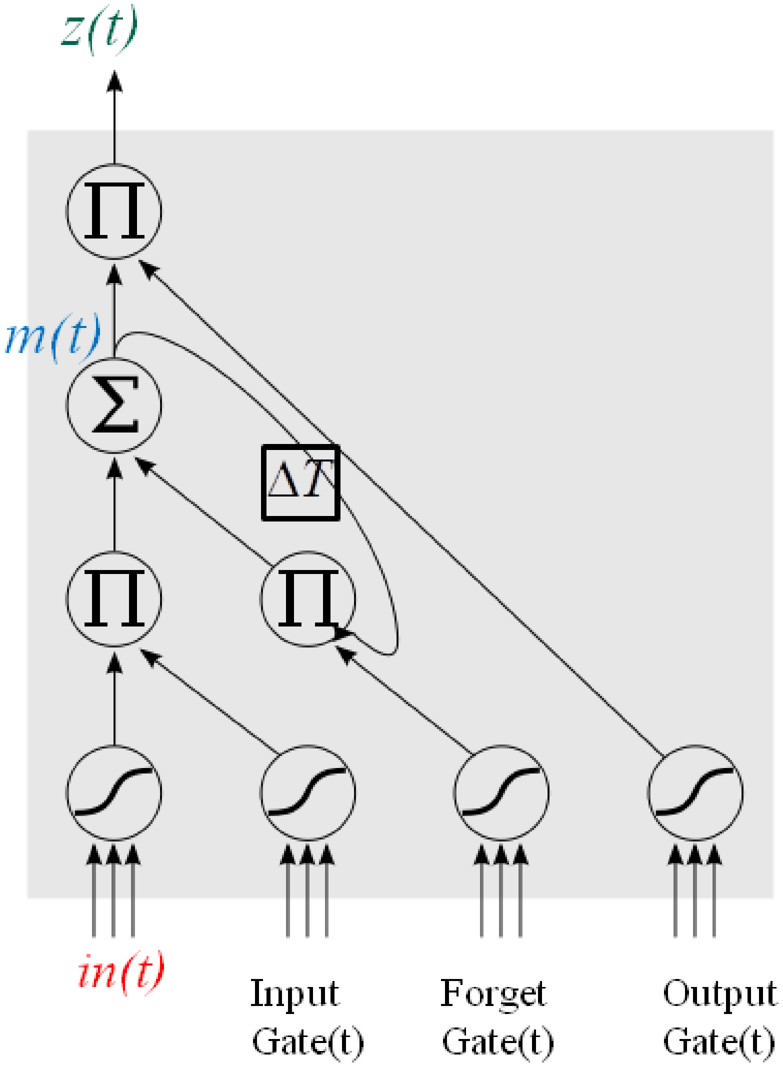
Long Short Term Memory (LSTM)

- As a recurrent structure the Long Short Term Memory (LSTM) approach has been very successful

- Basic idea: at time T a newspaper announces that the Siemens stock is labelled as “buy”. This information will influence the development of the stock in the next days. A standard RNN will not remember this information for very long. One solution is to define an extra input to represent that fact and that is on as along as “buy” is valid. But this is handcrafted and does not exploit the flexibility of the RNN. A flexible construct which can hold the information is a long short term memory (LSTM) block.

- The LSTM was used very successful for reading handwritten text

Long Short Term Memory

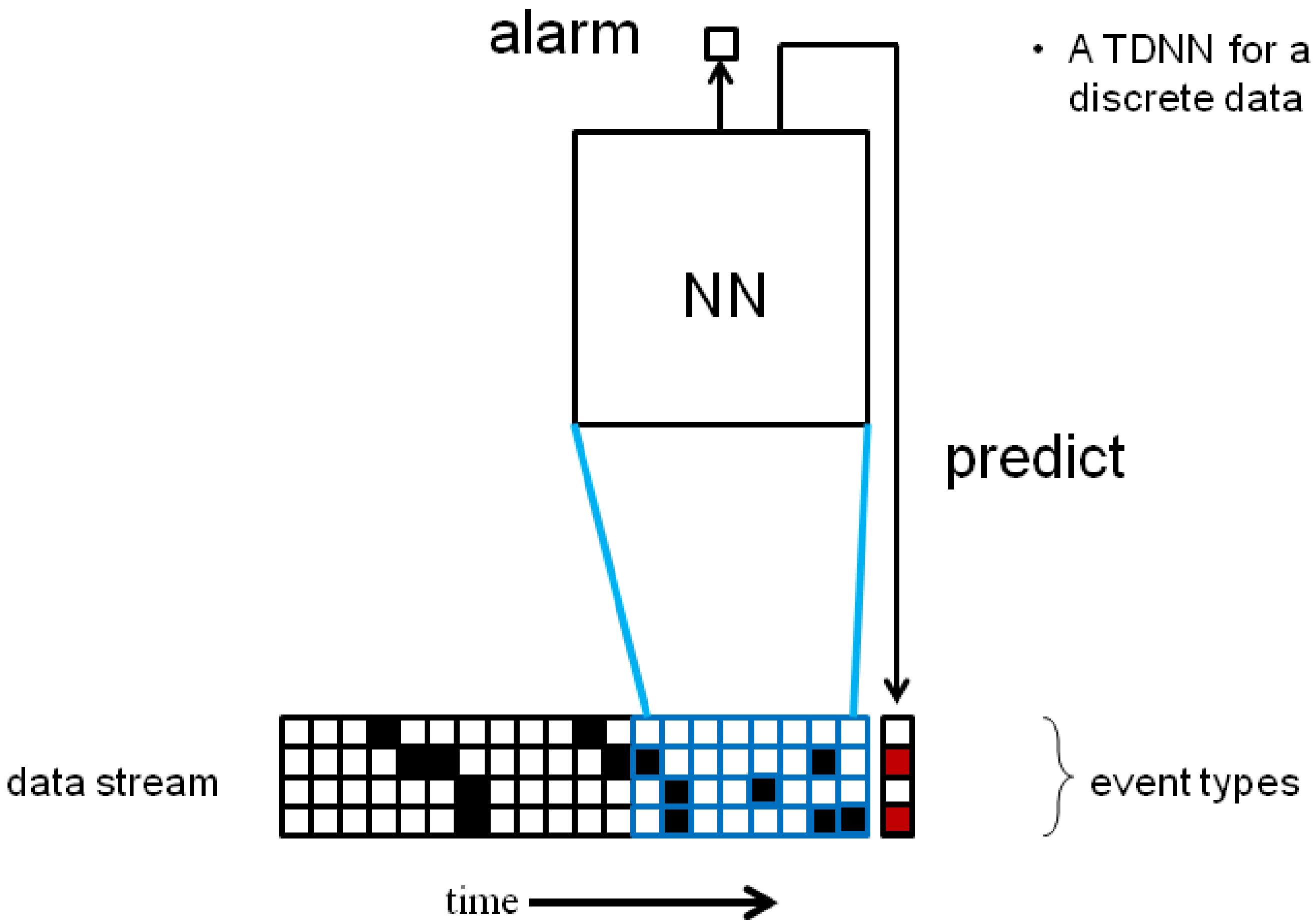


- Shown is one LSTM hidden neuron with input *in(t)* and output *z(t)*
- *in(t)* and all three gates can get input from the inputs *x(t-1)* and from the outputs of all hidden neurons
- When input gate and output gate are open and the forgetting gate is closed, we get a regular latent neuron in a regular recurrent neural net
- When the forgetting gate and the input gate are open the hidden state *m(t)* is

$$m(t) = m(t-1) + in(t)$$

TDNN for Discrete Data

- Analysis of a discrete data stream (alarming, prediction)



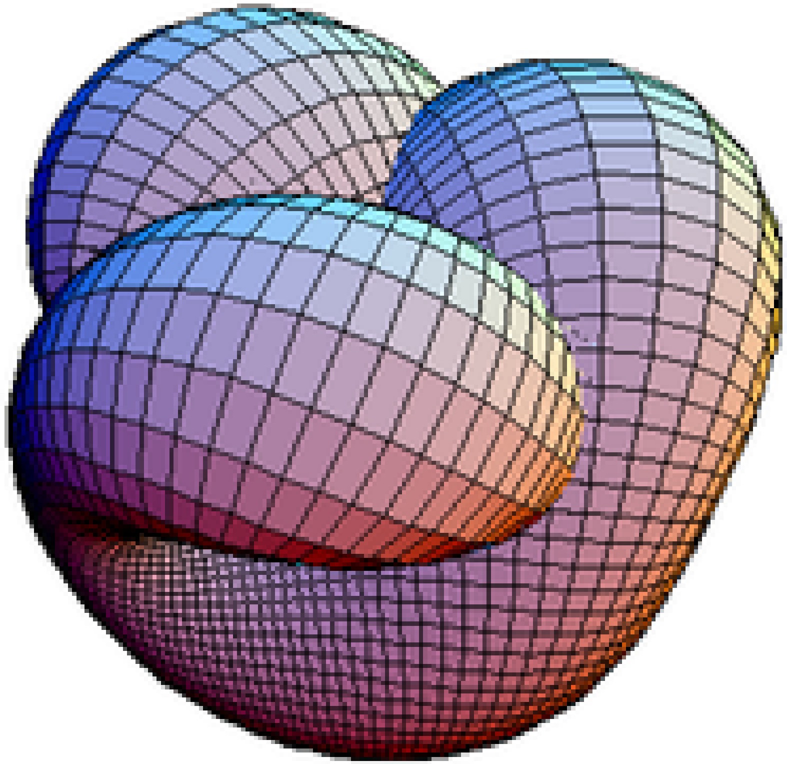
- A TDNN for a multivariate discrete data stream

APPENDIX: Approximation Accuracy of Neural Networks

Vector Space View

- In the discussion on fixed basis functions, we learned that a basis function is a fixed vector in a vector space and that the model function lives in the vector space spanned by the basis functions
- In a neural network the vector space itself can be adapted by tuning the inner weights in the neural network!
- Thus many vector spaces can be modelled by the same neural network leading to great modelling flexibility with fewer parameters
- Considering all parameters in the neural network, the functions that can be represented live in a **nonlinear manifold**

Example of a Nonlinear Manifold (not generated by a neural network)



Complexity Measure

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

$$\int_{\Re^d} |\omega||\tilde{f}(\omega)| \, d\omega = C_f,$$

where $\tilde{f}(\omega)$ is the Fourier transform of $f(\mathbf{x})$. C_f penalizes (assigns a high value to) functions with 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 $\mathbf{x} \in \Re^M$, the neural network has H hidden units
- The approximation error *AF* is the mean squared distance between a target function $f(\mathbf{x})$ and the model $f_{\mathbf{w}}(\mathbf{x})$

$$AF = \int_{B_r} (f(\mathbf{x}) - f_{\mathbf{w}}(\mathbf{x}))^2 \mu(d\mathbf{x}). \tag{1}$$

μ 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 AF_{Neur}

$$AF_{Neur} \leq \frac{(2rC_f)^2}{H}. \tag{2}$$

- 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

$$AF_{BF} \propto \frac{1}{H}^{\frac{2}{M}} = \left(\sqrt[M]{M_\phi^2}\right)^{-1}$$

With $AF_{BF} = 10^{-1}$, we get with this approximation

$$H = 10^{M/2}$$

and be see the exponential increase of the required basis functions with input dimensions M (“curse of dimensionality”)

- For important function classes it could be shown that C_f only increases weakly (e.g., proportional) with M

- 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.