Neural Networks

Volker Tresp

Summer 2014

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 dimensions

Flexible Models: Neural Networks

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

$$\widehat{y}_i = f(\mathbf{x}_i) = \sum_{h=0}^{H-1} w_h \mathsf{sig}(\mathbf{x}_i^T \mathbf{v}_h)$$

- 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

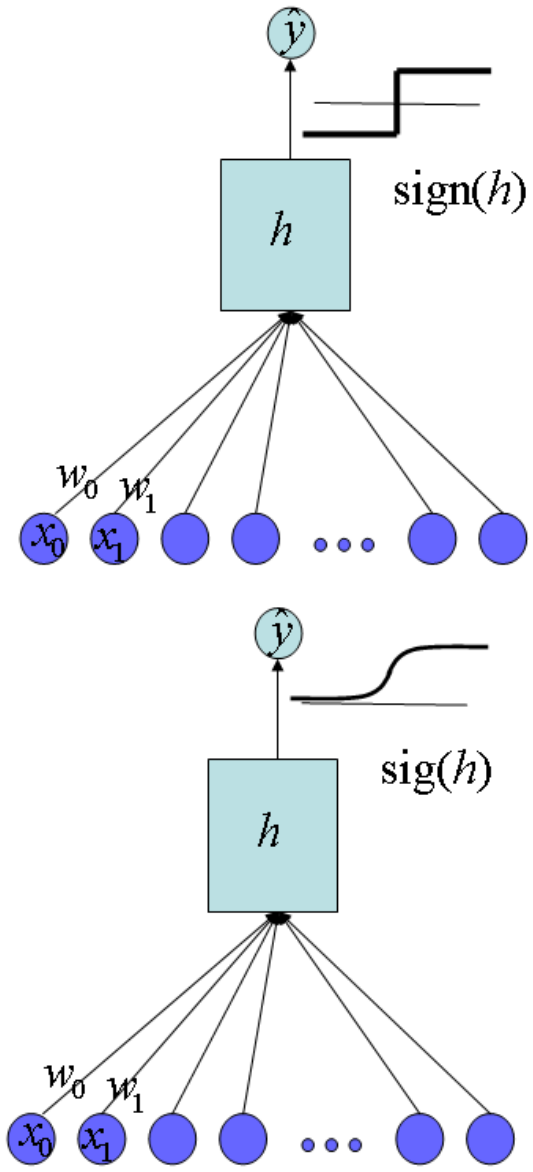
$$z_i = \mathrm{sig}(\mathbf{x}_i^T \mathbf{v}_h) = \mathrm{sig}\left(\sum_{j=0}^{M-1} v_{h,j} x_{i,j}\right)$$

  using the *logistic function*

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

- 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

$$h(\mathbf{x}_i) = \sum_{j=0}^{M-1} w_j x_{i,j}$$

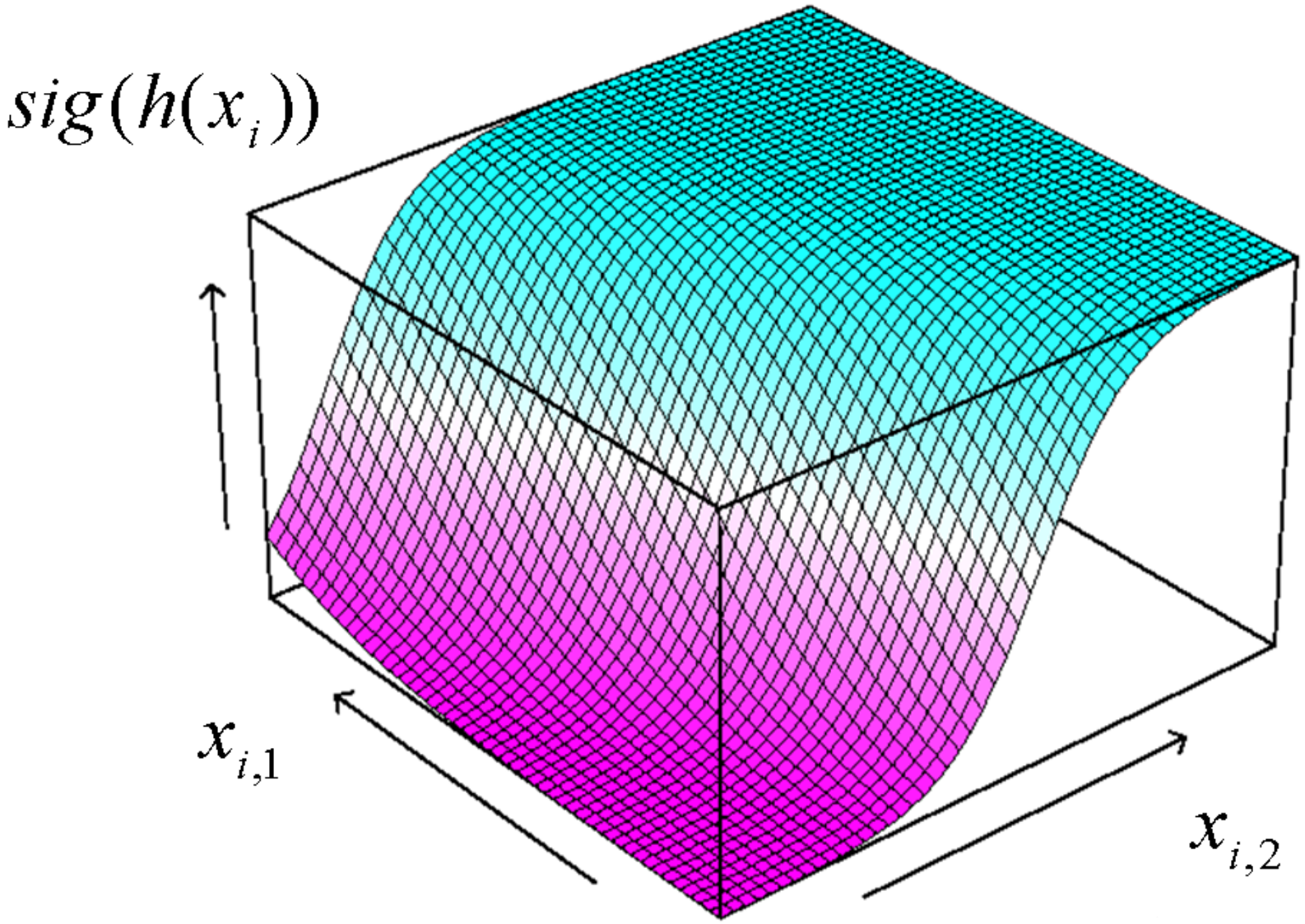
(note: $x_{i,0} = 1$ is a constant input, so that w_0 corresponds to the bias)

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

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

Sigmoidal neuron: $\widehat{y}_i = \text{sig}(h(\mathbf{x}_i))$

Transfer Function



Characteristic Hyperplane

- Definition of the hyperplane

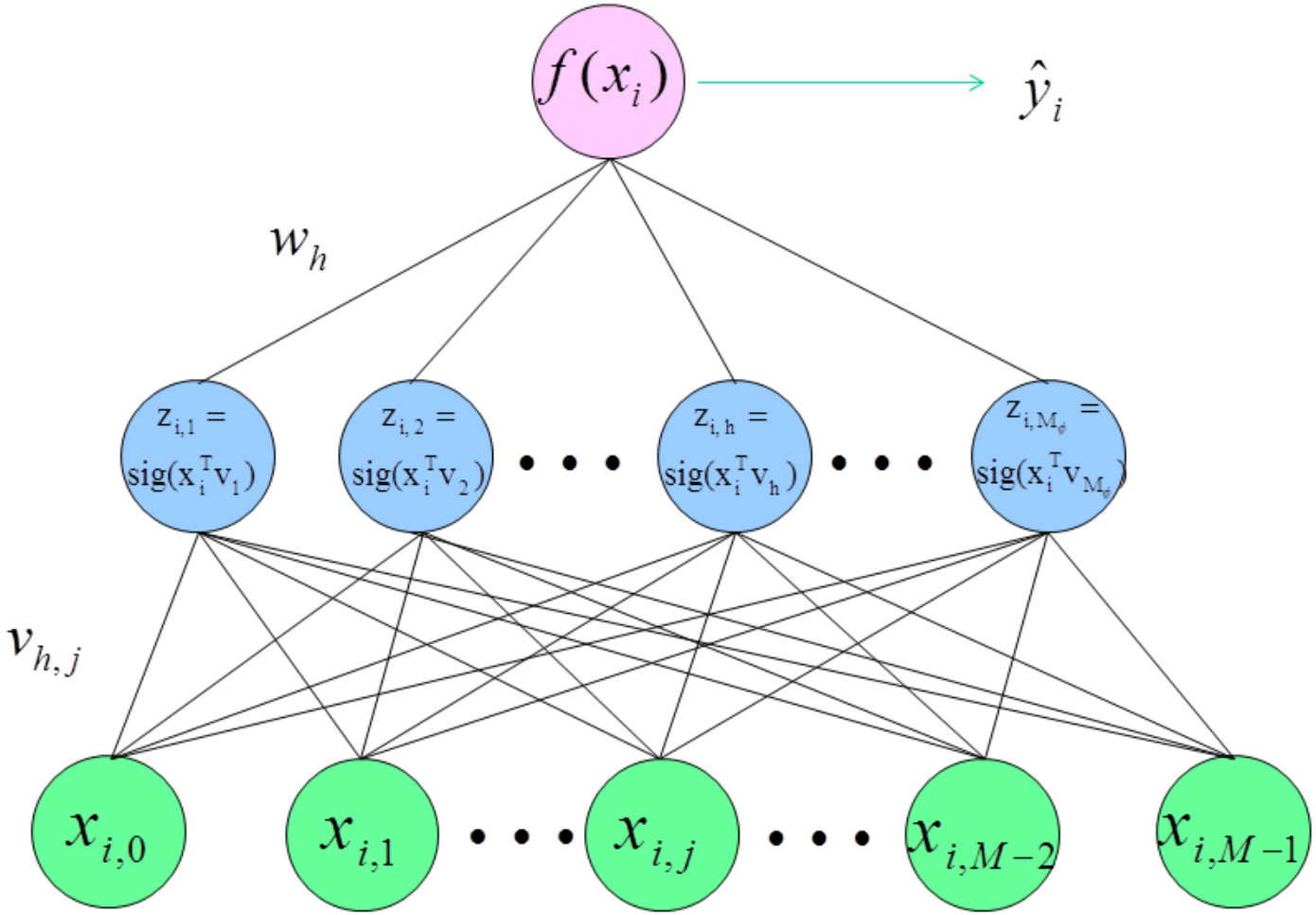
$$\text{sig}\left(\sum_{j=0}^{M-1} v_{h,j} x_{i,j}\right) = 0.5$$

  which means that:

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

- "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

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

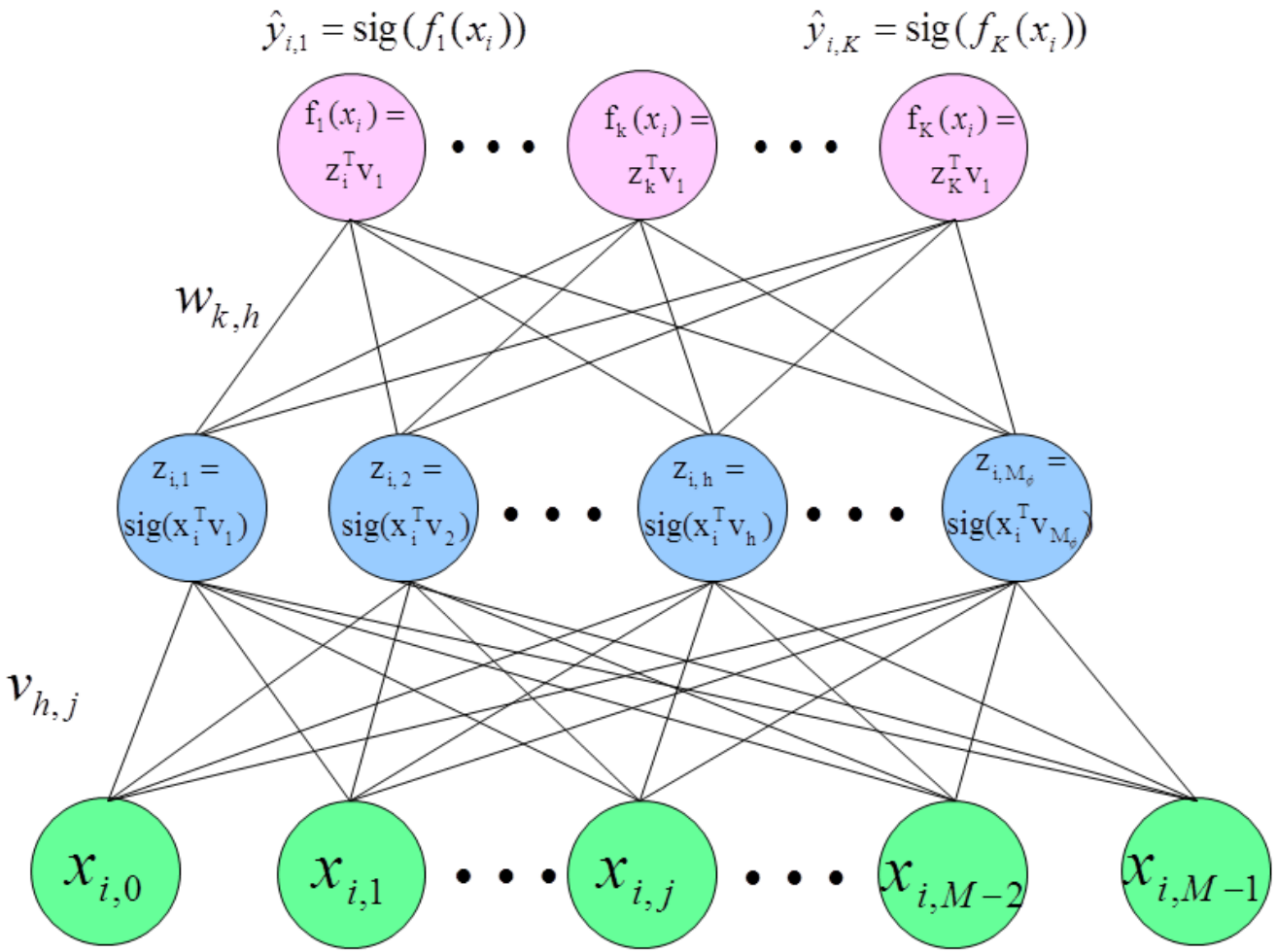
- For multi class tasks, one uses several output neurons. For example, to classify K digits

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

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

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

Architecture of a Neural Network for Several Classes



Learning with Several Outputs

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

$$\text{cost}(\mathbf{w}, \mathbf{v}) = \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{i,k} - \text{sig}(f_k(\mathbf{x}_i, \mathbf{w}, \mathbf{v})))^2$$

- The least squares solution for $\mathbf{v}$ cannot be calculated in closed-form
- Typically both $\mathbf{w}$ and $\mathbf{v}$ are trained via gradient descent

Adaption of the Output Weights

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

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

  where

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

  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

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

Adaption of the Input Weights

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

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

  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
  - 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

$$\text{cost}^{pen}(\mathbf{w},\mathbf{v}) = \sum_{i=1}^{N}\sum_{k=1}^{K}(y_{i,k}-\text{sig}(f_k(\mathbf{x}_i,\mathbf{w},\mathbf{v}))^2+\lambda_1\sum_{h=0}^{M_\phi-1}w_h^2+\lambda_2\sum_{h=0}^{H-1}\sum_{j=0}^{M}v_{h,j}^2$$

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

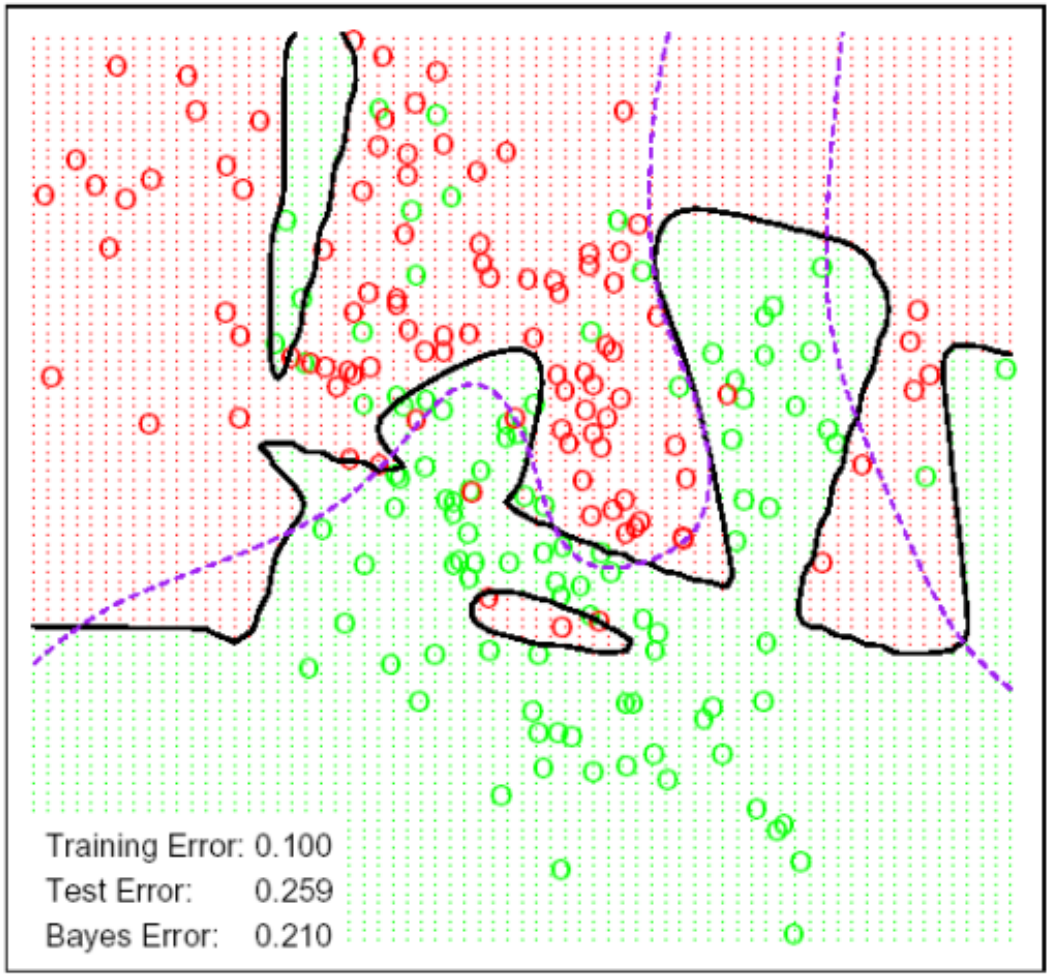
$$w_{k,h} \leftarrow w_{k,h} + \eta\left(\delta_{i,k}z_{i,h} - \lambda_1 w_{k,h}\right)$$

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

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)

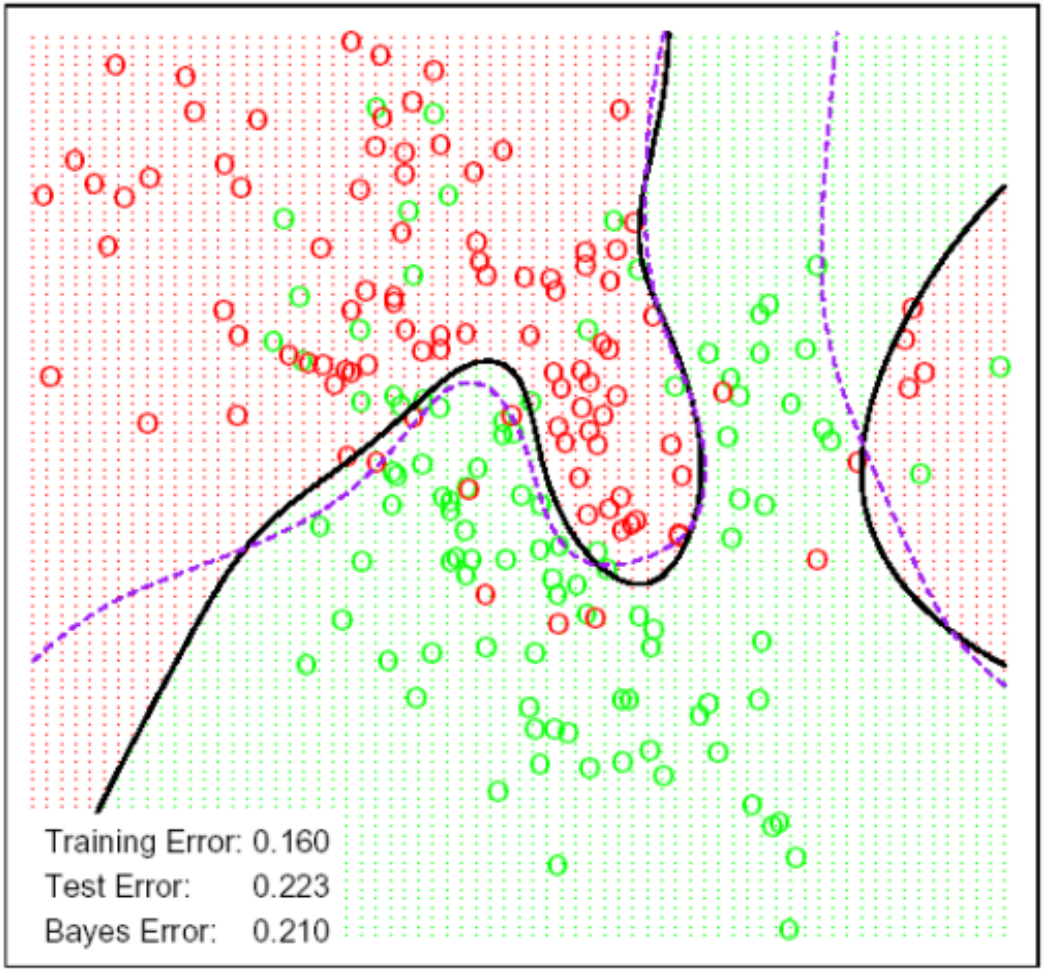
Neural Network - 10 Units, No Weight Decay



Same Example with Regularization

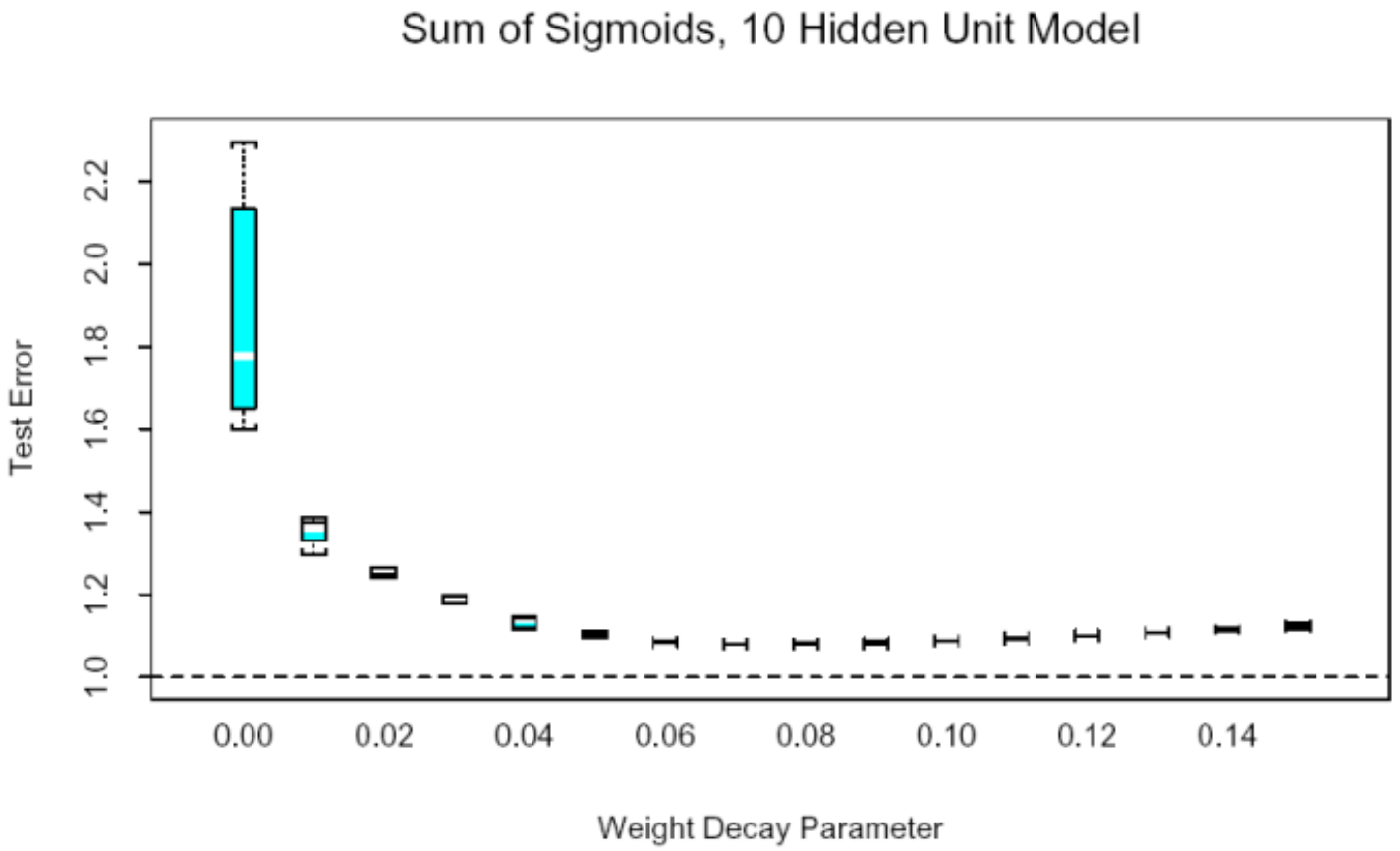
- With regularization ($\lambda_1 = \lambda_2 = 0.2$) the separating plane is closer to the true class boundaries
- The training error is smaller with the unregularized network, the test error is smaller with the regularized network

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
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 $y \in \{0, 1\}$
- Instead of the sum-squared-error cost function, use the cross-entropy cost function (logistic cost function) ($y \in \{0, 1\}$)

$$cost(arg) = -y \log arg - (1 - y) \log(1 - arg)$$

with

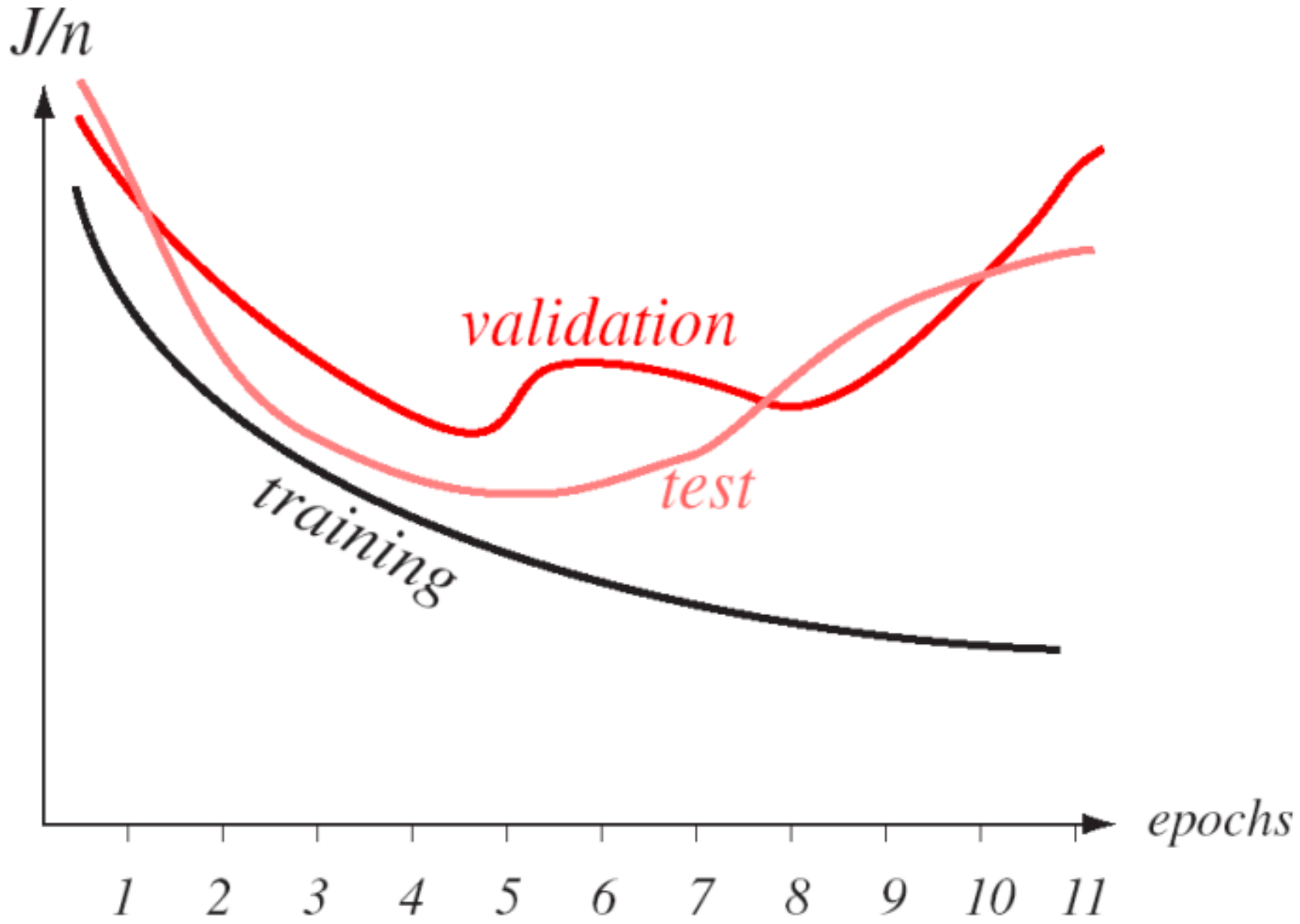
$$\frac{\partial cost(arg)}{\partial arg} = \sum_{i=1}^{N} (y_i - \text{sig}(arg))$$

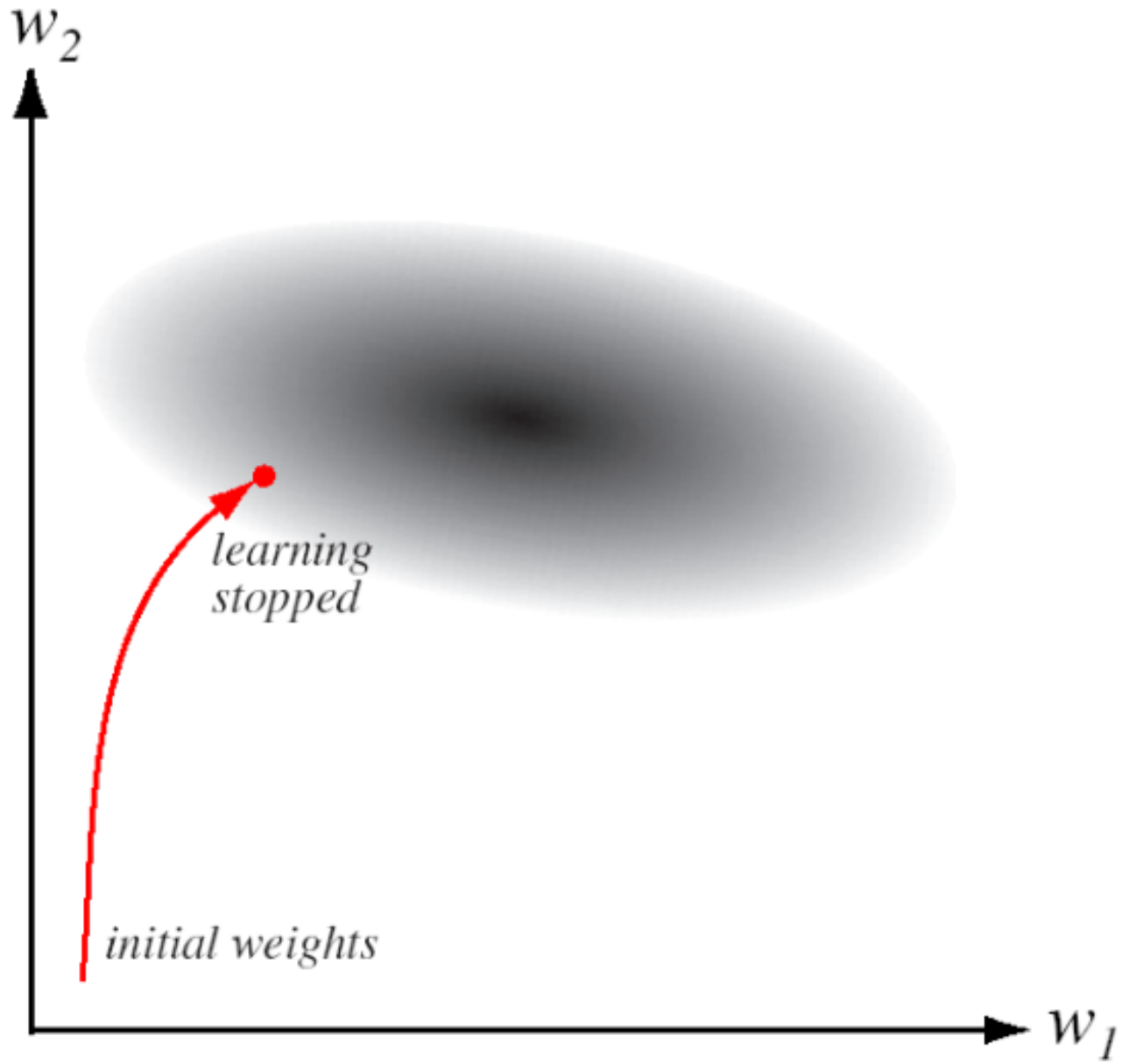
- Compare: for the squared error, we had

$$\frac{\partial cost(arg)}{\partial arg} = \sum_{i=1}^{N} (y_i - \text{sig}(arg))\text{sig}'(arg)$$

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")

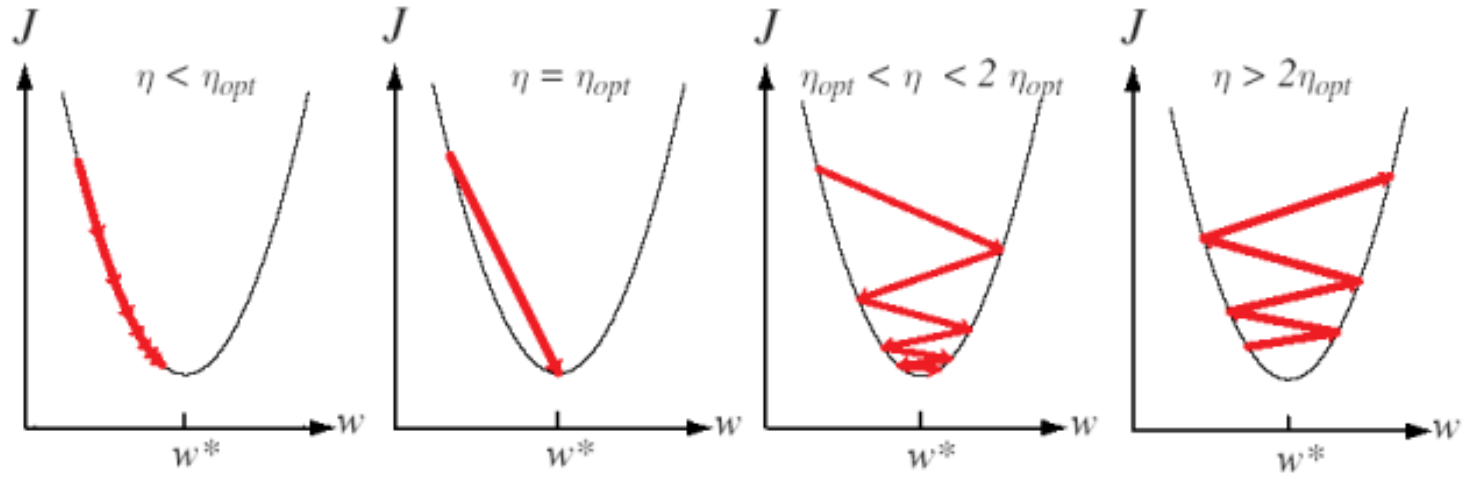
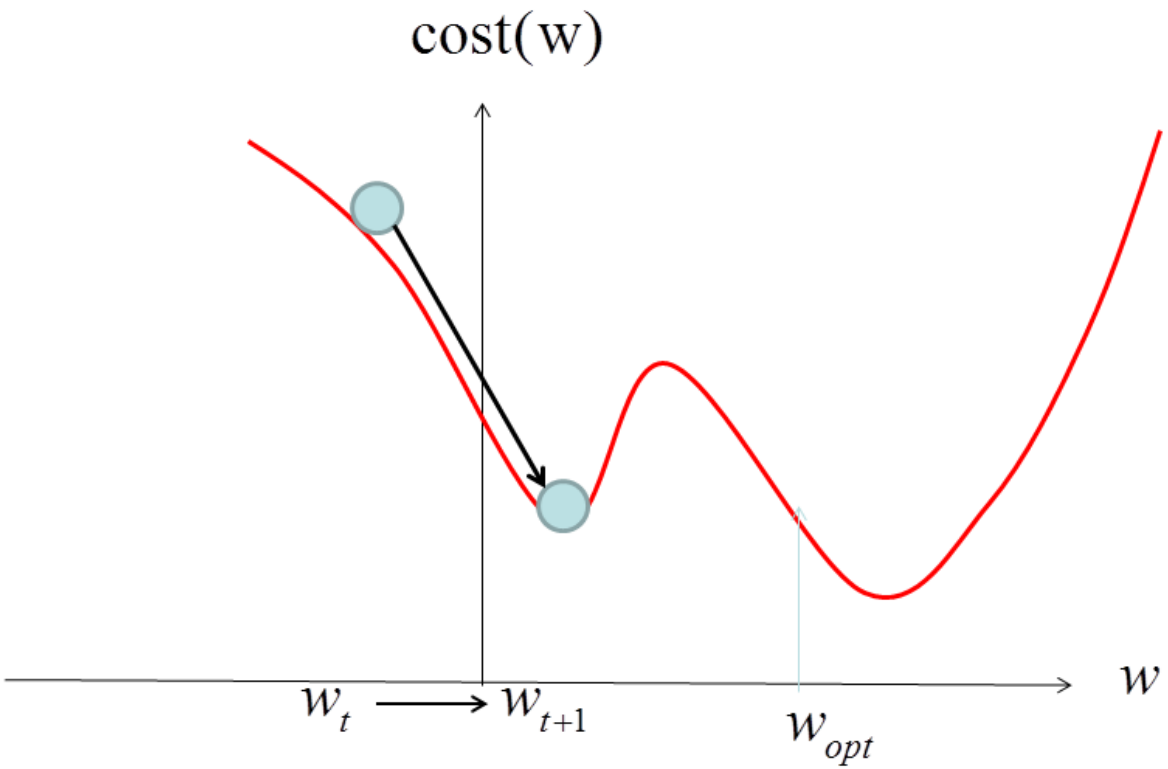
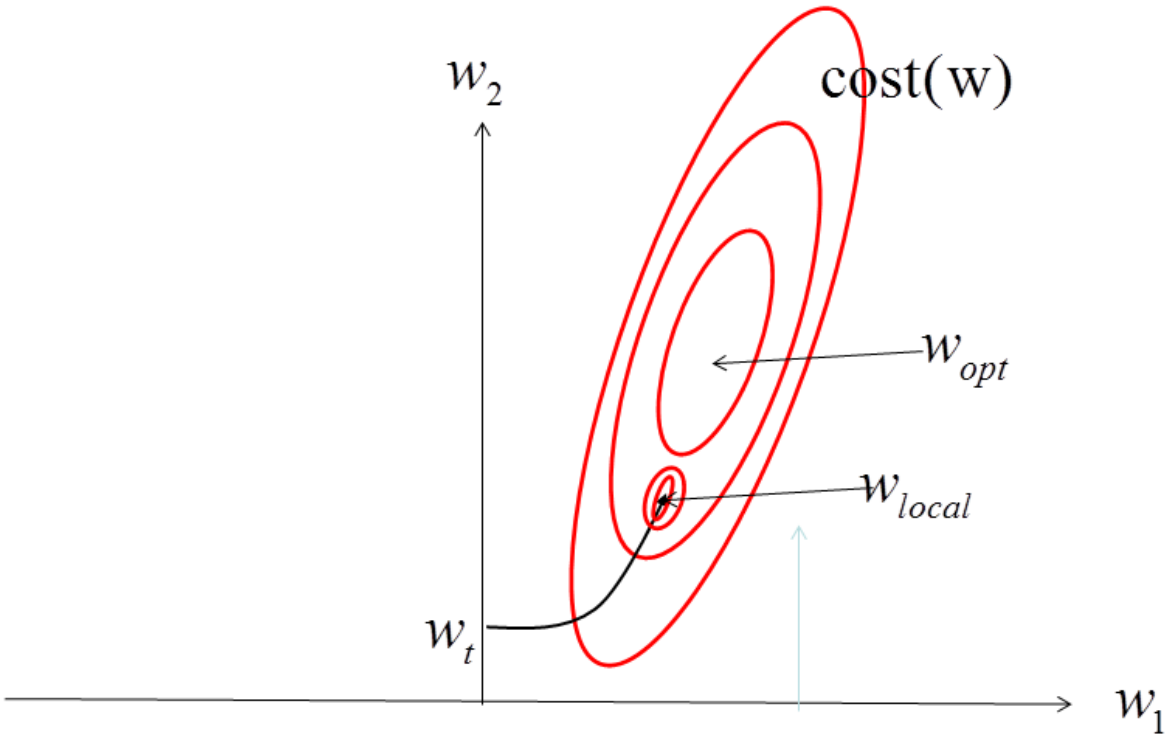


FIGURE 6.16. Gradient descent in a one-dimensional quadratic criterion with different learning rates. If $\eta < \eta_{opt}$, convergence is assured, but training can be needlessly slow. If $\eta = \eta_{opt}$, a single learning step suffices to find the error minimum. If $\eta_{opt} < \eta < 2\eta_{opt}$,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

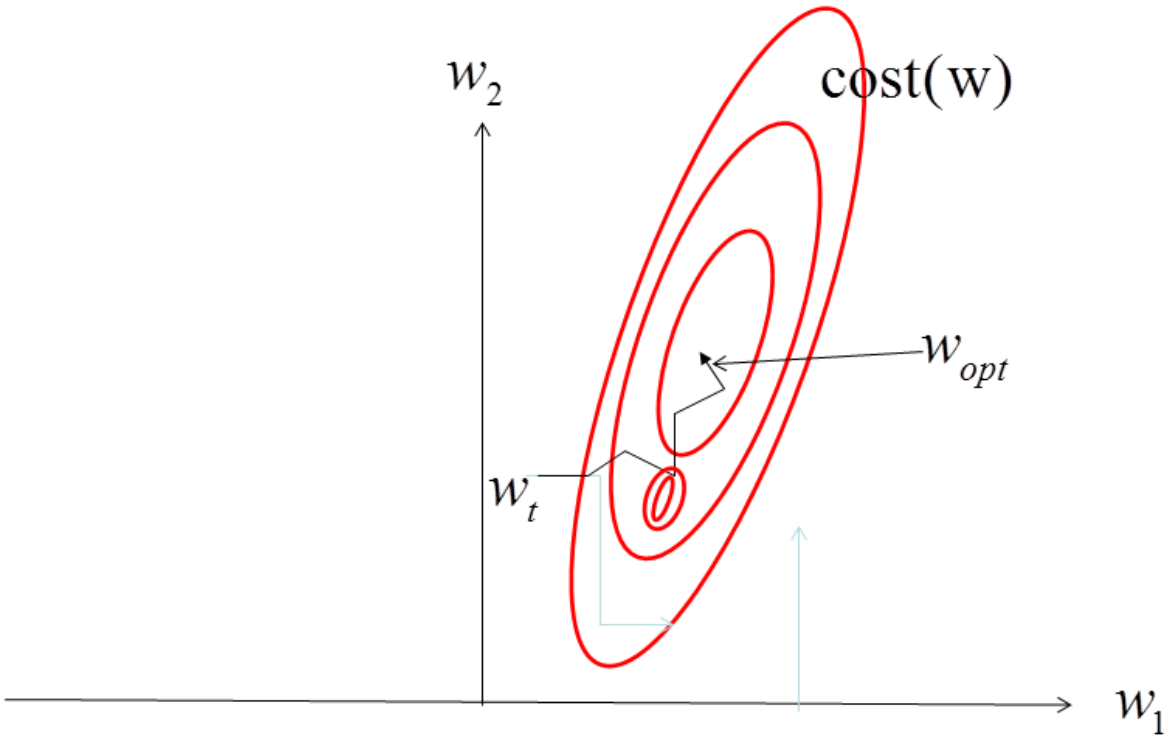
Local Solutions



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)

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

$$x_t = f(x_{t-1}, \ldots, x_{t-M_x}, u_{t-1}, \ldots, u_{t-M_u}) + \epsilon_t$$

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

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

- We approximate, using a neural network,

$$f(x_{t-1}, \ldots, x_{t-M_x}, u_{t-1}, \ldots, u_{t-M_u})$$

$$\approx f_{NN}(x_{t-1}, \ldots, x_{t-M_x}, u_{t-1}, \ldots, u_{t-M_u})$$

  and obtain the cost function

$$\text{cost} = \sum_{t=1}^{N} (x_t - f_{NN}(x_{t-1}, \ldots, x_{t-M_x}, u_{t-1}, \ldots, u_{t-M_u}))^2$$

- Te neuwrl network can be trained as before with simple back propagation
- 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 x_t cannot be measure directly (e.g., if there is noise on the measurements, see Kalman filter), then simple backpropagation is not sufficient, and one needs recurrent adaptation rules
- We assume a model of the form

$$x_t = f(x_{t-1}, \ldots, x_{t-M_x}, u_{t-1}, \ldots, u_{t-M_u}) + \epsilon_t$$

$$y_t = x_t + \delta_t$$

- Examples:
  - Backpropagation through time (BPTT)
  - Real-Time Recurrent Learning (RTRL)
  - Dynamic Backpropagation

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, defined as

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

  where $\tilde{f}(w)$ is the Fourier transform of $f(x)$. C_f penalizes high frequency components!

- The task is to approximate $f(x)$ with a given C_f
- The input vector is $x \in \Re^M$, the neural network has M_ϕ hidden units
- The approximation error *AF* is the mean squared distance between a target function $t(x)$ and $f(x)$

$$AF = \int_{B_r} (t(x) - f(x))^2 \mu(dx). \tag{1}$$

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

- Barron showed that for each $t(x)$, for which C_f is finite there is a neural network with one hidden layer, such that AF_{Neur}

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

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

- Note that for approximations with fixed basis functions, one obtains

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

- 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.