SECTION FOR DIGITAL SIGNAL PROCESSING
DEPARTMENT OF MATHEMATICAL MODELLING
TECHNICAL UNIVERSITY OF DENMARK

DTU

Introduction to Artificial Neural Networks

Jan Larsen

1st Edition

Contents

Preface

The present note is a supplement to the textbook *Digital Signal Processing* [13] used in the DTU course 04361 Digital Signal Processing (Digital Signalbehandling).

The note addresses introduction to signal analysis and classification based on artificial feed-forward neural networks.

Parts of the note are based on former 04364 course note: *Introduktion til Neurale Netværk, IMM, DTU, Oct. 1996* (in Danish) by Lars Kai Hansen and Morten With Pedersen.

Jan Larsen
Lyngby, November 1999

The manuscript was typeset in 11 points Times Roman and Pandora using $\LaTeX\,2_\varepsilon$.

1 Introduction

In recent years much research has directed towards adaptive models for design of flexible signal processing systems. Adaptive models display the following advantageous properties:

- The ability to learn a signal processing task from acquired examples of how the task should be resolved. A general task is to model a relation between two signals. In this case the learning examples simply are related samples of these signals. The learning (also referred to as supervised learning) is often done by adjusting some parameters (weights) such that some cost function is minimized. This property may be valuable in situations where it is difficult or impossible to exactly explain the physical mechanisms involved in the task.

- The possibility of continuously tracking changes in the environments, i.e., handling of non-stationary signals.

- The ability of generalizing to cases which were not explicitly specified by the learning examples. For instance, the ability to estimate the relation between two signals which were not used for training the filter.

Bernard Widrow pioneered the development of *linear adaptive systems* and early artificial neural network models in the sixties, and they have proved to be very successful in numerous applications areas: system identification, control, speech and image processing, time-series analysis, pattern recognition/classifiaction and datamining. This is mainly due to the model's ability to adapt to changing environmental conditions and development of simple and easy implementable algorithms like the Least Mean Squares algorithm. While the bulk of theoretical results and algorithms exist for linear systems, non-linearity is notoriously inherent in many applications. An illustrative example is that many physical systems display very complex behavior such as chaos and limit cycles, and are consequently intrinsically nonlinear. The obvious drawbacks of dealing with nonlinear models are:

- The class of nonlinear models contains, in principle, all models which are not linear. Thus it is necessary to delimit subclasses of nonlinear models which are applicable in a wide range of signal processing tasks. Moreover optimal performance requires adaptation of the *model structure* to the specific application.

- The computational complexity of nonlinear models often is significantly larger than linear models.

- Theoretical analysis often is very involved and intractable.

The field of adaptive signal processing based on artificial neural networks is an extremely active research field and has matured considerably during the past decade. The field is highly interdisciplinary and combines many approaches to signal processing in solving real world problems.

Neural networks is a very fascinating topic as more conventional algorithms does not solve significant problems within e.g., signal processing, control and pattern recognition - challenges which is handled easily by the human brain, e.g., focusing the attention on a specific speaker in a room with many speakers or recognizing/designating and understanding the nature of a sound signal. In other words: Obviously there exist solutions to many complicated problems but it is often not possible to state in details. This note is devoted to artificial neural networks which is an attempt to approach the marvelous world of a real neural network: the human brain.

For elaborate material on neural network the reader is referred to the textbooks:

- Christopher Bishop: *Neural Networks for Pattern Recognition* [1].
- Simon Haykin: *Neural Networks: A Comprehensive Foundation* [4].
- John Hertz, Anders Krogh and Richard G. Palmer: *Introduction to the Theory of Neural Computation* [5].
- Brian Ripley: *Pattern Recognition and Neural Networks* [14].

1.1 Definitions of Neural Networks

1.1.1 Information Processing in Large Networks of Simple Computers

The human brain - also covered by this definition - is characterized by:

- Human brain has $10^{11} = 100$ billion neurons. The thickness of a bank note is approx. 0.1 mm, i.e., the stack of 100 billion bank notes has the length of 100 km.
- Each neuron has 10^4 connections, i.e., the network is relatively sparsely connected.
- Neurons fire every few milliseconds.
- Massive parallel processing.

A neuron (nervous cell) is a little computer which receive information through it dendrite tree, see Fig. 1. The cell calculates continuously it state. When the collective input to the neuron exceeds a certain threshold, the neuron switches from an inactive to an active state - the neuron is firing. The activation of the neuron is transmitted along the axon to other neurons in the network. The transition of the axon signal to another neuron occur via the synapse. The synapse itself it also a computer as it weigh, i.e., transform the axon signal. Synapses can be either excitatory or inhibitory. In the excitatory case the firing neuron contributes to also activating the receiving neuron, whereas for inhibitory synapses, the firing neuron contributes to keep the receiving neuron inactive.

Artificial neural networks using state-of-the-art technology do however not provide this capacity of the human brain. Whether a artificial system with comparable computational capacity will display human like intelligent behavior has been questioned widely in the literature, see e.g., [18]. In Fig. 2 a general artificial neural network is sketched.

1.1.2 Learning/Adaptation by Examples

This is most likely the major reason for the attraction of neural networks in recent years. It has been realized that programming of large systems is notoriously complex: "when the system is implemented it is already outdated". It is possible to bypass this barrier through learning.

The learning-by-example paradigm as opposed to e.g., physical modeling is most easily explained by an example. Consider automatic recognition of hand-written digits where the digit is presented to the neural network and task is to decide which digit was written. Using the learning paradigm one would collect a large set of example of hand-written

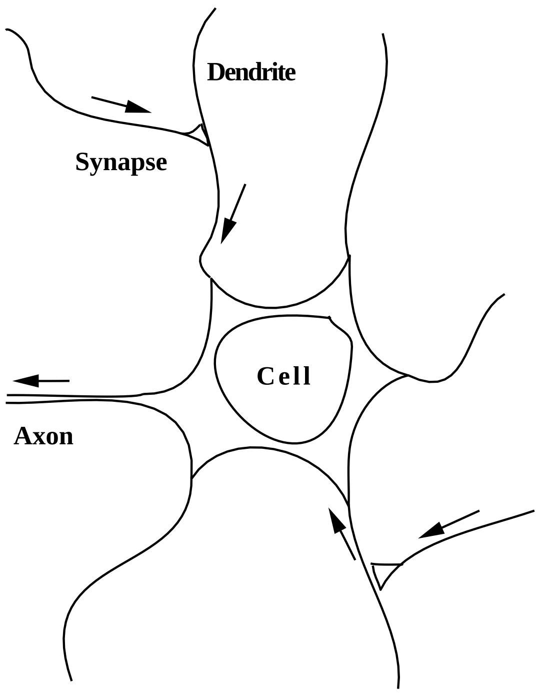


Figure 1: The biological nervous cell – the neuron.

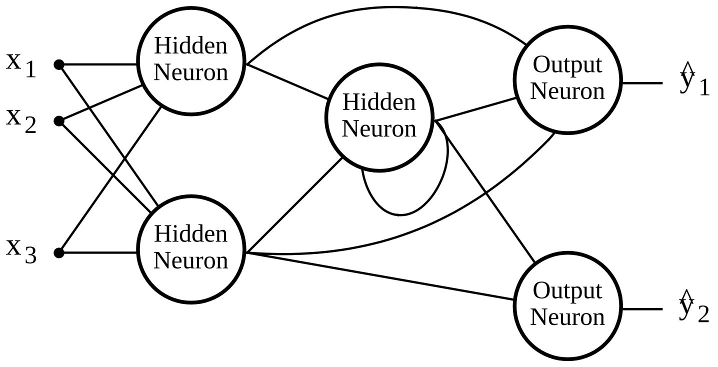


Figure 2: The general structure of an artificial neural network. x_1, x_2, x_3 are 3 inputs and $\widehat{y}_1, \widehat{y}_2$ 2 outputs. Each line indicates a signal path.

digits and learn the nature of the task by adjusting the network synaptic connection so that the number of errors is as small as possible. Using physical modeling, one would try to characterize unique features/properties of each digit and make a logical decision

Approach	Method	Knowledge acquisition	Implementation
System & information theory	Model data, noise, physical constraints	Analyze models to find optimal algorithm	Hardware implementation of algorithm
AI expert system	Emulate human expert problem solving	Observe human experts	Computer program
Trainable neural nets	Design architecture with adaptive elements	Train system with examples	Computer simulation or NN hardware

Table 1: Comparison of information processing approaches [2].

based on the presensense/absense of certain properties as illustrated in Fig. 3. In Table 1

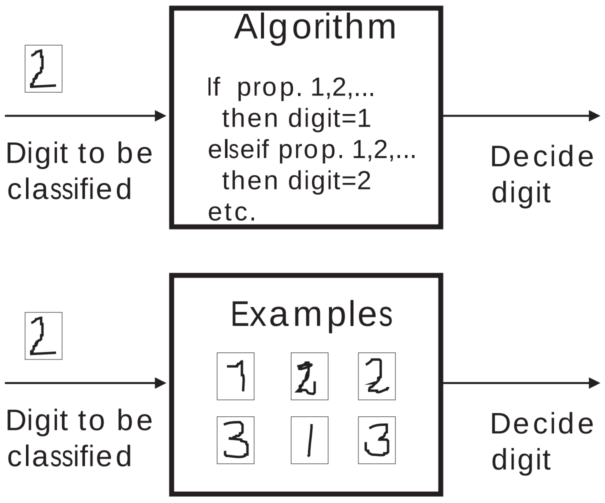


Figure 3: Upper panel: Physical modeling or programming. Lower panel: Learning by example.

a comparison of different information processing approaches is shown.

1.1.3 Generic Nonlinear Dynamical Systems

Such systems are common in daily life though difficult to handle and understand. The weather, economy, nervous system, immune system are examples of nonlinear systems which displays complex often chaotic behavior. Modern research in chaotic systems investigate fundamental properties of chaotic systems while artificial neural networks is an example but also a general framework for modeling highly nonlinear dynamical systems.

1.2 Research and Applications

Many researchers currently show interest in theoretical issues as well as application related to neural networks. The most important conferences and journals related to signal processing are listed below:

Conferences

- *IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP.* Web: `http://www.ieee.org/conferences/tag/sct1sp.html`
- *Advances Neural Information Processing Systems, NIPS.* Web: `http://www.cs.cmu.edu/Groups/NIPS/`
- *IEEE Workshop on Neural Networks for Signal Processing, NNSP.* Web: `http://eivind.imm.dtu.dk/nnsp2000`

Journals

- *Neural Computation.*
- *IEEE Transactions on Signal Processing.*
- *IEEE Transactions on Neural Networks.*
- *Neural Networks.*

Real world industrial/commercial applications of neural networks is e.g., found in IEEE Transaction on Neural Networks; Special Issue on Everyday Applications of Neural Networks, July 1997 and at International Conference on Acoustics, Speech and Signal Processing (ICASSP). A selection of these applications are: check reading, intelligent control of steel plants, credit card fraud detection, electric load forecasting, economic forecasting, software quality modeling, and cork quality classification.

Since late eighties many companies has shown increasingly interest in soft and hardware for neural networks applications. In Denmark a small number of companies has specialized in neural networks and many more routinely use neural networks in their R&D departments. A large number of commercial software packages like *Brainmaker* or *Neural Networks Toolbox* for MATLAB$^{\text{TM}}$ are currently available. Also neural network hardware products for fast processing has been developed. Hecht-Nielsen Neurocomputers Inc. was one of the first companies which marketed a PC plug-in acceleration card and INTEL manufactured a chip (ETANN) based on advanced analog VLSI design. Current trend is, however, to use standard general computers or programmable chips like digital signal processors.

1.3 Historical Outline

The research was initiated by McCulloch & Pitts [12] in 1943 who proposed a simple parametric nonlinear computational model of a real neuron. Rosenblatt [15], [16] proposed around 1960 a layered neural network consisting of *perceptrons* and an algorithm for adjusting the parameters of single layer perceptron network so that the network was able to implement a desired task. At the same time Widrow and Hoff [20] proposed the MADALINE neural network which resembles the perceptron network. Widrow pioneered the use of neural networks within signal processing and in [19] a review of this work can be found. However, the work in 1969 by Minsky and Papert was crucial to the development as they showed that the one-layer perceptron network was not capable of implementing simple tasks (as e.g., the XOR problem) and algorithms for adjusting the weights of multi layered perceptron networks were not invented.

Until the eighties the interest on nonlinear systems and neural networks became sparse. However, the extensively increased power of the computers in the eighties enabled to study more complex phenomena and a lot of progress was made within the study of chaos. Furthermore, around 1985 [11] an algorithm for adjusting the parameters (learning) of a multi-layered neural network – known as the *back-propagation algorithm* – was rediscovered. This turned on an enormous interest for neural networks.

In the DARPA study (1988) [2] a number of prominent neural network scientists devised the directions of the future neural network studies. They concluded that neural networks may provide a very useful tool within a broad range of applications.

Brief History[1]

1943 McCulloch and Pitts: Modeling bio-systems using nets of simple logical operations.

1949 Hebb: Invented a biologically inspired learning algorithm. Connections which are used gain higher synaptic strength. On the other hand, if a connection is not used it synaptic strength tends to zero.

1958 Rosenblatt: The Perceptron – a biologically inspired learning algorithm. The hardware implementation was a large "adaptive" switchboard.

1950's: Other types of simple nonlinear models, e.g., the Wiener and Hammerstein model.

1960 Widrow and Hoff: Learning rules for simple nets. Hardware implementation and signal processing applications.

1969 Minsky & Papert: negative analysis of the simple perceptron.

1982 Hopfield: Analogy between magnetism and associative memory – *The Hopfield model.*

1984 Hinton *et al.*: Supervised learning for general Boltzmann machines with hidden units significantly change the premises for Minsky and Papert's analysis.

1969–1986: The neural network blackout period.

1986 Rumelhart *et al.*: Rediscovery of the "Backpropagation of errors" algorithm for feed-forward neural networks.

1987: First commercial neuro computers: The Hecht-Nielsen ANZA PC add-on boards. The Science Application International Corp. DELTA PC add-on board.

1988 DARPA Study: The DARPA study headed by Widrow demonstrated the potential of neural networks in many application areas – especially signal processing – and had a great impact on research.

2000: Many commercial products and focused research.

[1]A more elaborate survey is found in [3].

Since 1988 the field has matured significantly and thousands of researchers work in field of neural networks or related areas. In Denmark the Danish Research Councils (SNF, STVF) supported the establishment of The Computational Neural Network Center (CONNECT) in 1991 with partners from The Niels Bohr Institute, DTU and Risø National Laboratory. CONNECT studied the theory, implementation and application of neural computation. An up-to-date idea of the current research can be found at

`http://eivind.imm.dtu.dk/thor`

2 Feed-forward Neural Network

The structure of the 2-layer feed-forward neural network is show in Fig. 4. The 2-layer

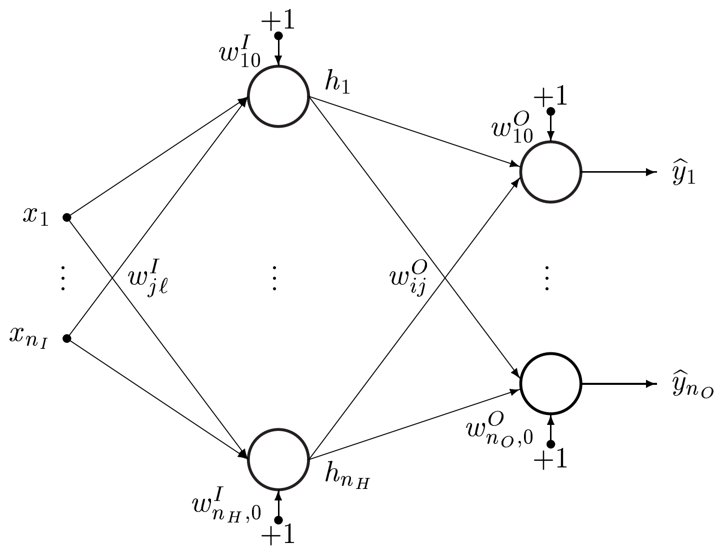


Figure 4: Two-layer (n_I, n_H, n_O) feed-forward neural network architecture.

feed-forward network has n_I inputs, n_H hidden neurons, and n_O output neurons; for short hand the nomenclature (n_I, n_H, n_O) is used. The network is a graphical representation of a layered computation: the hidden unit activation $h_1, \cdots, h_{n_I}$ in the first layer is calculated from the inputs $x_1, \cdots, x_{n_I}$. Next the output $\widehat{y_1}, \cdots, \widehat{y}_{n_O}$ are calculated from the hidden unit activations. The processing in networks is given

$$h_j(\boldsymbol{x}) = \psi\left(\sum_{\ell=1}^{n_I} w^I_{j\ell} x_\ell + w^I_{j0}\right) \tag{1}$$

$$\widehat{y}_i(\boldsymbol{x}) = \psi\left(\sum_{j=1}^{n_H} w^O_{ij} h_j(\boldsymbol{x}) + w^O_{i0}\right) \tag{2}$$

where

- $\boldsymbol{x} = [1, x_1, \cdots, x_{n_I}]$ is the input vector.

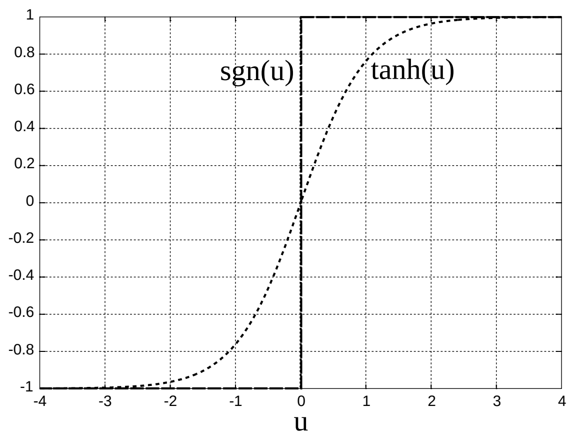


Figure 5: Two examples of typical activation functions: The sign function $\mathrm{sgn}(u)$ (solid line) and the hyperbolic tangent $\tanh(u)$ (dotted line).

- $\psi(u)$ is a non-linear activation function which usually is has a sigmoidal shape, e.g., $\psi(u) = \tanh(u)$, see Fig. 5.
- $w^I_{j\ell}$ is the weight *from* input ℓ *to* hidden neuron j.
- w^O_{ij} is the weight *from* hidden neuron j *to* output i.
- w^I_{j0}, w^O_{i0} are bias weights or thresholds.

A simple matrix formulation of the processing is possible by defining

- $\boldsymbol{W}^I$ the $(n_H, n_I + 1)$ input-hidden weight matrix and $\boldsymbol{W}^O$ the $(n_O, n_H + 1)$ hidden-output weight matrix:

$$\boldsymbol{W}^I = \{w^I_{j\ell}\} = \begin{bmatrix} (\boldsymbol{w}^I_1)^\top \\ \vdots \\ (\boldsymbol{w}^I_j)^\top \\ \vdots \\ (\boldsymbol{w}^I_{n_H})^\top \end{bmatrix} \qquad \boldsymbol{W}^O = \{w^O_{ij}\} = \begin{bmatrix} (\boldsymbol{w}^O_1)^\top \\ \vdots \\ (\boldsymbol{w}^O_i)^\top \\ \vdots \\ (\boldsymbol{w}^O_{n_O})^\top \end{bmatrix} \tag{3}$$

  where $(\boldsymbol{w}^I_j)^\top$ is the j'th row of the input-hidden weight matrix and $(\boldsymbol{w}^H_i)^\top$ is the i'th row of the hidden-output weight matrix.

- $\boldsymbol{x} = \{x_\ell\}$ the $(n_I + 1, 1)$ input vector with $x_0 \equiv 1$.
- $\boldsymbol{h} = \{h_j\}$ the $(n_H + 1, 1)$ hidden vector with $h_0 \equiv 1$.
- $\widehat{\boldsymbol{y}} = \{\widehat{y}_i\}$ the $(n_O, 1)$ output vector.
- $\boldsymbol{\psi}(\boldsymbol{u}) = [\psi(u_1), \cdots, \psi(u_n)]$ the element-by-element vector activation function.

The processing is then given by:

$$\boldsymbol{h} = \boldsymbol{\psi}(\,\boldsymbol{W}^I \cdot \boldsymbol{x}\,) \tag{4}$$
$$\widehat{\boldsymbol{y}} = \boldsymbol{\psi}(\boldsymbol{W}^O \cdot \boldsymbol{h}\,) = f(\boldsymbol{x}, \boldsymbol{w}) \tag{5}$$

For short hand, we notice the networks can viewed as a nonlinear function of the input vector $\boldsymbol{x}$ parameterized by the *column* weight vector $\boldsymbol{w} = \{\mathrm{vec}(\boldsymbol{W}^I), \mathrm{vec}(\boldsymbol{W}^H)\}$ which is the collection of all network weights. The total number of weights m equals $(n_I + 1)n_H + (n_H + 1)n_O$.

The processing in the individual neuron is thus a sum of its inputs followed by the non-linear activation function, as show in Fig. 6 for hidden neuron j.

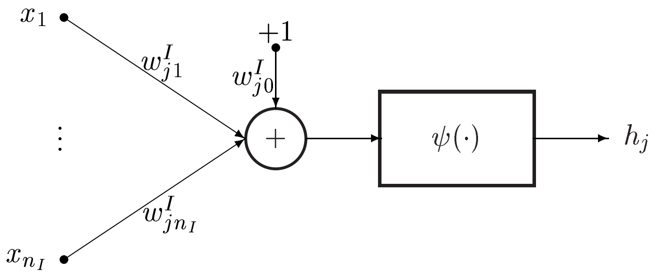


Figure 6: Processing in a neuron.

2.1 Geometrical Interpretation

A geometrical interpretation of the processing in a neuron is possible. Consider choosing the sign function as activation function, i.e., $\psi(u) = \mathrm{sgn}(u)$, then the processing of hidden unit j is $h_j = \mathrm{sgn}(\boldsymbol{w}^I_j \boldsymbol{x})$ where $(\boldsymbol{w}^I_j)^\top$ is the j'th row of $\boldsymbol{W}^I$. Define the $n_I \Leftrightarrow 1$ dimensional *hyperplane* $\mathcal{H}^I_j$

$$\boldsymbol{x}^\top \boldsymbol{w}^I_j = 0 \Leftrightarrow w^I_{j0} + \widetilde{\boldsymbol{x}}^\top \widetilde{\boldsymbol{w}}^I_j = 0 \tag{6}$$

where $\widetilde{\boldsymbol{w}}^I_j$ and $\widetilde{\boldsymbol{x}}$ are truncated versions of the weight and input vector, respectively, omitting the first component. The hyperplane thus separates the regions in input space for which the output of the neuron is $+1$ and $\Leftrightarrow 1$, respectively, as shown in Fig. 7. That is a single neuron are able to *linearly separate* input patterns into two classes. When $\psi(u) = \tanh(u)$ a smooth transition from $\Leftrightarrow 1$ to $+1$ will occur perpendicular to the hyperplane, and the output will be a continuous function of the input. A two-layer network can perform more complex separation/discrimation of input patterns. Consider $(2, 2, 1)$ network in which all neurons have sign activation functions. An example is shown in Fig. 8.

Example 2.1 The famous XOR problem in Fig. 9 which Minsky and Papert showed could not be solved by a single perceptron [5, Ch. 1.2] can be solved by a $(2, 2, 1)$ network as shown in Fig. 10. which Minsky and Papert showed could not be solved by a single perceptron [5, Ch. 1.2] can be solved by a $(2, 2, 1)$ network as shown in Fig. 10.

■

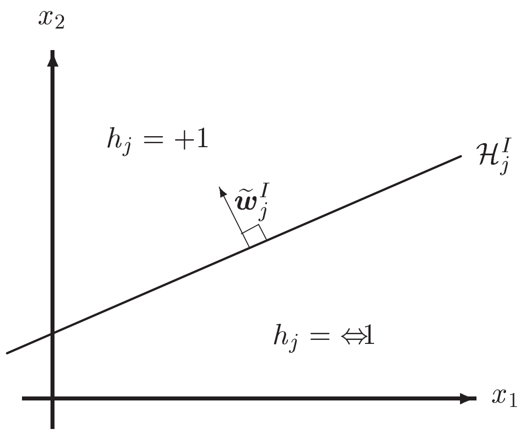


Figure 7: Separating hyperplane. $\widetilde{\boldsymbol{w}}_j^I$ is the normal vector of the hyperplane.

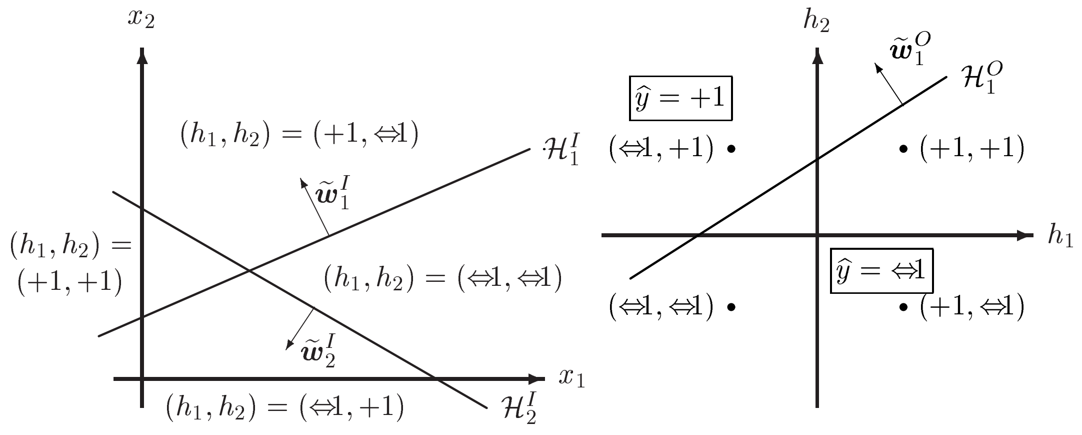


Figure 8: Example of separation in a $(2, 2, 1)$ feed-forward neural network. The area below $\mathcal{H}_1^I$ and $\mathcal{H}_2^I$ in input space is thus assigned output $\hat{y} = +1$; the remaining input space is assigned $\hat{y} = ⇔1$.

2.2 Universal Function Approximation

A 2-layer feed-forward neural network with n_I continuous inputs, hyperbolic tangent hidden unit activation functions and a single *linear* output neuron, i.e., $\psi(u) = u$, has the

x_1	x_2	$\hat{y}$
-1	-1	-1
-1	+1	+1
+1	-1	+1
+1	+1	-1

Figure 9: The XOR problem.

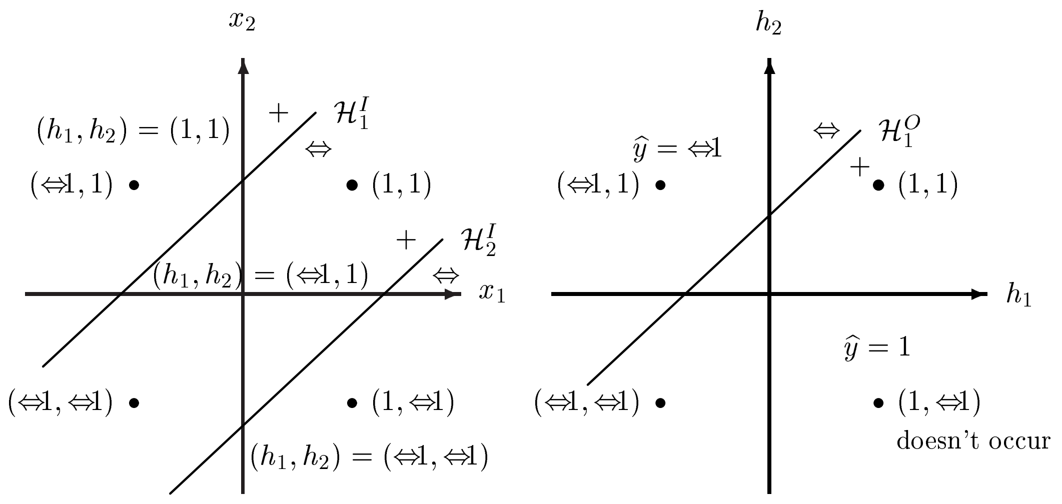

x_2
$(h_1, h_2) = (1,1)$
$+$
$\mathcal{H}_1^I$
$\Leftrightarrow$
$(\Leftrightarrow 1, 1)$ •
• $(1,1)$
$+$
$\mathcal{H}_2^I$
$(h_1, h_2) = (\Leftrightarrow 1, 1)$
$\Leftrightarrow$
x_1
$(\Leftrightarrow 1, \Leftrightarrow 1)$ •
• $(1, \Leftrightarrow 1)$
$(h_1, h_2) = (\Leftrightarrow 1, \Leftrightarrow 1)$
h_2
$\widehat{y} = \Leftrightarrow 1$
$\Leftrightarrow$
$\mathcal{H}_1^O$
$(\Leftrightarrow 1, 1)$ •
$+$
• $(1,1)$
h_1
$\widehat{y} = 1$
$(\Leftrightarrow 1, \Leftrightarrow 1)$ •
• $(1, \Leftrightarrow 1)$
doesn't occur

Figure 10: Solving the XOR problem by a (2,2,1) feed-forward perceptron neural network. Note that the locations of the hyperplanes are not unique.

property of universal approximation of any continuous function, $\widehat{y} = g(\boldsymbol{x})$ to any desired accuracy provides the number of hidden units are large, see e.g., [7]. That is, as $n_H \to \infty$ the approximation error $||g(\boldsymbol{x}) \Leftrightarrow f(\boldsymbol{x}, \boldsymbol{w})||$ tends to zero[2].

The universal approximation theorems are existence theorems, hence, they do not provide any guideline for selecting the network weights or (limited) number of hidden unit to reach a prescribed accuracy. Training of neural networks and selection of proper network architecture/structure are important issues dealt with in what follows.

Example 2.2 This examples shows how the a feed-forward neural network with hyperbolic tangent activation function in the hidden layer and linear output neuron are able to approximate the (one-dimensional) nonlinear function

$$\begin{aligned} g(x) \;=\; & \left[\exp\left(\Leftrightarrow 20(x \Leftrightarrow 0.5)^2\right) \Leftrightarrow \exp\left(\Leftrightarrow 10(x+0.6)^2\right)\right] \cdot \left[\frac{1}{1+\exp(4x \Leftrightarrow 2)}\right] \cdot \\ & \left[1 \Leftrightarrow \frac{1}{1+\exp(4x+2)}\right]. \end{aligned} \tag{7}$$

In Fig. 11 the graph of $g(x)$ is shown. A $(1,8,1)$ network with weight matrices

$$\boldsymbol{W}^I = \begin{bmatrix} 3.65 & 4.06 & 1.87 & 1.19 & \Leftrightarrow 1.84 & \Leftrightarrow 2.88 & \Leftrightarrow 4.25 & \Leftrightarrow 5.82 \\ 4.94 & 6.65 & 5.39 & 6.19 & 7.26 & 8.05 & 7.41 & 8.37 \end{bmatrix}^\top \tag{8}$$

$$\boldsymbol{W}^O = [0.0027, \Leftrightarrow 0.218, \Leftrightarrow 0.0793, 0.226, 0.0668, 0.159, 0.177, \Leftrightarrow 0.259, \Leftrightarrow 0.0758] \tag{9}$$

produces the approximation shown in Fig. 12.

■

[2] $||\cdot||$ denotes a norm.

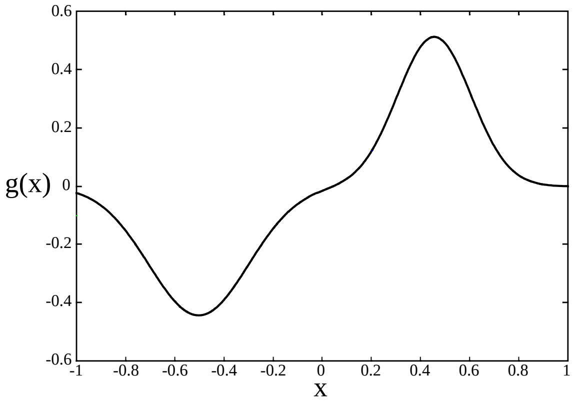
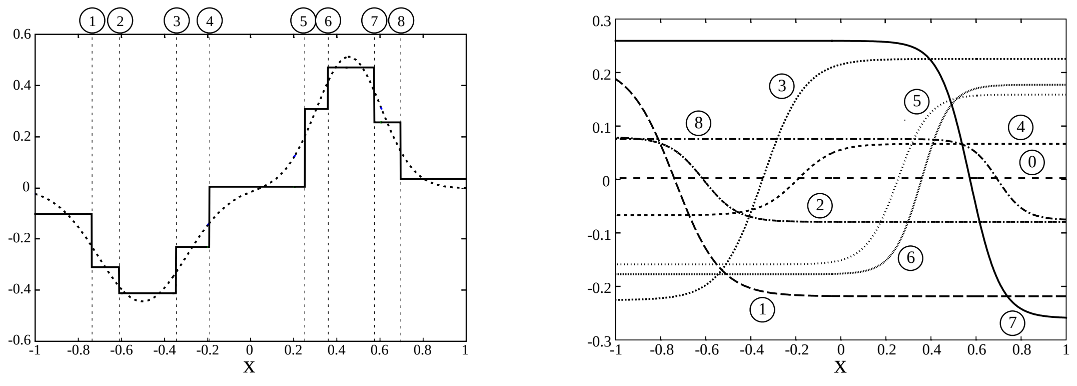


Figure 11: Graph of $g(x)$ given by Eq. (7).



(a) (b)

Figure 12: *Panel (a)*: $g(x)$ (dash-dotted line) and the piece-wise constant approximation (solid line) produced by the neural network using sign function hidden layer activation. The vertical dashed lines are the hyperplane locations of the eight hidden neurons. *Panel (b)*: The contributions from the individual hidden neurons multiplied by corresponding output weights. Thus adding all contributions will produce a curve close to the original $g(x)$. The neurons are indicated by numbers in circles, thus neuron no. 0 refers to the bias input, cf. Fig. 6.

2.3 Regression

Modeling the statistical relation between the stochastic continuous output (response) y and input (regressor) $\boldsymbol{x}$ is referred to as *regression*. A typical regression model is the additive noise model:

$$y = \widehat{y} + \varepsilon = g(\boldsymbol{x}) + \varepsilon \tag{10}$$

where ε is an random error in y which can not be explained by the input. Exploiting the universal function approximation capability of the feed-forward neural network, the neural network regression model is given by

$$y = \widehat{y} + e = f(\boldsymbol{x}, \boldsymbol{w}) + e \tag{11}$$

where e is random error[3].

2.4 Signal Processing Applications

Consider digital signals $x(k)$ and $y(k)$. Signal processing applications can be cast into a regression framework:

$$y(k) = f(\boldsymbol{x}(k), \boldsymbol{w}) + e(k) = \widehat{y}(k) + e(k) \tag{12}$$

where $f(\cdot)$ represents the neural network with hyperbolic tangent hidden units and linear output unit, $\boldsymbol{w}$ is the network weight vector, $\boldsymbol{x}(k) = [x(k), x(k \Leftrightarrow 1), \cdots, x(k \Leftrightarrow n_I + 1)]$ is the input signal vector, and $e(k)$ is the error signal which expresses the deviation of the estimated output $\widehat{y}(k)$ from the output signal $y(k)$.

Example 2.3 If the network consist of a single linear neuron, then

$$y(k) = \boldsymbol{w}^\top \boldsymbol{x}(k) + e(k) \tag{13}$$

In that case the neural network reduces to a linear adaptive FIR filter [17] with $y(k)$ as the target signal.

$\blacksquare$

2.4.1 Nonlinear System Identification

The identification of an unknown nonlinear system is shown in Fig. 13.

2.4.2 Nonlinear Prediction

Nonlinear prediction is shown in Fig. 14.

2.5 Classification

Consider a classification problem for which each (continuous) input pattern $\boldsymbol{x}$ belongs to one, and only one[4], specific class $\mathcal{C}_i$, $i = 1, 2, \cdots, c$ out of c possible classes. An example is depicted in Fig. 15. E.g., $\boldsymbol{x}$ could represent a set of features describing a hand-written digit, and $\mathcal{C}_1, \cdots, \mathcal{C}_{10}$ would represent the 10 digits. Another example is that the input is a signal vector $\boldsymbol{x}(k)$ and the objective is to decide the membership of c possible groups. Consider e.g., a music signal, then the groups could represent semantic annotations of the signal like: jazz music, rock music, classical music, other music.

Even though every pattern uniquely belongs to a specific class, there may overlap among classes, as illustrated in Fig. 15. Thus a statistical framework is deployed in which the aim is to model the conditional probabilities $p(\mathcal{C}_i|\boldsymbol{x})$, $i = 1, 2, \cdots, c$. That is, the probability that a specific input pattern $\boldsymbol{x}$ belongs to class $\mathcal{C}_i$. Knowing these probabilities

[3]In general, ε and e are different, due to the fact that the neural network can not implement the underlying target function $g(\boldsymbol{x})$ exactly.

[4]This is referred to as mutually exclusive classes.

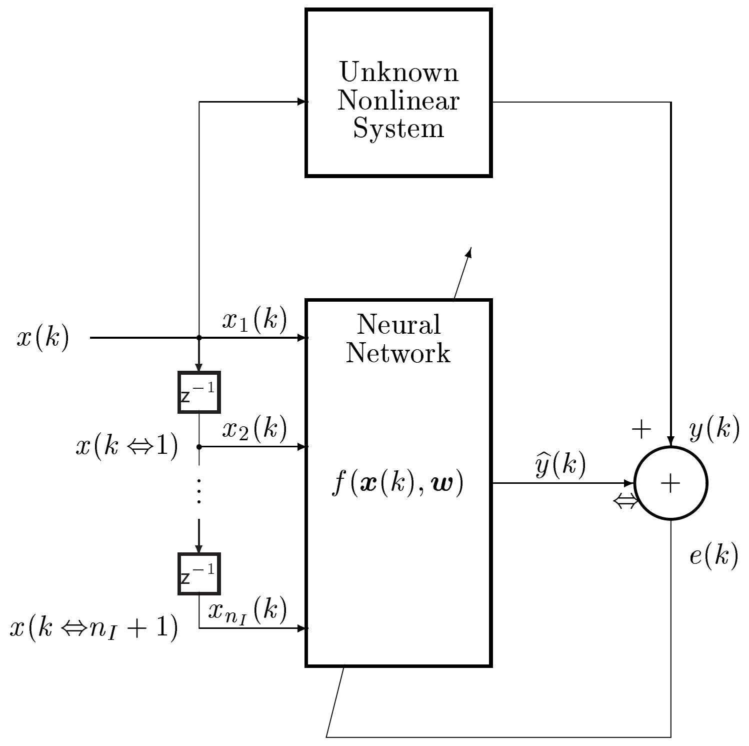


Figure 13: Identification of an unknown nonlinear system. The error signal $e(k)$ is used to adapt the parameters of the neural network. $\widehat{y}(k)$ is the neural network's prediction of nonlinear system output.

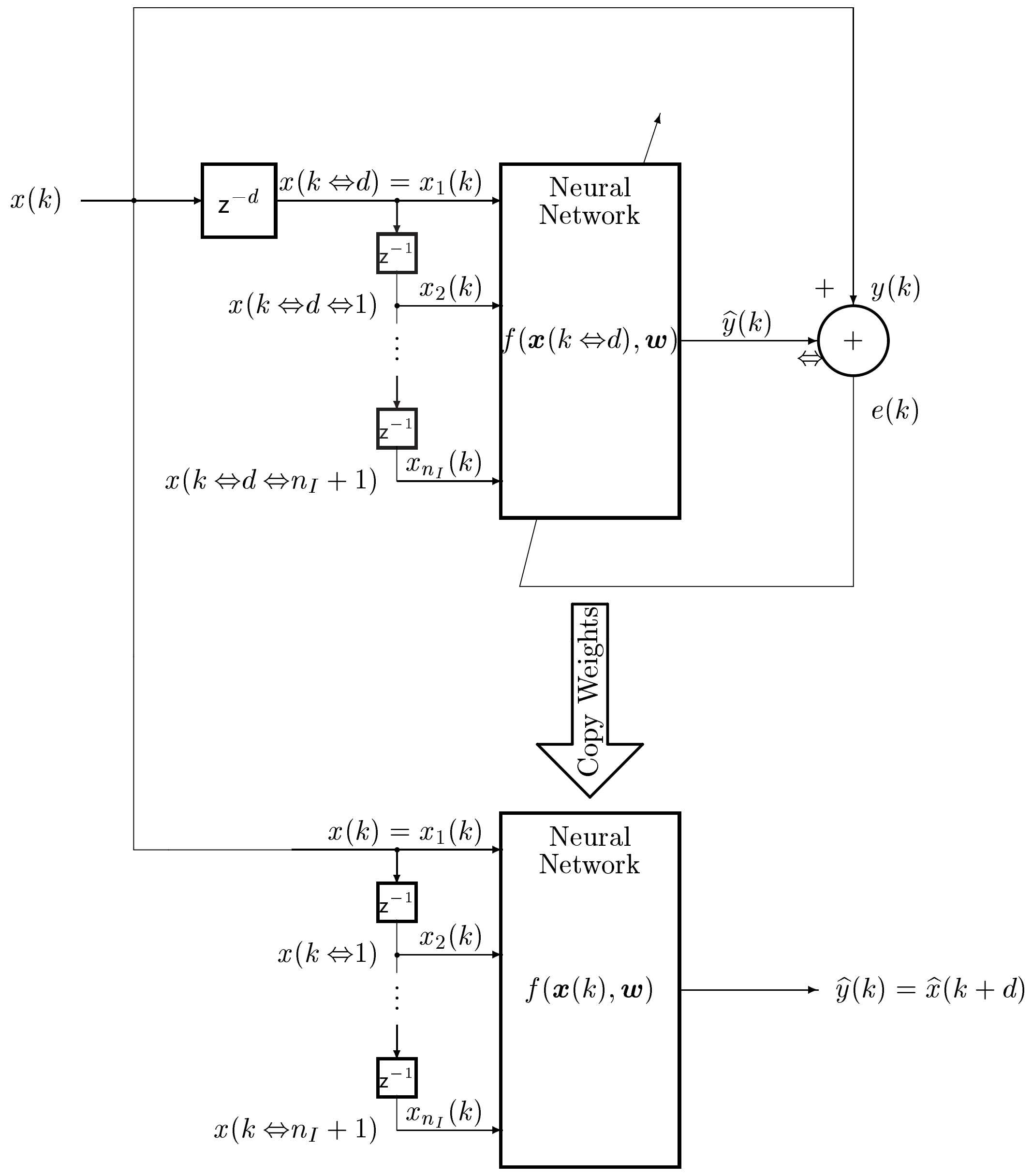
allows for an optimal class assignment. By assigning class $i = \arg\max_j p(\mathcal{C}_j|\boldsymbol{x})$ to input pattern $\boldsymbol{x}$, the probability of misclassification (misclassification percentage) is minimized according to Bayes rule [1]. Following [6], the outputs, $\widehat{y}_1, \cdots, \widehat{y}_c$ of the neural network represent *estimates* of the true conditional probabilities $p(\mathcal{C}_i|\boldsymbol{x})$, then the number of misclassifications is minimized by assigning class $\mathcal{C}_i$ to the input pattern $\boldsymbol{x}$ for which $\widehat{y}_i$ is maximum. The network is then a (n_I, n_H, c) network. See Appendix B for a detailed description of the neural network classifier.

3 Neural Network Training

Training or adaptation of the neural network for a specific task (regression/classification) is done by adjusting the weights of the network so as to minimize the discrepancy between the network output and the desired output on a set of collected *training data*:

$$\mathcal{T} = \{\boldsymbol{x}(k), y(k)\}_{k=1}^{N_{\text{train}}} \tag{14}$$

where k index the k'th training example (e.g., time index k) and N_{train} is the number of available training examples. $y(k)$ is the target (desired) output associated with input $\boldsymbol{x}(k)$. The output is for simplicity assumed to be a scalar, although the results are easily modified to allow for vector outputs.



$x(k)$
z^{-d}
$x(k \Leftrightarrow d) = x_1(k)$
z^{-1}
$x_2(k)$
$x(k \Leftrightarrow d \Leftrightarrow 1)$
$x_{n_I}(k)$
$x(k \Leftrightarrow d \Leftrightarrow n_I + 1)$
Neural Network
$f(\boldsymbol{x}(k \Leftrightarrow d), \boldsymbol{w})$
$\widehat{y}(k)$
$+$
$y(k)$
$\Leftrightarrow$
$e(k)$
Copy Weights
$x(k) = x_1(k)$
$x(k \Leftrightarrow 1)$
$x(k \Leftrightarrow n_I + 1)$
$f(\boldsymbol{x}(k), \boldsymbol{w})$
$\widehat{y}(k) = \widehat{x}(k + d)$

Figure 14: Nonlinear d step ahead prediction. In the adaptation phase, the objective of the neural network is to predict $x(k)$ from the delayed signal vector $\boldsymbol{x}(k \Leftrightarrow d)$. Once the network is adapted, the d step ahead prediction is obtained by feeding a neural network with $\boldsymbol{x}(k)$ using the adapted weights $\boldsymbol{w}$. Thus this copy of the network produces an prediction of $x(k + d)$.

The rest of the note will focus on regression/time-series processing problems although the techniques easily are adopted for classification problems, see Appendix B.

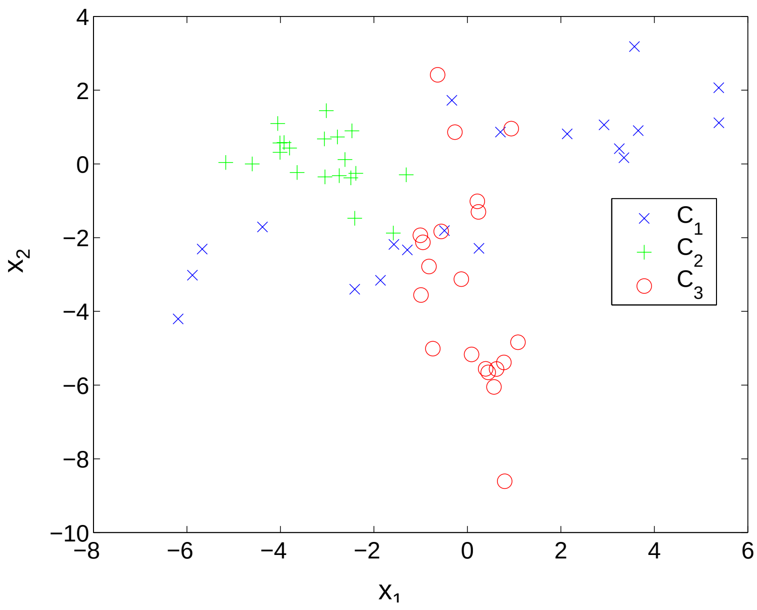


Figure 15: 3 classes in a 2D input space. The objective of classification is to separate classes by learning optimal decision boundaries from a set of training patterns. Once decision boundaries are learned new patterns can be automatically classified. Each pattern uniquely belongs to a class; however, the classes may be overlapping input space, thus the best classifier still will misclassify a certain percentage of the patterns.

As a measure of performance for specific weights using the available training data we often use the mean square error (MSE) cost function[5]

$$\begin{aligned} S_{\mathcal{T}}(\boldsymbol{w}) &= \frac{1}{2N_{\text{train}}} \sum_{k=1}^{N_{\text{train}}} (y(k) \Leftrightarrow \widehat{y}(k))^2 \\ &= \frac{1}{2N_{\text{train}}} \sum_{k=1}^{N_{\text{train}}} (y(k) \Leftrightarrow f(\boldsymbol{x}(k), \boldsymbol{w}))^2 \\ &= \frac{1}{2N_{\text{train}}} \sum_{k=1}^{N_{\text{train}}} e^2(k) \qquad (15) \end{aligned}$$

The value of the cost function is for each example k obtained by applying the input $\boldsymbol{x}(k)$ to the network using weights $\boldsymbol{w}$ and comparing the output of the network $\widehat{y}(k)$ with the desired output $y(k)$. The cost function is positive expect when choosing weights for which $e(k) \equiv 0$ for all examples. The smaller cost, the smaller is the network error on average over training examples.

The cost function $S_{\mathcal{T}}(\boldsymbol{w})$ is a continuous differentiable function of the weights and the training is done by adjusting the $m = n_I \cdot n_H + 2 \cdot n_H + 1$ weights contained in $\boldsymbol{w}$ so as to minimize the cost function. Multivariate function minimization is a well-known topic in numerical analysis and numerous techniques exist.

[5]The factor of 2 in the denominator is included by convenience only.

The cost function is generally not a convex function of the weights which means that there exist many local optima and further the global optimum is not unique. There are no practical methods which are guaranteed to yield the global optimum in reasonable time so one normally resort to searching for a *local optimum*. The necessary condition for a local optimum $\widehat{\boldsymbol{w}}$ (maximum,minimum,saddle point) is that the *gradient* of the cost function with respect to the weights are zero, as shown by

$$\boldsymbol{\nabla} S_{\mathcal{T}}(\widehat{\boldsymbol{w}}) = \left.\frac{\partial S_{\mathcal{T}}(\boldsymbol{w})}{\partial \boldsymbol{w}}\right|_{\boldsymbol{w}=\widehat{\boldsymbol{w}}} = \left[\left.\frac{\partial S_{\mathcal{T}}(\boldsymbol{w})}{\partial w_1}\right|_{\boldsymbol{w}=\widehat{\boldsymbol{w}}}, \cdots, \left.\frac{\partial S_{\mathcal{T}}(\boldsymbol{w})}{\partial w_m}\right|_{\boldsymbol{w}=\widehat{\boldsymbol{w}}} \right]^{\top} = \mathbf{0} \qquad (16)$$

This set of m equations are signified the *normal equations*. It should be stressed that determining one minimum $\widehat{\boldsymbol{w}}$ from all training data is an off-line method as opposed to on-line methods like the LMS algorithm for linear adaptive filters [17] which (in principle) continuously determines the optimal solution for each sample.

Example 3.1 Considering a single linear unit neural network, the model is $y(k) = \boldsymbol{w}^{\top}\boldsymbol{x}(k) + e(k)$. The cost function is quadratic in the weights, the optimal solution is unique and corresponds to the Wiener filter [13, Ch. 11].

■

3.1 Gradient Descend

Gradient descend is an iterative technique for solving the normal equations in Eq. (16). From an arbitrary initial guess $\boldsymbol{w}^{(0)}$ a change $\Delta\boldsymbol{w}^{(0)}$ which ensures a descend in the cost function. Thus if the weights iteratively are updated according to

$$\boldsymbol{w}^{(j+1)} = \boldsymbol{w}^{(j)} + \eta \Delta \boldsymbol{w}^{(j)} \qquad (17)$$

where $\boldsymbol{w}^{(j)}$ denotes the solution in iteration j, and $\eta > 0$ is a suitable *step-size*, then the cost is assured to decrease.

In gradient descend, $\Delta\boldsymbol{w}^{(j)} = \Leftrightarrow \boldsymbol{\nabla} S_{\mathcal{T}}(\boldsymbol{w}^{(j)})$. The the update is thus chosen as the direction where the cost has the steepest descend, i.e., in the direction of the negative gradient.

In order to perform gradient descend the partial derivatives of the cost function for the neural network is required. This is the topic of Section 3.2. Moreover a suitable step-size needs to be selected which is discussed below.

As stopping criterion for training more possibilities exist. An obvious choice is to terminate training when the change in cost function from one iteration to another is small, i.e.,

$$S_{\mathcal{T}}(\boldsymbol{w}^{j}) \Leftrightarrow S_{\mathcal{T}}(\boldsymbol{w}^{(j+1)}) < \tau_{\text{cost}} \qquad (18)$$

where τ_{cost} is a small constant. Using this stopping criterion does, however, not ensure that the weights are in the vicinity of a minimum which is determined by the condition $\boldsymbol{\nabla} S_{\mathcal{T}}(\boldsymbol{w}) = \mathbf{0}$. Another stopping criterion is thus

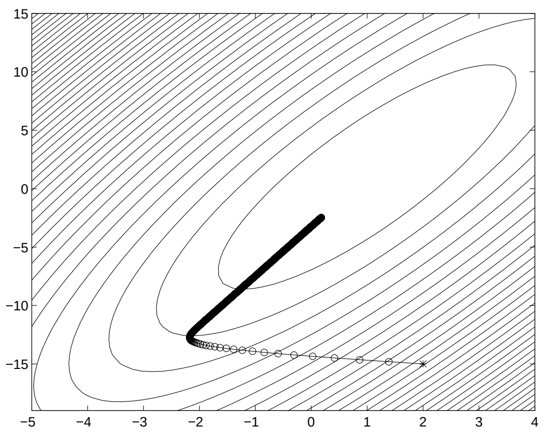
$$\|\boldsymbol{\nabla} S_{\mathcal{V}}(\boldsymbol{w}^{(j)})\|_2 < \tau_{\text{grad}} \qquad (19)$$

where τ_{grad} is a suitable small constant. $\|\boldsymbol{u}\|_2 = \sum_i u_i^2$ denotes the 2-norm or Euclidean length.

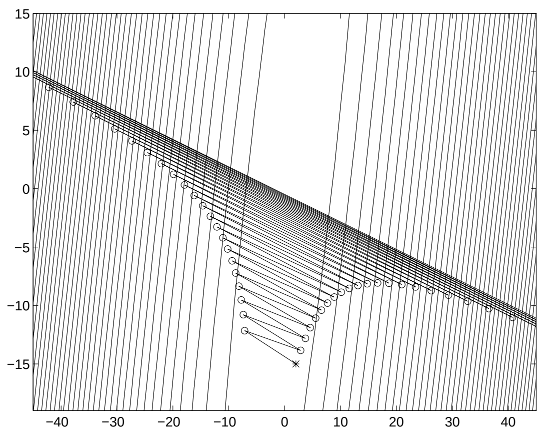
Generally the neural network training is time consuming, many iteration in Eq. (17) is normally required. Thus usually also a limit on the maximum number of iterations is desirable.

3.1.1 Choosing a Fixed Step-Size, η

The most simple choice is to keep the step-size – also denoted the learning rate – fixed and constant in every iteration. The convergence, however, is very sensitive to the choice. If η is very small, the change in weights are small, and often the assured decrease in the cost is also very small. This consequently leads to a large number of iterations to reach the (local) minimum. On the other hand, if the η is rather large, the cost function may increase and divergence is possible. Fig. 16 illustrates these situations for a simple quadratic cost function with two weights and minimum in $\boldsymbol{w} = [1, 1]^\top$. The optimal



(a)



(b)

Figure 16: The figures show contours of the cost function as well as the trace of iteration when performing gradient descent for quadratic cost function with two weights. The iterations starts in $(2, \Leftrightarrow 15)$ indicated by an asterisk. *Panel (a)*: 500 iterations using small step-size $\eta = 0.01$. The iterations are very close and after some time the trace shift it direction toward the valley of the minimum. The gradient in this valley is low, and since η also is small, the total number of iterations become pretty large. *Panel (b)*: Large step-size $\eta = 0.15$. In each iteration the cost function increases and divergence is inevitable.

choice of the step-size is in-between the extreme values in Fig. 16. The choice of η is very problem dependent, so the straight forward strategy is trial and error. In panel (a) of Fig. 17 the training is done with $\eta = 0.1$ and the stopping criterion is $\|\boldsymbol{\nabla} S_{\mathcal{T}}\|_2 < 0.01$ is obtained in 207 iterations.

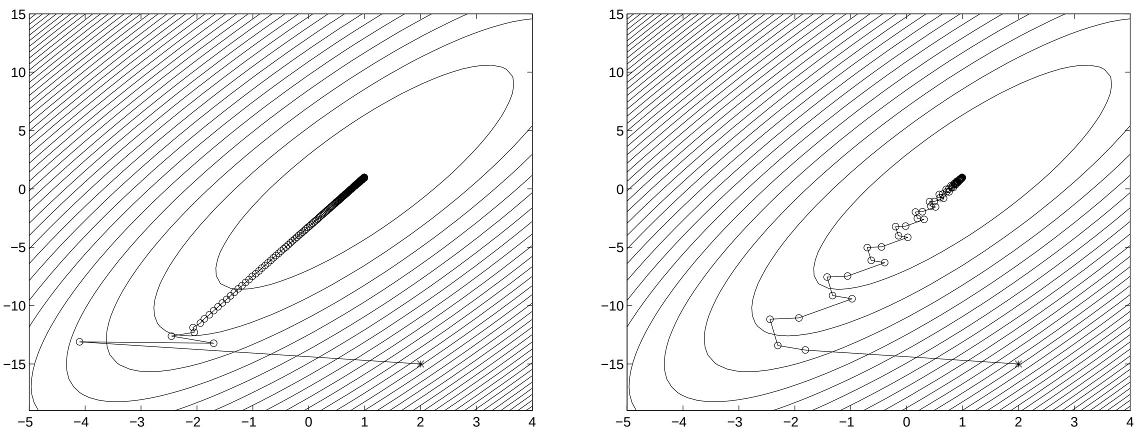
3.1.2 Choosing Step-Size by Line Search

Using line search, the step-size is adapted in each iteration. Exact line search is done by choosing η so as to minimize $S_{\mathcal{T}}(\boldsymbol{w}^{(j)} + \eta \cdot \Delta\boldsymbol{w}^{(j)})$. Exact line search is time consuming which leads to various inexact line search techniques which consist in choosing η so that the cost function is decrease

$$S_{\mathcal{T}}(\boldsymbol{w}^{(j)} + \eta \cdot \Delta\boldsymbol{w}^{(j)}) < S_{\mathcal{T}}(\boldsymbol{w}^{(j)}) \tag{20}$$

A simple heuristic method is bisection which is summarized in the following algorithm:

1. Initialize: $\eta = 1$, $\Delta\boldsymbol{w}^{(j)} = \Leftrightarrow\boldsymbol{\nabla} S_{\mathcal{T}}(\boldsymbol{w}^{(j)})$.



(a) (b)

Figure 17: *Panel (a)*: Training using an appropriate step-size. *Panel (b)*: Training using simple line search.

2. **while** $S_{\mathcal{T}}(\boldsymbol{w}^{(j)} + \eta \Delta \boldsymbol{w}^{(j)}) > S_{\mathcal{T}}(\boldsymbol{w}^{(j)})$
3. $\eta \leftarrow 0.5 \cdot \eta$.
4. **end**

In panel (b) of Fig. 17 training is shown when using bisection line search. An additional benefit of automatic step-size selection is improved convergence as compared with the appropriate fixed η in panel (a) of Fig. 17. The stopping criterion $\|\boldsymbol{\nabla} S_{\mathcal{T}}\|_2 < 0.01$ is now obtained in only 72 iterations. Experience indicates that convergence using bisection line search is as good or better than a fixed appropriate step-size.

3.2 Backpropagation

Gradient descent training techniques require the computation of the gradient $\boldsymbol{\nabla} S_{\mathcal{T}}(\widehat{\boldsymbol{w}})$ in each iteration. This section provides a detailed derivation of the gradient computation which reveals a computationally efficient structure: *backpropagation of errors* [11].

The gradient vector $\partial S_{\mathcal{T}}(\boldsymbol{w})/\partial \boldsymbol{w}$ is according to Eq. (15)

$$\begin{aligned} \frac{\partial S_{\mathcal{T}}(\boldsymbol{w})}{\partial w_i} &= \frac{1}{N_{\text{train}}} \sum_{k=1}^{N_{\text{train}}} \frac{\partial e^2(k)}{\partial \boldsymbol{w}} \\ &= \frac{1}{N_{\text{train}}} \sum_{k=1}^{N_{\text{train}}} e(k) \frac{\partial e(k)}{\partial w_i} \\ &= \Leftrightarrow \frac{1}{N_{\text{train}}} \sum_{k=1}^{N_{\text{train}}} e(k) \frac{\partial \widehat{y}_i(k)}{\partial w_i} \end{aligned} \tag{21}$$

as $e(k) = y(k) \Leftrightarrow \widehat{y}_i(k) = y(k) \Leftrightarrow f(\boldsymbol{x}(k), \boldsymbol{w})$.

According to Eq. (2)

$$\widehat{y}_i(k) = \psi(u_i^O(k)) = \psi\left(\sum_{j=1}^{n_H} w_{ij}^O h_j(k) + w_{i0}^O\right) \tag{22}$$

where $u_i^O(k) = \boldsymbol{h}^\top \boldsymbol{w}_i^O$ is the linear input to output neuron i. Recall that only single output networks ($n_O = 1$, i.e., $i = 1$) are studied in this section. The derivative w.r.t. hidden-output weights is accordingly

$$\frac{\partial \widehat{y}_i(k)}{\partial w_{ij}^O} = \psi'(u_i^O(k))h_j(k) \tag{23}$$

where $\psi'(u)$ is the derivative. If $\psi(u) = \tanh(u)$ then $\psi'(u)) = 1 \Leftrightarrow \tanh^2(u) = 1 \Leftrightarrow \psi^2(u)$. Using Eq. (21)

$$\frac{\partial S_{\mathcal{T}}(\boldsymbol{w})}{\partial w_{ij}^O} = \Leftrightarrow \frac{1}{N_{\text{train}}} \sum_{k=1}^{N_{\text{train}}} \delta_i^O(k) h_j(k) \tag{24}$$

with $\delta_i^O(k) = e(k)\psi'(u_i^O(k))$.

The derivatives w.r.t. input-hidden weights are found using the chain rule:

$$\begin{aligned} \frac{\partial \widehat{y}_i(k)}{\partial w_{j\ell}^I} &= \frac{\partial \widehat{y}_i(k)}{\partial h_j(k)} \cdot \frac{\partial h_j(k)}{\partial w_{j\ell}^I} \\ &= \psi'(u_i^O(k))w_{ij}^O \cdot \psi'(u_j^I)x_\ell(k) \end{aligned} \tag{25}$$

as Eq. (1) reads

$$h_j(k) = \psi(u_j^I(k)) = \psi\left(\sum_{l=1}^{n_I} w_{j\ell}^I x_\ell + w_{j0}^I\right) \tag{26}$$

Combining Eq. (21) and (25) yields:

$$\begin{aligned} \frac{\partial S_{\mathcal{T}}(\boldsymbol{w})}{\partial w_{j\ell}^I} &= \Leftrightarrow \frac{1}{N_{\text{train}}} \sum_{k=1}^{N_{\text{train}}} \delta_i^O(k) w_{ij}^O \cdot \psi'(u_j^I) x_\ell(k) \\ &= \Leftrightarrow \frac{1}{N_{\text{train}}} \sum_{k=1}^{N_{\text{train}}} \delta_j^I(k) x_\ell(k) \end{aligned} \tag{27}$$

with $\delta_j^I(k) = \delta_i^O(k) w_{ij}^O \cdot \psi'(u_j^I)$. Notice that Eq. (24) and (27) has the same *structure* albeit different definition of the error δ.

For a layered neural network with an arbitrary number of layers it can be shown that the derivative of the cost function will have the general structure

$$\frac{\partial S_{\mathcal{T}}(\boldsymbol{w})}{\partial w} = \Leftrightarrow \frac{1}{N_{\text{train}}} \sum_{k=1}^{N_{\text{train}}} \delta_{\text{to}} x_{\text{from}} \tag{28}$$

where δ_{to} is the error for the unit *to* which the weight is connected, and x_{from} is the linear activation *from* the unit to which the weight is connected. Due to the propagation of the error signal $e(k)$ backwards in the network as via the δ signals, the algorithm was named backpropagation [11], see also Fig. 18.

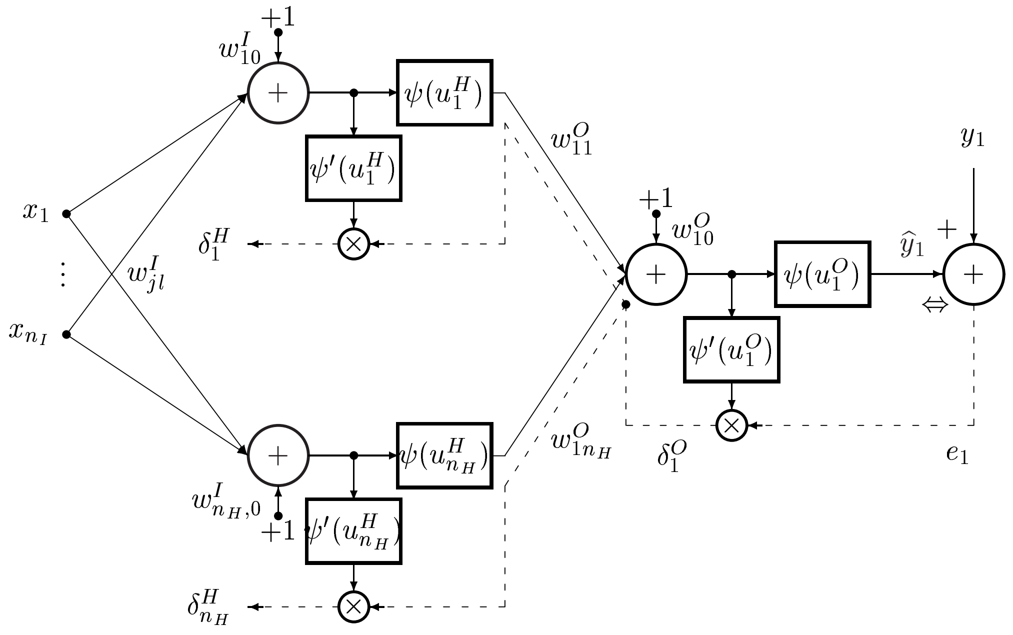


Figure 18: Backpropagation of errors: first signals are fed forward through the solid connections, then the error is formed at the output and propagated backwards through the dashed lines to compute δ's, i.e., is the error of the individual neuron.

3.2.1 Summary of Gradient Descend Algorithm

1. Pick a network structure, i.e., n_I, n_H and $n_O = 1$.
2. Initialize the weights of network $\boldsymbol{w}^{(o)}$ randomly so that the neurons are not saturated (close to ± 1) nor operating in the linear region.
3. For all training examples pass the input through the network to produce hidden activations $h_j(k)$ and outputs $\hat{y}(k)$.
4. Compute error signal $e(k) = y(k) \Leftrightarrow \hat{y}(k)$.
5. Compute gradients of the cost function using backpropagation, i.e., first $\delta^O(k)$ and gradients of the hidden-output weights; then $\delta^I_j(k)$ and gradients of the input-hidden weights.
6. Perform line search via bisection.
7. Update weight estimate.
8. Is stopping criterion fulfilled. If yes stop; otherwise go to 3.

4 Generalization

When the network is trained on N_{train} examples to yield minimal cost the aim is to apply the trained network on future data, see e.g., the time-series prediction case in Fig. 14. In general, there is no theory supporting that the network also performs well on future data, consequently a validation procedure is required. Suppose that $\widehat{\boldsymbol{w}}$ is the trained weights then the *generalization error* is defined as the expected square error on an arbitrary test sample $(\boldsymbol{x}, y)$ which is independent on training samples, given by

$$G(\widehat{\boldsymbol{w}}) = E_{\boldsymbol{x},y}\left[(y \Leftrightarrow f(\boldsymbol{x}, \widehat{\boldsymbol{w}}))^2\right] \tag{29}$$

where $E_{\boldsymbol{x},y}[\cdot]$ denotes statistical expectation w.r.t. both y and $\boldsymbol{x}$. Note that the generalization error depends on the training data and the number of examples through the estimated weights $\widehat{\boldsymbol{w}}$. The generalization error as a function of number of examples is referred to as the *learning curve* [5]. Obviously the learning curve in general is a decreasing function of the number of training examples. However, in some cases a so-called learning transition is noticed: for a critical number of examples the generalization error drops dramatically. That is, the number of examples needs to be greater than the critical number in order to learn the underlying problem.

The joint probability function of $\boldsymbol{x}$ and y is generally unknown so the generalization error needs to be estimated from data. Ultimately, if one have a large set of independent test examples the ensemble average can be substituted by a example (time) average provided the signals are ergodic [8], i.e., the generalization estimate (test error) becomes

$$\widehat{G}(\widehat{\boldsymbol{w}}) = \frac{1}{N_{\text{test}}} \sum_{k=1}^{N_{\text{test}}} (y(k) \Leftrightarrow f(\boldsymbol{x}, \widehat{\boldsymbol{w}})^2 \tag{30}$$

where k index the test samples. Often lack of data preclude estimating generalization error this way. Other methods for generalization error estimation are viable [9] but beyond the scope of this note.

Example 4.1 Consider a simple linear system identification where the true relation between input and output signals is given by

$$y(k) = w^* x(k) + \varepsilon(k) \tag{31}$$

where w^* is the true weight and $\varepsilon(k)$ is a white random signal with variance σ_ε^2 which is independent on the input $x(k)$. Further the input signal is assumed to be white with variance σ_x^2.

The linear adaptive filter model is deployed,

$$y(k) = wx(k) + e(k) \tag{32}$$

where $e(k)$ is the error signal. Minimizing the mean square error cost Eq. (15) on N_{train} examples lead cf. [13, Ch. 11] to the estimate

$$\widehat{w} = \frac{r_{xy}(0)}{r_{xx}(0)} \tag{33}$$

where $r_{xy}(0) = N_{\text{train}}^{-1} \sum_{k=1}^{N_{\text{train}}} x(k)y(k)$ is the estimated crosscorrelation function at lag zero and $r_{xx}(0) = N_{\text{train}}^{-1} \sum_{k=1}^{N_{\text{train}}} x^2(k)$ is the estimated autocorrelation function at lag zero.

The generalization error is then by using Eq. (31)

$$G(\widehat{w}) = E_{x,y}[(y \Leftrightarrow \widehat{w}x)^2] = E_{x,\varepsilon}[((w^* \Leftrightarrow \widehat{w})x + \varepsilon)^2] = (w^* \Leftrightarrow \widehat{w})^2\sigma_x^2 + \sigma_\varepsilon^2 \tag{34}$$

$G(\widehat{w}) \geq \sigma_\varepsilon^2$ and the minimum is obtained when the estimate equal the true weight, $w^* = \widehat{w}$. As $N_{\text{train}} \to \infty$, $\widehat{w}$ in Eq. (33) tends to the Wiener solution w^*.

■

4.1 Overtraining

If the network is to complex, i.e., has to many weights and the input-output relation is noisy the network often learns the noise. That is, the network performs very well on the training data but will possess large generalization error. In Fig. 19 overtraining is illustrated. There is always the hidden agenda in modeling: the ultimate goal is to minimize

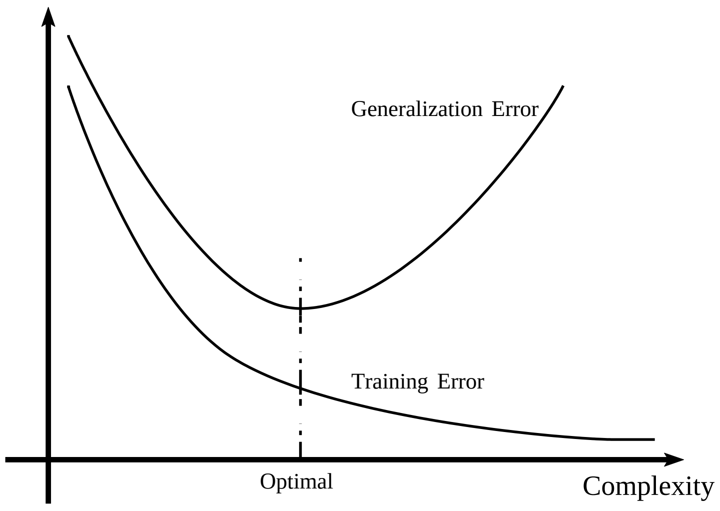


Figure 19: Overtraining: when the network complexity (number of weights) is large the network learns the noise in the training data and the resulting generalization error is high. On the other hand, when the complexity is low, the approximation capability of the network is limited cf. the universal approximation theorem Section 2.2. Thus the network commits many systematic errors and both the training and generalization errors are large. As a consequence, there exists an optimal complexity trade off.

generalization error rather than training error; however, only the training data are available. Are we facing a dilemma? Fortunately it possible to estimate generalization error, e.g., by reserving some data for testing and compute the estimate via Eq. (30). Then overtraining can be prevented by choosing a network with low estimated generalization error. The drawback is that many test examples are necessary in oder to estimate the generalization reliably. Another possibility for preventing overfitting is do reduce the network complexity by removing some of the weights. The *Optimal Brain Damage* (OBD) method [10] is a systematic method for pruning weights in the network. OBD is further discussed in Section 5. Finally, regularization can be used to prevent overtraining. Section 5 discusses the use of weight decay regularization.

4.2 Local Minima

Often many almost equally good solution exist. Due to symmetry properties of the network many solution are equal, e.g., the hidden neurons can be interchanged without altering the output. In other local minima, the error on a large number of training examples will be low but the error large on the remaining examples. Depicting the cost function as a mountain scenery, these local minima will be high altitude valleys. In such valleys the gradient descent algorithm can be get stuck for many iterations.

Both overfitting and local minima leads to the recommendation that the overall training procedure is replicated a number of times using different initial conditions. In addition to more good networks also it is possible to evaluate the statistical nature of the training.

5 Neural Network Architecture Optimization

Selection of neural network architecture/structure for a specific problem is very involved. Once the structure is selected, e.g, a feed-forward neural network, it needs to be optimized in order to ensure low generalization error and consequently avoiding overfitting.

5.1 Regularization Using Weight Decay

A simple regularization method is weight decay which in practice is accomplished by augmenting the cost function by a penalty term which penalizing high magnitude weights, as shown by

$$C_{\mathcal{T}}(\boldsymbol{w}) = S_{\mathcal{T}}(\boldsymbol{w}) + \frac{\kappa}{2}\sum_i w_i^2 \tag{35}$$

where $\kappa = \alpha/N_{\text{train}}$ is a positive normalized weight decay constants and α the weight decay. The regularization term forces the weight magnitudes against zero, as the cost is large for large magnitude weights. That is, (local) minima far from origo are blurred. In gradient descend, the weight decay gives a decay proportional to the weight, $w_i^{(j+1)} = (1 \Leftrightarrow \alpha/N_{\text{train}})w_i^{(j)}$. If a weight is superfluous it will slowly decay to zero, whereas essential weights only will be slightly influenced by the decay, see also [5].

5.2 Architecture Optimization

Weight decay regularization limits the dynamical range for the weights but they are not necessarily set to zero. The optimal architecture can be obtained by two different strategies. The first is a growing architecture in which starts from a minimal network, e.g., a single neuron. Then additional neurons are added during training when required. The second strategy consists in training a relatively large network which subsequently is pruned be removing weights (or neurons) until an optimal architecture is achieved, where minimal generalization error normally is used as criterion.

The pruning algorithm is summarized in

1. Train a sufficiently large network.

2. Rank weights according to importance.

3. Eliminate the least significant weight(s).

4. Retrain the pruned network. If generalization error is minimal stop; otherwise go to step 2.

5.2.1 Optimal Brain Damage

Optimal Brain Damage [10] is a method to determine the importance of a weight which is referred to as *saliency*. Saliency is defined as the change in mean square error cost function due to eliminating a specific weight.

Suppose that $\widehat{\boldsymbol{w}}$ minimizes the augmented cost function $C_{\mathcal{T}}(\boldsymbol{w})$ in Eq. (35). A second order Taylor series expansion of the mean square error cost $S_{\mathcal{T}}(\boldsymbol{w})$ is

$$S_{\mathcal{T}}(\widehat{\boldsymbol{w}} + \delta \boldsymbol{w}) = S_{\mathcal{T}}(\widehat{\boldsymbol{w}}) + \delta \boldsymbol{w}^{\top} \boldsymbol{\nabla} S_{\mathcal{T}}(\widehat{\boldsymbol{w}}) + \frac{1}{2} \delta \boldsymbol{w}^{\top} \boldsymbol{H}(\widehat{\boldsymbol{w}}) \delta \boldsymbol{w} \tag{36}$$

where $\delta \boldsymbol{w}$ is the weight change and $\boldsymbol{H}(\widehat{\boldsymbol{w}})$ is the Hessian matrix of the cost function in $\widehat{\boldsymbol{w}}$, i.e., the second order derivative w.r.t. the weights. As $\widehat{\boldsymbol{w}}$ is assumed to be a local minimum, then $\boldsymbol{\nabla} C_{\mathcal{T}}(\widehat{\boldsymbol{w}}) = \boldsymbol{\nabla} S_{\mathcal{T}}(\widehat{\boldsymbol{w}}) + \kappa \boldsymbol{w} = \boldsymbol{0}$ which means $\boldsymbol{\nabla} S_{\mathcal{T}}(\widehat{\boldsymbol{w}}) = \Leftrightarrow \kappa \boldsymbol{w}$. Using this expression in Eq. (36) gives

$$\begin{aligned} \delta S_{\mathcal{T}} &= S_{\mathcal{T}}(\widehat{\boldsymbol{w}}) + \delta \boldsymbol{w}) \Leftrightarrow S_{\mathcal{T}}(\widehat{\boldsymbol{w}}) \\ &= \Leftrightarrow \delta \boldsymbol{w}^{\top} \kappa \widehat{\boldsymbol{w}} + \frac{1}{2} \delta \boldsymbol{w}^{\top} \boldsymbol{H}(\widehat{\boldsymbol{w}}) \delta \boldsymbol{w} \end{aligned} \tag{37}$$

If the perturbation $\delta \boldsymbol{w}$ corresponds to eliminating the j'th weight then

$$\delta \boldsymbol{w} = [0, \cdots, 0, \Leftrightarrow \widehat{w}_j, 0, \cdots, 0]^{\top} \tag{38}$$

The change in MSE cost is cf. Eq. (37)

$$\delta S_{\mathcal{T}j} = \left(\kappa + \frac{1}{2} \frac{\partial^2 S_{\mathcal{T}}(\widehat{\boldsymbol{w}})}{\partial w_j^2} \right) \widehat{w}_j^2 \tag{39}$$

In OBD it is assumed that the change in MSE cost when eliminating more weights simultaneously can be approximated by the sum of the individual changes. This corresponds to assuming off-diagonal terms in the Hessian are zero ,i.e.,

$$\frac{\partial^2 S_{\mathcal{T}}(\widehat{\boldsymbol{w}})}{\partial w_i \partial w_j} = 0, \;\; i \neq j \tag{40}$$

For networks trained using the MSE cost function the following approximation of the second derivative can be applied:

$$\frac{\partial^2 S_{\mathcal{T}}(\boldsymbol{w})}{\partial w_i^2} \approx \frac{1}{2N_{\text{train}}} \sum_{k=1}^{N_{\text{train}}} \left(\frac{\partial y(k)}{\partial w_i} \right)^2 \tag{41}$$

Note that $\partial y(k)/\partial w_i$ is easily computed using backpropagation.

OBD algorithm:

1. Select a fully connect network with sufficient number of hidden neurons.
2. For the current network architecture compute the weight estimate $\widehat{\boldsymbol{w}}$ by minimizing the augmented cost function $C(\boldsymbol{w})$.
3. Compute saliencies $\delta S_{\mathcal{T}j}$ for all weights.
4. Rank weights according to saliency.

5. Eliminate a number of weights with small saliencies[6].

6. Compute an estimate of the generalization error and check if has reached its minimum. If not go to step 2; otherwise stop.

In Fig. 20 the elapse of training and test errors during OBD pruning is shown. The example is produced with the software presented in Section A. Both training and test sets comprised 100 examples generated by a "random" true network with 4 inputs, 2 hidden neurons and one output. In total the true network has 13 weights. The model network has initially 4 inputs, 5 hidden neurons and one output, i.e., 31 weights. Weight decay A weight decay of $\alpha = 0.01$ was deployed. As expected, the training error increases

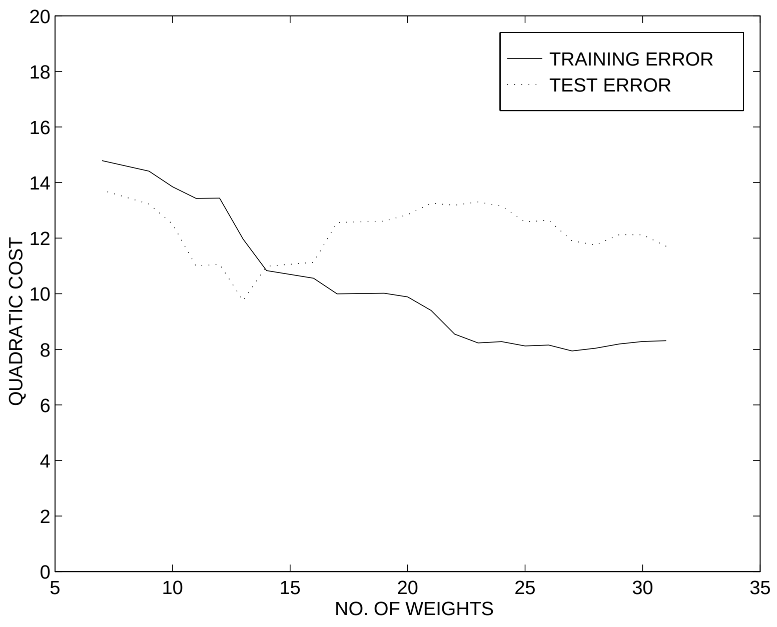


Figure 20: Elapse of training and test errors during OBD pruning. Notice that the pruning is from left to right, as the number of weights decreases during the pruning session.

as the number of weights decreases since the approximation capability is reduced. In principle, the training error should always increase with decreasing complexity; however, due to local minima this not always in the case. The test error increases somewhat in the beginning of the pruning session, but decreases significantly when the number of weights gets closer the true number, 13. This is due to the fact that the networks ability to adapt to specialities in the training set becomes less pronounced as the complexity decreases. When the number of weights are smaller than 13 the test error increases again since it becomes unable to reproduce the underlying rule. The trade off consist in having enough weights to implement the underlying rule but to few to model noise in the training set.

Fig. 21 shows the network architecture in different phases of the pruning.

[6]Normally one weight or a certain percentage of the remaining weights are pruned in each iteration

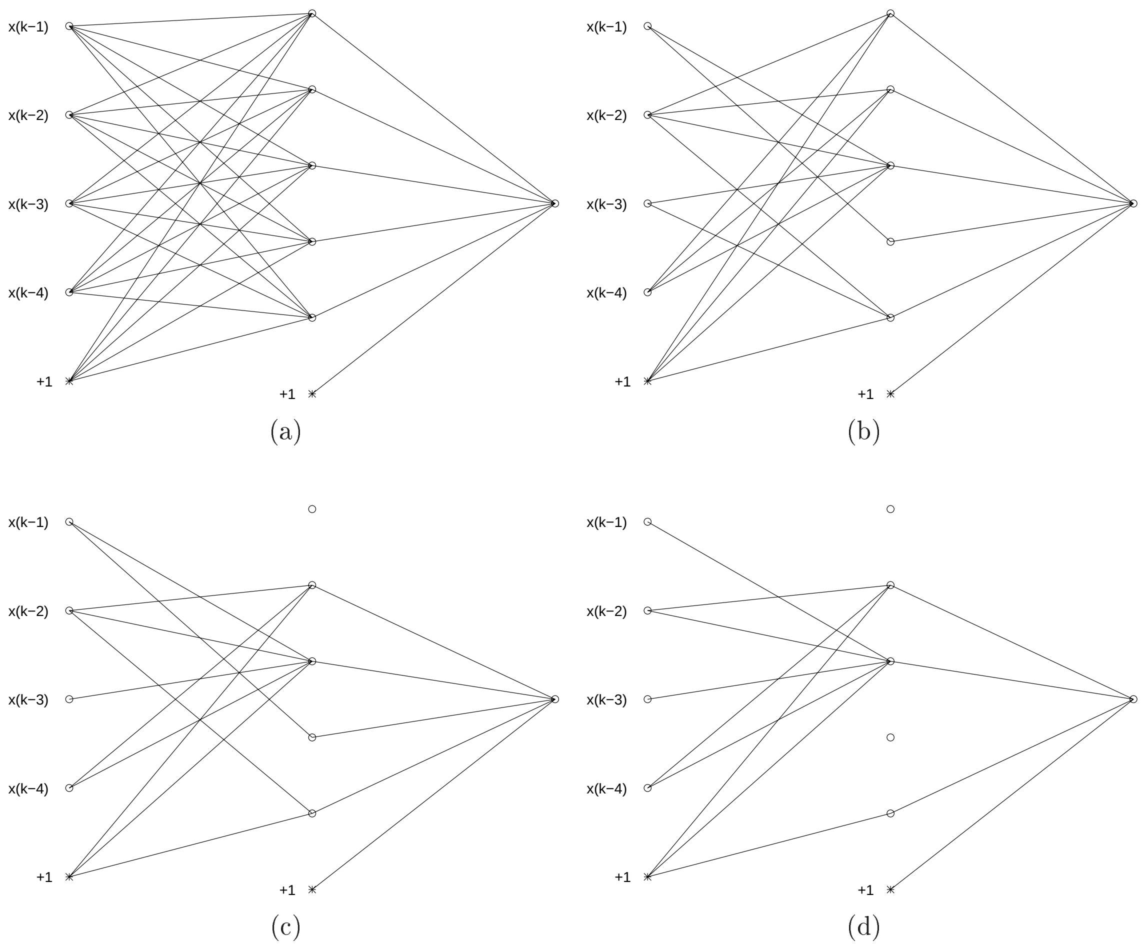


Figure 21: Network architecture during pruning. *Panel (a)*: 31 weight fully connected network. *Panel (b)*: Network with 21 weights and all hidden units. *Panel (c)*: network with 16 weights one hidden unit is removed. *Panel (d)*: 13 weight network with lowest test error. Compared to the 16 weight network, this network only uses 3 hidden neurons and the lower hidden unit receives input from the bias input only, i.e., the 3 lower weights can in principle be replaced by one common bias input for the output layer. That is, the resulting network effectively, like the true network, only uses 2 hidden neurons.

A Neural Network Regression Software

Suppose the number of training and test examples are N_{train}, N_{test}, respectively. Further, that the number of inputs, hidden and output neurons are n_I, n_H and n_O. The training data are required to be stored in the following matrices:

`train_inp`:	a $N_{\text{train}} \times n_I$ matrix of training input data. Row k is the k'th training example of the input vector $\boldsymbol{x}^\top(k)$.
`train_tar`:	a $N_{\text{train}} \times n_O$ matrix of training output/target data.

Similarly the test data are stored in `test_inp` and `test_tar`.

The weights of the network are assembled in the input-to-hidden matrix `Wi` and the hidden-to-output matrix `Wo` which are defined by

$$\mathtt{Wi} = \begin{bmatrix} \mathtt{Wi}_{11} & \mathtt{Wi}_{12} & \cdots & \mathtt{Wi}_{1,n_I} & \mathtt{Wi}_{1,\mathrm{bias}} \\ & & \ddots & & \\ \mathtt{Wi}_{n_H,1} & \mathtt{Wi}_{n_H,2} & \cdots & \mathtt{Wi}_{n_H,n_I} & \mathtt{Wi}_{n_H,\mathrm{bias}} \end{bmatrix} \tag{42}$$

where $\mathtt{Wi}_{j,\ell}$ is the weight from input ℓ to hidden neuron j. Further,

$$\mathtt{Wo} = \begin{bmatrix} \mathtt{Wo}_{11} & \mathtt{Wo}_{12} & \cdots & \mathtt{Wo}_{1,n_H} & \mathtt{Wo}_{1,\mathrm{bias}} \\ & & \ddots & & \\ \mathtt{Wo}_{n_O,1} & \mathtt{Wo}_{n_O,2} & \cdots & \mathtt{Wo}_{n_O,n_H} & \mathtt{Wi}_{n_O,\mathrm{bias}} \end{bmatrix} \tag{43}$$

where $\mathtt{Wo}_{i,j}$ is the weight from hidden unit j to output neuron i.

A.1 MATLAB Functions in the Neural Regression Package

A.1.1 Function Overview

`nr_netprun.m`	Main function.
`nr_calcul.m`	Calculates the cost function and the gradient.
`nr_cost_c.m`	Calculates the cost function augmented by weight decay.
`nr_cost_e.m`	Calculates the cost function.
`nr_dimen.m`	Calculates the number of non-zero weights.
`nr_extract.m`	Extraction of weight matrices from the reshaped vector.
`nr_forward.m`	Propagate examples forward through network calculating all hidden- and output unit outputs.
`nr_getdata.m`	Create input and output data from a teacher network.
`nr_gradient.m`	Calculate the partial derivatives of the cost function.
`nr_linesear.m`	Performs a simple line search.
`nr_linesearch.m`	Line search with Wolfe-Powell conditions.
`nr_plotnet.m`	Neural regression plot network.
`nr_plotsal.m`	Neural regression plot saliency.
`nr_prune.m`	Prunes a number of weights away using OBD.
`nr_pseuhess.m`	Calculates the pseudo Hessian (diagonal) elements and gradient of the cost function.
`nr_tanhf.m`	Fast hyperbolic tangent.
`nr_train.m`	Train the network with gradient descent followed by pseudo Gauss-Newton.

`nr_trainx.m`	Train network (conjugate gradient version).
`nr_two_norm.m`	Euclidean length of the total weight vector.
`nr_winit.m`	Uniform weight initialization.

A.1.2 Main Function `nr_netprun.m`

Pseudo Code

```
function nr_netprun
  1. Initialize algorithm parameters
  2. Initialize weights using nr_winit
  3. Train the network using nr_train
  4. Evaluate training and test performance using nr_cost_e and nr_err_frac
  5. while $\dim(\boldsymbol{w}) > \text{mindim}$ repeat
     1. Prune the network using prune
     2. Calculate dimension $\dim(\boldsymbol{w})$
     3. Retrain the network using train
     4. Evaluate training and test performance using nr_cost_e and nr_err_frac
     end
end
```

A.1.3 Subroutine `nr_calcul.m`

Call

```
function [f,df] = nr_calcul(W,Dim,alpha_i, alpha_o,Inputs,Targets);

%NR_CALCUL     Calculates the cost function and the gradient
%   [f,df] = nr_calcul(X, Dim, alpha_i, alpha_o, Inputs, Targets);
%   Calculates the cost function value f and the gradient df for
%   neural network the function is operating on the vector W created
%   by reshaping matrices Wi and Wo in Dim vector the dimensions of
%   those matrices are stored.
```

A.1.4 Subroutine `nr_cost_c.m`

Call

```
function [cost] = nr_cost_c(Wi,Wo,alpha_i,alpha_o,Inputs,Targets)
%NR_COST_C     Quadratic cost function with quadratic weight decay term
%   [cost] = NR_COST_C(Wi,Wo,alpha_i,alpha_o,Inputs,Targets)
%
%   Input:
%        Wi      :  Matrix with input-to-hidden weights
%        Wo      :  Matrix with hidden-to-outputs weights
%        alpha_i :  Weight decay parameter for input weights
%        alpha_o :  Weight decay parameter for output weights
%        Inputs  :  Matrix with examples as rows
%        Targets :  Matrix with target values as rows
%   Output:
%        Cost    : Value of augmented quadratic cost function
%
%   See also NR_COST_E
%
%   Neural Regression toolbox, DSP IMM DTU
```

A.1.5 Subroutine nr_cost_e.m

Call

```
function [error] = nr_cost_e(Wi,Wo,Inputs,Targets)
%NR_COST_E     Calculate the quadratic cost function
%   [error] = NR_COST_E(Wi,Wo,Inputs,Targets) calculates the value of
%   the quadratic cost function, i.e., 0.5*(sum of squared errors)
%
%   Input:
%        Wi      :  Matrix with input-to-hidden weights
%        Wo      :  Matrix with hidden-to-outputs weights
%        Inputs  :  Matrix with examples as rows
%        Targets :  Matrix with target values as rows
%   Output:
%        error   : Value of quadratic cost function
%
%   See also NR_COST_C
%
%   Neural Regression toolbox, DSP IMM DTU
```

A.1.6 Subroutine nr_dimen.m

Call

```
function [dim] = nr_dimen(Wi,Wo)
%NR_DIMEN      Number of non-zero-weights
%   [dim] = NR_DIMEN(Wi,Wo) calculates the number of non-zero weights
%   in the network, i.e. the dimension of the total weight vector
%
%   Input:
%        Wi      :  Matrix with input-to-hidden weights
%        Wo      :  Matrix with hidden-to-outputs weights
%   Output:
%        dim     :  Number of non-zero weights
%
%   Neural Regression toolbox, DSP IMM DTU
```

A.1.7 Subroutine nr_extract.m

Call

```
function [Wi,Wo] = nr_extract(WW,D)

%NR_EXTRACT    Extraction weight matrices from the reshaped vector
%   [Wi,Wo] = extract(WW,D)
%
%   Input:
%        WW : the vector of the dimensions D(1)+D(2)+D(3)+D(4)
%        D  : the vector with stored dimensions of weight matrices
%   Output:
%        Wi :  the matrix with input-to-hidden weights
%        Wo :  the matrix with hidden-to-output weights
```

A.1.8 Subroutine nr_forward.m

Call

```
function [Vj,yj] = nr_forward(Wi,Wo,Inputs)
%NR_FORWARD    Propagate example forward through the network
%   [Vj,yj] = NR_FORWARD(Wi,Wo,Inputs) propagates examples forward
%   through network calculating all hidden- and output unit outputs
%
%   Input:
%        Wi     :  Matrix with input-to-hidden weights
%        Wo     :  Matrix with hidden-to-outputs weights
%        inputs :  Matrix with example inputs as rows
%
%   Output:
%        Vj  :  Matrix with hidden unit outputs as rows
%        yj  :  Vector with output unit outputs as rows
%
%   Neural Regression toolbox, DSP IMM DTU
```

A.1.9 Subroutine nr_getdata.m

Call

```
function [tr_i,tr_t,te_i,te_t] = getdata(Ni,t_Nh,No,ptrain,ptest,noise)
%NR_GETDATA    Create input and output data from a teacher network
%    [tr_i,tr_t,te_i,te_t] = getdata(Ni,t_Nh,No,ptrain,ptest,noise)
%    creates input and output data from a 'teacher' network. The
%    outputs are contaminated with additive white noise.
%
%    Inputs:
%         Ni     :  Number of external inputs to net
%         t_Nh   :  Number of hidden units for the 'teacher' net
%         No     :  Number of output units
%         ptrain :  Number of training examples
%         ptest  :  Number of test examples
%         noise  :  Relative amplitude of additive noise
%    Outputs:
%         tr_i, te_i :  Inputs for training & test set
%         tr_t, te_t :  Target values
%
%    See also NR_NETPRUN
%
%    Neural Regression toolbox
```

A.1.10 Subroutine nr_gradient.m

Call

```
function [dWi,dWo] = nr_gradient(Wi,Wo,alpha_i,alpha_o,Inputs,Targets)
%NR_GRADIENT   Calculate the partial derivatives of the quadratic cost
%   [dWi,dWo] = nr_gradient(Wi,Wo,alpha_i,alpha_o,Inputs,Targets)
%   calculate the partial derivatives of the quadratic cost wrt. the
%   weights. Derivatives of quadratic weight decay are included.
%
%   Input:
%        Wi      :  Matrix with input-to-hidden weights
%        Wo      :  Matrix with hidden-to-outputs weights
%        alpha_i :  Weight decay parameter for input weights
%        alpha_o :  Weight decay parameter for output weights
%        Inputs  :  Matrix with examples as rows
%        Targets :  Matrix with target values as rows
%   Output:
%        dWi     :  Matrix with gradient for input weights
%        dWo     :  Matrix with gradient for output weights
%
%   See also NR_PSEUHESS, NR_TRAIN
%
%   Neural Regression toolbox, DSP IMM DTU
```

A.1.11 Subroutine nr_linesear.m

Call

```
function [eta] = nr_linesear(Wi, Wo, Di, Do, alpha_i, alpha_o, Inputs, ...
    Targets, pat)
%NR_LINESEAR   Simple linesearch
%    [eta] = linesear(Wi,Wo,Di,Do,alpha_i,alpha_o,Inputs,Targets,pat)
%    performs a simple linesearch in a direction in parameter space,
%    determining the 'optimal' steplength by iterative decrease.
%
%    Input:
%        Wi      :  Matrix with input-to-hidden weights
%        Wo       :  Matrix with hidden-to-outputs weights
%        Di       :  Matrix with input search direction
%        Do       :  Matrix with output search direction
%        alpha_i :  Weight decay parameter for input weights
%        alpha_o :  Weight decay parameter for output weights
%        Inputs  :  Matrix with examples as rows
%        Targets :  Matrix with target values as rows
%        pat      :  Patience; max number of decreases
%    Output:
%        eta      :  'Optimal' step length
%
%    See also NR_TRAIN, NR_COST_C
%
%    Neural Regression toolbox, DSP IMM DTU
```

A.1.12 Subroutine nr_linesearch.m

Call

```
function [f2,df2,WW,eta] = nr_linesearch(WW, D, alpha_i, alpha_o, ...
    Inputs, Targets, h0, slopeX, eta, fX, GX, WWmask)

%NR_LINESEARCH Line search with Wolfe-Powell conditions
%   [f2,df2,WW,eta] = nr_linesearch(WW, D, alpha_i, alpha_o, Inputs,
%       Targets, h0, slopeX, eta, fX, GX, WWmask)
%
%   Input:
%     WW     :  weight vector (the matrices reshaped into one vector)
%     D      :  the vector with stored dimensions of the weight matrices
%     h0     :  direction vector
%     slopeX :
%     eta    :  guessed step size
%     fX     :  function value for the starting point
%     GX     :  gradient at the starting point
%     WWmask :  weights mask
%
%   Output :
%     f2     : function value for the minimizer
%     df2    : gradient for the minimizer
%     WW     : output weights
%     eta    : output step size
```

A.1.13 Subroutine nr_plotnet.m

Call

```
function nr_plotnet(Wi, Wo, plottype)

%NR_PLOTNET    Plot network
%   NR_PLOTNET(Wi, Wo, plottype)
```

```
%
%   Input:
%        Wi       :  Matrix with input-to-hidden weights
%        Wo       :  Matrix with hidden-to-outputs weights
%        plottype :  type of plot
%
%    See also NR_TRAIN
%
%    Neural Regression toolbox, DSP IMM DTU
```

A.1.14 Subroutine nr_plotsal.m

Call

```
function nr_plotsal(Wi, Wo, alpha_i, alpha_o, Inputs, Targets, ...
    plottype)

%NR_PLOTSAL    Neural regression plot saliency
%   NR_PLOTSAL(Wi, Wo, alpha_i, alpha_o, Inputs, Targets, plottype)
%
%   Input:
%        Wi       :  Matrix with input-to-hidden weights
%        Wo       :  Matrix with hidden-to-outputs weights
%        alpha_i  :  Weight decay for input-to-hidden weights
%        alpha_o  :  Weight decay for hidden-to-output weights
%        Inputs   :  Input to the neural network
%        Targets  :  Targets
%        plottype :  Type of plot
%
%    See also NR_TRAIN, NR_PLOTNET
%
%    Neural Regression toolbox, DSP IMM DTU
```

A.1.15 Subroutine nr_prune.m

Pseudo Code

```
function nr_prune
    1. Calculate diagonal elements of the un-regularized cost $S_{\mathcal{T}}(\boldsymbol{w})$ using nr_pseuhess
    2. Calculate saliencies
    3. Prune a fixed number of the weight with smallest saliencies
end
```

Call

```
function [Wi_new,Wo_new] = nr_prune(Wi,Wo,alpha_i,alpha_o,Inputs,Targets,
                                    kills)
%NR_PRUNE      Prune weights with Optimal Brain Damage
%   [Wi_new,Wo_new] = nr_prune(Wi,Wo,alpha_i,alpha_o,Inputs,Targets,kills)
%   prunes a number of weights away using Optimal Brain Damage
%
%   Input:
%        Wi       :  Matrix with input-to-hidden weights
%        Wo       :  Matrix with hidden-to-outputs weights
%        alpha_i  :  Weight decay parameter for input weights
%        alpha_o  :  Weight decay parameter for output weights
%        Inputs   :  Matrix with examples as rows
%        kills    :  Number of weights to eliminate
%   Output:
%        Wi_new   :  Matrix with reduced input-to-hidden weights
%        Wo_new   :  Matrix with reduced hidden-to-outputs weights
%
%   See also NR_TRAIN, NR_PSEUHESS
%
%   Neural Regression toolbox, DSP IMM DTU
```

A.1.16 Subroutine nr_pseuhess.m

Call

```
function [dWi,dWo,ddWi,ddWo] = nr_pseuhess(Wi, Wo, alpha_i, alpha_o,...
                                            Inputs, Targets)
%NR_PSEUHESS   Pseudo Hessian elements and the partial derivatives.
%   [dWi,dWo,ddWi,ddWo] = NR_PSEUHESS(Wi,Wo,alpha_i,alpha_o,Inputs,Targets)
%   calculates the pseudo Hessian elements AND the partial derivatives
%   of the quadratic cost function  wrt. the weights. Derivatives of
%   quadratic weight decay are included.
%
%   Input:
%        Wi       :  Matrix with input-to-hidden weights
%        Wo       :  Matrix with hidden-to-outputs weights
%        alpha_i  :  Weight decay parameter for input weights
%        alpha_o  :  Weight decay parameter for output weights
%        Inputs   :  Matrix with examples as rows
%        Targets  :  Matrix with target values as rows
%   Output:
%        dWi      :  Matrix with gradient for input weights
%        dWo      :  Matrix with gradient for output weights
%        ddWi     :  Matrix with pseudo Hessian for input w.
%        ddWo     :  Matrix with pseudo Hessian for output w.
%
%   Neural Regression toolbox, DSP IMM DTU
```

A.1.17 Subroutine nr_tanhf.m

Call

```
function y=tanhf(x)
```

```
%NR_TANHF      Fast hyperbolic tangent
%   y=tanhf(x) calculates the fast hyperbolic tangent:
%   y=1 - 2./(exp(2*x)+1);
%
%   Neural Regression toolbox, DSP IMM DTU
```

A.1.18 Subroutine `nr_train.m`

Pseudo Code

```
function nr_train
    1. Do gradient descent training
        1. Calculate gradient, $\nabla$, using gradient
        2. while norm($\nabla$) > prescribed value and no. of iteration < maxitr repeat
            1. Determine step-size using nr_linesear
            2. Update weights
            3. Recalculate the gradient using nr_gradient
           end
       end
    2. Do pseudo Gauss-Newton training
        1. Calculate pseudo Gauss-Newton direction using nr_pseuhess
        2. while norm($\nabla$) > prescribed value and no. of iteration < maxitr repeat
            1. Determine step-size using nr_linesear
            2. Update weights
            3. Recalculate the pseudo Gauss-Newton direction using nr_pseuhess
           end
       end
end
```

Call

```
function [Wi_tr,Wo_tr] = nr_train(Wi, Wo, alpha_i, alpha_o, Inputs, ...
    Targets, gr, psgn, neps)

%NR_TRAIN      Train network
%   [Wi_tr,Wo_tr] = nr_train(Wi, Wo, alpha_i, alpha_o, Inputs, ...
%   Targets, gr, psgn, neps) trains a network with gradient descent
%   followed by pseudo Gauss-Newton
%
%   Input:
%        Wi      :  Matrix with input-to-hidden weights
%        Wo      :  Matrix with hidden-to-outputs weights
%        alpha_i :  Weight decay parameter for input weights
%        alpha_o :  Weight decay parameter for output weights
%        Inputs  :  Matrix with examples as rows
%        Targets :  Matrix with target values as rows
%        gr      :  Max. number of gradient descent steps
%        psgn    :  Max. number of pseudo Gauss-Newton steps
%        neps    :  Gradient norm stopping criteria
%   Output:
%        Wi_tr   :  Matrix with trained input-to-hidden weights
%        Wo_tr   :  Matrix with trained hidden-to-outputs weights
%
%   Neural Regression toolbox, DSP IMM DTU
```

A.1.19 Subroutine `nr_trainx.m`

Call

```
function [Wi_tr,Wo_tr, Etrain] = nr_trainx(Wi, Wo, alpha_i, alpha_o, ...
    Inputs, Targets, gr, psgn, neps, figh, method)

%NR_TRAINX     Train network (conjugate gradient version)
%   [Wi_tr,Wo_tr] = nr_trainx(Wi, Wo, alpha_i, alpha_o, Inputs,
%   Targets, gr, psgn, neps, figh)
%   Train the network with gradient descent followed by pseudo
%   Gauss-Newton
%
%   Input:
%        Wi      :  Matrix with input-to-hidden weights
%        Wo      :  Matrix with hidden-to-outputs weights
%        alpha_i :  Weight decay parameter for input weights
%        alpha_o :  Weight decay parameter for output weights
%        Inputs  :  Matrix with examples as rows
%        Targets :  Matrix with target values as rows
%        psgn    :  Max. number of steps in pseudo Gauss-Newton
%                   or N dimensional passes Conjugate Gradient
%        gr      :  Max. number of gradient descent steps
%                   not used for CG
%        neps    :  Gradient norm stopping criteria not used for CG
%        figh    :  figure handle. If no plotting is desired use
%                   figh=0;
%        method  :  defined only when Conjugate Gradient method used
%                       FR - Fletcher-Reeves
%          HS - Hestenes-Stiefel
%   PR - Polak-Ribiere (default)
%
%    Output:
%        Wi_tr   :  Matrix with trained input-to-hidden weights
%        Wo_tr   :  Matrix with trained hidden-to-outputs weights
%
%    Neural Regression toolbox, DSP IMM DTU

%    JL97, MWP97, Anna 1999
```

A.1.20 Subroutine `nr_two_norm.m`

Call

```
function [n] = nr_two_norm(dWi,dWo)
%NR_TWO_NORM   Euclidean length of the total weight vector.
%   [n] = two_norm(dWi,dWo) calculates the Euclidean length of the
%   total weight vector, i.e. the 2-norm.
%
%   Input:
%        dWi      :  Matrix with input-to-hidden gradient
%        dWo      :  Matrix with hidden-to-outputs gradient
%   Output:
%        n        :  2-norm of total gradient
%
%   Neural Regression toolbox, DSP IMM DTU
```

A.1.21 Subroutine `nr_winit.m`

Call

```
function [Wi,Wo]=nr_winit(Ni,Nh,No,range,seed)

%NR_WINIT      Initial weight in neural network
%   [Wi,Wo] = NR_WINIT(Ni,Nh,No,range,seed) initialize the
%   weight in a neural network with a uniform distribution
%
%   Input:
%        Ni    : no. of input neurons
%        Nh    : no. of hidden neurons
%        No    : no. of output neurons
%        range : weight initialization uniformly over
%                [-range;range]/Ni and [-range;range]/Nh, respectively.
%        seed  : a integer seed number, e.g., sum(clock*100)
%    Output:
%        Wi: Input-to-hidden weights
%        Wo: Hidden-to-output initial weights
%
%    Neural Regression toolbox, DSP IMM DTU
```

B Neural Network Classification Software

B.1 Network Classifier Architecture

Suppose that the input (feature) vector is denoted by $\boldsymbol{x}$ with $\dim(\boldsymbol{x}) = n_I$. The aim is to model the posterior probabilities $p(\mathcal{C}_i|\boldsymbol{x})$, $i = 1, 2, \cdots, c$ where $\mathcal{C}_i$ denotes the i'th class. Then the Bayes optimal[7] classifier assigns class label $\mathcal{C}_i$ to $\boldsymbol{x}$ if $i = \arg\max_j p(C_j|\boldsymbol{x})$.

[7]That is, minimal probability of misclassification.

Following [6], the outputs, $\widehat{y}_i$, of the neural network represent *estimates* of the posterior probabilities, i.e., $\widehat{y}_i = \widehat{p}(\mathcal{C}_i|\boldsymbol{x})$; hence, $\sum_{i=1}^{c} p(\mathcal{C}_i|\boldsymbol{x}) = 1$. That is, we need merely to estimate $c \Leftrightarrow 1$ posterior probabilities, say $p(\mathcal{C}_i|\boldsymbol{x})$, $i = 1, 2, \cdots, c \Leftrightarrow 1$, then the last is calculated as $p(\mathcal{C}_c|\boldsymbol{x}) = 1 \Leftrightarrow \sum_{i=1}^{c-1} p(\mathcal{C}_i|\boldsymbol{x})$.

Define a 2-layer feed-forward network with n_I inputs, n_H hidden neurons and $c \Leftrightarrow 1$ outputs by:

$$h_j(\boldsymbol{x}) = \tanh\left(\sum_{\ell=1}^{n_I} w_{j\ell}^I x_\ell + w_{j0}^I\right) \tag{44}$$

$$\phi_i(\boldsymbol{x}) = \sum_{j=1}^{n_H} w_{ij}^O h_j(\boldsymbol{x}) + w_{i0}^O \tag{45}$$

where $w_{j\ell}^I$, w_{ij}^O are the input-to-hidden and hidden- to-output weights, respectively. All weights are assembled in the weight vector $\boldsymbol{w} = \{w_{j\ell}^I, w_{ij}^O\}$.

In order to interpret the network outputs as probabilities a *modified* normalized exponential transformation similar to SoftMax is used,

$$\widehat{y}_i = \frac{\exp(\phi_i)}{\sum_{j=1}^{c-1}\exp(\phi_j) + 1}, \; i = 1, 2, \cdots, c \Leftrightarrow 1, \qquad \widehat{y}_c = 1 \Leftrightarrow \sum_{i=1}^{c-1} \widehat{y}_i. \tag{46}$$

The network architecture is shown in Fig. 22.

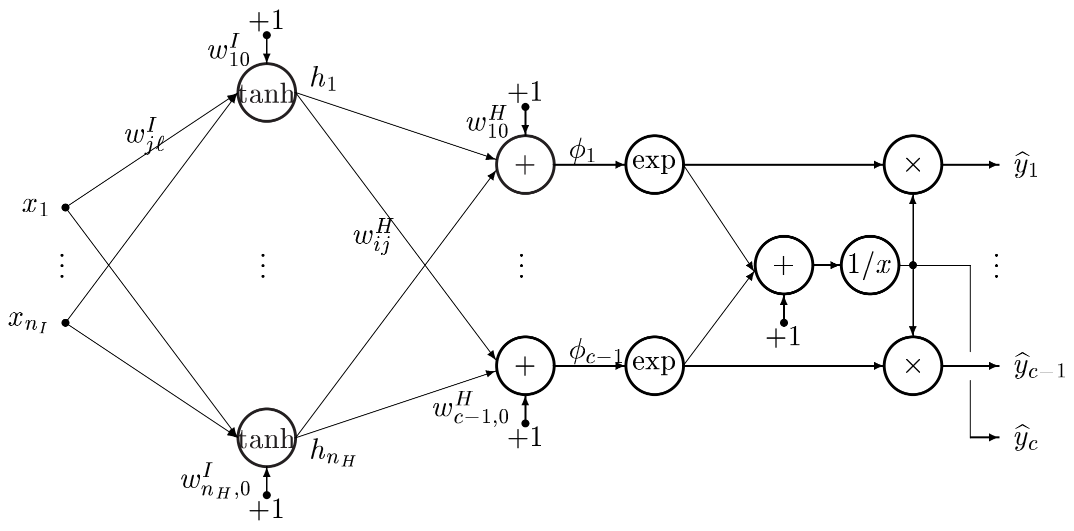


Figure 22: Neural network classifier architecture.

B.2 Training and Regularization

Assume that we have a training set $\mathcal{T}$ of N_{train} related input-output pairs $\mathcal{T} = \{(\boldsymbol{x}(k), \boldsymbol{y}(k))\}_{k=1}^{N_{\text{train}}}$ where

$$y_i(k) = \begin{cases} 1 & \text{if } \boldsymbol{x}(k) \in \mathcal{C}_i \\ 0 & \text{otherwise} \end{cases} \tag{47}$$

The likelihood of the network parameters is given by (see e.g., [6]),

$$p(\mathcal{T}|\boldsymbol{w}) = \prod_{k=1}^{N_{\text{train}}} p(\boldsymbol{y}(k)|\boldsymbol{x}(k), \boldsymbol{w}) = \prod_{k=1}^{N_{\text{train}}} \prod_{i=1}^{c} (\widehat{y}_i(k))^{y_i(k)} \tag{48}$$

where $\widehat{\boldsymbol{y}}(k) = \widehat{\boldsymbol{y}}(\boldsymbol{x}(k), \boldsymbol{w})$ is a function of the input and weight vectors. The training error is the normalized negative log-likelihood

$$S_{\mathcal{T}}(\boldsymbol{w}) = \Leftrightarrow \frac{1}{N_{\text{train}}} \log p(\mathcal{T}|\boldsymbol{w}) \equiv \frac{1}{N_{\text{train}}} \sum_{k=1}^{N_{\text{train}}} \ell\left(\boldsymbol{y}(k), \widehat{\boldsymbol{y}}(k); \boldsymbol{w}\right) \tag{49}$$

with $\ell(\cdot)$ denoting the loss given by

$$\ell\left(\boldsymbol{y}(k), \widehat{\boldsymbol{y}}(k); \boldsymbol{w}\right) = \log\left(1 + \sum_{j=1}^{c-1} \exp(\phi_j(\boldsymbol{x}(k)))\right) \Leftrightarrow \sum_{i=1}^{c-1} y_i(k)\phi_i(\boldsymbol{x}(k)). \tag{50}$$

The objective of training is minimization of the regularized cost function[8] which is defined as the negative log-likelihood augmented by separate weight decay regularization for input-to-hidden and hidden-to output layers, as shown by,

$$C(\boldsymbol{w}) = S_{\mathcal{T}}(\boldsymbol{w}) + \kappa_I \cdot |\boldsymbol{w}^I|^2 + \kappa_O \cdot |\boldsymbol{w}^O|^2 \tag{51}$$

where $\boldsymbol{w} = [\boldsymbol{w}^I, \boldsymbol{w}^O]$ with $\boldsymbol{w}^I$, $\boldsymbol{w}^O$ denoting the input-to-hidden weights and hidden-to-output weights, respectively. $\kappa_I \equiv \alpha_I/(2N_{\text{train}})$ and $\kappa_O \equiv \alpha_O/(2N_{\text{train}})$ are the weight decay parameters.

Training provides the estimated weight vector $\widehat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} C(\boldsymbol{w})$ and is done using a gradient or a pseudo Gauss-Newton scheme. The gradient scheme reads,

$$\boldsymbol{w}^{\text{new}} = \boldsymbol{w}^{\text{old}} \Leftrightarrow \eta \cdot \boldsymbol{\nabla}(\boldsymbol{w}^{\text{old}}) \tag{52}$$

where η is the step-size (line search parameter). η is found using simple bisection line search, i.e., η is successively divided by a factor of 2 until a decrease in the cost is observed. For that purpose we require the gradient, $\boldsymbol{\nabla}(\boldsymbol{w}) = \partial C/\partial \boldsymbol{w}$, of the cost function given by,

$$\frac{\partial C}{\partial \boldsymbol{w}}(\boldsymbol{w}) = \Leftrightarrow \frac{1}{N_{\text{train}}} \sum_{k=1}^{N_{\text{train}}} \sum_{i=1}^{c-1} \left[y_i(k) \Leftrightarrow \widehat{y}_i(k)\right] \frac{\partial \phi_i(\boldsymbol{x}(k))}{\partial \boldsymbol{w}} + \frac{\alpha_I}{N_{\text{train}}} \cdot \boldsymbol{w}^I + \frac{\alpha_O}{N_{\text{train}}} \cdot \boldsymbol{w}^O \tag{53}$$

The pseudo Gauss-Newton scheme is given by

$$\boldsymbol{w}^{\text{new}} = \boldsymbol{w}^{\text{old}} \Leftrightarrow \eta \cdot \widetilde{\boldsymbol{J}}^{-1}(\boldsymbol{w}^{\text{old}}) \boldsymbol{\nabla}(\boldsymbol{w}^{\text{old}}) \tag{54}$$

$\widetilde{\boldsymbol{J}}(\boldsymbol{w})$ is the pseudo Hessian of the cost function defined as the diagonal of the full Hessian, $\widetilde{\boldsymbol{J}}(\boldsymbol{w}) = \{J_{mn}\} = \partial^2 C/\boldsymbol{w}\boldsymbol{w}^\top$ where

$$J_{mm}(\boldsymbol{w}) = \frac{1}{N_{\text{train}}} \sum_{k=1}^{N_{\text{train}}} \sum_{i=1}^{c-1} \sum_{j=1}^{c-1} \widehat{y}_i(k) \left[\delta_{ij} \Leftrightarrow \widehat{y}_j(k)\right] \frac{\partial \phi_i(\boldsymbol{x}(k))}{\partial w_m} \frac{\partial \phi_j(\boldsymbol{x}(k))}{\partial w_m} + \frac{\alpha_m}{N_{\text{train}}} \tag{55}$$

where α_m equals α_I or α_O depending on whether the m'th weight is an input-to-hidden or an hidden-to-output weight. Above δ_{ij} denotes the Kronecker delta and we have used the Gauss-Newton approximation to the Hessian.

[8]This might be viewed as a maximum a posteriori (MAP) method.

B.3 MATLAB Functions in the Neural Classification Package

Suppose the number of training and test examples are N_{train}, N_{test}, respectively. Further, that the number of inputs, hidden and output neurons are n_I, n_H and $n_O = c \Leftrightarrow 1$, where c is the number of classes. The training data are required to be stored in the following matrices:

`train_inp`:	a $N_{\text{train}} \times n_I$ matrix of training input data. Row k is the k'th training example of the input vector $\boldsymbol{x}^\top(k)$.
`train_tar`:	a $N_{\text{train}} \times 1$ vector of training output/target data. The k'th element is $i \in [1; c]$ if example k is classified as class $\mathcal{C}_i$.

Similarly the test data are stored in `test_inp` and `test_tar`.

The weights of the network are assembled in the input-to-hidden matrix `Wi` and the hidden-to-output matrix `Wo` which are defined by

$$\mathtt{Wi} = \begin{bmatrix} \mathtt{Wi}_{11} & \mathtt{Wi}_{12} & \cdots & \mathtt{Wi}_{1,n_I} & \mathtt{Wi}_{1,\text{bias}} \\ & & \ddots & & \\ \mathtt{Wi}_{n_H,1} & \mathtt{Wi}_{n_H,2} & \cdots & \mathtt{Wi}_{n_H,n_I} & \mathtt{Wi}_{n_H,\text{bias}} \end{bmatrix} \tag{56}$$

where $\mathtt{Wi}_{j,\ell}$ is the weight from input ℓ to hidden neuron j. Further,

$$\mathtt{Wo} = \begin{bmatrix} \mathtt{Wo}_{11} & \mathtt{Wo}_{12} & \cdots & \mathtt{Wo}_{1,n_H} & \mathtt{Wo}_{1,\text{bias}} \\ & & \ddots & & \\ \mathtt{Wo}_{n_O,1} & \mathtt{Wo}_{n_O,2} & \cdots & \mathtt{Wo}_{n_O,n_H} & \mathtt{Wi}_{n_O,\text{bias}} \end{bmatrix} \tag{57}$$

where $\mathtt{Wo}_{i,j}$ is the weight from hidden unit j to output neuron i.

B.3.1 Function Overview

`nc_netprun.m`	Main function.
`nc_cl_error.m`	Calculates number of erroneous classified examples.
`nc_cl_probs.m`	Calculates posterior probabilities.
`nc_cost_c.m`	Calculates the cost function augmented by weight decay.
`nc_cost_e.m`	Calculates the cost function.
`nc_dimen.m`	Calculates the number of non-zero weights.
`nc_err_frac.m`	Calculate the fraction of erroneous classified examples.
`nc_eucnorm.m`	Calculates the Euclidean length of the weight vector.
`nc_forward.m`	Propagate examples forward through network calculating all hidden- and output unit outputs.
`nc_getdata.m`	Neural classifier get forensic glass data set.
`nc_gradient.m`	Calculate the partial derivatives of the cost function.
`nc_linesear.m`	Performs a simple line search.

`nc_plotnet.m`	Neural classifier plot network.
`nc_plotsal.m`	Neural classifier plot saliency.
`nc_prune.m`	Prunes a number of weights away using OBD.
`nc_pseuhess.m`	Calculates the pseudo Hessian (diagonal) elements and gradient of the cost function.
`nc_softmax.m`	Performs the softmax operation.
`nc_tanhf.m`	Fast hyperbolic tangent.
`nc_train.m`	Train the network with gradient descent followed by pseudo Gauss-Newton.
`nc_winit.m`	Uniform weight initialization.

B.3.2 Main Function `nc_netprun.m`

Pseudo Code

```
function nc_netprun
  1. Initialize algorithm parameters
  2. Initialize weights using nc_winit
  3. Train the network using nc_train
  4. Evaluate training and test performance using nc_cost_e and nc_err_frac
  5. while $\dim(\boldsymbol{w}) > \mathrm{mindim}$ repeat
       1. Prune the network using prune
       2. Calculate dimension $\dim(\boldsymbol{w})$
       3. Retrain the network using train
       4. Evaluate training and test performance using nc_cost_e and nc_err_frac
     end
end
```

B.3.3 Subroutine `nc_cl_error.m`

Call

```
function [errors,probs,class] = nc_cl_error(Wi,Wo,Inputs,Targets)

% Calculates number of erroneous classified examples, estimated classes
% and posterior probabilities
% Input:
%        Wi      : Matrix with input-to-hidden weights
%        Wo      : Matrix with hidden-to-outputs weights
%        Inputs  : Matrix with examples as rows
%        Targets : Matrix with target values as rows
% Output:
```

```
%          errors: the no. erroneous classified examples
%           class: vector of estimated classes (No. of examples,1)
%           probs: matrix of posterior probabilities
%                  (No. of examples,no. of classes)
%
%
JL97,MWP97
```

B.3.4 Subroutine nc_cl_probs.m

Call

```
function [probs] = nc_cl_probs(Wi,Wo,Inputs)

% Calculates posterior probabilities for each example
% Input:
%        Wi      : Matrix with input-to-hidden weights
%        Wo      : Matrix with hidden-to-outputs weights
%        Inputs  : Matrix with examples as rows
% Output:
%           probs: matrix of posterior probabilities
%                  (No. of examples,no. of classes)
%
%
JL97,MWP97
```

B.3.5 Subroutine nc_cost_c.m

Call

```
function [cost] = nc_cost_c(Wi,Wo,alpha_i,alpha_o,Inputs,Targets)
%
% Calculates the value of the negative log-likelihood cost function,
% augmented by quadratic weight decay term
%
% Input:
%        Wi      :  Matrix with input-to-hidden weights
%        Wo      :  Matrix with hidden-to-outputs weights
%        alpha_i :  Weight decay parameter for input weights
%        alpha_o :  Weight decay parameter for output weights
%        Inputs  :  Matrix with examples as rows
%        Targets :  Matrix with target values as rows
% Output:
%        Cost    : Value of augmented negative log-likelihood cost
%                  function
%
MWP97
```

B.3.6 Subroutine nc_cost_e.m

Call

```
function [error] = nc_cost_e(Wi,Wo,Inputs,Targets)
%
% Calculate the value of the negative log likelihood cost
%
% Input:
%        Wi      :  Matrix with input-to-hidden weights
%        Wo      :  Matrix with hidden-to-outputs weights
%        Inputs  :  Matrix with input features as rows
%        Targets :  Column vector  with target class as elements
% Output:
%        error   :  Value of negative log likelihood
%
MWP97
```

B.3.7 Subroutine nc_dimen.m

Call

```
function [dim] = nc_dimen(Wi,Wo)
%
% Calculates the number of non-zero weights in the
% network, i.e. the dimension of the total weight vector
%
% Input:
%        Wi      :  Matrix with input-to-hidden weights
%        Wo      :  Matrix with hidden-to-outputs weights
% Output:
%        dim     :  Number of non-zero weights
%
MWP97
```

B.3.8 Subroutine nc_err_frac.m

Call

```
function [rate,probs,class] = nc_err_frac(Wi,Wo,Inputs,Targets)

% Calculate the fraction of erroneous classified examples, estimated
% classes and posterior probabilities
% Input:
%        Wi      : Matrix with input-to-hidden weights
%        Wo      : Matrix with hidden-to-outputs weights
%        Inputs  : Matrix with examples as rows
%        Targets : Matrix with target values as rows
% Output:
%            rate: the fraction of erroneous classified examples
%           class: vector of estimated classes (No. of examples,1)
%           probs: matrix of posterior probabilities
%                   (No. of examples,no. of classes)
%
%
JL97,MWP97
```

B.3.9 Subroutine `nc_eucnorm.m`

Call

```
function [n] = nc_eucnorm(dWi,dWo)
%
% Calculates the Euclidean length of the total weight vector,
%
% Input:
%        dWi      :  Matrix with input-to-hidden gradient
%        dWo      :  Matrix with hidden-to-outputs gradient
% Output:
%        n        :  Euclidean norm sqrt(|w|.^2)
%
MWP97
```

B.3.10 Subroutine `nc_forward.m`

Call

```
function [Vj,phi] = nc_forward(Wi,Wo,Inputs)
%
% Propagate examples forward through network calculating all hidden-
% and output unit outputs. Note: There is no softmax included.
%
% Input:
%        Wi     :  Matrix with input-to-hidden weights
%        Wo     :  Matrix with hidden-to-outputs weights
%        inputs :  Matrix with example inputs as rows
%
% Output:
%        Vj  :  Matrix with hidden unit outputs as rows
%        phi :  Matrix with output unit outputs as rows
%
MWP97
```

B.3.11 Subroutine `nc_getdata.m`

Call

```
function [tr_i,tr_t,te_i,te_t] = nc_getdata

%NC_GETDATA    Neural classifier get forensic glass data
```

```
%  [tr_i,tr_t,te_i,te_t] = nc_getdata
%  Use an example data set: the forensic glass data from the Proben
%  collection as data
%
%  Neural classifier, DSP IMM DTU, MWP97
```

B.3.12 Subroutine `nc_gradient.m`

Call

```
function [dWi,dWo] = nc_gradient(Wi,Wo,alpha_i,alpha_o,Inputs,Targets)
%
% Calculate the partial derivatives of the negative log-likelihood
cost.
% wrt. the weights. Derivatives of quadratic weight decay are included.
%
% Input:
%        Wi      :  Matrix with input-to-hidden weights
%        Wo      :  Matrix with hidden-to-outputs weights
%        alpha_i :  Weight decay parameter for input weights
%        alpha_o :  Weight decay parameter for output weights
%        Inputs  :  Matrix with examples as rows
%        Targets :  Matrix with target values as rows
% Output:
%        dWi     :  Matrix with gradient for input weights
%        dWo     :  Matrix with gradient for output weights
%
MWP97
```

B.3.13 Subroutine `nc_linesear.m`

Call

```
function [eta] =
nc_linesear(Wi,Wo,Di,Do,alpha_i,alpha_o,Inputs,Targets,pat)
%
% This function performs a simple line search in a direction
% in parameter space, determining the 'optimal' step length
% by iterative bisection.
%
% Input:
%        Wi      :  Matrix with input-to-hidden weights
%        Wo      :  Matrix with hidden-to-outputs weights
%        Di      :  Matrix with input search direction
%        Do      :  Matrix with output search direction
%        alpha_i :  Weight decay parameter for input weights
%        alpha_o :  Weight decay parameter for output weights
%        Inputs  :  Matrix with examples as rows
%        Targets :  Matrix with target values as rows
%        pat     :  Patience; max number of bisections
% Output:
%        eta     :  'Optimal' step length
%
MWP97
```

B.3.14 Subroutine `nc_plotnet.m`

Call

```
function nc_plotnet(Wi,Wo,plottype)

%NC_PLOTNET    Neural classifier plot network
%   NC_PLOTNET(Wi, Wo, plottype)
%
%   Input:
%        Wi       :  Matrix with input-to-hidden weights
%        Wo       :  Matrix with hidden-to-outputs weights
%        plottype :  type of plot
%
%
%    Neural classifier, DSP IMM DTU
```

B.3.15 Subroutine `nc_plotsal.m`

Call

```
function nc_plotsal(Wi, Wo, alpha_i, alpha_o, Inputs, Targets, ...
    plottype)

%NC_PLOTSAL    Neural classifier plot saliency
%   NC_PLOTSAL(Wi, Wo, alpha_i, alpha_o, Inputs, Targets, plottype)
%
%   Input:
%        Wi       :  Matrix with input-to-hidden weights
%        Wo       :  Matrix with hidden-to-outputs weights
%        alpha_i  :  Weight decay for input-to-hidden weights
%        alpha_o  :  Weight decay for hidden-to-output weights
%        Inputs   :  Input to the neural network
%        Targets  :  Targets
%        plottype :  Type of plot
%
%
%    Neural Classifier toolbox, DSP IMM DTU
```

B.3.16 Subroutine `nc_prune.m`

Pseudo Code

```
function nc_prune
    1. Calculate diagonal elements of the un-regularized cost $S_{\mathcal{T}}(\boldsymbol{w})$ using nc_pseuhess
    2. Calculate saliencies
    3. Prune a fixed number of the weight with smallest saliencies
end
```

Call

```
function [Wi_new,Wo_new] = nc_prune(Wi,Wo,alpha_i,alpha_o,Inputs,Targets,
                           kills)
% This function prunes a number of weights away using Optimal Brain
Damage.
%
% Input:
%        Wi      :  Matrix with input-to-hidden weights
%        Wo      :  Matrix with hidden-to-outputs weights
%        alpha_i :  Weight decay parameter for input weights
%        alpha_o :  Weight decay parameter for output weights
%        Inputs  :  Matrix with examples as rows
%        kills   :  Number of weights to eliminate
% Output:
%        Wi_new  :  Matrix with reduced input-to-hidden weights
%        Wo_new  :  Matrix with reduced hidden-to-outputs weights
%                                                             JL97,
MWP97
```

B.3.17 Subroutine `nc_pseuhess.m`

Call

```
function [dWi,dWo,ddWi,ddWo] = nc_pseuhess(Wi,Wo,alpha_i,alpha_o,Inputs,
                                           Targets)
%
% Calculates the pseudo Hessian (diagonal) elements AND the partial
% derivatives of % the negative log-likelihood cost function wrt.
% the weights. Derivatives of quadratic weight decay are included.
%
% Input:
%        Wi      :  Matrix with input-to-hidden weights
%        Wo      :  Matrix with hidden-to-outputs weights
%        alpha_i :  Weight decay parameter for input weights
%        alpha_o :  Weight decay parameter for output weights
%        Inputs  :  Matrix with examples as rows
%        Targets :  Matrix with target values as rows
% Output:
%        dWi     :  Matrix with gradient for input weights
%        dWo     :  Matrix with gradient for output weights
%        ddWi    :  Matrix with pseudo Hessian for input weights
%        ddWo    :  Matrix with pseudo Hessian for output weights
%
MWP97
```

B.3.18 Subroutine `nc_softmax.m`

Call

```
function probs=nc_softmax(phi)

% Carry out the softmax operation
% Input:  phi the matrix of outputs of the network from forward.m.
%         rows are the individual output neurons.
% Output: the matrix of posterior probabilities. Each row is the
%         are individual class prob for a specific example.
%
JL97
```

B.3.19 Subroutine `nc_tanhf.m`

Call

```
function y=nc_tanhf(x)

% Fast tanh y=tanhf(x)
%y=1 - 2./(exp(2*x)+1);
```

B.3.20 Subroutine `nc_train.m`

Pseudo Code

```
function nc_train
  1. Do gradient descent training
     1. Calculate gradient, $\boldsymbol{\nabla}$, using gradient
     2. while norm($\boldsymbol{\nabla}$) $>$ prescribed value and no. of iteration $<$ maxitr repeat
        1. Determine step-size using nc_linesear
        2. Update weights
        3. Recalculate the gradient using nc_gradient
        end
     end
  2. Do pseudo Gauss-Newton training
     1. Calculate pseudo Gauss-Newton direction using nc_pseuhess
     2. while norm($\boldsymbol{\nabla}$) $>$ prescribed value and no. of iteration $<$ maxitr repeat
        1. Determine step-size using nc_linesear
        2. Update weights
        3. Recalculate the pseudo Gauss-Newton direction using nc_pseuhess
        end
     end
end
```

Call

```
function [Wi_tr,Wo_tr] = nc_train(Wi,Wo,alpha_i,alpha_o,Inputs,Targets,gr,
                         psgn,neps,figh)
%
% Train the network with gradient descent followed by pseudo Gauss-
Newton
%
% Input:
%        Wi      :  Matrix with input-to-hidden weights
%        Wo      :  Matrix with hidden-to-outputs weights
%        alpha_i :  Weight decay parameter for input weights
%        alpha_o :  Weight decay parameter for output weights
%        Inputs  :  Matrix with examples as rows
%        Targets :  Matrix with target values as rows
%        gr      :  Max. number of gradient descent steps
%        psgn    :  Max. number of pseudo Gauss-Newton steps
%        neps    :  Gradient norm stopping criteria
%        figh    :  figure handle. If no plotting is desired use
%                   figh=0;
% Output:
%        Wi_tr   :  Matrix with trained input-to-hidden weights
%        Wo_tr   :  Matrix with trained hidden-to-outputs weights
%
MWP97
```

B.3.21 Subroutine `nc_winit.m`

Call

```
function [Wi,Wo]=nc_winit(Ni,Nh,No,range,seed)

% Uniform weight initialization
% Input:
%      Ni: no. of input neurons
%      Nh: no. of hidden neurons
%      No: no. of output neurons
%   range: weight initialization uniformly over [-range;range]/Ni
%          and [-range;range]/Nh, respectively.
%    seed: a integer seed number, e.g., sum(clock*100)
% Output:
%      Wi: Input-to-hidden weights
%      Wo: Hidden-to-output initial weights
%                                                               JL97
```

References

[1] C.M. Bishop: *Neural Networks for Pattern Recognition*, Oxford, UK: Oxford University Press, 1995.

[2] Defense Advanced Research Projects Agency: *DARPA Neural Network Study*, AFCEA International Press, 1988.

[3] R.C. Eberhart and R.W. Dobbins: *Early Neural Network Development History: The Age of Camelot* IEEE Engineering in Medicine and Biology, September 1990, 15-18, (1990).

[4] S. Haykin: *Neural Networks: A Comprehensive Foundation*, New York, New York: Macmillan College Publishing Company, 1994.

[5] J. Hertz, A. Krogh & R.G. Palmer: *Introduction to the Theory of Neural Computation*, Redwood City, California: Addison-Wesley Publishing Company, 1991.

[6] M. Hintz-Madsen, L.K. Hansen, J. Larsen, M.W. Pedersen & M. Larsen: "Neural Classifier Construction using Regularization, Pruning and Test Error Estimation," *Neural Networks*, vol. 11, no. 9, pp. 1659–1670, Dec. 1998.

[7] K. Hornik: "Approximation Capabilities of Multilayer Feedforward Networks," *Neural Networks*, vol. 4, pp. 251–257, 1991.

[8] J. Larsen: Jan Larsen: *Correlation Functions and Power Spectra*, 04361 Digital Signal Processing course note, 4th Edition, IMM, DTU, 1999

[9] J. Larsen: *Design of Neural Network Filters*, Ph.D. Thesis, Electronics Institute, Technical University of Denmark, March 1993.

[10] Y. Le Cun, J.S. Denker, S.A. Solla: "Optimal Brain Damage," in D.S. Touretzsky (ed.) *Advances in Neural Information Processing Systems II* San Mateo: Morgan Kaufman, pp. 598–605, 1990.

[11] J.L. McClelland & D.E. Rumelhart (eds.): *Parallel Distributed Processing, Explorations in the Microstructure of Cognition. Vol. 1: Foundations*, Cambridge, Massachusetts: MIT Press, 1986.

[12] W.S. McCulloch & W. Pitts: "A Logical Calculus of Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–133, 1943.

[13] J.G. Proakis & D.G. Manolakis: *Digital Signal Processing: Principles, Algorithms and Applications*, 3rd edition, Upper Saddle River, New Jersey: Prentice-Hall, Inc., 1996.

[14] B.D. Ripley: *Pattern Recognition and Neural Networks*, Cambridge, UK: Cambridge University Press, 1996.

[15] F. Rosenblatt: "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain," *Psychological Review*, vol. 65, pp. 336–408, 1958.

[16] F. Rosenblatt: *Principles of Neurodynamics*, New York, New York: Spartan, 1962.

[17] J.Aa. Sørensen: *Adaptive FIR Filters*, 04361 Digital Signal Processing course note, IMM, DTU, 1999.

[18] D.G. Stork (ed.): *HAL's Legacy: 2001's Computer as Dream and Reality*, Cambridge, Massachusetts: MIT Press, 1997.

[19] B. Widrow & M.A. Lehr: "30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1415–1441, Sept. 1990.

[20] B. Widrow & M.E. Hoff, Jr.: "Adaptive Switching Circuits," *IRE WESCON Convention Record*, part 4, pp. 96–104, 1960.