GNOSIS

Generalized Networks for the
Optimal Synthesis of Information Systems

Version 3.0

Users' Manual

September 1998

BARRON ASSOCIATES, INC.
Jordan Building
1160 Pepsi Place, Suite 300
Charlottesville, VA 22901-0807

(804) 973-1215
FAX (804) 973-4686
EMAIL: barron@bainet.com

FOREWORD

This users' manual and the software it describes have been prepared by Barron Associates, Inc. to provide a tool for computer-based modeling of complex processes. The *GNOSIS* tool is an outgrowth of 33 years of continuous R&D in batch-supervised, inductive, numerical modeling employing an algebraic neural network approach. The modeling is accomplished with users' databases of observations; from these *GNOSIS* learns the *parameter values* and (optionally) *structures* of network models. These models have just-sufficient complexity, considering the functional relationships implicit in the data and the number of data points.

The authors gratefully acknowledge the important contributions provided by their present and former colleagues toward developing the underlying theory and the software for *GNOSIS*. Dr. Andrew R. Barron established the fundamental algorithmic techniques and modeling criteria embodied in the *GNOSIS* algorithm, drawing partly on the work of A.G. Ivakhnenko, H. Akaike, J.H. Friedman, and the ideas of colleagues at Adaptronics, Inc. and Barron Associates, Inc. (BAI). *GNOSIS* builds upon previous BAI software tools, including *ASPN II*, *ASPN-IIc*, *CLASS*, and *Dyn3*.

Support for portions of this work by government and industry sources has been greatly appreciated. In particular, support of the Flight Dynamics Directorate, Wright Aeronautical Laboratory (AFSC), United States Air Force, is gratefully acknowledged. This support has been received principally via Small Business Innovative Research Contracts.

All opinions expressed in this manual (and in the software) are those of Barron Associates, Inc., which is solely responsible for its content.

Barron Associates, Inc. welcomes questions and suggestions concerning *GNOSIS*, this manual, and our technical services.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Information in this document is subject to change without notice and does not represent a commitment on the part of Barron Associates, Inc. No part of this manual may be reproduced or transmitted in any form for any purpose without the express written permission of Barron Associates, Inc.

Product names printed in this manual are the trademarks or registered trademarks of their manufacturers. UNIX is a registered trademark of AT&T Bell Laboratories, Macintosh is a registered trademark of Apple Computer, Inc., SUN is a trademark of Sun Microsystems, Inc., and Windows is a trademark of Microsoft.

Barron Associates, Inc.
Jordan Building
1160 Pepsi Place, Suite 300
Charlottesville, VA 22901-0807

GNOSIS SOFTWARE LICENSE AGREEMENT

1. GRANT OF LIMITED-USE LICENSE. Barron Associates, Inc. (BAI) hereby grants _____ _____ (Licensee) and Licensee hereby accepts a perpetual, nontransferable, nonexclusive license to use the *GNOSIS* software program(s) identified in Attachment 1 (hereinbelow referred to as "Licensed *GNOSIS* Program(s)") in strict compliance with the terms of this Agreement. Licensee agrees that the Licensed *GNOSIS* Program(s) will be used only for the Licensee's internal research and development purposes and only on the Designated Computer and at the Designated Site stated in Attachment 1. Licensee may move the Licensed *GNOSIS* Program(s) to a new equivalent Designated Computer or to a new Designated Site with BAI's prior written consent, which consent shall not be unreasonably withheld. Without BAI's prior written consent, Licensee shall have neither the power nor the right to sell, assign, license, pledge, or otherwise transfer, whether voluntarily or by operation of law, any of its rights under this Agreement or in and to the Licensed *GNOSIS* Program(s), and any such attempt shall be void and of no effect.

2. PAYMENT AND DELIVERY. Customer shall pay to BAI in United States dollars the one-time License Fee(s) set forth on Attachment 1. Payment terms are Net 30 days. Delivery is F.O.B., Charlottesville, Virginia, prepay and bill. Interest at the maximum rates permitted by law shall be added by BAI to any overdue amounts owed to BAI by Licensee. Licensee shall pay all shipping and insurance charges for delivery of the Licensed *GNOSIS* Program(s) to the Designated Site. Unless BAI receives specific shipping instructions from Licensee, BAI shall select the method of shipment. Estimated delivery dates are approximate only and are based upon prompt receipt of the necessary information and documentation from Licensee. BAI shall not be liable for any delay in delivery due to causes beyond its reasonable control. All amounts payable to BAI under this Agreement are payable in full without deduction for any custom duties or foreign or U.S. Federal, state, or local excise, sales, use or other taxes. Licensee agrees to pay promptly to BAI the amount of all taxes (including without limitation, sales, use, privilege, ad valorem, personal property, withholding, or excise taxes, and customs duties, however designated) which arise as a result of the transactions contemplated hereunder, but exclusive of federal, state, or local income taxes based on BAI's net income.

3. LICENSEE'S AGREEMENT AS TO CERTAIN RESPONSIBILITIES AND USE. Licensee agrees not to: (1) distribute, rent, lease, lend, or sublicense, or otherwise transfer the Licensed *GNOSIS* Program(s) or Licensee's rights hereunder; (2) remove or obscure proprietary rights (e.g., trademark and copyright) notices; (3) alter, reverse-engineer, decompile, disassemble, make any attempt to discover the source code of, or create derivative works based on the software; (4) utilize the software on any service bureau, time-sharing, or interactive cable system; or (5) make telecommunication data transmission of the software. Licensee shall be permitted to make one copy of the software received from BAI, such copy to be used for back-up purposes only. LICENSEE AGREES TO USE DUE CARE AND TO TAKE ALL REASONABLE STEPS TO PROTECT THE LICENSED *GNOSIS* PROGRAM(S) FROM UNAUTHORIZED REPRODUCTION, PUBLICATION, DISCLOSURE, OR DISTRIBUTION, AND TO ADVISE BAI PROMPTLY IN WRITING NOT LATER THAN TEN (10) CALENDAR DAYS FROM ITS LEARNING OF OR BECOMING AWARE OF THE OCCURRENCE OF ANY OF THE ABOVE EVENTS.

4. COPYRIGHT AND TRADE SECRETS. The Licensed *GNOSIS* Program(s) contain trade secrets owned by BAI, involves or involve the proprietary rights of BAI, and is or are protected by U.S. copyright law and by the laws of most other countries. This Agreement does not constitute a sale of, nor does it otherwise convey ownership of the Licensed *GNOSIS* Program(s) to Licensee or any other party. The Licensed *GNOSIS* Program(s), its or their structure, and its or their code are valuable property of BAI. BAI retains title to and full ownership thereof and all other proprietary rights related thereto, including, without limitation, copyright and trademark rights.

5. TERM. Without limitation to any of its other rights, BAI may terminate this Agreement if Licensee fails to comply with any of the terms and conditions of this Agreement. Upon termination of this Agreement, Licensee shall purge all electronic memories of the Licensed *GNOSIS* Program(s). Licensee's obligations under this Agreement shall survive any termination of this Agreement.

6. UPDATES. If Attachment 1 indicates that Licensee has purchased *GNOSIS* Hotline Support and Software Maintenance, then, during the indicated period thereof, BAI will furnish Licensee a copy of any Update(s) of the Licensed *GNOSIS* Program(s). As used herein, the term "Update" shall mean any general version or release of a Licensed *GNOSIS* Program that contains an addition, improvement, enhancement, or other change thereto which is not

separately identified and priced by BAI in its published price schedule and which is marketed by BAI under the same name as that of the Licensed *GNOSIS* Program in question. At the sole discretion of BAI, BAI will furnish new programs that are intended for use in conjunction with the Licensed *GNOSIS* Program(s). Upon delivery to Licensee, by whatever means of delivery, any *GNOSIS* related Updates and New Programs become immediately subject to all terms and conditions of this license agreement. Licensee may renew the *GNOSIS* Hotline Support and Software Maintenance for successive annual periods by paying the annual license fee in effect at each time of renewal.

7. LIMITED WARRANTY AND LIMITATION OF LIABILITY. BAI warrants the magnetic diskette or tape on which the software is recorded to be free from defects in material and workmanship for a period of ninety (90) days from the date of delivery as evidenced by a copy of the receipt. If during this period a defect in the tape or diskette should occur, Licensee may return the same to BAI and it will be replaced without charge. Nothing in this Limited Warranty shall obligate BAI to make Updates available to its customers generally or to Licensee in particular.

BAI warrants that for a period of ninety (90) days from the date of delivery of the Licensed *GNOSIS* Program(s) to Licensee, said program(s) will, if properly installed on the Designated Computer, perform in conformance with the applicable User Documentation; provided, however, that BAI's sole obligation and liability for any breach of the Warranty shall be, in BAI's sole option, to: (i) replace the copy of such defective Licensed Program, (ii) repair or correct the copy of such defective Licensed Program and its User Documentation so that it or they function or read in accordance with such specifications, or (iii) refund a mutually-agreed portion not to exceed seventy-five percent (75%) of the License Fee paid for each defective Licensed Program.

IN NO EVENT SHALL BAI's LIABILITY TO LICENSEE EXCEED THE AMOUNT PAID BY LICENSEE TO BAI HEREUNDER. ANY WARRANTY MADE BY BAI HEREUNDER SHALL BE VOID IN THE EVENT THAT LICENSEE MODIFIES THE LICENSED *GNOSIS* PROGRAM(S), OR USES OR OPERATES THEM ON A COMPUTER OTHER THAN THE DESIGNATED COMPUTER OR AT A LOCATION OTHER THAN THE DESIGNATED SITE. EXCEPT AS EXPRESSLY SET FORTH ABOVE, (1) THE SOFTWARE AND ITS ASSOCIATED DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND BY BAI INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE; AND (2) BAI DOES NOT WARRANT THAT THE SOFTWARE AND ITS ASSOCIATED DOCUMENTATION WILL MEET LICENSEE'S REQUIREMENTS, BE ERROR FREE, OR OPERATE WITHOUT INTERRUPTION, AND THE LICENSEE ASSUMES THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE. IN NO EVENT WILL BAI BE LIABLE FOR SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION OR OTHER PECUNIARY LOSS) EVEN IF BAI HAS BEEN ADVISED OF THE POSSIBILITY THAT SUCH DAMAGES MAY ARISE. WITHOUT LIMITATION TO THE FOREGOING, BAI SHALL HAVE NO LIABILITY WHATSOEVER, WHETHER IN AN ACTION BASED ON CONTRACT OR TORT, ARISING OUT OF OR IN CONNECTION WITH THE PERFORMANCE OR NON-PERFORMANCE OF ANY COMPUTER PROGRAM WHICH RESULTS IN WHOLE OR IN PART FROM THE USE BY THE LICENSEE OF THE LICENSED *GNOSIS* SOFTWARE.

8. OTHER TERMS AND CONDITIONS. No remedy specified in this Agreement is intended to be exclusive, and the rights and remedies specified herein are in addition to any others conferred on BAI by law or in equity. Licensee agrees that should it default in any of the covenants or agreements herein, Licensee shall pay all costs and expenses, including reasonable attorney's fees, which may arise from BAI's enforcement of this Agreement against Licensee. If any provision of this Agreement shall be held by a court of competent jurisdiction to be contrary to law, the remaining provisions of the Agreement shall remain in full force and effect. This Agreement is governed by the laws of the Commonwealth of Virginia, U.S.A. This Agreement, together with Attachment 1 checked and initialed by both parties, constitutes the entire, complete, and exclusive statement of the agreement between the parties with respect to the Licensed *GNOSIS* Program(s) and User Documentation, and supersedes all prior proposals, understandings, or agreements, written or oral with respect thereto. Although the terms and conditions of this Agreement may conflict with or vary from certain terms and conditions, if any, specified by Licensee's order form, BAI's acceptance of Licensee's order and of this Agreement is made expressly on the condition that this Agreement shall govern.

For:	_____	For:	Barron Associates, Inc.
By:	_____	By:	_____
Name:	_____	Name:	Roger L. Barron
Title:	_____	Title:	President

Date: _____ Date: _____

GNOSIS Software License Agreement

Attachment 1

Re Agreement between _____ (Licensee)

and Barron Associates, Inc. (BAI), dated_____:

Designated Computer:

(1) Type/Model _____

(2) Serial No. _____

(3) Operating System/Level _____

(4) Distribution Medium _____

Designated Site: _____

_____

Licensed *GNOSIS* Program	License Fee	User Documentation	Estimated Delivery Date

Initialled for Licensee by: _____

Initialled for BAI by: _____

1

INTRODUCTION

Many scientific, engineering, marketing, and economic phenomena are complex and difficult to model. This usually is owing to a large number of conditions that influence the variable or variables to be modeled or an incomplete understanding of the underlying principles governing the system. The Barron Associates, Inc. Generalized Networks for the Optimal Synthesis of Information Systems *(GNOSIS)* software suite provides a versatile and powerful method for modeling a variety of complex phenomena. *GNOSIS* can solve diversified problems such as:

- modeling systems with continuous-valued outputs (e.g., what is the temperature inside a reactor?),
- classifying data into two or more categories (e.g., will the market go up, down, or stay the same?)
- predicting the future values of time-series data (e.g., what will tomorrow's temperature be?)

GNOSIS offers a number of technical advantages over both traditional statistical modeling and other neural network software tools. These include:

- building blocks that can, on their own, model complex nonlinear data interrelationships,
- a network training algorithm that is orders of magnitude faster than the gradient-descent methods employed by most neural network algorithms, and
- a 'C'-language code generator that allows the user to compile and incorporate networks in custom applications on any computer.

Additionally, with the optional Algorithm for Synthesis of Polynomial Networks – III (*ASPN-III*) structure-learning feature, *GNOSIS* can rapidly sort through thousands of candidate inputs and structures to find a feedforward estimation network of just-sufficient complexity that is designed to perform optimally on unseen data. The structure-learning capability can significantly reduce the time required to obtain a good model, while enhancing model robustness; the details of this option are discussed in Chapter 6 of this manual.

BACKGROUND

An artificial neural network (ANN) synthesis algorithm is a mathematical technique for learning relationships between a set of observed data (inputs) and corresponding dependent data (outputs). In statistics, the construction of such models is called *regression*, but unlike traditional regression, ANNs model complex systems by combining numerous, relatively simple, elements (or neurons). This structure, as the name implies, was originally suggested by the physiology of the human brain and has been particularly well-suited for the modeling of numerous complex, nonlinear systems.

Common neural network techniques use nodal elements (or neurons) that combine their inputs linearly and pass the result through a nonlinear transformation (e.g., sigmoid or radial basis function). In multi-layer networks, elements are grouped in *layers*, and the inputs to each element on any given layer are the outputs of the previous layer. Figure 1.1 shows a two-layer fully-connected feedforward neural network.

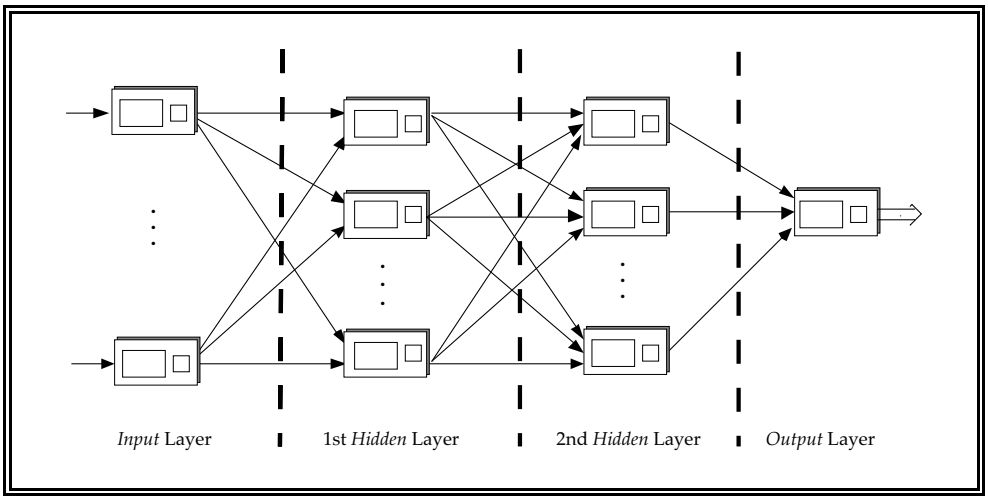


Figure 1.1: A Two-Layer Feedforward Neural Network

GNOSIS creates its models by analyzing user-provided data (called the *synthesis* or *training database*) and finds the best mathematical relationship between measurable input variables and an output variable or variables. The user must, therefore, create a *database* for *GNOSIS* by:

1. Deciding which variables are to be estimated or classified (the *output variables*).
2. Deciding which of the measurable input variables (the *candidate input variables*) potentially influence the output variables.
3. Measuring the candidate input variables and compiling this information with corresponding "truth" measurements (or calculations) of the output variables.

The goal of *GNOSIS* is to generate models of data that are accurate, simple, robust, transparent, and easy to implement. *Accurate models* closely reflect the relationships between measured inputs and their respective output or outputs. *Simple models* require only a few mathematical manipulations of the inputs to calculate the outputs. *Robust models* have behavior that is well-defined, and these models act reliably for specified conditions; they are

also insensitive to moderate uncertainties in the data and perform well on data that the model has not seen during synthesis. *Transparent models*, by the clarity of their form, lead the analyst to further insights into the process being modeled. The final attribute of good models, *ease of implementation*, becomes essential if the models are to be used in time-critical applications.

STRUCTURE LEARNING

If you purchased the *ASPN-III* structure learning feature, *GNOSIS* can be used to find the best feedforward (no time delays or feedback) model that transforms observable variables into one or more estimated dependent variables. The software begins by postulating the simplist possible model (a linear relationship between a selected one of the input variables and the output variable). *GNOSIS* then introduces incremental complexity increases to the model until a just-sufficient level of complexity is reached. "Just sufficient" complexity means that a complexity-constrained modeling criterion (predicted squared error) has been minimized, signifying that, for the given training database, any further growth in model complexity would lead to deterioration - not improvement - in the ability of the model to generalize from its training. In other words, when constructing a network via *ASPN-III*, *GNOSIS* uses information theory to find a balance between model accuracy and simplicity. A proper balance ensures the best model performance when the model is later interrogated on "unseen" data, i.e., data not available during model synthesis. A condition of *overfitting* occurs if the model is allowed to become more complex than justified by the quantity of synthesis data; overfitting leads to loss of performance on unseen data (see Figures 1.2 and 1.3).

To reduce the risk of overfitting, the structure learning algorithm used by *GNOSIS* employs the predicted squared error (PSE) criterion to determine the relative worth of each trial element. Elements with a high degree of complexity are penalized and are therefore less likely to appear in the final model. This "complexity penalty" is a function of the number of parameters in the network, the number of observations in the database, and a measure of the information content in the synthesis database. As a result, the larger and more diverse the database in relation to the number of model parameters, the more complex and therefore more accurate the *GNOSIS* model can justifiably be.

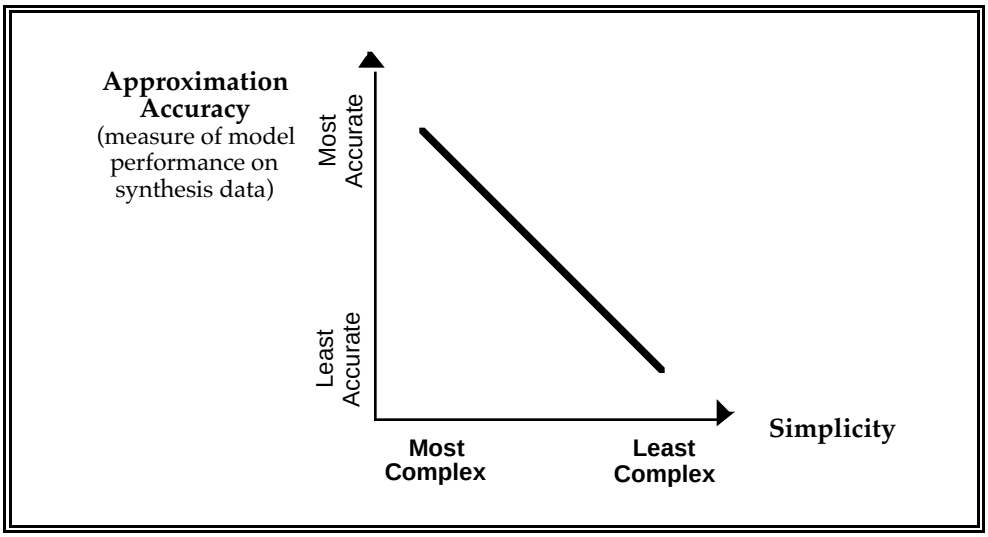


Figure 1.2: Relationship between Model Simplicity and Approximation Accuracy

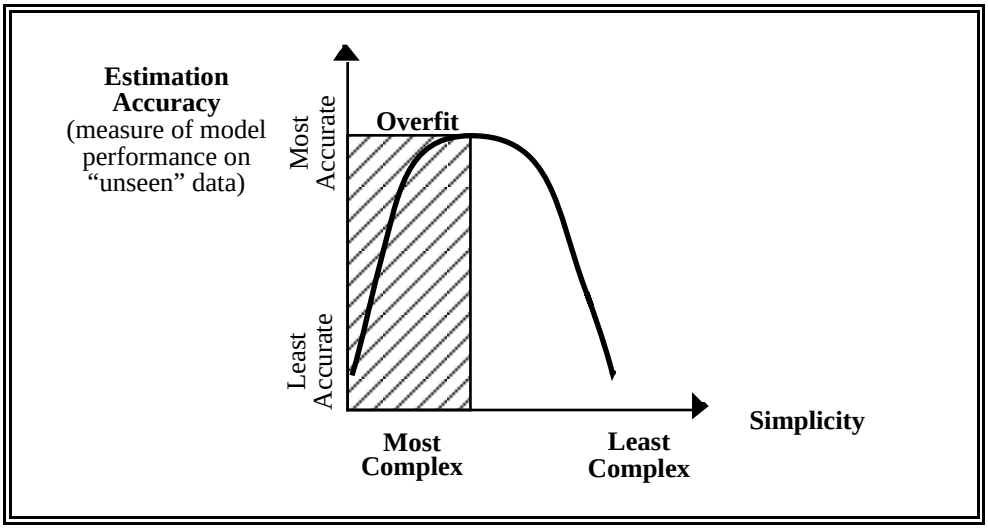


Figure 1.3: Relationship between Model Simplicity and Estimation Accuracy

2

GETTING STARTED

HARDWARE AND SOFTWARE REQUIREMENTS

GNOSIS is currently supported in three environments: UNIX (primarily SUN Operating System and Hewlett-Packard workstations), Macintosh computers (with 68030, 68040, or PowerPC microprocessors), and DOS machines (486 or Pentium microprocessors and Windows 95/NT).† The software functions identically in each of these environments.

INSTALLING THE PROGRAM

GNOSIS and its associated control and test files are not compressed and do not require any special installation software to transform them into a usable form. To install *GNOSIS* on your computer, simply copy the contents of the distribution disk or tape to a hard drive on your computer.

QUICK START TUTORIAL

A quick start tutorial session will help you become familiar with the general process of model creation using *GNOSIS*. Further details are given in the remainder of this manual. The synthesis process consists of three main parts:

- database creation and manipulation
- model synthesis (training)
- model evaluation

Because *GNOSIS* can be used to solve many different types of modeling problems, the following three examples are provided in the next sections. The first tutorial contains the most detail, and succeeding examples refer to it on occasion.

---

† *GNOSIS* is written in *ANSI C* and uses a command-line interface. As a result, it can be ported to a number of different platforms relatively easily. If you do not have one of the platforms listed above, contact Barron Associates, Inc., to inquire about the possibility of porting the software to your specific computer and operating system.

- Tutorial 1: estimation problem using a General Structure File (GSF)
- Tutorial 2: estimation problem using a Network Description File (NDF)
- Tutorial 3: classification problem using a General Structure File

Executing the various commands required in these examples may be different depending on the platform. For example, on DOS or UNIX platforms, type the command lines shown in the tutorials. On a Macintosh, double click on the *GNOSIS* icon, which creates a dialog box for entering command lines.

For all the tutorials, run *GNOSIS* in the same directory with the tutorials files. Either copy the desired tutorial files to the *GNOSIS* directory, or copy the *GNOSIS* executable and license files into the `GNOSIS\TUTORIALS` directory. All input files mentioned in the quick start tutorial session can be found in `GNOSIS\TUTORIALS`.

TUTORIAL 1: ESTIMATION PROBLEM USING A GENERAL STRUCTURE FILE

GNOSIS uses three types of control files (GSF, NDF, and SLF) to synthesize neural network models. These files contain the parameters and options used in training and evaluating a model, and can be modified to achieve better performance from the neural network.

Database Creation and Manipulation

To become familiar with use of a General Structure File (GSF), the first step is to create a database that implicitly describes the problem to be solved. The time-series dynamic estimation problem of this first tutorial session consists of optimizing the parameters of a two-input, two-output neural network. The data to be used for training and evaluation of this neural network are stored in the file, `feedback.dat`, which is shown in Figure 2.1.

GNOSIS keeps track of data columns using the labels in the header line of the database. The first column stores the observation index, n, which is not used by *GNOSIS*, but can remain in the database for the user's convenience.

n	x1	x2	y1	y2
0	0.000000	0.000000	-0.000064	0.000860
1	0.500000	0.500000	1.624742	0.501206
2	1.000000	1.000000	3.001583	0.501587
3	1.500000	1.500000	4.126927	0.009568
4	2.000000	2.000000	5.000255	-1.000652
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
245	2.500000	-3.500000	21.125971	10.108940
246	2.000000	-3.000000	17.498762	10.737074
247	1.500000	-2.500000	12.123316	7.464598
248	1.000000	-2.000000	7.500667	4.466553
249	0.500000	-1.500000	3.624975	2.180159

Figure 2.1: Database File, `feedback.dat`

To use some of the data for synthesis (training) and some for later evaluation, split the database into two subsets and create files accordingly. When determining the database splitting ratio (percentage of observations that will be in each data subset), consider that:

1. The *training subset* should have as many observations as possible to obtain an accurate model (generally around 80% of the observations.)
2. The *evaluation subset* should contain enough observations to provide a statistically sound evaluation.

This network (i.e., model) will have internal feedback and time delays, and the database is split such that the network will be trained on the first 80% to help it identify the time dependencies. The first 200 observations are in `feedback.tra` while the last 50 observations are in `feedback.eva`. For problems that don't involve time-series data, avoid splitting a database such that the first part is used for training and the last part for evaluation. However, for this example, the inputs are in the same domain (same population) during both the first and last parts of the database.

Network Synthesis (Training)

For this tutorial problem, the database has been created using the following nonlinear auto-regressive moving-average equations (NARMAX)

$$y_1[n] = 3\,x_1[n] - x_1[n]\,x_2[n-1] + 0.5\,x_2^2[n] + \varepsilon[n]$$

and

$$y_2[n] = x_1[n] - x_1[n\text{-}1]\,x_2[n] + 0.01\,x_2[n-1]\,y_2^2[n-1] + \varepsilon[n]$$

in which "n" is the observation number and ε[n] represents noise.

Normally one would not know the neural network structure and parameters, but for now it is helpful in gaining understanding of *GNOSIS*.

The single hidden layer for the polynomial neural network that can model these equations is shown in Figure 2.2. The small boxes embedded in the inside left of the node boxes represent the number of samples to delay the respective inputs.

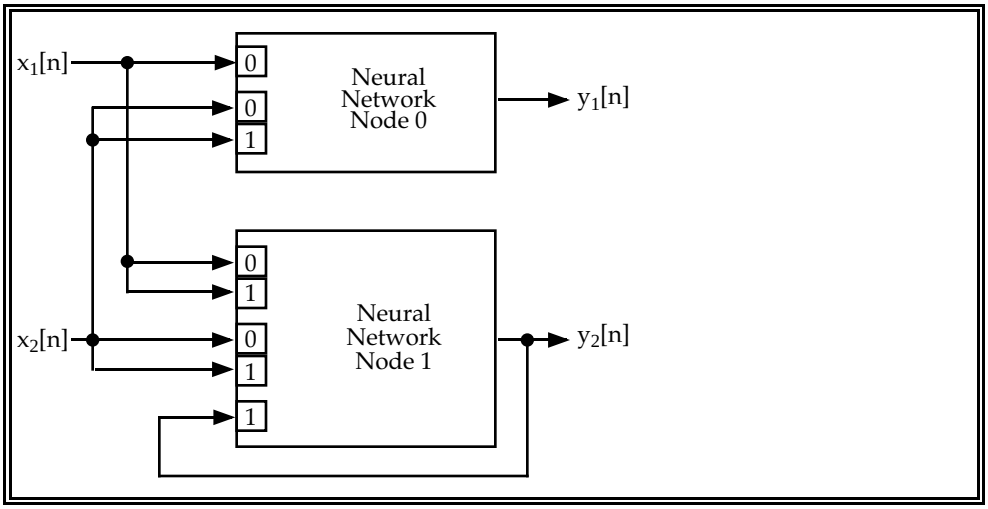


Figure 2.2: Network Structure for Estimation Problem

Training in this case is the process of optimizing the parameters of the given model structure so as to identify the relationship between the database input variables and the output variables.

In its modeling, *GNOSIS* builds a network of polynomial equations. This construction can be controlled in more than one way. In this example, a GSF is used to specify the network to be built. The GSF instructs *GNOSIS* to build a network containing one hidden layer with the same number of elements as there are outputs in the training database. Each element has as inputs all the training database inputs as well as the delayed inputs and/or feedbacks specified in the GSF. Details on the specific fields in the GSF are explained in Chapter 4.

The GSF for this problem is found in the file `feedback.gsf` and is shown in Figure 2.3.

```
General Structure File for GNOSIS.
Copyright 1998 by Barron Associates, Inc. All Rights Reserved.


-------------------------------------------------------------------------------------
MODIFIER                    ARGUMENTS    |COMMENTS
-------------------------------------------------------------------------------------
Inputs                      USE_VSTRING  |USE_VSTRING, or enter input variables by name,
                                         |ending with'|'
Outputs                     USE_VSTRING  |USE_VSTRING, or enter output variables by name,
                                         |ending with'|'
Vstring                     -xxyy        |Specify inputs/outputs: y, x, -, integer
                                         |repeat count (only used if USE_VSTRING
                                         |option entered for Inputs or Outputs)
Distortion_Function         QUADRATIC    |Indicate distortion (loss) function for
                                         |GNOSIS to use (QUADRATIC/LOGISTIC)
Normalize                   NO           |Normalize inputs (YES/NO)
Unitize                     NO           |Normalize/Unitize outputs (YES/NO)
                                         |(only used if Distortion is QUADRATIC)
Init_Network_With_Database  YES          |Use database data to fill shift registers
                                         |with "true" data before optimizing (YES/NO)
-------------------------------------------------------------------------------------
Input_Delays                CUSTOM       |Format for input delays (LINEAR/LOG/CUSTOM)
                                         |LINEAR: delay[i] = previous_delay + Space_Delay
                                         |LOG: delay[i] = Start_Delay + 2^i - 1
                                         |CUSTOM: enter delays in In_Custom_Delays
Start_Delay                 0            |Smallest delay used in each node >= 0
Space_Delay                 3            |Spacing between delays >= 1
Max_Delay                   15           |Maximum delay for each node > Start_Delay
Custom_Delays               0 1          |Delay values >= 0 separated by spaces or tabs
-------------------------------------------------------------------------------------
Use_Prior_Data_Base_Outputs YES          |Use past values of DATABASE output columns
                                         |as network inputs (YES/NO)
Use_Prior_Node_Outputs      NO           |Use past values of NODE output(s) as inputs
                                         |to node - i.e. feedback (YES/NO)
Output_Delays               CUSTOM       |Format for output delays (LINEAR/LOG/CUSTOM)
                                         |LINEAR: delay[i] = previous_delay + spacing
                                         |LOG: delay[i] = start + 2^i - 1
                                         |CUSTOM: enter delays in In_Custom_Delays
Start_Delay                 1            |Smallest delay used in each node >= 1
Space_Delay                 3            |Spacing between delays >= 1
Max_Delay                   16           |Maximum delay for each node > Start_Delay
Custom_Delays               1            |Delay values >= 1 separated by spaces or tabs
-------------------------------------------------------------------------------------
Limit_Nodes                 IO           |Set which nodes to limit.  Nodes from input
                                         |and output layers limited with database
                                         |training values.  (IO/ALL/NONE)
Limit_Range                 1.0          |Percentage of full range (0.01 - 1.0) that
                                         |node outputs  will be limited to. (only used
                                         |if Limit_Nodes is ALL)
-------------------------------------------------------------------------------------
Node_Type                   COMP         |Specify polynomial structure type
                                         |(ADD/MULT/COMP/CUSTOM) ADD = additive,
                                         |MULT = multilinear, COMP = complete
Node_Degree                 2            |Maximum allowable degree of polynomial term
Node_Bias                   NO           |Include bias term in polynomial (YES/NO)
Custom_K_Matrix_File        ?.kmx        |File name of custom K matrix
Post_Trans                  LIN          |Choose post transformation (LIN/SIN/COS/SIG)
-------------------------------------------------------------------------------------
Write_Pix_File              YES          |Write node connections diagram and node
                                         |equations to .pix file
Write_Source_Code           YES          |Write source code to .c and .h files
-------------------------------------------------------------------------------------
```

Figure 2.3: General Structure File, `feedback.gsf`, for Tutorial 1

GNOSIS uses a command-line interface to allow the user to choose execution modes. Chapter 4, "Using *GNOSIS*," describes the available command-line options in detail. To execute *GNOSIS* in training mode, enter the following command

gnosis -d feedback.tra -t feedback.gsf -b feedback

```
************************************************
# GNOSIS Version 3.0
# Copy #0001 licensed to Barron Associates, Inc.

# Copyright 1995-1998, Barron Associates, Inc.
# All rights reserved
# GNOSIS
      Optimizes network structure and/or optimizes
      parameters of neural networks.
************************************************
# Cmdline: gnosis -d feedback.tra -t feedback.gsf -b feedback
# GNOSIS: Sizing input database 'feedback.tra'...
# Database: 5 variables, 200 observations, 1 block(s)
# GNOSIS: Scanning general structure file 'feedback.gsf'...
# GNOSIS: Initializing the network structure...
# GNOSIS: Network input variable(s): x1 x2 y1 y2
# GNOSIS: Network output variable(s): y1 y2
# GNOSIS: Reading input database...
# GNOSIS: Optimizing the parameters for y1...
# ILS: Iteration  0, Lambda 0.0e+00, rScore 1.08483
# ILS: Iteration  1, Lambda 0.0e+00, rScore 0.00804415
# ILS: Iteration  2, Lambda 0.0e+00, rScore 0.00804415
# ILS: Iteration  3, Lambda 1.0e-01, rScore 0.00804415
# ILS: Parameter change below tolerance.
# GNOSIS: Optimizing the parameters for y2...
# ILS: Iteration  0, Lambda 0.0e+00, rScore 1.00459
# ILS: Iteration  1, Lambda 0.0e+00, rScore 0.0145046
# ILS: Iteration  2, Lambda 1.0e-01, rScore 0.0145046
# ILS: Iteration  3, Lambda 1.0e-02, rScore 0.0145046
# ILS: Parameter change below tolerance.
# GNOSIS: Evaluating the network...
# GNOSIS: Writing estimation file 'feedback.est'...
# GNOSIS: Writing statistics file 'feedback.sts'...
# GNOSIS: RMS Error: y1 = 0.00103293  y2 = 0.0459645
# GNOSIS: Norm RMS Error: y1 = 0.000183732  y2 = 0.0133094
# GNOSIS: R Squared: y1 = 1  y2 = 0.999823
# GNOSIS: Writing network description file 'feedback.mdl'...
# GNOSIS: Writing graphic representation of network to 'feedback.pix'...
# GNOSIS: Writing source code to 'feedback.c'...
# GNOSIS: Writing header code to 'feedback.h'...
# GNOSIS: DONE @3.90 sec
```

Figure 2.4: Display of Training Run for Tutorial 1

As indicated on the screen, *GNOSIS* generates a number of files when network synthesis is complete. The files are named using the specified base with the appropriate extension appended:

- `feedback.mdl` model file: model structure, and parameters for future *GNOSIS* use; same format as Network Description File
- `feedback.pix` structure file: model training score, structure, and parameters in a semi-graphical form
- `feedback.c` source code file: C source code for model implementation
- `feedback.h` source code file: C header file with sample main program
- `feedback.est` estimation file: true and modeled output from training database
- `feedback.sts` statistics file: model performance metrics on training database

To verify that everything worked correctly, look in the structure file, `feedback.pix`. For the first equation, the significant parameters obtained by *GNOSIS* on the training data for Node5 will be close to those originally defined in the representative equations: 3, –1, and 0.5; all other terms should have coefficients that are nearly zero. For the second equation the coefficients will not match the equations exactly because `feedback.gsf` limited *GNOSIS* to

second-order interactions of inputs, and $x_2[n–1]\ y_2^{\,2}[n–1]$ is a third-order term. However, *GNOSIS* still performs well on this problem. Tutorial 2 will show how to customize these networks to get rid of unnecessary terms and include new terms.

Model Evaluation

GNOSIS automatically produces an estimation file as part of the training process. However, this file initially contains outputs that were estimated using data in the training dataset. *GNOSIS* should be rerun to get results on independent (unseen) data.

To evaluate the *GNOSIS* model stored in `feedback.mdl`, enter the command

gnosis -d feedback.eva -e feedback.mdl -b feedback_eval

The generated display appears in Figure 2.5, and results are saved in two files:

- `feedback_eval.est` estimation file: true and modeled output from evaluation database
- `feedback_eval.sts` statistics file: model performance metrics on evaluation database

```
************************************************
# GNOSIS Version 3.0
# Copy #0001 licensed to Barron Associates, Inc.

# Copyright 1995-1998, Barron Associates, Inc.
# All rights reserved
# GNOSIS
      Optimizes network structure and/or optimizes
      parameters of neural networks.
************************************************
# Cmdline: gnosis -d feedback.eva -e feedback.mdl -b feedback_eval
# GNOSIS: Sizing input database 'feedback.eva'...
# Database: 5 variables, 50 observations, 1 block(s)
# GNOSIS: Scanning input network description file 'feedback.mdl'...
# GNOSIS: Initializing the network structure...
# GNOSIS: Network input variable(s): x1 x2 y1 y2
# GNOSIS: Network output variable(s): y1 y2
# GNOSIS: Reading input database...
# GNOSIS: Evaluating the network...
# GNOSIS: Writing estimation file 'feedback_eval.est'...
# GNOSIS: Writing statistics file 'feedback_eval.sts'...
# GNOSIS: RMS Error: y1 = 0.00123978  y2 = 0.0495621
# GNOSIS: Norm RMS Error: y1 = 0.000213843  y2 = 0.0124968
# GNOSIS: R Squared: y1 = 1  y2 = 0.999845
# GNOSIS: DONE @0.37 sec
```

Figure 2.5: Display of Evaluation Run for Tutorial 1

If many network models are tested for a given problem, compare all the evaluation statistics files to determine the best model. The best model should not exhibit a significant difference between the model performance on its training data and the evaluation data.

TUTORIAL 2: ESTIMATION PROBLEM USING A NETWORK DESCRIPTION FILE

The second tutorial session uses the same database as the first, but a Network Description File (NDF) is used during network synthesis rather than a General Structure File (GSF). Thus, the database creation and manipulation, network evaluation, and network implementation stages of model development are the same as in Tutorial 1.

Network Synthesis (Training)

The Network Description File or NDF provides an alternative means of describing the network structure for *GNOSIS* to optimize. Whereas the General Structure File may be somewhat simpler and easier to work with, the NDF is much more flexible. The Network Description File (NDF) for this tutorial, `feedback.ndf`, appears in Figure 2.6. The specific fields in the NDF are explained in detail in Chapter 4. The network structure represents the equations given for the problem in Tutorial 1, except that the noise is not modeled.

```
GNOSIS NETWORK DESCRIPTION
              Inputs: x1 x2 |
              Outputs: y1 y2 |
              Class_Single_Output: NO
              Number_of_Layers: 3
              Distortion_Function: QUADRATIC
              Normalize: NO
              Unitize: NO
              Limit_Nodes  NONE
              Init_Network_With_Database: NO

NETWORK STRUCTURE
-----------------------------------------------
INPUT_LAYER:
              NODE_0-0 DESCRIPTION:
                   Parameters:       0        1
                   Output_Range:     0        0
              NODE_0-1 DESCRIPTION:
                   Parameters:       0        1
                   Output_Range:     0        0
-----------------------------------------------
LAYER 1:
              Number_of_Nodes: 2
              Core_Transformation: POLY
              Post_Transformation: LIN
              NODE_1-0 DESCRIPTION:
                   Input_Node(s): 2
                        NODE 0 - 0
                             Delays: 0
                        NODE 0 - 1
                             Delays: 0 1
                   Parameters:
                        0.1 -0.1 0.1
                   Set_of_Indices:
                        1 0 0
                        1 0 1
                        0 2 0
                   Optimize:         YES
                   Output_Range:     0        0
              NODE_1-1 DESCRIPTION:
                   Input_Node(s): 3
                        NODE  0 - 0
                              Delays: 0 1
                        NODE  0 - 1
                              Delays: 0 1
                        NODE  1 - 1
                              Delays: 1
                   Parameters:
                        0.1   -0.1  0.1
                   Set_of_Indices:
                        1 0 0 0 0
                        0 1 1 0 0
                        0 0 0 1 2
                   Optimize:         YES
                   Output_Range:     0        0
-----------------------------------------------
OUTPUT_LAYER:
              NODE_2-0 DESCRIPTION:
                   Input_Node: 1 - 0
                   Parameters:       0        1
                   Output_Range:     0        0
              NODE_2-1 DESCRIPTION:
                   Input_Node: 1 - 1
                   Parameters:       0        1
                   Output_Range:     0        0
```

Figure 2.6: Network Description File, `feedback.ndf`, for Tutorial 2

To execute *GNOSIS* in the training mode using the NDF, enter the following command:

gnosis -d feedback.tra -t feedback.ndf -b feedback

To evaluate the model, follow the steps outlined in Tutorial 1.

TUTORIAL 3: CLASSIFICATION PROBLEM USING A GENERAL STRUCTURE FILE

The third tutorial describes how to use *GNOSIS* to classify species of the iris flower using a well-known database.

Database Creation and Manipulation

In Fisher's iris database [Fisher, 1936], there are three classes (or species) of iris, with 50 observations recorded for each class. Sample data contained in `iris.dat` are shown in Figure 2.7. Note that each class (or species) is entered as a character string. *GNOSIS* considers each unique string to be a different class.

Sepal_length	Sepal_width	Petal_length	Petal_width	Class
5.1	3.5	1.4	.02	Species_1
4.9	3	1.4	.02	Species_1
...	...	...	...	...
7	3.2	4.7	1.4	Species_2
...	...	...	...	...
7.1	3	5.9	2.1	Species_3
...	...	...	...	...

Figure 2.7: Fisher's Iris Flower Data

As before, the original iris data should be split into a *training database* and an *evaluation database*. Because there are no time-dependencies in the iris data, the data should be split randomly. Sample training (`iris.tra`) and evaluation (`iris.eva`) databases are provided.

Network Synthesis (Training)

Fisher's iris data represent a classification problem. The optimum network structure for classification problems has C – 1 output nodes, where C is the number of output classes, and each output represents the probability of its respective class (1, 2, ... , C – 1). Note that because probabilities must sum to unity, the probability of class C, P(C), is determined by P(C) = 1 – [P(1) + P(2) + ... + P(C–1)].

The *GNOSIS* neural network structure for this classification problem appears in Figure 2.8.

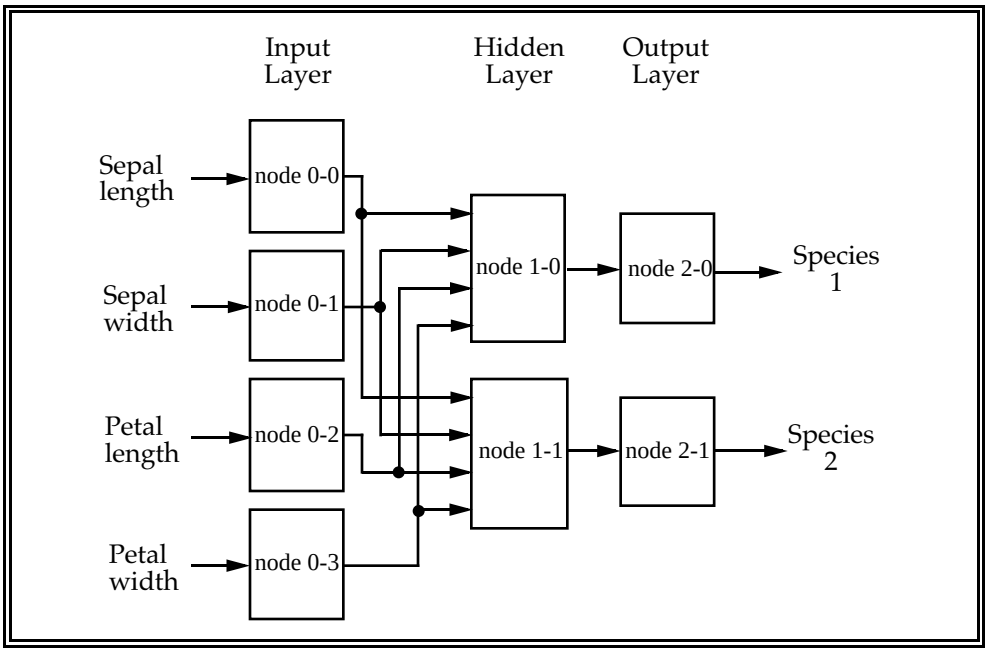


Figure 2.8: Network Structure for Iris Classification Problem

Since, for this problem, the network requires only one hidden layer, the General Structure File (GSF) is a quick and easy way to guide *GNOSIS* in finding an optimal solution. The GSF provided for the iris problem, `iris.gsf`, is shown in Figure 2.9. Train the classification network by entering the following command:

gnosis -d iris.tra -t iris.gsf -b iris

For classification problems, *GNOSIS* writes the estimation file, `iris.est` with the true probability versus the probability based on the network outputs. *GNOSIS* also includes "confusion" matrices in the statistics file, `iris.sts`, which provide a simple representation of how well observations were classified. See Chapter 5 for details.

Model Evaluation

Evaluate the model *GNOSIS* created, using the evaluation data, by entering on the command line:

gnosis -d iris.eva -e iris.mdl -b iris_eval

GNOSIS writes the statistics file, `iris_eval.sts`, and estimation file, `iris_eval.est`. Use the statistics file to determine how well *GNOSIS* performed. The logistic loss score of the evaluation run *seems* poor. A "good" score is usually close to zero, and scores close to or greater than one are usually "bad." However, in this example, *GNOSIS* actually classified 29 out of 30 observations correctly. The nature of the logistic-loss function, used for training classification networks, reduces the utility of using score as a measure of network performance. Percent correct classifications is frequently a better measure of network performance.

```
General Structure File for GNOSIS.
Copyright 1998 by Barron Associates, Inc. All Rights Reserved.

-------------------------------------------------------------------------------------
MODIFIER                    ARGUMENTS    |COMMENTS
-------------------------------------------------------------------------------------
Inputs                      USE_VSTRING  |USE_VSTRING, or enter input variables by name,
                                         |ending with'|'
Outputs                     USE_VSTRING  |USE_VSTRING, or enter output variables by name,
                                         |ending with'|'
Vstring                     xxxy         |Specify inputs/outputs: y, x, -, integer
                                         |repeat count (only used if USE_VSTRING
                                         |option entered for Inputs or Outputs)
Distortion_Function         LOGISTIC     |Indicate distortion (loss) function for
                                         |GNOSIS to use (QUADRATIC/LOGISTIC)
Normalize                   NO           |Normalize inputs (YES/NO)
Unitize                     NO           |Normalize/Unitize outputs (YES/NO)
                                         |(only used if Distortion is QUADRATIC)
Init_Network_With_Database  NO           |Use database data to fill shift registers
                                         |with "true" data before optimizing (YES/NO)
-------------------------------------------------------------------------------------
Input_Delays                CUSTOM       |Format for input delays (LINEAR/LOG/CUSTOM)
                                         |LINEAR: delay[i] = previous_delay + Space_Delay
                                         |LOG: delay[i] = Start_Delay + 2^i - 1
                                         |CUSTOM: enter delays in In_Custom_Delays
Start_Delay                 0            |Smallest delay used in each node >= 0
Space_Delay                 3            |Spacing between delays >= 1
Max_Delay                   15           |Maximum delay for each node > Start_Delay
Custom_Delays               0            |Delay values >= 0 separated by spaces or tabs
-------------------------------------------------------------------------------------
Use_Prior_Data_Base_Outputs NO           |Use past values of DATABASE output columns
                                         |as network inputs (YES/NO)
Use_Prior_Node_Outputs      NO           |Use past values of NODE output(s) as inputs
                                         |to node - i.e. feedback (YES/NO)
Output_Delays               CUSTOM       |Format for output delays (LINEAR/LOG/CUSTOM)
                                         |LINEAR: delay[i] = previous_delay + spacing
                                         |LOG: delay[i] = start + 2^i - 1
                                         |CUSTOM: enter delays in In_Custom_Delays
Start_Delay                 1            |Smallest delay used in each node >= 1
Space_Delay                 3            |Spacing between delays >= 1
Max_Delay                   16           |Maximum delay for each node > Start_Delay
Custom_Delays               1            |Delay values >= 1 separated by spaces or tabs
-------------------------------------------------------------------------------------
Limit_Nodes                 IO           |Set which nodes to limit.  Nodes from input
                                         |and output layers limited with database
                                         |training values.  (IO/ALL/NONE)
Limit_Range                 1.0          |Percentage of full range (0.01 - 1.0) that
                                         |node outputs  will be limited to. (only used
                                         |if Limit_Nodes is ALL)
-------------------------------------------------------------------------------------
Node_Type                   COMP         |Specify polynomial structure type
                                         |(ADD/MULT/COMP/CUSTOM) ADD = additive,
                                         |MULT = multilinear, COMP = complete
Node_Degree                 1            |Maximum allowable degree of polynomial term
Node_Bias                   YES          |Include bias term in polynomial (YES/NO)
Custom_K_Matrix_File        ?.kmx        |File name of custom K matrix
Post_Trans                  LIN          |Choose post transformation (LIN/SIN/COS/SIG)
-------------------------------------------------------------------------------------
Write_Pix_File              YES          |Write node connections diagram and node
                                         |equations to .pix file
Write_Source_Code           YES          |Write source code to .c and .h files
-------------------------------------------------------------------------------------
```

Figure 2.9: General Structure File, `iris.gsf`, for Tutorial 3

3

DATA FILE FORMAT AND PREPARATION

INTRODUCTION

GNOSIS requires a database of input and output data sequences, with optional weights, which it uses to determine the parameters in and sometimes the structure of a neural network. *GNOSIS* is a supervised learning tool, due to its reliance on a "truth" output in the database. To manipulate the database of candidate inputs and true outputs, Barron Associates provides a proprietary tool, *DB*, with *GNOSIS*. *DB* is a command-line oriented program with detailed on-line help; a general description of *DB* functions is included in this section.

GENERAL DATA FILE FORMAT

GNOSIS uses a specific form of numerical text database, divided into a header and a data section. The header is the first line of the database and lists the variable names, optionally followed by size specifiers. The variable names label the columns of the database; each is followed by a tab. Names may include letters, digits, and underscore, but must start with a letter. The sizes include the number of blocks and the numbers of observations in each block, each followed by a space:

```
var1 var2 ... var_last num_blocks num_obs1 num_obs2 … num_obs_last_block
```

The data section is composed of blocks, observations, and variables. A block (typically a time series) is made up of observations, and each observation consists of a set of tab-delimited variable values. Each block may contain a different number of observations; blocks are separated by a line containing four hyphens. The number of variables, blocks, and observations are limited only by available memory. Once block sizes are determined from either the size specifiers or by counting the observations between block separators, the separators are ignored. Therefore, the separators are optional if sizes are specified. If the separators are misplaced according to the size specifiers, the block sizes specified prevail.

All variables appear as vertical columns in the database and may include inputs, outputs, and optional observation weights. Variable values should be numerical, except for single output

classification data (see below). Weights should be zero or positive numbers. Observations (or samples) appear as rows in the database. Each number in the observation is separated by tab. Each observation must have the same number of variables and is terminated by a newline. Make sure the last line of data is followed by a newline, or it may not be read.

An example database file, depicted in Figure 3.1, has four variables including a column of observation weights; observations are in two blocks, four in the first block, and five in the second. The weights have value only in relation to each other and may use any positive scale; so weights of 0.1, 0.2, and 0.25 are equivalent to those in Figure 3.1. Weighting an observation with zero is the same as removing the observation. Doubling an observation weight is the same as duplicating the observation with the original weight. Weights can be used to discount data of dubious value and emphasize data of known quality, or to make up for underrepresented points in the training region of interest; see the Database Tips section following.

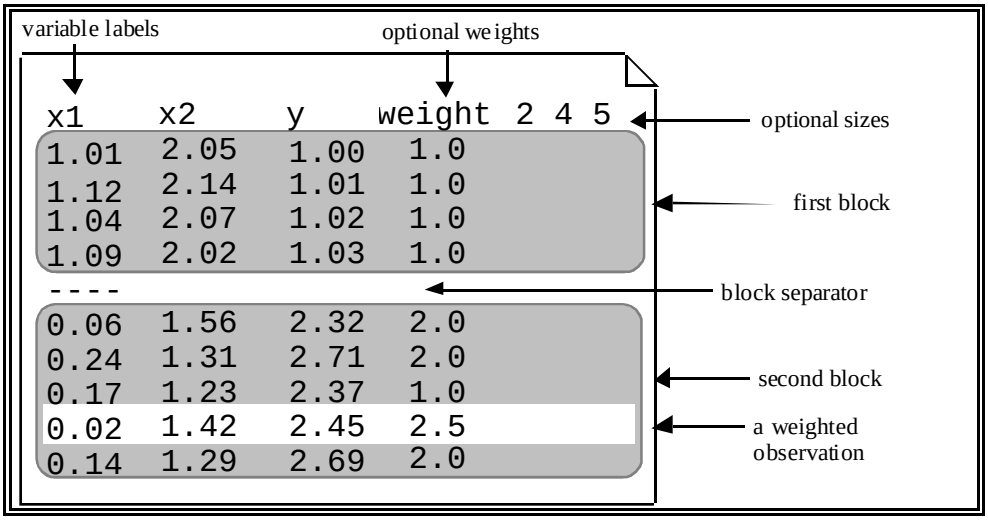


Figure 3.1: Example Database File

When creating a database for a classification problem, the user has two options for output specification:

- making a single output for which each class is a possible value, or
- making probability output variables for each class.

The single-output file format permits the user to enter strings as values for each observation. The strings may include letters, digits, and underscore, but must begin with a letter. An example of a single-output file format (without the optional sizes) appears in Figure 3.2. For this example, x1 and x2 are inputs, Class is the name of the output, and mouse, dog, and cat are the possible classes.

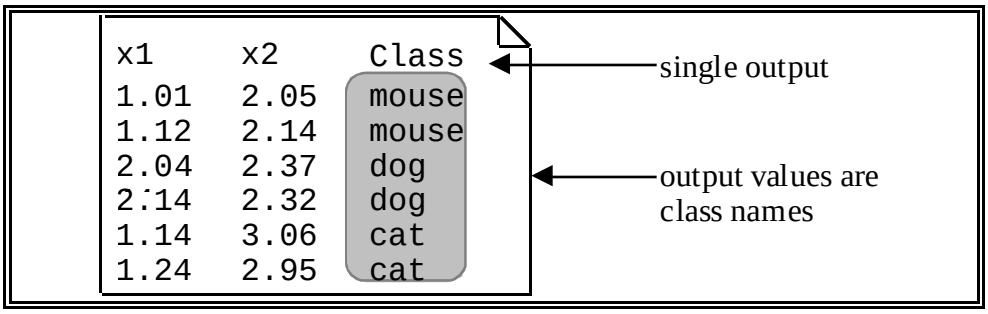


Figure 3.2: Example Single-Output File for Classification Problem

True class values can also be specified with the probability-outputs file format as in Figure 3.3. The values for each class output are probabilities that the input data values represent the given class. For measured data, enter a number between 0 (data do not represent this class) and 1 (data represent this class).

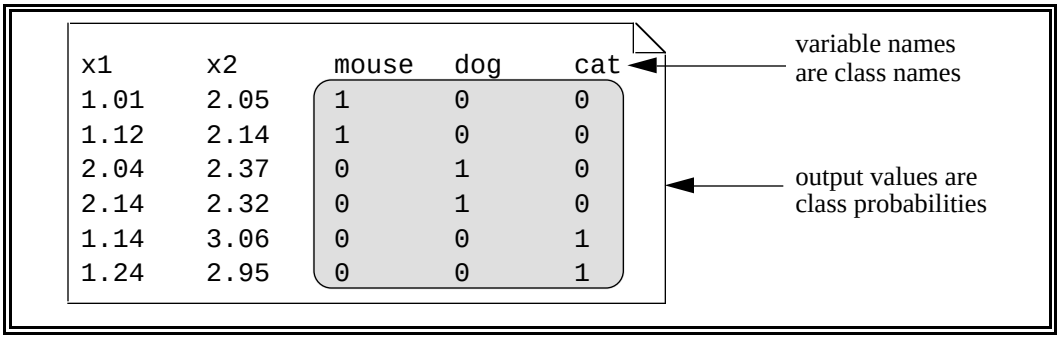


Figure 3.3: Example Probability-Outputs File for Classification Problem

DB DATABASE MANIPULATION TOOL

DB is a general purpose tool for manipulating any database formatted as described above; numeric and classification string data are preserved. The program provides functions to modify or sort existing variables, append new variables to or select observations from an existing database, or create a new database. Several functions display statistics or metrics for variables in a database. *DB* can split a database into two new databases for *GNOSIS* training and evaluation, or merge multiple databases into one. Finally, *DB* can shift time-series data to produce delay or advance variables.

A brief definition of each *DB* function is given below. To get detailed help, enter

db -help -function

where function is one of the commands listed below; also see Appendix D. Most functions require an input and output database filename. In general, filenames should be distinct, although in practice many functions will properly overwrite an input database with output data.

Following are *DB* functions which modify existing databases; i.e., the output database has the same variables as the input database:

`-clip` Limits one or more variables to specified numeric ranges.

-zero Zeroes one or more variables inside or outside specified numeric ranges.
-sort Sorts a database using one or more variables as keys.

Following are *DB* functions which augment existing databases; i.e., new variables are either appended to the database or output alone in a new database:

-calc Calculates new variables from expressions using existing variables and/or constants with operators and built-in functions such as sin and log.
-delay Creates new variables which are delayed or advanced values of existing variables.
-random Creates a new variable using a uniform or gaussian random number generator.
-sequence Creates new variables as linear sequences beginning from existing variable values or constants.

Following are *DB* functions which make new databases by combining or extracting from existing databases:

-merge Merges all of the variables, or all of the observations from one or more databases.
-split Splits the observations from one database randomly into two partial databases.
-select Deletes or extracts a list of variables or observations based on block/observation range or sequence, or value of one or more variables.

Following are *DB* functions which produce displays rather than databases:

-help Shows the syntax of every *DB* function, or detailed help for any specified function.
-corr Shows the linear correlation between pairs of variables.
-histo Shows the distribution with histogram bins for one or more variables.
-stats Shows the range, mean, median, standard deviation and variance for one or more variables.

For single-output classification data, the numeric *DB* functions use the underlying indices of the strings for calculations and produce new data as numbers. For example, "-calc new_var=old_var -a", where old_var is a column of class strings, preserves old_var as strings and appends a column, new_var, which are the integer indices of the strings, beginning with 0.

DB functions are specified and parameterized using command-line options, one function per *DB* invocation. Multiple *DB* command lines may be written in a command file named `db.cmd`. If *DB* is invoked without parameters, it processes all of the commands found in the `db.cmd` command file. *DB* command files can be documented by comment lines beginning with '#'. The commented command file capability allows the construction of more complex database transformations. For example:

```
# Add observation numbers for identification; preserve original database.in
db -sequence database.in OBS 1 1 -out database.tmp1

# Characterize TORQUE
db -stats database.in TORQUE
db -histo database.in TORQUE 5
```

```
# Convert variable TORQUE to 0 below threshold 10000, 1 above the threshold
db -zero database.tmp1 TORQUE 10000 $ -out database.tmp2
db -clip database.tmp2 TORQUE 0 10000 -out database.tmp1
db -calc database.tmp1 AT_THRESHOLD=TORQUE/10000 -a -out database.tmp2
db -select database.tmp2 -del TORQUE -out database.tmp1

# Split 80% of observations into training database, 20% into evaluation database
db -split database.tmp1 -obs 0.8 -out database
```

When using *DB* on a UNIX platform, single or double quotes should enclose any expressions which include UNIX control characters such as '*' For example:

```
db -calc database.in Qbar="Rho * V * V/2" -a -out database.out
```

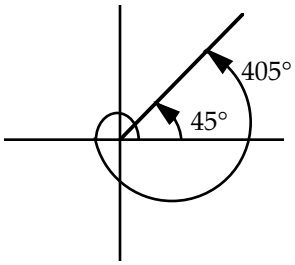
To redirect *DB* displays, such as the output from -stats, -histo, and -corr functions, to a UNIX or DOS file, append "> file" to the *DB* command. On a Macinstosh, console output can be saved after the run; Command Q exits the program and offers a Save option.

DATABASE TIPS FOR GETTING THE BEST MODEL PERFORMANCE

The quality of a generated model is dependent upon the data used to train the model. There are several issues to consider when developing a *training database*: numbers of observations and variables, and quantification of variables, outliers, and data region boundaries.

To provide adequate data, include enough observations with sufficient variety to represent the possible combinations of variable values. Also, if using *GNOSIS* for a classification problem, equal numbers of exemplars for all classes in the database will improve the modeling process, assuming the prior expectations of the classes are equal. For instance, if a total of 1,000 observations are available for five classes, it is desirable to have approximately 200 observations for each class. If one or more of the classes must be underrepresented (i.e., the desired quantity of data is not available), be sure to include as many observations as possible. Add a column of constant observation weights to the database using the *DB* –calc weight=1.0 function. Then increase the weights of underrepresented classes until the weight sum for each class is equal. This will assure proper weighting, but is much less desirable than having a database comprised of unique exemplars.

The second database issue involves the correct quantification of the database values. The variable values should be properly quantified by being single-valued and reflecting true magnitudes. For example, there is only a single case in which one can use degrees (or radians) and adhere to the two guidelines for quantifying data (see Figures 3.4 - 3.6):

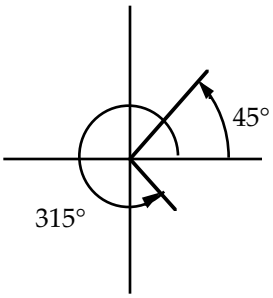


Values of *degree* are not restricted

Characteristics:

Not single-valued (i.e. 45° = 405°).
Value does not reflect magnitude (i.e. 405° is as close to 0° as 45°).

Figure 3.4: Unrestricted Angle Values

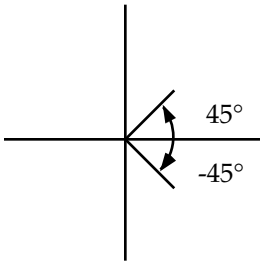


$0° < degree \leq 360°$

Characteristics:

Single-valued.
Value does not reflect magnitude (i.e. 45° is as close to 0° as 315°).

Figure 3.5: Improperly Restricted Angle Values



$-180° < degree \leq 180°$

Characteristics:

Single-valued.
Value reflects magnitude.

Figure 3.6: Properly Restricted Angle Values

If two of the variables contain the parts of a number in polar coordinates (*magnitude, angle*), it may be safer to use Cartesian coordinate (*x,y*) equivalents instead. They can be calculated using the *DB* -calc function:

$$x = magnitude * cos(angle) \qquad y = magnitude * sin(angle)$$

Likewise, sin(angle) and cos(angle) should be used in lieu of angle if angle is unbounded.

Many time series data are cyclical or have harmonic content. An example is the total power demand imposed on an electrical grid; this usually has diurnal and seasonal periodicities. It can be helpful to encode time (t) as two variables (t_1 and t_2) for each period of interest; thus

$$t_1 = \sin\left(\frac{t - t_{ref}}{T}\right) \qquad t_2 = \cos\left(\frac{t - t_{ref}}{T}\right)$$

where t_{ref} is a reference time (perhaps 0000 hours on the first day of Winter) and T is the period of interest (24 hours, 365 days, etc.).

To explain the next database issues, outliers and data region boundaries, the term *region* must be defined. Each input variable consists of certain typical values that define a one-dimensional region (a mathematical domain). Combining *n* of these variables and their respective value domains, an *n*-dimensional region is created.

A database may contain one or more *outliers*, observations that are a great distance from the heart of the database *n*-dimensional region. For instance, if an observation in the iris data had a petal width measurement of 6 feet, it would be reasonable to assume that an error had been made and that the observation should be removed from the database. A help in identifying if the database contains *outliers* is to calculate the median and mean values of each variable, using the *DB* -stats function. If the mean and median have greatly dissimilar magnitudes and/or have values that are unreasonable for the problem at hand, then it is likely that outliers are present. The *DB* -histo function can also help with detection of outliers by displaying histogram bin counts for each variable. While *GNOSIS* is less sensitive to outliers than many other modeling techniques, outliers can still have a large impact on model performance and should usually be purged from the *training* database. If, however, feedback or delays exist in a time-series database, outlier observations should not generally be removed. If appropriate, an outlier may be replaced by an average of its immediate neighbors.

The last database issue deals with the boundaries of the *n*-dimensional region. Empirical models generally perform less well near the boundaries of their training databases. The best way to compensate for this degraded performance is to extend the boundaries of the original training database to create a new training database that consists of an *extended region* and the *region of interest*. (see Figure 3.7). This is done with the understanding that the model should only be interrogated in the region of interest. Note that the extended region should not be excessively large relative to the region of interest, or the incorrect region may be modeled!

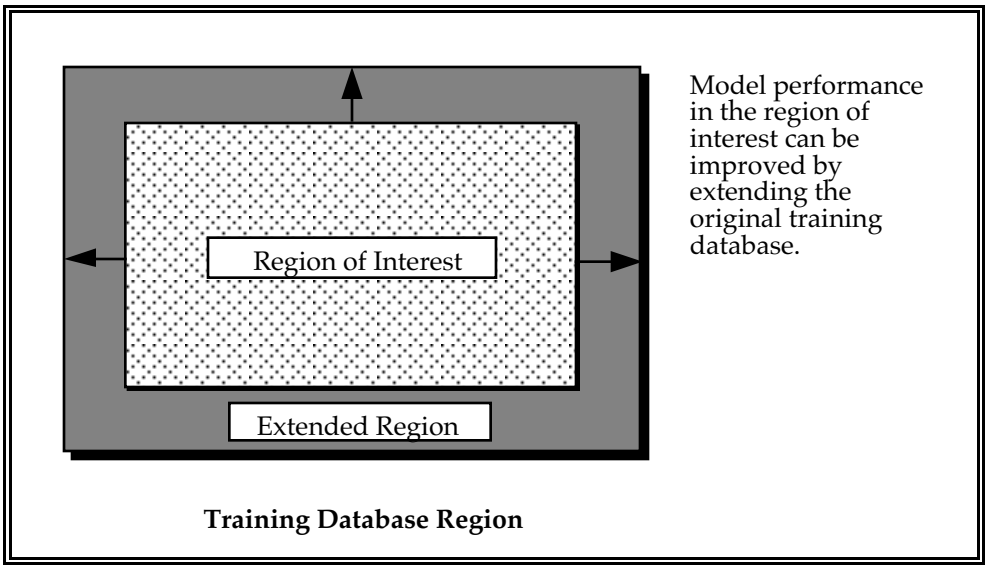


Figure 3.7: Extended Training Database Region

If the training database cannot be extended beyond the region of interest, include, if possible, a heavier proportion (tighter density) of observations near the edges of the region (and particularly its corners or vertices) than in its interior (see Figure 3.8).

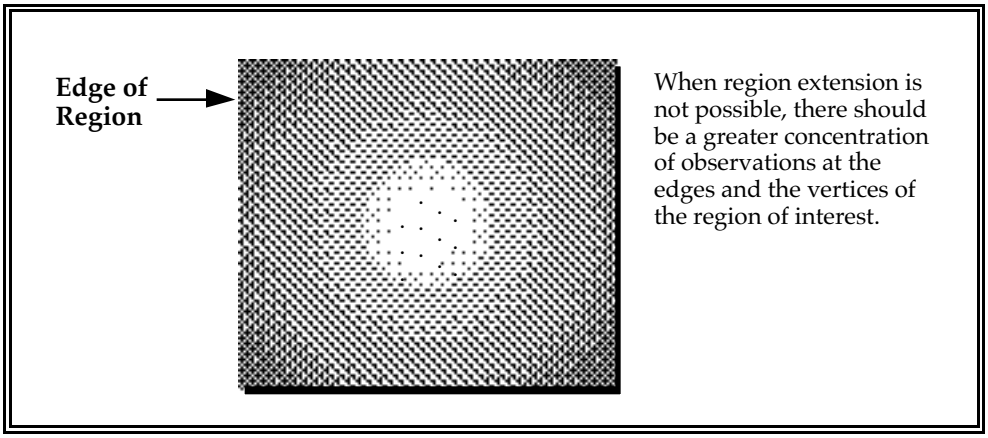


Figure 3.8: Distribution of Observations in Unextended Region

For data with a large number of potential inputs, it quickly becomes very difficult to ensure uniform and dense population of the data space. Witness the *curse of dimensionality*. The optional structure-learning (*ASPN-III*) feature of *GNOSIS* can avoid many of the problems associated with large-dimensional data sets by selecting appropriate inputs and growing models made up of low-dimensional components.

RESERVE SOME DATA FOR EVALUATION PURPOSES

To evaluate a model, it is a good idea to set aside some data for independent testing. The training database and evaluation database both should be representative of the entire data

population. The training database is used to generate a model, and the evaluation database is then used to test the quality of this model on unseen, statistically similar data.

When using *GNOSIS* for a classification problem (i.e., when using the logistic-loss function), or for an estimation problem with no time-delays, randomly split the database using the *DB -split -obs* function. Randomized database splitting is especially important if the data are listed systematically (i.e., roughly sorted by one of the variables) because one needs to increase the likelihood that all sections of the region of interest will be adequately represented in each database. Such representation is particularly important if the model is meant to be valid over the whole region, since a model is not likely to do well in an area strange to it.

When using *GNOSIS* for an estimation problem with time-delays or feedback, the relationships between the inputs and outputs for a given observation depend on sequential observations in the same time-series. The *GNOSIS* database format lends itself well to multiple collections of time-series data through the use of blocks. Randomly split some blocks into a training database and the rest into an evaluation database using the *DB -split -block* function.

4

USING *GNOSIS*

GNOSIS can generate models or evaluate existing models. The training function is used to create a model that best describes the relationship between input variables and output variables in a training database. *GNOSIS* provides three options for how the network synthesis is controlled during training:

- general network structure specification,
- unique network structure specification, and
- structure-learning (optional).

The evaluation function is used to assess the performance of an existing model on data it has not seen before. Both the training and evaluation functions are accessed using command-line options.

This chapter is organized into five sections. The first describes command line usage. The next two detail how to train fixed-structure models using the first two options described above. The fourth briefly describes how to decide which training option is appropriate for what you are trying to accomplish. Evaluation usage is presented in the final section. The optional structure learning algorithm is discussed in Chapter 6.

COMMAND-LINE USAGE

GNOSIS uses the syntax conventions of a UNIX style command. All *GNOSIS* command options are prefixed with a '-'; option arguments may follow the option. Command line text can be stored in a file named `gnosis.cmd`. Double click the gnosis icon on a Macintosh or Windows 95/NT PC, or enter:

```
gnosis
```

at a UNIX or DOS prompt (after loading Windows 95/NT), and *GNOSIS* will read the command line from `gnosis.cmd`. The command line file, `gnosis.cmd`, can have as many lines as desired, specifying one or more *GNOSIS* runs. Blank lines or lines beginning with '#' are skipped; all other lines are processed as *GNOSIS* commands. The `gnosis.cmd` file provided with *GNOSIS* has training and evaluation commands for all tutorials.

All other *GNOSIS* specifications are given in the general structure file (GSF), network description file (NDF), or optional structure learning file (SLF); output is written to result files and standard output on the screen. See Chapter 5, "Interpreting Generated Output," for more information about the result files. On a Macintosh, standard output can be sent to a file after the run; Command Q exits the program and offers a Save option. On UNIX or DOS platforms, standard output is redirected by appending a ">file" to the *GNOSIS* command. On DOS or UNIX platforms, *GNOSIS* can be included in the PATH; otherwise, *GNOSIS* expects to find input files and outputs result files in the same directory as the executable file.

Syntax for the the command line options:

```
gnosis -h(elp)
gnosis -d(ata) data_file -t(rain) control_file -b(ase) name
gnosis -d(ata) data_file -e(val) model_file -b(ase) name
```

Full spelling is optional; one character after the '-' is sufficient. Option `-d` gives the data file used to optimize network parameters or evaluate an existing network. Option `-e` gives the model file to be evaluated; typically the extension is mdl. Option `-t` gives the NDF, GSF, or optional SLF control file governing the training process; typically the control file has an extension of ndf, gsf, or slf. *GNOSIS* recognizes a GSF by finding the word "general" (case-insensitive) in the first line of the control file, an SLF by finding "learning". Otherwise, the control file is processed as an NDF. To be sure your control file is properly recognized by *GNOSIS*, copy a GSF or SLF from the tutorials or an NDF from a model file produced by *GNOSIS*, and modify it for your purposes. Option -b gives the base name to be used for all output files; appropriate extensions are added to the base for each output file name. The base name is also used in the function names of the output source code. When doing multiple *GNOSIS* runs, use a unique base name for each to avoid overwriting the output files.

TRAINING WITH A GENERAL STRUCTURE (USING A GSF)

The General Structure File (GSF) provides a quick and easy means for obtaining a simple neural network for a given database. Suppose you want to implement a neural network having three inputs (α, β, γ) and two outputs (A, B), as depicted in Figure 4.1. Using the GSF, *GNOSIS* sets up a network with three layers. The input (normalization) layer has one node per input; the hidden and output (unitization) layers each have one node per output. Time delays and feedbacks between the layers may be specified in the GSF. Normalization and unitization layers are explained later in this section; a hidden layer is any layer between the normalization and unitization layers. Note that the network in Figure 4.1 has a fully connected hidden layer; i.e. each node receives all of the outputs from the previous layer.

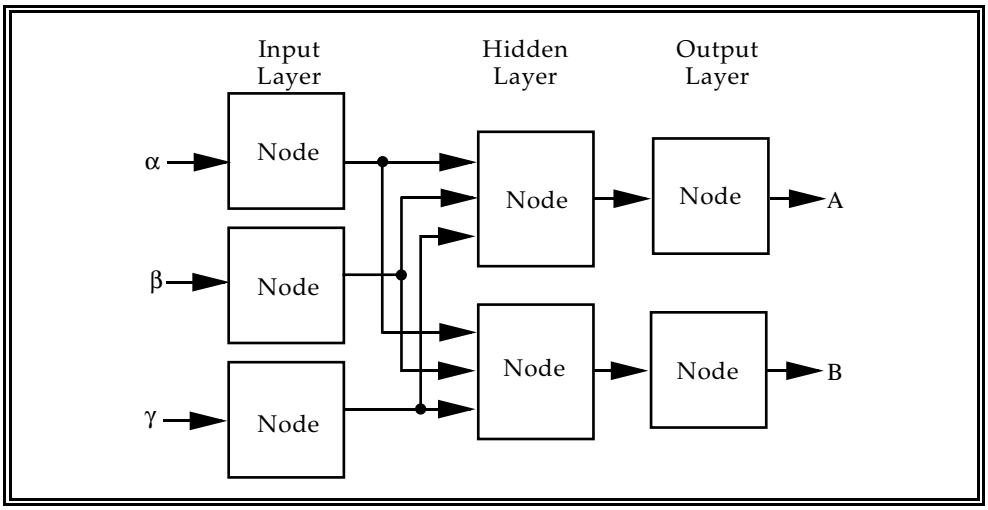


Figure 4.1: Network Structure for *GNOSIS* GSF

To run *GNOSIS* with a GSF, enter the following command:

gnosis -d datafile -t generalstructurefile -b basename

GNOSIS will read in the options from the GSF and create a simple network to meet the selected options. A sample GSF appears in Figure 4.2; "general" must appear in line 1. The GSF follows a particular format intended to provide the easiest use. Each line of information contains three parts - modifier, arguments, and comments - such that the user need only edit the arguments. The comments explain what each option means and what the argument options are. *GNOSIS* searches for the modifier and uses whatever follows as its arguments.

```
General Structure File for GNOSIS.
Copyright 1998 by Barron Associates, Inc. All Rights Reserved.

-------------------------------------------------------------------------------------
MODIFIER                      ARGUMENTS    |COMMENTS
-------------------------------------------------------------------------------------
Inputs                        USE_VSTRING  |USE_VSTRING, or enter input variables by name,
                                           |ending with'|'
Outputs                       USE_VSTRING  |USE_VSTRING, or enter output variables by name,
                                           |ending with'|'
Vstring                       yyxx-x       |Specify inputs/outputs: y, x, -, integer
                                           |repeat count (only used if USE_VSTRING
                                           |option entered for Inputs or Outputs)
Weights                       weight       |Optional observation weight variable
Distortion_Function           QUADRATIC    |Indicate distortion (loss) function for
                                           |GNOSIS to use (QUADRATIC/LOGISTIC)
Normalize                     NO           |Normalize inputs (YES/NO)
Unitize                       NO           |Normalize/Unitize outputs (YES/NO)
                                           |(only used if Distortion is QUADRATIC)
Init_Network_With_Database    NO           |Use database data to fill shift registers
                                           |with "true" data before optimizing (YES/NO)
-------------------------------------------------------------------------------------
Input_Delays                  CUSTOM       |Format for input delays (LINEAR/LOG/CUSTOM)
                                           |LINEAR: delay[i] = previous_delay + Space_Delay
                                           |LOG: delay[i] = Start_Delay + 2^i - 1
                                           |CUSTOM: enter delays in In_Custom_Delays
Start_Delay                   0            |Smallest delay used in each node >= 0
Space_Delay                   3            |Spacing between delays >= 1
Max_Delay                     15           |Maximum delay for each node > Start_Delay
Custom_Delays                 0            |Delay values >= 0 separated by spaces or tabs
-------------------------------------------------------------------------------------
Use_Prior_Data_Base_Outputs   NO           |Use past values of DATABASE output columns
                                           |as network inputs (YES/NO)
Use_Prior_Node_Outputs        NO           |Use past values of NODE output(s) as inputs
                                           |to node - i.e. feedback (YES/NO)
Output_Delays                 CUSTOM       |Format for output delays (LINEAR/LOG/CUSTOM)
                                           |LINEAR: delay[i] = previous_delay + spacing
                                           |LOG: delay[i] = start + 2^i - 1
                                           |CUSTOM: enter delays in In_Custom_Delays
Start_Delay                   1            |Smallest delay used in each node >= 1
Space_Delay                   3            |Spacing between delays >= 1
Max_Delay                     16           |Maximum delay for each node > Start_Delay
Custom_Delays                 1            |Delay values >= 1 separated by spaces or tabs
-------------------------------------------------------------------------------------
Limit_Nodes                   ALL          |Set which nodes to limit.  Nodes from input
                                           |and output layers limited with database
                                           |training values.  (IO/ALL/NONE)
Limit_Range                   1.0          |Percentage of full range (0.01 - 1.0) that
                                           |node outputs  will be limited to. (only used
                                           |if Limit_Nodes is ALL)
-------------------------------------------------------------------------------------
Node_Type                     COMP         |Specify polynomial structure type
                                           |(ADD/MULT/COMP/CUSTOM) ADD = additive,
                                           |MULT = multilinear, COMP = complete
Node_Degree                   2            |Maximum allowable degree of polynomial term
Node_Bias                     YES          |Include bias term in polynomial (YES/NO)
Custom_K_Matrix_File          ?.kmx        |File name of custom K matrix
Post_Trans                    LIN          |Choose post transformation (LIN/SIN/COS/SIG)
-------------------------------------------------------------------------------------
Write_Pix_File                YES          |Write node connections diagram and node
                                           |equations to .pix file
Write_Source_Code             YES          |Write source code to .c and .h files
-------------------------------------------------------------------------------------
```

Figure 4.2: Example General Structure File

MODIFIER DESCRIPTIONS

Each of the fields in the General Structure File are explained in detail below, in the order they appear in the GSF. Most of these fields are application independent. This means you can run *GNOSIS* with any setting (provided the setting is logically sound, i.e., a boolean variable gets set to YES or NO and not to 5.78) regardless of the data file. However, the first three GSF fields--Inputs, Outputs, and Vstring--are application specific. If they are not set correctly for each data file used, program execution will terminate.

Inputs and **Outputs [USE_VSTRING** or strings**] |**

The inputs and the outputs can be specified to *GNOSIS* one of two ways. One way is to type in the names of the variables as they appear in the database; see Figure 4.3. The other way is the Vstring option, selected by entering `USE_VSTRING` as the argument for Inputs or Outputs. In either case, a '|' must appear after the arguments so *GNOSIS* stops reading arguments. Inputs and Outputs are processed by the same method; if `USE_VSTRING` is specified for either, the Vstring method is used for both.

Vstring [x, y, - codes**]**

If either Inputs or Outputs have `USE_VSTRING` arguments, then *GNOSIS* reads the `Vstring` modifier and its arguments. All variables of the database are assigned to be inputs, outputs, or other with 'x', 'y', or '-' respectively. For example, the `Vstring , -xxxxyy`, is appropriate for a database header such as

```
unwanted_data   Input_1   Input_2   Input_3   Input_4   Output_1   Output_2
```

A number preceding an 'x', 'y', or '-' tells *GNOSIS* to repeat the character that immediately follows, so the data above could also be represented using the `Vstring, -4xyy.` To get a better idea of how the `Inputs`, `Outputs`, and `Vstring` modifiers are assigned, see Figure 4.3 below.

```
Database  Header
A       B       Alpha    Beta     Class    Gamma

            Use  Vstring
Inputs      USE_VSTRING           |
Outputs     USE_VSTRING           |
Vstring     yyxx-x                |Outputs A, B; Inputs Alpha, Beta, Gamma; ignore Class

            Enter  Inputs  and  Outputs  by  Name
Inputs      Alpha Beta Gamma      |
Outputs     A B                   |
Vstring     don't care            |ignore Class

            Single-Output  Classification  Data
Inputs      USE_VSTRING           |
Outputs     USE_VSTRING           |
Vstring     --xxyx                |Outputs are classes listed under Class
```

Figure 4.3 : Examples of I/O Indication

If a single variable is listed after Outputs or one 'y' is encoded in Vstring, and the specified database column contains class strings, Outputs is expanded to the set of classes found in the database. See Chapter 3 for the single-output database format.

Weights [string**]**

The weights entry is optional; if omitted, all database observations are weighted equally. To change the weighting, enter the variable name of a column giving relative weights to all observations; weights must be zero or positive. Weights affect the gradient calculations in both network parameter optimization and network structure learning, for both quadratic and logistic-loss functions, and affect the corresponding reported scores.

Distortion_Function [QUADRATIC/LOGISTIC]

Select `QUADRATIC` to use the squared-error loss function for estimation problems. Select `LOGISTIC` to use the logistic-loss function for classification problems.

Normalize and Unitize [YES/NO]

If the database input and/or output variable values vary greatly in their orders of magnitude, it may be difficult to obtain high accuracy in parameter optimization. Select `Normalize` and/or `Unitize` to transform the data so that optimization occurs on data values of similar magnitude. *GNOSIS* provides input and output layers with the sole purpose of normalizing or unitizing the data values. If normalization or unitization are not selected, these nodes "pass" the input values unchanged to their outputs.

The parameters of the normalizing and unitizing layers are computed, not optimized as are the rest of the network parameters. The normalizing and unitizing nodes are linear polynomial nodes of the form

$$y(t) = P0 + P1\, x(t)$$

where P0 and P1 are given by:

Normalization:	P0 = – mean/sigma	P1 = 1.0/sigma
Pass Through:	P0 = 0.0	P1 = 1.0
Unitization:	P0 = mean	P1 = sigma

and y(t) is the node output, x(t) is the node input (which could be database inputs or outputs), "mean" is the mean of the training database values of x(t), and "sigma" is the standard deviation of the training database values of x(t).

Enabling `Normalize` causes *GNOSIS* to normalize input values of the neural network. Enabling `Unitize` causes *GNOSIS* to train the network on normalized output data and then appropriately unitize the outputs of the neural network to obtain the original units of the data. `Unitize` is not available for `LOGISTIC` distortion function; pass through parameters are always used for the output layer.

Init_Network_With_Database [YES/NO]

This option only affects networks with input and/or output time delays. Enable this flag to fill the input layer nodes with database inputs, and fill output layer nodes with database outputs before network processing. All other nodes are initialized with zeroes. To initialize all nodes with zeroes, select `NO`. If `YES` is selected, the number of observations used for initialization vs. evaluation is reported in the statistics file.

Input_Delays [LINEAR/LOG/CUSTOM]

Input delays can be set in one of three ways: `CUSTOM`, `LINEAR`, and `LOG`. See Figure 4.4 for examples of each option. For static feedforward networks, there are no delays at the inputs. The nodes should have one delay with a value of zero; select `CUSTOM` and enter the zero delay in `Custom_Delays`.

Select `LINEAR` to set the input delays linearly such that

delay[i] = `previous_delay` + spacing

where the starting value, the spacing value, and the maximum delay value are specified with `Start_Delay`, `Space_Delay`, and `Max_Delay` respectively.

Select `LOG` to set the input delays logarithmically such that

$$delay[i] = start + 2^{i} - 1$$

where the starting value and the maximum delay value are specified with `Start_Delay` and `Max_Delay`.

```
Static Network, with Input Delay of 0 Using CUSTOM Format
Input_Delays              CUSTOM     |
Start_Delay               don't care |
Space_Delay               don't care |
Max_Delay                 don't care |
Custom_Delays             0          |

Input Delays of 0, 3, 6, and 9 Using LINEAR Format
Input_Delays              LINEAR     |
Start_Delay               0          |
Space_Delay               3          |
Max_Delay                 9          |
Custom_Delays             don't care |

Input Delays of 1, 2, 4, and 8 Using LOG Format
Input_Delays              LOG        |
Start_Delay               1          |
Space_Delay               don't care |
Max_Delay                 8          |
Custom_Delays             don't care |
```

Figure 4.4: Examples of Input Delay Specification

`Start_Delay [integer]`
If `LINEAR` or `LOG` is selected for `Input_Delays` or `Output_Delays`, then `Start_Delay` gives the first delay value. The first delay value must be zero or more for `Input_Delays`, and one or more for `Output_Delays`, described below.

`Space_Delay [integer >= 1]`
If `LINEAR` is selected for `Input_Delays` or `Output_Delays`, then `Space_Delay` gives the spacing value between sequential delays.

`Max_Delay [integer]`
If `LINEAR` or `LOG` is selected for `Input_Delays` or `Output_Delays`, then `Max_Delay` gives the maximum delay value. The last delay value in a linear or log sequence will be less than or equal to `Max_Delay`. The maximum delay must be more than `Start_Delay`.

`Custom_Delays [integers] |`
If `CUSTOM` is selected for `Input_Delays` or `Output_Delays`, then `Custom_Delays` gives the delay values. Enter the desired delay values, separating each value with spaces or tabs and ending with a '|'. Delays must be zero or more for `Input_Delays`, and one or more for `Output_Delays`.

Use_Prior_Data_Base_Outputs [YES/NO]

Feedback tends to be difficult for neural network synthesis; enabling this flag can assist *GNOSIS* in achieving optimal network performance that may not have been realized by simply feeding back network outputs. Select `Use_Prior_Data_Base_Output` to use prior (delayed) values of the true network output as candidate inputs to the network. The number of samples to delay the true network parameters before input to the network are specified in the `Output_Delays` section.

Once *GNOSIS* has determined the optimal neural network transformations, edit the output model file (Network Description File) so that fed-back network estimates of node outputs are used rather than prior (delayed) values of the true parameters. By editing the Network Description File (NDF) in this manner, *GNOSIS* generally has a better chance of obtaining optimal network transformations, because *GNOSIS* begins the network synthesis using already-optimized parameters. Disable the `Unitize` option prior to network synthesis, to ensure that hidden layer outputs are equivalent (if being fed back) to using delayed database output values as inputs.

Use_Prior_Node_Outputs [YES/NO]

Enable this flag to use past values of hidden layer node outputs as inputs to themselves (self-feedback). The number of samples to delay the node outputs before input to the network is specified in the `Output_Delays` section.

Output_Delays [LINEAR/LOG/CUSTOM]

Output delays actually refer to feedback delays. Therefore, the `Output_Delays` option and associated fields are used only if `Use_Prior_Data_Base_Outputs` is `YES` or `Use_Prior_Node_Outputs` is `YES`. All output delay values must be at least one. Otherwise, output delays are specified in the same way as input delays.

Limit_Nodes [IO/ALL/NONE]

Select which nodes of the network will limit their outputs. When training a network, *GNOSIS* stores the output range for each node. For input and output nodes, *GNOSIS* computes the maximum and minimum values from the database. For hidden nodes, *GNOSIS* computes the range of the node output. During evaluation, these minimum and maximum values may be used to limit nodal outputs. Select `IO` to limit only the node outputs of the input and output layers of the network. Select `ALL` to limit the outputs of all nodes in the network. Select `NONE` to do no limiting. For `LOGISTIC` distortion function, the output layer nodes are not limited in any case. For sigmoid post transformation, hidden layer nodes are not limited, since the transformation limits implicitly.

Limit_Range [`decimal`**]**

If `Limit_Nodes` is `ALL` and `Post_Trans` is not `SIG`, then `Limit_Range` gives the percentage of the full range of values that the hidden layer nodal outputs will be limited to in evaluations and generated source code usage. Valid values are 0.01 through 1.0, but values at or near 1.0 are recommended.

Node_Type [ADD/MULT/COMP/CUSTOM]

Select the polynomial type of all hidden layer nodes. ADD specifies additive polynomials for the structure of the network elements. Additive polynomials have no cross terms. The degree of the polynomial (specified with Node_Degree) indicates the highest power to which each element input variable can be raised. For example, if Node_Degree is 3, the most complex element structure that *GNOSIS* attempts for a two input element is $\theta_0 + \theta_1x_1 + \theta_2x_1^2 + \theta_3x_1^3 + \theta_4x_2 + \theta_5x_2^2 + \theta_6x_2^3$.

MULT specifies multilinear polynomials for the structure of the network elements. The degree of the polynomial indicates the maximum number of input variables that may be included in a cross-product term. No input is raised to a power greater than one. For example, if Node_Degree is 3, the most complex element structure that *GNOSIS* attempts for a three input element is $\theta_0 + \theta_1x_1 + \theta_2x_2 + \theta_3x_3 + \theta_4x_1x_2 + \theta_5x_1x_3 + \theta_6x_2x_3 + \theta_7x_1x_2x_3$.

COMP specifies complete polynomials for the network element structures. In this case, Node_Degree specifies the highest degree to which node polynomial inputs will be exponentiated and cross-multiplied. For example, when *GNOSIS* tries elements with two inputs and Node_Degree is 3, the most complex structure attempted is $\theta_0 + \theta_1x_1 + \theta_2x_1^2 + \theta_3x_1^3 + \theta_4x_2 + \theta_5x_2x_1 + \theta_6x_2x_1^2 + \theta_7x_2^2 + \theta_8x_2^2x_1 + \theta_9x_2^3$.

Experience shows that in many applications just the cross terms, x_ix_j, represent nonlinearities very effectively. Eliminating all "raising to a power" terms from a complete polynomial structure reduces complexity of the element, yet retains its power to introduce nonlinearities when such are needed to produce the desired output.

CUSTOM specifies that polynomial powers are given in a custom K matrix; see Custom_K_Matrix_File below.

Node_Degree [integer >= 1]

Enter the highest degree of the hidden layer polynomial elements *GNOSIS* should try during network synthesis.

Node_Bias [YES/NO]

Enable this flag to include a bias term (constant) in the hidden layer polynomial elements.

Custom_K_Matrix_File [name.kmx]

If CUSTOM is selected for Node_Type, then *GNOSIS* reads the $\underline{\underline{K}}$ matrix for the hidden layer polynomials from a file. Read the information on the $\underline{\underline{K}}$ matrix in the "Training with the Structure Specified (Using an NDF)" section that appears later in this chapter (page 4-11).

To specify a $\underline{\underline{K}}$ matrix, first determine how many core inputs each hidden layer element has. Each row in the $\underline{\underline{K}}$ matrix file must have as many numbers as there are core inputs for the network. The number of core inputs is the number of input variables times the number of input delays plus (if using database outputs as inputs) the number of output variables times the number of output delays plus (if using node outputs as inputs) the

number of output delays. Each row represents a term of the polynomial, and the user may have as many terms as desired. Each line in the file should contain integers referring to the power to which the corresponding core inputs are raised, separated by spaces or tabs.

To find the core input to which each $\underline{\underline{K}}$ matrix column refers, first run *GNOSIS* with another `Node_Type` option so that an output network description file (the `*.mdl` file) can be studied. See the "Training with the Structure Specified (Using an NDF)" section for details on the NDF $\underline{\underline{K}}$ matrix format.

For a network with two inputs, one output, two input delays, and three output delays, Figure 4.5 represents a valid custom $\underline{\underline{K}}$ matrix file corresponding to the expression:

$\theta_0\, x_1(t)\, x_1(t-1)\, y^2(t-1) + \theta_1\, x_2^2(t)\, x_2^2(t-1)\, y(t-3)$
$+ \theta_2\, y(t-3) + \theta_3\, x_1(t)\, x_1(t-1)\, x_2(t-1)\, y(t-2)$

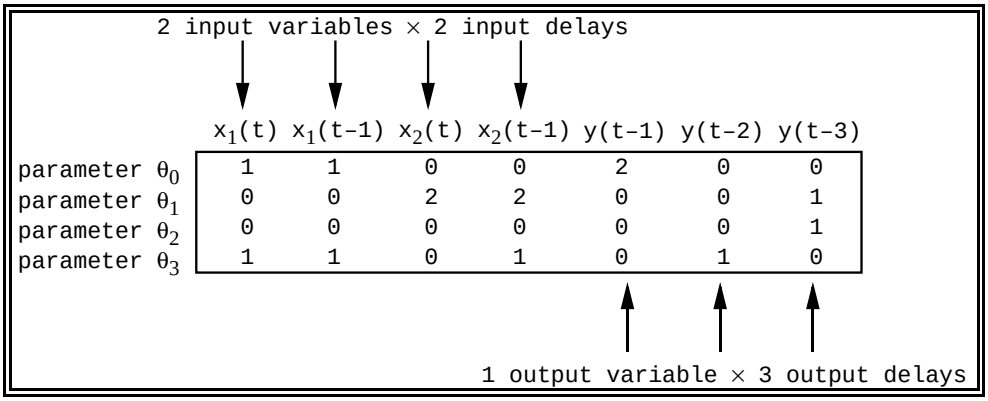


Figure 4.5: Example Custom $\underline{\underline{K}}$ Matrix File

`Post_Trans [LIN/SIN/COS/SIG]`

Select the post transformation for all hidden layer nodes. The post transformation, h(z), is applied to the output of the polynomial core transformation to produce the output of the node. The available transformations are:

`LIN` $h(z) = z$
`SIN` $h(z) = \sin(z)$
`COS` $h(z) = \cos(z)$
`SIG` $h(z) = \dfrac{1}{1+e^{-\beta z}}$ β = sigmoid gain, default 1.0

`Write_Pix_File [YES/NO]`

Enable this flag to write a node connections diagram and node equations to a `*.pix` file, where `'*'` is the base of the database filename.

`Write_Source_Code [YES/NO]`

Enable this flag to write C-language source code to `*.c` and `*.h` files, where `'*'` is the base of the database filename. The source code and bai_src library can be used in C-language programs to evaluate the network modeled by *GNOSIS*.

TRAINING WITH THE STRUCTURE SPECIFIED (USING AN NDF)

The Network Description File (NDF) serves two purposes:

1. It allows the user to specify a network structure and initial parameter values, and indicates which parameters are to be optimized (i.e., acts as an input file).
2. It communicates to the user the found optimum coefficient values (i.e., acts as an output file).

Consider a neural network with three inputs (α, β, γ) and two outputs (A, B) that matches the following equations:

$$A(t) = \theta_0\,\alpha(t) + \theta_1\,\alpha(t-1) + \theta_2\,\beta(t-1) + \theta_3\,\beta(t-3)$$
$$B(t) = \theta_4\,\beta(t) + \theta_5\,\beta(t-1) + \theta_6\,\gamma(t) + \theta_7\,\gamma(t-1)$$

This neural network structure would have two hidden nodes and could be diagrammed as in Figure 4.6. The corresponding NDF is shown in Figure 4.7.

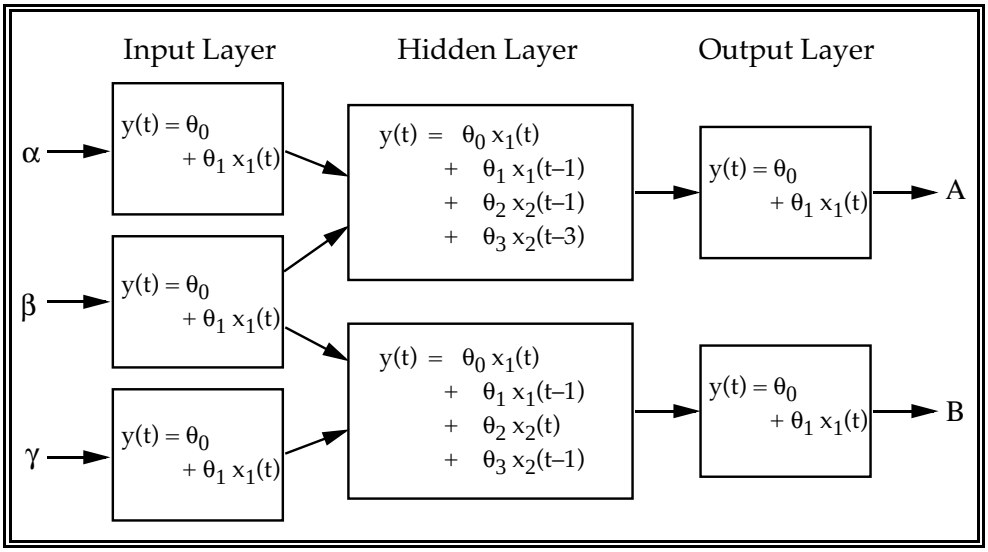


Figure 4.6: Network Structure for *GNOSIS* NDF

Note that in Figure 4.6, the variable, x, is used generically to represent the input to a node; thus, x_1 in one node is not the same variable as x_1 in another node. Also note that unlike the GSF-specified structure shown in Figure 4.1, the NDF-specified structure is not fully connected and only calls out certain specific terms in the polynomial nodes.

```
GNOSIS NETWORK DESCRIPTION
              Inputs: Alpha Beta Gamma  |
              Outputs: A B  |
              Class_Single_Output: NO
              Weights: weight
              Number_of_Layers: 3
              Distortion_Function: QUADRATIC
              Reg_Penalty: 0.0001
              Normalize: NO
              Unitize: NO
              Limit_Nodes:  NONE
              Init_Network_With_Database: NO

NETWORK STRUCTURE
---------------------------------------

INPUT_LAYER:
              NODE_0-0 DESCRIPTION:
                   Parameters:       0    1
                   Output_Range:     0    0
              NODE_0-1 DESCRIPTION:
                   Parameters:       0    1
                   Output_Range:     0    0
              NODE_0-2 DESCRIPTION:
                   Parameters:       0    1
                   Output_Range:     0    0
---------------------------------------

LAYER 1:
              Number_of_Nodes:      2
              Core_Transformation:  POLY
              Post_Transformation:  SIG
              Sigmoid_Gain:         1.0
              NODE_1-0 DESCRIPTION:
                   Input_Node(s):    2
                        NODE 0 - 0
                             Delays:  0    1
                        NODE 0 - 1
                             Delays:  1    3
                   Parameters:
                        0.1   -0.1  0.1   -0.1
                   Set_of_Indices:
                        1  0  0  0
                        0  1  0  0
                        0  0  1  0
                        0  0  0  1
                   Optimize:         YES
                   Output_Range:     0    0
              NODE_1-1 DESCRIPTION:
                   Input_Node(s):    2
                        NODE 0 - 1
                             Delays:  0    1
                        NODE 0 - 2
                             Delays:  0    1
                   Parameters:
                        0.1   -0.1  0.1   -0.1
                   Set_of_Indices:
                        1  0  0  0
                        0  1  0  0
                        0  0  1  0
                        0  0  0  1
                   Optimize:         YES
                   Output_Range:     0    0
---------------------------------------

OUTPUT_LAYER:
              NODE_2-0 DESCRIPTION:
                   Input_Node:  1 - 0
                   Parameters:       0    1
                   Output_Range:     0    0
              NODE_2-1 DESCRIPTION:
                   Input_Node: 1 - 1
                   Parameters:       0    1
                   Output_Range:     0    0
---------------------------------------
```

Figure 4.7: Example Network Description File (NDF)

Normalization and Unitization Layers

See the `Normalize` and `Unitize` options in Chapter 4 for an explanation of normalization and unitization layers. The zeroth layer of the network contains one "normalizing" node per network input. If normalization is not desired, this layer is configured to simply "pass through"

the input data. The last layer contains one "unitizing" node per network output. This layer has the same number of nodes as the second-to-last layer.

Node Identification Convention

In the NDF, nodes are labeled as `NODE L-N`, where `L` is the layer number and `N` is the node number on the given layer. The layer numbers start at zero at the input/normalization layer and increment by one at each succeeding layer. The node numbers within a layer start at zero and increment sequentially as well.

Specification of Delay Values

For hidden layer nodes, the NDF specifies the source of the inputs and the delays associated with each input, just as they would appear in a difference equation. For example, if the difference equation of a node were

$$y(t) = x(t) + x(t-1) + y(t-1) + y(t-2)$$

the NDF input delay specification for the node would be:

```
NODE_1-0 DESCRIPTION:
      Input Node(s): 2
            NODE 0 - 0
                  Delays: 0 1
            NODE 1 - 0
                  Delays: 1 2
```

Every hidden layer node must have at least one input node, but an input node can come from any normalization or hidden layer node. Delay values of input nodes from the same layer or a later layer must be one or more. The delay values may be listed in any order.

Core and Post Transformations

Each node in the network performs a core transformation function on the inputs to produce a single output. Then a post transformation function operates on the output of the core transformation to produce the output of the node. Currently *GNOSIS* uses a polynomial core transformation for all nodes, and a linear post transformation for all input and output layer nodes. The post transformation for each hidden layer is selectable and may vary from layer to layer. If linear, sine, or cosine transformations are selected, hard limits may be applied with the `Limit_Nodes` and `Limit_Range` options. If the sigmoid transformation is selected, hard limits are not applied due to the clipping nature of the function. An initial sigmoid gain of 1.0 is recommended; increasing the gain makes the output more closely approximate a hard limit.

Set of Indices (or $\underline{\underline{K}}$ matrix)

The inputs to the core transformation function are referred to as core inputs. Each input to a node provides as many core inputs as the number of delays specified for that node input. So for a node of the form

$$y(t) = \theta_0 + \theta_1 x_1(t-1) + \theta_2 x_1^3(t) + \theta_3 x_1(t)\, x_2(t-3) + \theta_4 x_2^2(t-1)$$

there are four core inputs even though the node only has two external inputs. The core inputs are $x_1(t–1)$, $x_1(t)$, $x_2(t–3)$, and $x_2(t–1)$.

To relate the core transformation function to *GNOSIS*, the user must specify a $\underline{\underline{K}}$ matrix. Rows of the $\underline{\underline{K}}$ matrix represent the parameters of the polynomial terms, the columns stand for the core inputs, and the individual matrix elements specify the power to which the corresponding input is raised. The θ_0 bias term is handled by a row of zeroes in the $\underline{\underline{K}}$ matrix.

Thus the $\underline{\underline{K}}$ matrix corresponding to

$$y(t) = \theta_0 + \theta_1 x_1(t–1) + \theta_2 x_1^3(t) + \theta_3 x_1(t)\, x_2(t–3) + \theta_4 x_2^2(t–1)$$

with the columns referring to $x_1(t–1)$, $x_1(t)$, $x_2(t–3)$, and $x_2(t–1)$, respectively, is

$$\underline{\underline{K}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

For each node in the hidden layers (all layers except the normalization and unitization layers), specify the $\underline{\underline{K}}$ matrix in the NDF by entering the $\underline{\underline{K}}$ matrix numbers starting on the line after `Set_of_Indices`:

```
Set_of_Indices:
      0 0 0 0
      1 0 0 0
      0 3 0 0
      0 1 1 0
      0 0 0 2
```

NETWORK DESCRIPTION FILE SYNTAX

Incorrectly specified NDFs usually bring about error messages, but syntax errors may go undetected and result in incorrect network descriptions. To ensure that the NDF is correctly developed, start with an existing NDF generated by *GNOSIS* using either a GSF or optional SLF. Output NDF's (`*.mdl` files) are generated by every run of *GNOSIS*, and any output NDF can also be used as an input NDF. By editing an existing NDF, one can change the network-specific parts of the file and prevent formatting errors.

The network must be described completely and in the correct order:

1. *GNOSIS* network description
2. Input layer
3. Hidden layers (in order)
4. Output layer

All NDF specifications must be separated by spaces or tabs, and may be followed by comments starting with '|'. Bold text below has been written as it should appear in the NDF; bracketed

text refers to user-specified text. N refers to an integer; decimal is any value with or without a decimal point. End the last line of text with a carriage return, or it may not be read by *GNOSIS*.

Network Description Section

`Inputs: [strings] |`
Specify the names of external inputs into the entire network, matching labels in the database file. There must be the same number of nodes in the input layer as strings. Input layer `Node_0-0` applies to the first string, `Node_0-1` to the second string, etc.

`Outputs: [strings] |`
Specify the names of outputs of the entire network, matching labels in the database file. There must be the same number of nodes in the last hidden layer, and nodes in the output layer as strings. For a single-output classification problem, the strings should be the set of class strings found as outputs in the database; see Chapter 3.

`Class_Single_Output: [string or NO]`
If `Outputs` is a list of class strings from a single-output format database, enter the name of the output column in the database header. Otherwise, select `NO` .

`Weights: [string]`
This entry is optional; use it to specify the name of an observation weighting column in the database file.

`Number_of_Layers: [integer >= 3]`
Specify the number of layers in the entire network. There must be at least three to account for one input normalization layer, one or more hidden layers, and one output unitization layer.

`Distortion_Function: [QUADRATIC/LOGISTIC]`
Select `QUADRATIC` for the squared-error loss function used for estimation problems. Select `LOGISTIC` for the logistic-loss function used for classification.

`Reg_Penalty: [decimal]`
Specify the penalty on the magnitude of network parameters. A larger penalty sacrifices network performance to keep the parameters smaller, which results in a network with smoother output and better able to generalize on unseen data.

`Normalize: [YES/NO]`
Select `YES` to normalize the inputs; input layer `Parameters` will be -mean/sigma and 1.0/sigma of the database inputs. Select `NO` to pass the inputs through to the hidden layers; input layer `Parameters` will be 0.0 and 1.0.

`Unitize: [YES/NO]`
If `Distortion_Function` is `QUADRATIC`, select `YES` to unitize the outputs; output layer `Parameters` will be mean and sigma of the database outputs. Select `NO` to pass the outputs of the last hidden layer through to the network output; output

layer `Parameters` will be 0.0 and 1.0. If `Distortion_Function` is `LOGISTIC`, unitization is not available and pass-through parameters are used.

`Limit_Nodes: [IO/ALL/NONE]`
`Limit_Range: [``decimal`**`]`**
`Init_Network_With_Database: [YES/NO]`
See the description of these fields in the GSF section above.

Input Layer Section

`INPUT_LAYER`:
Title for layer 0. Input layer nodes must follow in numerical order, one for each Inputs string. Each node has the following three fields:

`NODE_0-N DESCRIPTION:`
Title for node N, numbering sequentially from 0. The input to this node is the Nth database input listed in Inputs.

`Parameters: [``decimal decimal`**`]`**
Ignore for a training NDF. For an output NDF model file, these are the normalization or pass-through parameters.

`Output_Range: [``decimal decimal`**`]`**
Ignore for a training NDF. For an output NDF model file, these are the minimum and maximum values of the database inputs.

Hidden Layer Section(s)

`LAYER N:`
Title for layer N, numbering sequentially from 1.

`Number_of_Nodes: [``integer >= 1`**`]`**
Enter the number of nodes in this layer. A node description for each must follow.

`Core_Transformation: POLY`
The core transformation applied to all nodes in this layer. Polynomial is the only transformation available at this time.

`Post_Transformation: [LIN/SIN/COS/SIG]`
The post transformation applied to all nodes in this layer. The transformations available are linear, sine, cosine, and sigmoid:

`LIN` $h(z) = z$
`SIN` $h(z) = \sin(z)$
`COS` $h(z) = \cos(z)$
`SIG` $h(z) = \frac{1}{1+e^{-\beta z}}$ β = sigmoid gain, default 1.0

`Sigmoid_Gain: [``0.1 to 10.0`**`]`**
If the post transformation is `SIG`, enter a gain for β.

NODE_L-N DESCRIPTION:
Title for node N of layer L, numbering N sequentially from 0.

Input_Node(s): [integer >= 1]
Enter the number of nodes that are providing inputs into this node. Each input node has the following two fields:

NODE L - N
Enter the layer number L and node number N of the source node for this input. Inputs can come from any node of any hidden layer, including self-feedback.

Delays: [integers >= 0]
Enter one or more delays for this input. If an input variable, x, only occurs in difference equations as x(t), then the delay is 0. The delays may appear in any order, but the `Set_of_Indices` ($\underline{\underline{K}}$ matrix) must be specified according to the delay order. Feedback inputs must have delays of 1 or more.

Parameters: [decimals]
Specify the initial values for the parameters of this node. If any parameter is 0.0, *GNOSIS* randomly perturbs it (standard deviation of 0.001) to obtain initial values. Try different initial values if training results are poor. The output NDF model file shows the optimized parameters here.

Set_of_Indices: [one line integers per parameter]
Enter the $\underline{\underline{K}}$ matrix of powers. The number of rows must equal the number of parameters entered. A row of zeros signifies a bias term. The number of columns must match the sum of all the delays of the node inputs. The first column corresponds to the first delay of the first input, the second column corresponds to the second delay of the first input if it exists, otherwise to the first delay of the second input, and so on.

Optimize: [YES/NO]
Select `YES` to optimize all parameters for this node. Select `NO` to leave all parameters at their initial values.

Output_Range: [decimal decimal]
Ignore for a training NDF. For an output NDF model file, these are the minimum and maximum values of the node outputs during training.

Output Layer Section

OUTPUT_LAYER:
Title for the last layer. Output layer nodes must follow in numerical order, one for each `Outputs` string. Each node has the following three fields:

NODE_L-N DESCRIPTION:
Title for node N of output layer L, numbering N sequentially from 0. Layer number L is one more than the last hidden layer. The input to this node is node N of layer L-1.

Parameters: [decimal decimal**]**
Ignore for a training NDF. For an output NDF model file, these are the unitization or pass-through parameters.

Output_Range: [decimal decimal**]**
Ignore for a training NDF. For an output NDF model file, these are the minimum and maximum values of the database outputs.

TIPS FOR IMPROVING MODEL PERFORMANCE

Once a network has been created and incorporated into a NDF, more can be done to help *GNOSIS* generate the best possible model.

When executing in the training mode, *GNOSIS* starts the parameters at the initial values in the NDF, slightly perturbed if zero. If you have educated guesses about the values of some parameters, try setting them as initial parameter values. Even if you only know that a parameter should be negative, start it at some negative number. By trying specific starting values for the parameters, you are "pushing" *GNOSIS* in a certain initial direction. Starting the parameters near their optimal values can make the difference between finding the optimal parameters and getting trapped in a multi-modal search space. If you are getting poor results and have no idea of parameter values, you could try a sequence of synthesis runs starting parameters at different random values.

You may try multiple network structures or parameter initializations. For example, dynamic networks are networks with feedback; consider whether a dynamic network may be appropriate. In many applications, dynamic networks result in simpler implementations (i.e., fewer network parameters) than do static (feedforward) networks. This reduces the probability of overfitting and increases network accuracy and robustness. Another advantage of dynamic networks over static networks is that they have the ability to forecast arbitrarily far into the future (albeit with an accuracy that decreases somewhat with forecast horizon), rather than requiring that networks be trained for a specific, *a priori*-specified number of prediction time steps into the future.

The more you know or suspect about relationships between variables, the more suited to a particular problem you can make the network structure. Spend some time considering what makes sense as far as how inputs and outputs might combine or be fed back to arrive at a suitable output. You will probably benefit from making plots of the various input and output variables to find general relationships. If the plot of one input with one output appears to be a high-degree equation, you may want to introduce high-degree terms and cross terms in the appropriate set of indices ($\underline{\underline{K}}$ matrix).

In general, the training process will involve developing different NDFs to find the best model. You can change the number of delays for each input, the number of nodes and their connections in the network, the sets of indices ($\underline{\underline{K}}$ matrices), the initial parameter values, etc. After trying multiple structures, look at the results of each trial to see how the structures and parameters correspond to better and poorer results.

Chapter 6 describes use of the *ASPN-III* structure-learning option for *GNOSIS* feedforward estimation networks. Use of this option avoids being trapped in multi-modal search space, and *ASPN-III* automatically identifies a just-sufficient network complexity that minimizes model error without overfitting the training database.

WHEN TO USE THE GSF, SLF, AND NDF

The most important considerations in determining which run option to use are:

- Are the phenomena being modeled static or dynamic?
- How much *a priori* knowledge of the network structure is available?

If you are modeling a dynamic system with feedbacks and time delays or have a classification problem, then either the GSF or the NDF should be used. If you have little idea of the network structure and want a quick, simple solution, the GSF is the best choice. Otherwise, use the NDF, which is much more flexible and allows virtually any desired structure to be optimized and tested.

For estimation problems without feedbacks and time delays, structure learning via the *ASPN-III* option is available. Choosing an appropriate structure greatly increases the chances of obtaining a good model. Since one does not generally know *a priori* the structure needed for a neural network, it is helpful to use the *ASPN-III* structure-learning algorithm (Chapter 6) when there are a large number of potential inputs and structures.

EVALUATION

As part of the training process, *GNOSIS* evaluates a model on the training database. To evaluate a model on unseen data, however, *GNOSIS* must be run in evaluation mode:

gnosis -d datafile -e modelfile -b basename

where datafile is the evaluation database and modelfile is the `*.mdl` file (created during training) describing the model to be evaluated. When execution is complete, several output files are created by *GNOSIS*., named by adding extensions to the given basename These files are described in detail in Chapter 5.

In general, the evaluation process should be implemented for any model that looked promising after training. It is important to evaluate networks on previously unseen data to ensure that they score similarly on both seen and unseen data.

Once a satisfactory model has been produced, the model may also be evaluated by including the source code (created during training) in an application which reads database inputs and interrogates the network. An example application is given in the `*.h` file; see Chapter 5.

5

INTERPRETING GENERATED OUTPUT

GNOSIS generates output on the standard output display, and in files named using the basename given on the command line. *GNOSIS* overwrites any existing files with the same names, formed by appending to the basename:

`.mdl` for the model Network Description File, generated in the training mode.
`.sts` for the statistics file, generated in the training and evaluation modes.
`.est` for the estimation file, generated in the training and evaluation modes.

The following files are generated by default when training with an NDF, or when selected in a GSF or SLF:

`.pix` for the picture file
`.c` for the C-language source code file
`.h` for the C-language header file

This chapter is organized into three sections. The first section describes runtime messages printed to standard output. The second details files created only during training, while the third section describes files created in both training and evaluation.

MESSAGES TO STANDARD OUTPUT

When executing in the training or evaluation mode, *GNOSIS* writes information to the screen relaying what it is doing. Most of these runtime messages are self-explanatory. However, the ILS optimization messages require some background knowledge to understand.

Optimization Algorithm Messages

GNOSIS employs the Iterative Least Squares (ILS) search algorithm, a regularized Gauss-Newton optimization method that uses score-surface gradient and curvature (i.e., the pseudo-Hessian) information. If the score is not improving, the curvature information is weighted less and the search devolves into a pure gradient-descent algorithm (i.e., least mean squares). If the score is improving, however, the curvature information is weighted more heavily and the

search algorithm attempts to jump to the score-surface minimum in one iteration (i.e., Gauss-Newton).

The Levenberg-Marquardt method is used to vary smoothly between the extremes of the Gauss-Newton and gradient-descent algorithms; it achieves this by regularizing the pseudo-Hessian. If the score is not improving, *GNOSIS* introduces a regularization factor, lambda, to weight the gradient-descent information more. The current iteration number, lambda, and root of the best score to date are displayed; enter Control C to stop after the current iteration. Otherwise, *GNOSIS* continues iterating until it has satisfied one of the following stopping criteria:

 # ILS: Maximum number of iterations reached.
GNOSIS stops if 100 iterations have been tried, to prevent *GNOSIS* from running for an excessive time period. To extend the run beyond this maximum number of iterations, run *GNOSIS* again in training mode using the model file just created; *GNOSIS* then restarts with the parameters from the prior run.

 # ILS: Normalized score below tolerance.
GNOSIS stops if the normalized score is less than 1e–8, indicating very successful parameter optimization.

 # ILS: Normalized score change below tolerance.
GNOSIS stops if the normalized score is changing too slowly. This happens when the *GNOSIS* score has changed by less than 1e–10, three iterations in succession.

 # ILS: Maximum lambda exceeded.
Lambda refers to the Levenberg-Marquardt regularization factor used to vary smoothly between the extremes of the Gauss-Newton algorithm and the steepest-descent algorithm. If *GNOSIS* successively increases the gradient-descent weighting and the corresponding score ceases to improve, *GNOSIS* will stop. This message probably means that *GNOSIS* has reached a locally optimal point on the network response surface. The maximum lambda is 1e7.

 # ILS: Parameter change below tolerance.
GNOSIS stops if all the parameters have essentially converged. This message appears when the largest change in all the parameters is less than 1e–10 for three successive iterations.

Error Messages

If *GNOSIS* detects problems while processing NDF, GSF, SLF, and database files or while creating and optimizing the network, it terminates with an error message. Standard error messages are:

 # ERROR: Processing command line <option>
An option or argument on the command line is improperly specified. See Chaper 4 for command-line usage.

 # ERROR: Opening file <name>

The named file could not be found; execute *GNOSIS* from the same directory as the input files.

 # ERROR: Processing field <string >
 # ERROR: Processing field <string > for NODE L-N (NDF only)
 # ERROR: Processing field <string > for LAYER L (NDF only)
GNOSIS detected a problem with the named field in the NDF, GSF, or SLF. Either the field cannot be found, or in some cases is out of order, or the option specified for the field is improper. For example, entering a number for a YES/NO field, misspelling one of the valid choices for a text field, or entering a number out of range causes this error.

 # ERROR: Must have at least <N> and at most 256 Inputs and Outputs
Check the Inputs and Outputs specifications in the NDF, GSF, or SLF. A common cause for this error is omitting the terminating '|' from the Inputs or Outputs list. For estimation models, N is 1. For classification, N is 2; check that a single-output format database has at least two output values.

 # ERROR: Memory allocation
GNOSIS has insufficient system memory to continue allocating data. Free up or acquire more memory for your platform, or reduce the complexity of the network specified.

ERROR: Reading from database line <N>
The database is improperly formatted; see Chapter 3. Line 0 indicates a problem with the header line. Otherwise N gives the observation number of the first misformatted line.

 # ERROR: Indexing database for <name>
The database is improperly referenced. Check Inputs and Outputs in the NDF, GSF, or SLF for proper identification of database variables.

A handful of other special-case error messages may be reported by *GNOSIS;* instructions are self-evident or are given in the message. In all cases, # ERROR is a termination message; the problem must be resolved and *GNOSIS* rerun.

FILES CREATED DURING TRAINING

After training a network, optimizing the structure and/or parameters, *GNOSIS* creates a number of files. Since part of the training process involves evaluating the optimized network model on the training database, some of these output files are identical to those created when *GNOSIS* is run in evaluation mode. This section details those output files that are created only in training mode: the model file (`*.mdl`), picture file (`*.pix`), and source files (`*.c` and `*.h`).

Output Model File (`*.mdl` file)

The model file (*.mdl) has the same format as an input NDF, but provides the optimized parameter values and some additional information at the top of the file:

```
GNOSIS NETWORK DESCRIPTION
   Training_Database: feedback.traNorm_RMS_Error: y1 = 0.000183732
   y2 = 0.0133094
   — or —
   Percent_Misclassified: 0.0312568
```

For estimation models, the normalized root-mean-squared error for each output is the percent of standard deviation of true output not accounted for by the model output. The closer the errors are to zero, the more accurate the model. For classification models, the percent misclassified observations out of the total helps assess the accuracy and robustness of the model. See also the statistics file for more scoring information.

Output Picture File (*.pix file)

GNOSIS generates the picture file after training is complete. This feature can be disabled in the GSF or SLF; see Chapter 4. The picture file contains an ASCII diagram of the network structure and the node equations of the network. Input and output names are taken from the database and each node in the network is numbered. Inputs or nodes that are fed into multiple nodes are rewritten, so the network picture in the pix file of Figure 5.1 matches the network structure diagrammed in Figure 5.2

```
Network:          cannon.mdl
Trained on:       cannon.dat

 velocity-----Node1-+---Node3-+---Node6-----Node7---range
                    |         |
    gamma-----Node2-/         |
                              |
 velocity-----Node1-+---Node4-|
                    |         |
    gamma-----Node2-/         |
                              |
              Node3*-+---Node5-|
                     |        |
    gamma-----Node2-/         |
                              |
             velocit-----Node1-/

A '*' following a node indicates that the node's output is fed in here as an input.

Node1(t) = 1*velocity(t)
Node2(t) = 1*gamma(t)
Node3(t) = 235.85 - 0.108877*Node1(t) + 3.91077e-005*Node1^2(t) - 2.58303*Node2(t)
Node4(t) = -0.0161736*Node2^4(t) + 15413.2*Node1(t) - 0.00168485*Node1^2(t)*Node2^2(t)
           - 59325*Node2(t)
Node5(t) = 8.99247 + 6.7244e-006*Node3(t-1)*Node2(t)
Node6(t) = 16249.9 - 0.0611516*Node3(t)*Node5(t)*Node1(t)
           -0.00471466*Node3^2(t)*Node1(t) + 0.00172708*Node4(t)*Node5(t)
Node7(t) = 1*Node6(t)
```

Figure 5.1: Example Pix File (*.pix)

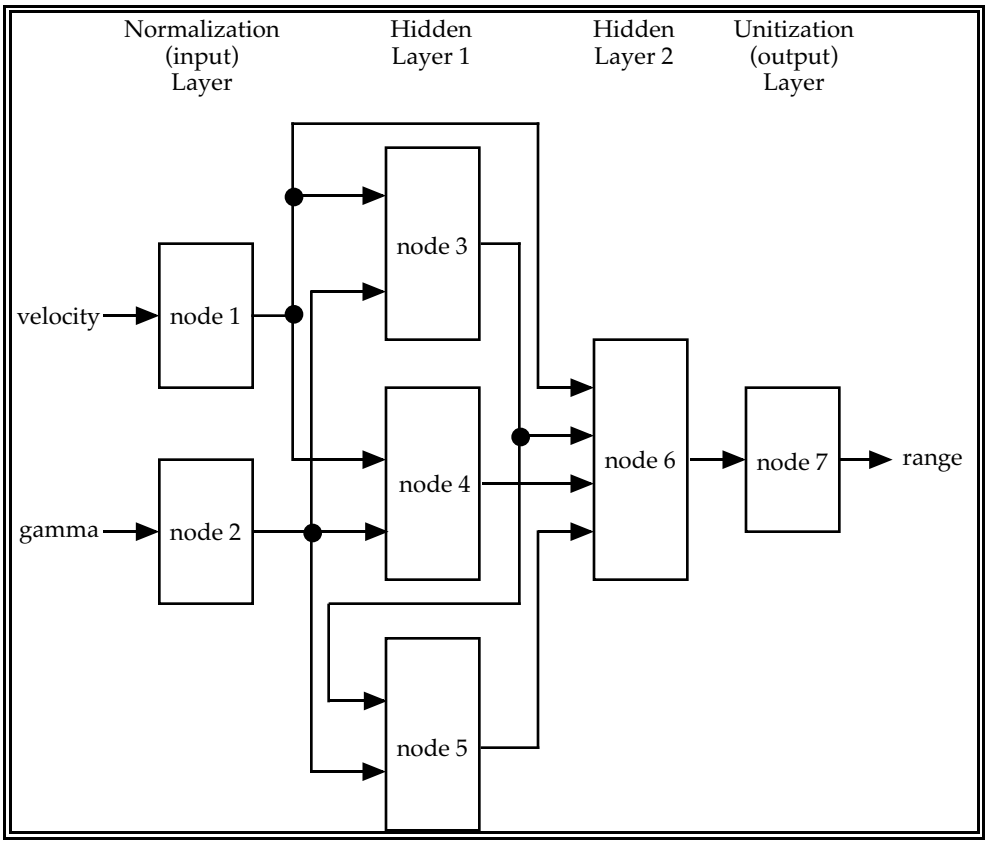


Figure 5.2: Network Structure of Example Pix File

Output Source Files (`*.c` and `*.h` files)

GNOSIS generates a C-language representation of the network after training is complete. This feature can be disabled in the GSF or SLF; see Chapter 4. The source code implements the synthesized network in user applications.

The header (`*.h`) file includes the prototypes of the functions that the application will call. Also included in the header file in the comment section is a sample C-language main program that executes the source code file. The fscanf line must be replaced to read inputs and outputs from your database; otherwise the program is ready to evaluate the network and produce a `user.est` output file. Link the application main program with the `*.c` source file and `bai_src.c` provided with the *GNOSIS* package.

Figure 5.3 shows a sample header file generated by *GNOSIS*. Function names depend on the basename, and parameter names depend on the database variables; look in the generated header file for the correct interface with the source code. Dynamic (having time-delays or feedbacks) networks include the option of using initial observations of data to initialize the network.

```
/*-------------------------------------------------------------------------------
    feedback.h

    created by GNOSIS using
        data:    feedback.tra
        control: feedback.gsf
-------------------------------------------------------------------------------*/

/*  Sample program to produce estimation file user.est from evaluation data feedback.eva
    Provide fscanf line to read your database; link with feedback.c and bai_src.c

#include <stdio.h>
#include "bai_src.h"
#include "feedback.h"

void main()
{
    FILE *ifp, *ofp;
    char str[256];
    int status;
    long obs = 0;
    double dummy;
    double True_y1, Est_y1;
    double True_y2, Est_y2;
    double x1;
    double x2;

    if (Init_feedback(true) == ERR_MEM) {
        printf("ERROR: Memory allocation; could not initialize network\n");
        exit(1);
    }

    ofp = fopen("user.est", "w");
    fprintf(ofp, "Obs\tTrue_y1\tModel_y1\tErr_y1\tTrue_y2\tModel_y2\tErr_y2\n");
    ifp = fopen("feedback.eva", "r");
    fgets(str, 256, ifp);

    while (true) {
        if (fscanf(ifp, "format", inputs, outputs, dummys) != count)
            break;

        status = Inter_feedback(&Est_y1, &Est_y2, x1, x2, True_y1, True_y2);
        if (status == INIT_NET)
            fprintf(ofp,"Initializing network with observation %ld\n", obs++);
        else
            fprintf(ofp, "%ld\t%g\t%g\t%g\t%g\t%g\t%g\n", obs++, True_y1, Est_y1, True_y1-Est_y1, True_y2,
                                                          Est_y2, True_y2-Est_y2);

        if (status == WARN_IN)
            fprintf(ofp,"Warning: Network input limited to the network training range\n");
        if (status == WARN_OUT)
            fprintf(ofp,"Warning: Network output limited to the network training range\n");
    }
    Close_feedback();
    fclose(ofp);
    fclose(ifp);
}
-------------------------------------------------------------------------------*/

#ifndef __feedback__
#define __feedback__

int Init_feedback(bool init_with_data);
    /* Allocates memory and initializes variables.  Call this before interrogating the network. */
    /* Set init_with_data = true to initialize the network from first observations. */
    /* Set init_with_data = false to initialize with zeroes. */
    /* Function returns OK or ERR_MEM. */

int Inter_feedback(double *Est_y1, double *Est_y2, double x1, double x2, double y1, double y2);
    /* Interrogates the network such that inputs: x1, x2, y1, y2, */
    /* propagate through the network producing new output(s): Est_y1, Est_y2, */
    /* Function returns OK, INIT_NET, WARN_IN, or WARN_OUT. */

void Close_feedback();
    /* Frees memory.  Call this once finished interrogating network. */

#endif
```

Figure 5.3: Example Header File (*.h)

FILES CREATED DURING TRAINING AND EVALUATION

When evaluating or training, *GNOSIS* generates a statistics file (`*.sts`) and an estimation file (`*.est`). Keep in mind that *GNOSIS* automatically evaluates the network on the training data. However, to determine the robustness of the model, reevaluate the network on data that *GNOSIS* has not seen in training .

Statistics File (`*.sts` file)

The statistics file provides various measures from the training or evaluation run to assess the performance of the network. The statistics file is different for estimation problems than for classification problems; an estimation statistics file is shown in Figure 5.4. Weighted standard deviation and mean are reported for each output, using the formulas:

$$\text{mean} = \text{sum} \;/\; \text{total\_weight}$$

$$\text{std}^2 = \frac{(\text{total\_weight} * \text{sum\_square} - \text{sum} * \text{sum})}{\text{total\_weight} * (\text{total\_weight} - \text{min\_weight})}$$

where total_weight = sum of all observation weights
min_weight = smallest positive observation weight
sum = sum of obs * weight for each obs, where obs is one input or output column
sum_square = sum of obs * obs * weight for each obs

If observations weights are not used, total_weight becomes the number of observations, min_weight becomes 1, and all weights are 1.0 in the sums. Large ratios between the maximum and minimum positive weights cause the standard deviation calculation to become inaccurate and should be avoided. To greatly deemphasize an observation, use a weight of 0.0 rather than a very small number, keeping the max-to-min weight ratio within reasonable bounds.

For each estimation output, a column of error statistics are reported, including RMS and normalized RMS error, mean and mean absolute error, standard deviation and the R^2 statistic. The error statistics are computable from the errors listed in the estimation file and use observation weights if available. RMS Error is the root-mean-square of the error between the true and modeled outputs. RMS Error is in units of the data, while Norm RMS Error is normalized by dividing RMS Error by the standard deviation of the database outputs. The mean of all errors and mean of all error magnitudes is also reported. The error standard deviation is the root of the difference between MSE and the square of mean error, in units of the data. The R^2 statistic is the percent of the output variance accounted for by the model, and is computed by dividing the difference between output and error variance by output variance. A perfect model would have an R^2 statistic of nearly 1.0.

Finally, the counts of all non-zero parameters in the hidden layers related to each output are reported. If nodes are shared between outputs, the parameter counts for all outputs may sum to more than the actual parameters in the hidden layers. The observation counts show the number of observations used to initialize the network, vs used for evaluation. The initialization count is zero unless `Init_Network_With_Database` was selected in the NDF or GSF for a network with time delays. The evaluation count is used for all of the mean error statistics described

above. The sum of the initialization and evaluation counts equals the total number of observations in the database.

```
Output Statistics:    y1                y2
  Mean:               2.192485409       0.1271482853
  Std Dev:            5.621964079       3.453536992

Error Statistics:     y1                y2
  RMS:                0.001032932955    0.04596445844
  Norm RMS:           0.0001837316888   0.01330938645
  Mean:               4.9525142e-05     7.284049404e-05
  Mean Abs:           0.0007910210404   0.03406389297
  Std Dev:            0.001031745002    0.04596440072
  R Squared:          0.9999999663      0.9998228607

Parameters:           27                27

Observations used for initialization: 1
Observations used for evaluation: 199
```

Figure 5.4: Estimation Statistics File (*.sts)

A classification statistics file has two classification tables plus two scoring statistics. Figure 5.5 shows how the observations in the database are classified by the model. The bottom-right corner value of the table shows the total number of observations in the database. Be sure that the database had an adequate representation of each class.

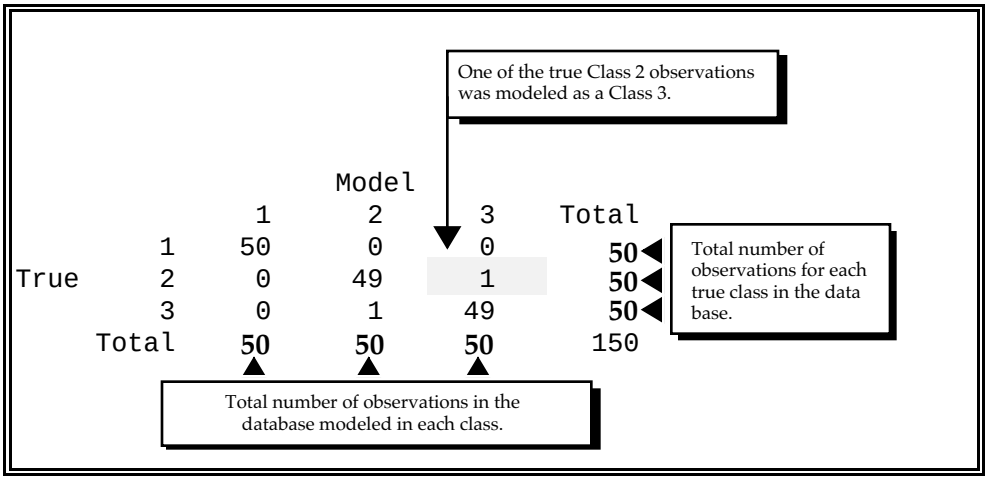


Figure 5.5: Statistics File "Number of Observations Classified"

A percentage (strictly speaking, a per unit) table is also a part of the statistics file; see Figure 5.6. The numbers in the table show, on a per unit basis, how many times each true class was modeled as a Class 1, Class 2, etc.

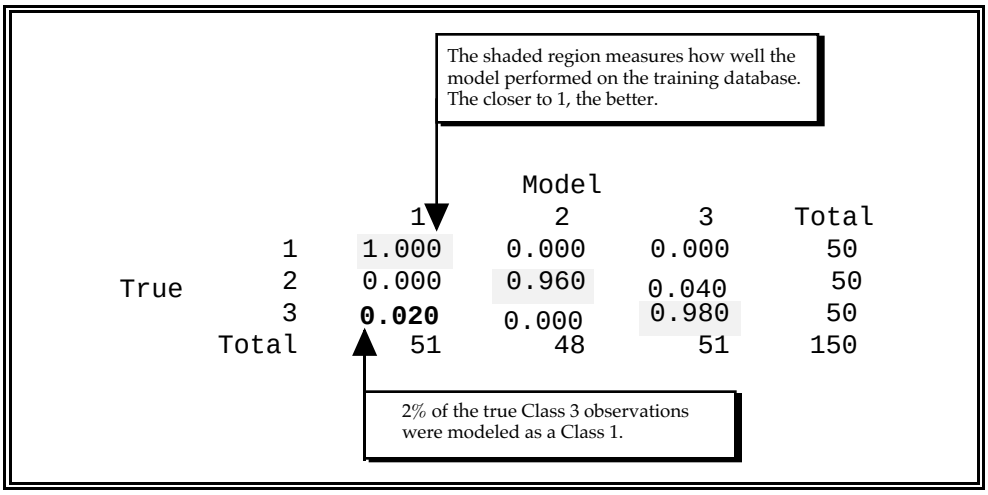


Figure 5.6: Statistics File "Percentage Table"

Finally the Logistic-Loss score and percent misclassified are recorded. The score is the sum of the distribution function outputs divided by the number of observations, and generally decreases as network accuracy improves. The percent misclassified is the sum of off-diagonal elements in the classification table, divided by the number of observations. An example of an entire statistics file for a classification problem is shown in Figure 5.7.

```
                Number of Observations Classified

                                          Model
                          Species_1  Species_2  Species_3      Total
                Species_1         7          0          0          7
True            Species_2         0          9          0          9
                Species_3         0          1         13         14
                     Total        7         10         13         30


                           Percentage Table

                                          Model
                          Species_1  Species_2  Species_3      Total
                Species_1     1.000      0.000      0.000          7
True            Species_2     0.000      1.000      0.000          9
                Species_3     0.000      0.071      0.929         14
                     Total        7         10         13         30


Logistic-Loss score:       2.482426695
Percent misclassified:     0.03333333333

Observations used for initialization: 0
Observations used for evaluation: 120
```

Figure 5.7: Classification Statistics File

Estimation File (`*.est` file)

The estimation file, Figure 5.8, presents for each observation in the database the observation number, the true outputs from the database, the outputs modeled by *GNOSIS*, and the corresponding errors. The top line provides labels for each column of data in the rest of the file:

Obs refers to the observation index, numbered sequentially from the first observation not used for network initialization.

True_<name> refers to the true values of an output as specified in the database file. <name> is the label of this output as specified in the database file and NDF.

Model_<name> refers to the values obtained by the *GNOSIS* model for each observation. For classification problems, the values are probabilities.

Err_<name> refers to the error between the true and modeled data for each observation (err = true – modeled).

Data are separated by tabs, for processing by text editors, spreadsheet and graphing tools.

Obs	True_y1	Model_y1	Err_y1	True_y2	Model_y2	Err_y2
1	1.62474	1.62411	0.000633096	0.501206	0.520212	-0.0190064
2	3.00158	2.9992	0.00238114	0.501587	0.519169	-0.0175824
3	4.12693	4.12436	0.00257039	0.009568	0.0032703	0.0062977
4	5.00026	4.99959	0.000666001	-1.00065	-1.02558	0.024929
5	5.62611	5.62492	0.00119184	-2.54055	-2.57067	0.0301137
6	5.50081	5.49984	0.000969839	-5.65855	-5.65105	-0.00749703
7	6.12454	6.1244	0.000143754	-6.00927	-5.98787	-0.0214039
8	4.00031	3.99956	0.000756331	-4.23643	-4.25268	0.0162501

Figure 5.8: Example Estimation File (*.est)

6

STRUCTURE LEARNING

INTRODUCTION

If you have no idea of the structure of the network needed to model a given phenomenon, structure learning may be the best choice for network synthesis. However, structure learning only builds feedforward networks. If feedbacks are needed to create an accurate model, then use the options discussed in Chapter 4 to create a dynamic neural network.

This chapter gives a description of the *ASPN-III* structure learning option, beginning with a simple tutorial. If this option was not purchased with the *GNOSIS* software, the features discussed below will not be available.

QUICK START STRUCTURE LEARNING TUTORIAL

Structure learning can be used to find the best feedforward (no feedback) model that transforms observable (candidate input) variables into estimated outputs.† This tutorial session reviews the general process of model creation using the structure learning capabilities of *GNOSIS*. A more detailed account of the structure learning features of *GNOSIS* is given following this tutorial.

A hypothetical "moon cannon" is used to generate a database for this tutorial: Suppose that a cannon on an airless moon is test-fired 50 times at various angles of muzzle elevation, called `gamma` (degrees), and muzzle velocities, `velocity` (m./sec.). The corresponding distances that the cannonball traveled, `range` (m.), are recorded,†† as shown in Figure 6.1. The goal is to predict `range` for any combination of `gamma` and `velocity` in future firings. To run this tutorial, go to the cannon directory provided with *GNOSIS*, where all the necessary files are located.

---

† Time delays of observable quantities are learned indirectly if the user pre-computes the candidate delayed variables and presents them to the *GNOSIS ASPN-III* option via the synthesis database (Chapter 3). *ASPN-III* then selects the most relevant time-delayed (i.e., leading) variables from those candidates.

†† In this case, computed from first principles.

Database Creation and Manipulation

As in the previous examples, a database must be created which describes the problem to be solved. The data for the problem at hand are stored in the file `cannon.dat`.

obs	range	velocity	gamma
1	28462	411.6	58.4
2	31737	475.9	24
.	.	.	.
.	.	.	.
.	.	.	.
49	33755	498.6	66.9
50	45438	491.1	44.8

Figure 6.1: Original Moon Cannon Data (cannon.dat)

From the `velocity` and `gamma` information, one may readily compute the horizontal and vertical components of the cannonball muzzle velocity vector, i.e., the initial conditions:

$$\dot{x}_0 = V_0 \cos \gamma_0 \qquad \dot{y}_0 = V_0 \sin \gamma_0$$

Placing $\dot{x}_0$ and $\dot{y}_0$ into the database in lieu of or in addition to V_0 and γ_0 is very advantageous. It can be shown that the analytical solution for `range`, expressed in terms of V_0 and γ_0, involves the function $\frac{V_0^2}{2g} \sin 2\gamma_0$. (Maximum `range` results when $\gamma_0 = 45$ deg., and higher or lower muzzle elevation angles reduce `range`.) Thus, the solution in terms of V_0 and γ_0 requires learning a trigonometric function, difficult to do accurately with only 40 or 50 data points.

Suppose we do not know about the trigonometry, and proceed to train a network using the original database "willy nilly." *GNOSIS* does the best it can. If we suspect difficulty with use of (V_0, γ_0) coordinates only, given that the database is small, we could model two functions, one for `range` when $0 \le \gamma_0 \le 45°$ and the other for `range` when $45° \le \gamma_0 \le 90°$. But a good general policy would be to present data to *GNOSIS* using more than one type of coordinate system (for model outputs just as much as for inputs), and turn the matter over to *GNOSIS* for resolution. Try this with the "moon cannon" database. Use (V_0, γ_0), ($\dot{x}_0$, $\dot{y}_0$), and (V_0, γ_0, $\dot{x}_0$, $\dot{y}_0$) input vectors in three separate synthesis runs, comparing results.

As in the other tutorials, the database should be split randomly, with approximately 80% of the observations going to the training database `cannon.tra` and 20% going to the evaluation database, `cannon.eva`. Both training and evaluation databases have been provided.

Model Synthesis (Training)

Once the training database has been established, start network synthesis via structure learning. To begin training, enter the command

gnosis -d cannon.tra -t cannon.slf -b cannon

The file `cannon.slf`, shown in Figure 6.2, is the Structure Learning File (SLF) described in detail later in this chapter. It controls the execution of *GNOSIS* when run in structure learning

mode. Repeat the training process using different SLF settings to find the model best suited to your needs. For each run, be sure to note which modifier settings in the SLF were used to create each result file.

```
Structure Learning File for GNOSIS.
Copyright 1998 by Barron Associates, Inc. All Rights Reserved.

-------------------------------------------------------------------------------
KEY_WORD                          ARGUMENTS     |COMMENTS
-------------------------------------------------------------------------------
Inputs                        USE_VSTRING  |USE_VSTRING, or enter input variables by name,
                                           |ending with'|'
Outputs                       USE_VSTRING  |USE_VSTRING, or enter output variables by name,
                                           |ending with'|'
Vstring                       -yxx--       |Specify inputs/outputs: y, x, -, integer
                                           |repeat count (only used if USE_VSTRING
                                           |option entered for Inputs or Outputs)
Weights                       weight       |Optional observation weight variable
-------------------------------------------------------------------------------
Normalize                     YES          |Normalize inputs (YES/NO)
Unitize                       YES          |Normalize/Unitize outputs (YES/NO)
-------------------------------------------------------------------------------
Pause_Between_Layers          NO           |Be prompted after completion of each layer
                                           |to decide whether to continue building the
                                           |structure (YES/NO)
Global_Optimization           QUADRATIC    |Optimize all parameters after learning the
                                           |network structure (NONE/QUADRATIC/LOGISTIC)
-------------------------------------------------------------------------------
Use_DB_Vars_As_Node_Inputs    YES          |Try database input variables as inputs to
                                           |nodes of each hidden layer past the first
                                           |hidden layer (YES/NO)
Sharing                       NO           |Outputs use best nodes from other outputs
                                           |(YES/NO)
Try_Whites                    NO           |Try white nodes on each hidden layer
                                           |(YES/NO)
Projection_Pursuit            YES          |Use Projection Pursuit (YES/NO)
Number_Of_L0_Elements         1            |Number of L0 nodes to keep for each hidden
                                           |layer
Max_Number_Of_Layers          2            |Maximum number of hidden layers for the
                                           |network
-------------------------------------------------------------------------------
Limit_Nodes                   ALL          |Set which nodes to limit.  Nodes from input
                                           |and output layers limited with database
                                           |training values. (IO/ALL/NONE)
Limit_Range                   1.0          |Percentage of full range (0.01 - 1.0) that
                                           |node outputs  will be limited to. (only used
                                           |f Limit_Nodes is ALL)
-------------------------------------------------------------------------------
Carving                       NO           |Carve away unneeded parameters in nodes
Node_Type                     COMP         |Specify polynomial structure type
                                           |(ADD/MULT/COMP) ADD = additive,
                                           |MULT = multilinear, COMP = complete,
Node_Degree                   2            |Maximum allowable degree of polynomial term
Max_Number_Of_Inputs          2            |Maximum number of inputs to any node
-------------------------------------------------------------------------------
Nearest_Neighbor              NO           |Use nearest neighbor algorithm to find
                                           |Sigma_P
Sigma_P                       0.0007       |Specify Sigma_P for each output (only used
                                           |if Nearest_Neighbor option is NO)
-------------------------------------------------------------------------------
Write_Pix_File                YES          |Write node connections diagram and node
                                           |equations to .pix file
Write_Source_Code             YES          |Write source code to .c and .h files
-------------------------------------------------------------------------------
```

Figure 6.2: Sample Structure Learning File (`cannon.slf`)

While training on the moon cannon data, *GNOSIS* develops a general relationship that forecasts `range`, when given new cases of `gamma` and `velocity`. Modeling information is displayed on the screen as execution proceeds; see Figure 6.3. More information about the messages displayed and files output by *GNOSIS* during network training is given at the end of this chapter.

```
************************************************
# GNOSIS Version 3.0
# Copy #0001 licensed to Barron Associates, Inc.

# Copyright 1995-1998, Barron Associates, Inc.
# All rights reserved
# GNOSIS
      Optimizes network structure and/or optimizes
      parameters of neural networks.
************************************************
# Cmdline: gnosis -d cannon.tra -t cannon.slf -b cannon
# GNOSIS: Sizing input database 'cannon.tra'...
# Database: 6 variables, 40 observations, 1 block(s)
# GNOSIS: Scanning structure learning file 'cannon.slf'...
# GNOSIS: Initializing the network structure...
# GNOSIS: Network input variable(s): Velocity Gamma
# GNOSIS: Network output variable(s): Range
# GNOSIS: Reading input database...
# GNOSIS: Beginning Network Synthesis algorithm...
# SL: ---- Layer 1 ------------
# SL: Out = Range, N = 0
# SL: Node =   3, rPSE = 0.0664963, RMS = 0.0664952
# SL: Out = Range, N = 1
# SL: Node =   3
# SL: ---- Layer 2 ------------
# SL: Out = Range, N = 0
# SL: Node =   3, rPSE = 0.0345262, RMS = 0.0345219
# GNOSIS: Layer limit reached.  2 hidden layer(s) successfully made
# GNOSIS: Optimizing the parameters for Range...
# ILS: Iteration  0, Lambda 0.0e+00, rScore 0.0405441
# ILS: Iteration  1, Lambda 1.0e-01, rScore 0.0405441
# ILS: Iteration  2, Lambda 1.0e-02, rScore 0.028184
# ILS: Iteration  3, Lambda 1.0e-03, rScore 0.0252902
# ILS: Iteration  4, Lambda 1.0e-04, rScore 0.0240246
# ILS: Iteration  5, Lambda 1.0e-03, rScore 0.0240246
# ILS: Iteration  6, Lambda 1.0e-04, rScore 0.0227805
# ILS: Iteration  7, Lambda 1.0e-05, rScore 0.0225097
# ILS: Iteration  8, Lambda 1.0e-04, rScore 0.0225097
# ILS: Iteration  9, Lambda 1.0e-05, rScore 0.0219331
# ILS: Iteration 10, Lambda 1.0e-06, rScore 0.0217879
# ILS: Iteration 11, Lambda 1.0e-07, rScore 0.0216935
# ILS: Iteration 12, Lambda 1.0e-08, rScore 0.0216918
# ILS: Iteration 13, Lambda 1.0e-07, rScore 0.0216918
# ILS: Iteration 14, Lambda 1.0e-06, rScore 0.0216918
# ILS: Iteration 15, Lambda 1.0e-05, rScore 0.0216918
# ILS: Iteration 16, Lambda 1.0e-04, rScore 0.0216918
# ILS: Iteration 17, Lambda 1.0e-03, rScore 0.0216918
# ILS: Iteration 18, Lambda 1.0e-02, rScore 0.0216918
# ILS: Iteration 19, Lambda 1.0e-01, rScore 0.0216918
# ILS: Iteration 20, Lambda 1.0e+00, rScore 0.0216918
# ILS: Iteration 21, Lambda 1.0e+01, rScore 0.0216918
# ILS: Iteration 22, Lambda 1.0e+02, rScore 0.0216918
# ILS: Iteration 23, Lambda 1.0e+03, rScore 0.0216918
# ILS: Iteration 24, Lambda 1.0e+04, rScore 0.0216918
# ILS: Iteration 25, Lambda 1.0e+05, rScore 0.0216918
# ILS: Normalized score change below tolerance.
# GNOSIS: Evaluating the network...
# GNOSIS: Writing estimation file 'cannon.est'...
# GNOSIS: Writing statistics file 'cannon.sts'...
# GNOSIS: RMS Error: Range = 70.3518
# GNOSIS: Norm RMS Error: Range = 0.00921419
# GNOSIS: R Squared: Range = 0.999915
# GNOSIS: Writing network description file 'cannon.mdl'...
# GNOSIS: Writing graphic representation of network to 'cannon.pix'...
# GNOSIS: Writing source code to 'cannon.c'...
# GNOSIS: Writing header code to 'cannon.h'...
# GNOSIS: DONE @2.75 sec
```

Figure 6.3: Display of Structure Learning Run

Model Evaluation

To evaluate the model generated during the training exercise above, enter the following command:

gnosis -d cannon.eva -e cannon.mdl -b cannon_eval

The files created during evaluation are described in Chapter 5.

STRUCTURE LEARNING FILE MODIFIER DESCRIPTIONS

The structure learning capabilities of *GNOSIS* are controlled entirely by a Structure Learning File (SLF). An example of a SLF with the proper syntax is shown in Figure 6.2; "learning" must appear in line 1. The SLF follows the same basic format as the GSF. Each structure learning field is explained in detail below, in the order they appear in the structure learning file.

Inputs and **Outputs [USE_VSTRING** or strings**] |**
Vstring [x, y, - codes**]**
Weights [string**]**
Normalize [YES/NO]
Unitize [YES/NO]

These fields are the same as for a GSF; see Chapter 4.

Pause_Between_Layers [YES/NO]

Enable this flag to use an interactive approach to network synthesis. The program stops at the end of each layer and asks if the next layer should be built. Otherwise network synthesis continues until there is no longer improvement or the maximum number of layers has been reached.

Global_Optimization [NONE/QUADRATIC/LOGISTIC]

Global optimization of all the parameters is available once the network structure is complete. *GNOSIS* uses the same ILS gradient-based search employed for GSF and NDF optimization. Select `NONE` to skip global optimization and retain the parameters developed during structure learning. Select `QUADRATIC` to use the squared-error loss function during global optimization, using the parameters from structure learning as initial values. Select `LOGISTIC` to use the logistic-loss function during global optimization. *GNOSIS* first resets hidden layer parameters to small random values, turns off unitization, resets output layer parameters for pass-through, and turns off node limiting for the output layer.

Use_DB_Vars_As_Node_Inputs [YES/NO]

Enable this flag to use input variables from the data file as candidate inputs to elements on all hidden layers. Otherwise data file inputs are provided only to the first layer.

Sharing [YES/NO]

Enable this flag to let the subnetworks in a multi-output model share their elements. The candidate inputs to a given layer of a particular network output will include the element outputs of the previous layer for all the other network outputs as well. If Sharing is disabled, an element found for one of the outputs is not used for any of the other outputs. The resulting network is a collection of n_o (number of outputs) independent subnetworks.

Try_Whites [YES/NO]

Enable this flag to try "white" elements. A white element is a linear combination element that consists of the weighted sum of all its inputs:

$$y = \theta_0 + \sum_{i=1}^{N} \theta_i \, x_i$$

where y is the element output, θ_i are the element parameters, and x_i are the element inputs.

Projection_Pursuit [YES/NO]

Projection pursuit finds candidate elements on each layer that will work well in linear combination with each other when the next layer is synthesized. To do this, a backfitting algorithm similar to the one used in projection pursuit regression is used. [Friedman and Tukey, 1974; Friedman and Stuetzle, 1981; Friedman, 1988]. Backfitting significantly increases execution time since the algorithm boosts the number of trial elements. However, projection pursuit often improves the network score.

First, *GNOSIS* attempts to find new elements that model the output all by themselves. These elements, called L_0 or non-projection pursuit elements, are created whether `Projection_Pursuit` is enabled or not. If `Projection_Pursuit` is enabled, *GNOSIS* then attempts to find an L_1 element with an output which, when added linearly to that of the best L_0 element, will model the output well. Next, an L_2 element is found that will work well when linearly combined with the best L_0 and the best L_1 elements. When the next layer is built, the L_0, L_1, and L_2 elements will all be available.

Number_of_L0_Elements [integer >= 1]

Enter the number of non-projection-pursuit elements to save on a given layer for a given output. If there are three outputs and `Number_of_L0_Elements` is four, 12 elements are saved on each layer, assuming `Projection_Pursuit` is disabled.

Max_Number_of_Layers [integer >= 1]

Enter the maximum number of hidden layers to build during network synthesis.

Limit_Nodes [IO/ALL/NONE]

Limit_Range [decimal]

These fields are the same as for a GSF; see Chapter 4.

Carving [YES/NO]

Enable this flag to eliminate (i.e., carve away) in each element the unneeded parameters and the basis functions they introduce. When determining which parameter to discard next, *GNOSIS* uses a "backward greedy algorithm" (see below) and the PSE as a stopping criterion.

A trial carve consists of:

1. removing a parameter from the element,
2. refitting the element (i.e., optimizing remaining parameters),
3. determining a score (with PSE), and
4. replacing the parameter (if determined to be needed).

First, *GNOSIS* finds the best parameter to remove with N trials, where N is the number of original parameters in the element. If the PSE allows it, the parameter is removed and *GNOSIS* looks for the next best parameter to remove (with N–1 trials), and so on for a maximum total of N + (N–1) + (N–2)… trials. This is known as "backward greedy" because once parameters are removed they are no longer considered by *GNOSIS*. A non-greedy algorithm would consider all combinations of parameters that could be removed before making the decision (with PSE) to remove anything. The number of trials that would be required for a non-greedy algorithm would be

$$\text{number of trials} = \sum_{i=1}^{N-1} \binom{N}{i}$$

where

$$\binom{N}{i} = \frac{N!}{i!\,(N-i)!}$$

For example, if an element had five original parameters, the greedy algorithm executed by *GNOSIS* would perform a maximum of 14 trials, whereas a non-greedy algorithm would require a maximum of 2500. The disparity between the two algorithms dramatically increases as the number of parameters increases.

`Node_Type [ADD/MULT/COMP]`

`Node_Degree [integer >= 0]`

These fields are the same as for a GSF, except the `CUSTOM` option is not available; see Chapter 4.

`Max_Number_Of_Inputs [integer >= 1]`

This modifier specifies the maximum number of inputs to any polynomial element in the network. For example, if `Max_Number_Of_Inputs` is 4, *GNOSIS* will try 1, 2, 3, and 4 input nodes during network synthesis.

`Nearest_Neighbor [YES/NO]`

The complexity penalty (the second term in the predicted squared error, PSE, criterion) is computed as

$$\frac{2K}{N}\,\sigma_p^2$$

where K is the number of non-zero parameters in the network, N is the number of observations in the database, and σ_p^2 is the prior estimate of the true error variance that does not depend on the network model being considered. σ_p^2 can be calculated a number of different ways.

Enable this flag to use a nearest-neighbor algorithm to set the values of σ_p, the prior estimate of the error standard deviation (see `Sigma_P` below). Usually the nearest-neighbor estimate is very conservative, as the estimation method of examining only local data "neighborhoods" is much less sophisticated than the global modeling performed by *GNOSIS*. Still, results from the simpler and quite different approach can provide a good starting point. Further, σ_p values may be reduced as modeling iterations reveal more about the tractability of the problem being addressed.

Sigma_P [decimals]

If Nearest_Neighbor is disabled, enter the normalized σ_p to use for each output. Otherwise ignore this field. Usually σ_p is set between 0.0 and 1.0 for the percentage of standard deviation of each database output not accounted for by the model. Smaller values cause a closer model fit. For cases with very large N, σ_p can be set above 1.0 to increase the complexity penalty.

Write_Pix_File [YES/NO]

Write_Source_Code [YES/NO]

These fields are the same as for a GSF; see Chapter 4.

STRUCTURE-LEARNING OUTPUT FILES

GNOSIS creates a number of files once network synthesis is complete, the same files described in Chapter 5. The statistics file has some additional entries after structure learning, as shown in Figure 6.4. Normalized σ_p from the SLF or computed by the nearest-neighbor algorithm is reported, as well as σ_p in units of the data. σ_p is the prior estimate of the standard deviation of each database output not accounted for by the model, and is used to develop complexity penalties (CP). The complexity penalty for each normalized and unitized σ_p is computed:

$$CP = \frac{2K}{N}\ \sigma_p^2$$

where K is the parameter count and N is the evaluation observation count. Finally, the root of the Predicted Squared Error gives an estimate of the model's performance on unseen data, and for each output is computed:

$$rPSE = \sqrt{RMS^2 + CP}$$

```
Output Statistics:    Range
  Mean:               33010
  Std Dev:            7635.153853

Error Statistics:     Range
  RMS:                70.35175568
  Norm RMS:           0.00921418966
  Mean:               0.1330276522
  Mean Abs:           54.64694694
  Std Dev:            70.35162991
  R Squared:          0.999915099

Parameters:           12

SL Statistics:        Range
  Norm SigmaP:        0.0007
  Norm CP:            2.94e-07
  Norm rPSE:          0.009230129527
  SigmaP:             5.344607697
  CP:                 17.13889886
  rPSE:               70.47345902

Observations used for initialization: 0
Observations used for evaluation: 40
```

Figure 6.4: Structure Learning Statistics File (cannon.sts)

STRUCTURE-LEARNING MESSAGES

While *GNOSIS* is performing structure learning, information is printed to the screen pertaining to the current status of network synthesis. As seen in the Figure 6.3, *GNOSIS* displays the number of trial elements attempted thus far for each layer, each named output, and each projection pursuit number N. If N = 0, L0 elements are currently being tried, if N = 1, L1 elements, and so on. For each trial, the best square root of predicted squared error and best root of mean squared error found so far are displayed. As the structure develops, RMS tends to get smaller as the fit improves. Since PSE combines RMS and the complexity penalty, it increases as parameters are added, and decreases as RMS is reduced. Structure development stops when there is no improvement in PSE. If no rPSE or RMS is displayed for a trial, all candidate elements were eliminated before fitting by special heuristics designed to remove redundant nodes.

After at least one element has been found for each output, the structure learning search can be terminated by entering Control C. Before that time, Control C is ignored. Once all the candidate elements are exhausted or the search is terminated, global optimization begins (if selected in the Structure Learning File). *GNOSIS* displays ILS iteration messages on the screen, as described in Chapter 5.

TIPS FOR IMPROVING MODEL PERFORMANCE

The quality of a generated model is strongly dependent upon the data that are used to train the model. Following the guidelines outlined in Chapter 3 can dramatically improve model performance. Determining the number of input variables to include in the database can also help *GNOSIS* find better solutions. However, this can be an evolving process. On the first *GNOSIS* training run, use just a few of the candidate input variables in a trial synthesis. This will establish quickly if the synthesis protocol is being followed correctly. Next, include all the available candidate input variables in a subset of the training database. If this constitutes a large number of input variables, *GNOSIS* will take relatively more time. *GNOSIS* will examine each candidate input variable in the context of its relevance to the model, and determine which of the candidates have significant influence on the output variables. Those input variables that do not significantly influence the output variables will not appear in the *GNOSIS* model. The second synthesis will therefore show which candidate input variables are valuable. For successive runs, reduce the number of candidate inputs and increase the number of training exemplars.

7

REFERENCES

Akaike, H., "Statistical predictor identification," *Ann. Inst. Stat. Math.*, Vol. 22, pp. 203-217, 1970.

Akaike, H., "Information theory and an extension of the maximum likelihood principle," *Proc. Second Int'l. Symp. on Information Theory*, B.N. Petrov and F. Csaki (Eds.), Akadémiai Kiadó Budapest, pp. 267 -281, 1973.

Angeline, P.J., G.M. Saunders, J.B. Pollack, "An evolutionary algorithm that constructs recurrent neural networks," *IEEE Trans. on. Neural Networks*, Vol. 5, No. 1, January 1994, pp. 54-65.

Anon., *CLASS (Polynomial Neural Network Classifier Synthesis Algorithm) Users' Manual*, Barron Associates, Inc., September 1992.

Anon., *IMP (Intelligent Modeling Product) Users' Manual*, Barron Associates, Inc., November 1995.

Barron, A.R., *Properties of the Predicted Squared Error: A Criterion for Selecting Variables, Ranking Models, and Determining Order*, Adaptronics, Inc., McLean, VA, 1981.

Barron, A.R., "Predicted Squared Error: A criterion for automatic model selection," *Self-Organizing Methods in Modeling: GMDH Type Algorithms*, S.J. Farlow (Ed.), Marcel Dekker, Inc., NY, Chap. 4, pp. 87-103, 1984.

Barron, A.R. and R.L. Barron, "Statistical learning networks: A unifying view," *1988 Symp. on the Interface: Statistics and Computing Science*, Reston, VA, Apr. 21-23, 1988, pp. 192-203.

Barron, A.R., "Statistical properties of artificial neural networks," *Proc. IEEE 1989 Conf. on Decision and Control*, Tampa, FL, December 13-15, 1989.

Barron, A.R., "Universal approximation bounds for superpositions of a sigmoidal function," *IEEE Trans. on Information Theory*, vol. 39, no. 3, pp. 930-945, May 1993.

Barron, A.R., "Approximation and estimation bounds for artificial neural networks," *Machine Learning*, Vol. 14, Kluwer Academic Publishers, Boston, pp. 115-133, 1994.

Barron, R.L., "Adaptive transformation networks for modeling, prediction, and control," *Proc. Joint National Conference on Major Systems*, IEEE/ORSA, October 25-26, 1971.

Barron, R.L., "Learning networks improve computer-aided prediction and control," *Computer Design*, August 1975.

Barron, R.L., A.N. Mucciardi, F.J. Cook, J.N. Craig, and A.R. Barron, "Adaptive learning networks: development and application in the United States of algorithms related to

GMDH," *Self-Organizing Methods in Modeling: GMDH Type Algorithms*, S.J. Farlow (Ed.), Marcel Dekker, Inc., NY, Chap. 2, pp. 25-65, 1984.

Barron, R.L., "Alternative strategies for reconfigurable flight controls," *Proc. 1984 NAECON*, pp. 1313-1320, May 21-25, 1984.

Barron, R.L. and D.W. Abbott, "Use of polynomial networks in optimum, real-time, two-point boundary-value guidance of tactical weapons," *Proc. 1988 Military Computing Conf.*, May 3-5, 1988.

Barron, R.L., R.L. Cellucci, P.R. Jordan, III, N.E. Beam, P. Hess, and A.R. Barron, "Applications of polynomial neural networks to FDIE and reconfigurable flight control," *Proc. 1990 NAECON*, May 1990.

Breiman, L. and J.H. Friedman, "Estimating optimal transformations for multiple regression and correlation," *J. Amer. Statist. Assoc.*, vol. 80, pp. 580-619, 1985.

Breiman, L., Friedman, J.H., Olshen, R.A., and Stone, C.J., *Classification and Regression Trees*, Wadsworth & Brooks, Monterey, CA, 1984.

Cherkassky, V.C., Friedman, J.H., Wechsler, H. eds., *From Statistics to Neural Networks*, Springer-Verlag, NY, 1994.

Daubechies, I., *Ten Lectures on Wavelets*, CBMS-NSF Regional Conference Series in Applied Mathematics, Capital City Press, Montpelier, Vermont, 1992.

Elder, J.F., IV, R.L. Cellucci, and R.L. Barron, *Users' Manual: ASPN-II — Algorithm for Synthesis of Polynomial Networks*, *Version 8.00*, Barron Associates, Inc., July 1990.

Farley, B.G. and W. A. Clark, "Simulation of self-organizing systems by digital computers," *IRE Trans. on Inform. Theory*, vol. PGIT-4, pp. 76-84, 1954.

Fayyad, M. , G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, eds. *Advances in Knowledge Discovery and Data Mining*, U, AAAI/MIT Press. 1996. (See especially: Elder, IV, J.F. and Pregibon, D.A Statistical Perspective on Knowledge Discovery in Databases, Chapter 4 .)

Fisher, R.A., "The use of multiple measurements in axonomic problems," *Annals of Eugenics*, vol. 7, pp. 179-188, 1936.

Friedman, J.H. and J.W. Tukey, "A projection pursuit algorithm for exploratory data analysis," *IEEE Trans. on Computers*, Vol. 23, pp. 881-889, 1974.

Friedman, J.H. and W. Stuetzle, "Projection pursuit regression," *J. Amer. Stat. Assoc.*, vol. 76, pp. 817-823, 1981.

Friedman, J.H., "Fitting functions to noisy scattered data in high dimensions," *Proc. 20th Symposium on the Interface: Computing Science and Statistics,* Reston, VA, April 1988.

Friedman, J.H., "Multivariate adaptive regression splines," *Annals of Statistics,* vol .19, no. 1, pp. 1-66, 1991.

Gabor, D., "Communication theory and cybernetics," *Trans. of IRE*, Vol. CT-1. no. 4, p. 19, 1954.

Gabor, D., P.L. Wilby, and R. Woodcock, "A universal non-linear filter, predictor and simulator which optimizes itself by a learning process," *J. IEE*, paper received October 1959.

Gilstrap, L.O., Jr., "An adaptive approach to smoothing, filtering and prediction," *Proc. 1969 NAECON*, pp. 275-280, 1969.

Gilstrap, L.O., Jr., "Keys to developing machines with high-level artificial intelligence," *ASME Design Engineering Conf.*, ASME Paper No. 71-DE-21, April 1971.

Hecht-Nielsen, R., *Neurocomputing*, Addison-Wesley Publ. Co., Reading, MA, 1989.

Hebb, D.O., *The Organization of Behavior*, John Wiley & Sons, Inc., NY, 1949.

Hess, P. and D.W. Abbott, *Multidimensional Search and Optimization with the OMNIsearch Algorithm,* Barron Associates, Inc., Informal Documentation for Century Computing, Inc., P.O. 5035, Contract F33615-84-C-3609, August 1988.

Honig, M.L. and D. G. Messerschmidt, *Adaptive Filters: Structures, Algorithms, and Applications*. Kluwer, Boston, 1984.

Hush, D.R. and B.G. Horne, "Progress in supervised neural networks: What's new since Lippmann?," *IEEE Signal Processing Magazine*, Jan. 1993.

Ivakhnenko, A.G., "The group method of data handling — A rival of stochastic approximation," *Soviet Automatic Control*, vol. 1, pp. 43 - 55, 1968.

Ivakhnenko, A.G., "Polynomial theory of complex systems," *IEEE Trans. on Systems, Man, & Cybernetics*, vol. SMC-1, no. 4, pp. 364-378, October 1971.

Lee, R.J., "Internal Circuitry of a Reron," privately published, Lib. of Congress Card #TK 7882.C5L4, June 1955.

Lee, R.J., "Generalization of learning in a machine," *Preprints Papers Presented at Fourteenth Natl. Meeting ACM*, pp. 21-1-21-4, September 1959.

Lee, R.J. and L.O. Gilstrap, Jr., "Learning machines," *Proc. Bionics Symp.*, USAF Wright Air Development Div., Dayton, OH, TR60-600, pp. 437-450, 1960.

Ljung, L. and T. Söderström, *Theory and Practice of Recursive Identification*, MIT Press, Cambridge, MA, 1983.

Mallows, C.L., "Some comments on Cp," *Technometrics*, vol. 15, pp. 661-675, 1973.

Marquardt, D.W., "An algorithm for least-squares estimation of non-linear parameters," *Journal SIAM,* vol. 11, pp. 431-441, 1963.

McCulloch, W.S. and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bull. Math. Biophys*, vol. 5, pp. 115-133, 1943.

Michie, D., Spiegelhalter, D.J., and Taylor, C.C., eds. *Machine Learning, Neural and Statistical Classification*. Ellis Horwood, New York, 1994.

Moddes, R.E.J., R.J. Brown, L.O. Gilstrap, Jr., R.L. Barron, et al., *Study of Neurotron Networks in Learning Automata*, Adaptronics, Inc., AFAL-TR-65-9, February 1965.

Papoulis, A., *Probability, Random Variables, and Stochastic Processes.*, McGraw-Hill, NY, 1986.

Press, W.H., B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, NY, 1986.

Reklaitis, G.V., A. Ravindran, and K.M. Ragsdell, *Engineering Optimization Methods and Applications*, John Wiley & Sons, pp. 105-106, 1983.

Rissanen, J., "Modeling by shortest data description," *Automatica*, vol. 11, pp. 465-471, 1978.

Rissanen, J., "A universal prior for integers and estimation by minimum description length," *Ann. Stat*., Vol. 11, No. 2, pp. 416-431, 1983.

Rosenblatt, F., "The perceptron," *Cornell Aeronaut. Lab. Rept.* VG-1196-G-1, January 1958.

Rumelhart, D.E., G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in D. E. Rumelhart and J. L. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*, M.I.T. Press, Cambridge, MA, 1986.

Shrier, S., R.L. Barron, and L.O. Gilstrap, "Polynomial and neural networks: Analogies and engineering applications," *Proc. 1st Ann. Conf. on Neural Networks*, vol. II, pp. 431-439, June 1987.

Snyder, R.F., R.L. Barron, R.J. Brown, and E.A. Torbett, *Advanced Computer Concepts for Intercept Prediction, Vol. I: Conditioning of Parallel Networks for High-Speed Prediction of Re-Entry Trajectories*, Adaptronics, Inc. Technical Summary Report, Contract DA-36-034-AMC-0099Z, Nike-X Proj. Ofc., U.S. Army Material Command, Redstone Arsenal, AL, 1964.

Söderström, T. and P. Stoica, *System Identification*, Prentice Hall, NY, 1989.

Specht, D.F., "Generation of polynomial discriminant functions for pattern recognition," *IEEE Trans. on Electronic Computers*, vol. EC-16, no. 3, pp. 308-319, June 1967.

Ward, D.G., B.E. Parker, Jr., and R.L. Barron, *Active Control of Complex Systems Via Dynamic (Recurrent) Neural Networks*, Barron Associates, Inc. Final Technical Report for Office of Naval Research, Contract N00014-89-C-0137, May 1992.

Ward, D.G., B.E. Parker, Jr., and R.L. Barron, *Principles of Function Estimation Using Artificial Neural Networks*, Barron Associates, Inc., September 1, 1993.

Ward, D.G., "Generalized networks for complex function modeling," *Proc. 1994 IEEE Systems, Man, & Cybernetics Conf.*, San Antonio, TX, October 2-5, 1994.

Weiss, S.M. and Kulikowski, C.A. *Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Networks, Machine Learning, and Expert Systems.* San Mateo, CA Morgan Kauffman, 1991.

Widrow, B., "Generalization and information storage in networks of Adeline neurons," *Self-Organizing Systems*, M.C. Yovits, G.T. Jacobi, and G.D. Goldstein (Eds.), Spartan Books, Washington, DC, pp. 435-461, 1962.

A

SERVICES AND SUPPORT

Thank you for purchasing *GNOSIS*. As owner of a *GNOSIS* usage license, for a period of one year you:

- will be notified automatically regarding upgrades and enhancements to *GNOSIS*, and
- are entitled to program maintenance and technical support up to the ceiling established by your purchase agreement.

This one-year maintenance is renewable annually for a nominal fee.

To help us keep our records current, please notify Barron Associates, Inc. of any name or address changes.

It is emphasized that you should create a backup copy of the software. Store it in a safe, cool place. The backup and original copies should be made accessible only to your authorized personnel.

Barron Associates seeks always to improve its service and support. If you find that a feature or technique needs further clarification in this Users' Manual, or something has been omitted from the index, please let us know. In addition, we would like to hear your comments or suggestions concerning the *GNOSIS* software.

TECHNICAL SUPPORT

If you are having difficulties with *GNOSIS* and cannot find the help you need in this manual, please contact us by:

- email - support is available by sending an email to:
  bird@bainet.com

- telephone – support is available at (804) 973-1215, Monday through Friday, 9:00 a.m. to 5:00 p.m. (U.S.A. Eastern Time). It will be helpful if you are seated at your computer when you make your call.
- FAX – support is available at (804) 973-4686, Monday through Friday, 9:00 a.m. to 5:00 p.m. (U.S.A. Eastern Time).
- mail – support is available by writing to

  Barron Associates, Inc.
  Jordan Building
  1160 Pepsi Place, Suite 300
  Charlottesville, VA 22901-0807
  U.S.A.

Please include your name, address, and telephone number.

We will be able to help you more efficiently if you can provide the following information when contacting us:

Computer Model,
Network Environment, and
Problem/Question.

B

DEVELOPMENT OF ARTIFICIAL NEURAL NETWORKS

Serious study of artificial neural networks began with the work of McCulloch and Pitts [1943],† who put forward a mathematical representation, based upon boolean algebra, for the gross behavior of neural networks. Hebb [1949] introduced neural network models in which the cells included delays and a refractory period, and the networks of these cells incorporated feedback connections producing reverberating chains. Lee [1952] proposed generalized logic learning elements for automata; selective reinforcement of total network behavior conditioned the appropriate boolean functions in each of the network elements. Farley and Clark [1954] suggested use of linear elements with output thresholding. Gabor [1954] proposed a filter that could learn with supervision.

Kolmogorov [1957] proved an important theorem on network representation of continuous functions. Rosenblatt [1957] drew a vital connection between studies of neural networks and of statistical inference; he showed that a network of the Farley and Clark elements can find a data-separating hyperplane if one exists. Rosenblatt [1958] also assembled hardware for a transformation network ("Perceptron"), using it to recognize patterns. Gabor et al. [1959] described results obtained with a "universal non-linear filter" which optimized itself by a learning process. The Gabor filter could compute "...94 terms of a polynomial, each term containing products and powers of the input quantities, with adjustable coefficients, and...form their sum". He showed that "...the most general functional of the past of a band-limited time function can be put in the form...of a polynomial of the samples". Widrow [1962] employed stochastic approximation to estimate sequentially the weights in a network of the Farley and Clark elements.

In 1963, R. L. Barron, Gilstrap, et al. introduced analytic nonlinear neural networks using polynomial nodal elements suggested by the earlier work of their colleague, Lee [1955, 1959, 1960]. It was demonstrated that these elements and networks could perform boolean logic,

† References are listed in Section 7.

estimate values of unknown variables, perform high-order predictions, and discriminate between patterns. Pre-structured feedforward networks were used, with the coefficients adjusted simultaneously using a statistical measure of performance on the synthesis database. The first application of the Barron/Gilstrap polynomial networks was for prediction of trajectories of atmosphere re-entry vehicles. It was demonstrated that polynomial networks are robust and competitive in accuracy with serial integration of equations of motion. Moreover, the polynomial networks were shown to be many orders of magnitude faster in solution speed [Snyder, et al. 1964].

With the propensity of large fixed networks to result in overfitted estimates, attention was turned in the 1970s to synthesis of networks for which the *structure is adaptively determined from the data*. Such network strategies were introduced by Ivakhnenko [1968, 1971] in Ukraine. Ivakhnenko used a one-element-at-a-time synthesis strategy and a statistical cross-validation procedure (with multiple data subsets) to select between candidate elements and discourage overfitting of the synthesis data. Akaike [1970, 1972] in Japan introduced information-theoretic, constrained-complexity statistical modeling criteria, opening the door for realization of rigorously optimized network structures and weights. Prior work in the United States and development and use of information-theoretic synthesis criteria and the Ivakhnenko strategy are discussed in R.L. Barron et al. [1984].

The types of nodal elements typically used in adaptively synthesized polynomial networks are second- and third-degree polynomial or multinomial functions in one, two, or three input variables. At the element level, the number of inputs is restricted to avoid a combinatorial explosion in the number of element possibilities that must be examined by the synthesis algorithm. The number of inputs to the network is limited only by the number of elements therein.

The basic Ivakhnenko strategy for feedforward estimation network structure learning (described here using elements involving two variables) is depicted in Figure B.1. On the first layer of the polynomial network, all possible pairs of inputs are considered and the best subset k_1 is temporarily saved. The value k_i is a program parameter that dictates the number of elements to save for layer i of the network. On the succeeding layers, all possible pairs of the intermediate variables z from the preceding layer(s) are considered and the best k_2 (k_3, etc.) are saved. When additional layers fail to provide more improvement, the network synthesis stops at the last layer that did produce improvement. The final network consists only of the elements driving the final (output) element (or elements, if multiple outputs are needed.)

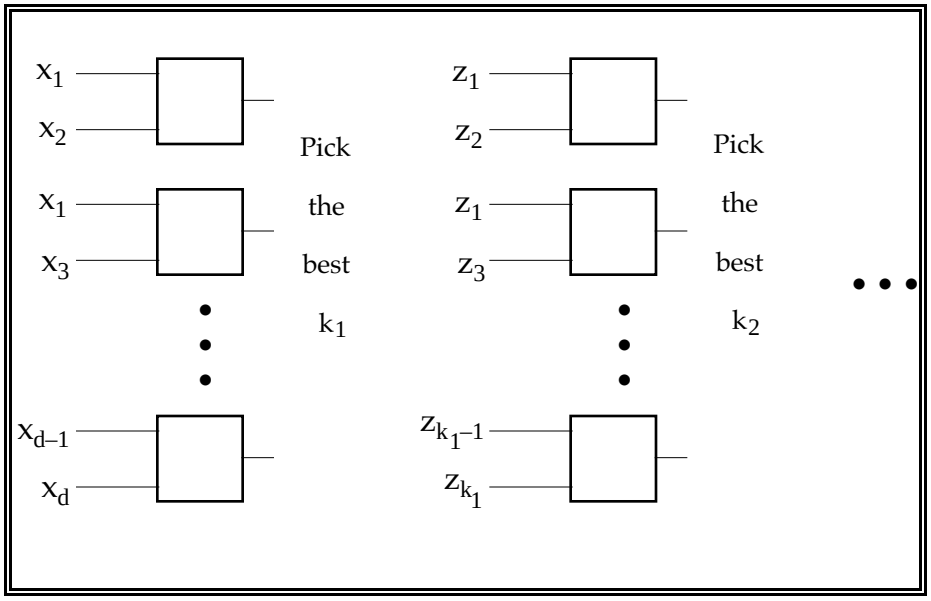


Figure B.1: An Adaptive Model Synthesis Strategy for Feedforward Estimation Networks

In the original Ivakhnenko algorithm, the parameters within each element were estimated so as to minimize on a training set of observations the sum of squared errors of the fit of the element to the final desired output. Cross-validation on a separate testing set was used to rank and select the best elements on each layer and to select the number of layers. (Ivakhnenko called this use of multiple data subsets during network synthesis and validation the *group method of data handling (GMDH)*.) The need to construct complete quadratic polynomials for every pair of variables forced early implementations of the algorithm to restrict the number of temporarily saved intermediate variables to be typically not more than 16.

A.R. Barron [1981, 1984, 1985] drew the connection between the subjects of neural network synthesis and statistical inference. Drawing upon the work of Akaike, Rissanen [1978], and Mallows [1973], he formalized the *predicted squared error (PSE)* and *minimum description length (MDL)* synthesis criteria and built synthesis algorithms upon them. These algorithms incorporated the PSE criterion at every phase of element design and element/network selection, including the "pruning" of terms in the individual elements to minimize their complexity. He formulated a method whereby candidate pairs were prescreened, permitting more elements to be considered on each layer. This also permitted use of more complicated elements, i.e., third-degree polynomials, with terms selected by the PSE criterion. The saved elements from *all* preceding layers were treated as candidate inputs to a given layer. Moreover, some one- and three-input elements were considered on each layer. The *PNETTR* synthesis algorithm by A.R. Barron [1979] was extensively applied to problems in nondestructive evaluation of materials, modeling of material characteristics, flight guidance and control, target recognition, intrusion detection systems, and scene classification; see R.L. Barron et al. [1984] and the references cited there.

The Barron Associates, Inc. (BAI) *ASPN-II* network synthesis algorithm by Elder, Cellucci, et al. [1990] permitted choice between the predicted squared error and minimum description

length criteria. This algorithm had more user flexibility in the specification of one-, two-, or three-input elements and in candidate forms of the polynomial elements. Moreover, at each layer a new element was considered that was a linear combination of all elements on the preceding layer.

The field of artificial neural networks continues to draw closer to statistical inference [Barron and Barron, 1988]. The work of Friedman and Tukey ["Projection Pursuit," 1974], Friedman ["Multivariable Adaptive Regression Splines," 1988] and others has been influential. High-order (high-degree) neural networks are being more widely studied [Giles and Maxwell, 1988]. The theoretical and estimation bounds on artificial neural networks have been derived [A.R. Barron, 1993, 1994]. *ASPN-IIc* and *ASPN-III* by BAI incorporate projection pursuit and other refinements in the *ASPN* structure-learning paradigm. BAI was first to apply the logistic-loss function, used extensively in the statistical community, to the construction of nonlinear classificatory neural networks. The first commercial product to have this feature was BAI's *CLASS* software [Anon., 1992].

Current work in artificial neural networks is placing increasing emphasis upon use of time delays and feedback (recurrent) connections that impart dynamic (hebbian) behavior [Ward, Parker, and Barron, 1992]. The BAI *GNOSIS* and predecessor *IMP* [1995] software are synthesis and evaluation tools that provide complete flexibility in the insertion of time delays and feedbacks within and external to artificial neural networks.

C

TECHNICAL / MATHEMATICAL DESCRIPTION†

INTRODUCTION

GNOSIS utilizes a generalized neural network architecture and learning algorithm that is capable of implementing a wide variety of neural and statistical function estimation paradigms, including basis functions, splines, polynomial neural networks, multi-layer perceptrons, recurrent networks, and others. The fundamental building block of *GNOSIS* is a generic nodal element that can perform a number of user-defined linear and nonlinear transformations. These nodal elements are combined into networks using an information-theoretic approach that reduces excess network complexity. An iterative Gauss-Newton training algorithm is used for network synthesis. In this Appendix it is shown how this algorithm is used to optimize the network for a variety of loss functions. The intent is to provide insight into both neural and statistical modeling by exploring the relationships between existing paradigms and by providing a technique that allows the best aspects of existing paradigms (including the Barron Associates, Inc. algorithms *ASPN-II, ASPN-IIc, CLASS, Dyn3*, and *IMP*) to be combined into novel function estimation strategies.

† The material in this paper was originally presented in [Ward, 1994]. (References are listed in Section 7.)

THE *GNOSIS* ARCHITECTURE

GNOSIS, like other tools for synthesis of artificial neural networks (ANNs), is a parallel, distributed information processing structure consisting of multiple, interconnected nodal elements. Figure C.1 shows a single fully-interconnected *GNOSIS* layer. Note the feedback connections in the generalized structure; this allows the network to be connected in either a feedforward or recurrent configuration.

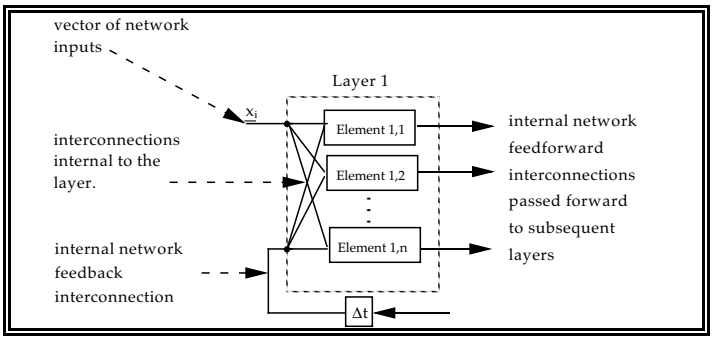


Figure C.1: A *GNOSIS* Layer

Each layer of *GNOSIS* is comprised of fundamental building blocks called nodes, elements, or nodal elements; a *GNOSIS* nodal element is shown in Figure C.2. This generalized nodal element is comprised of three important parts: (1) an algebraic or other series expansion, (2) a fixed linear or non-linear post-transformation function, h(·), and, (3) shift registers or delay banks to allow the series expansion to have access to prior input values. These shift-registers, along with the recurrent interconnections, provide the network with memory. The first two nodal element components are discussed in more detail below.

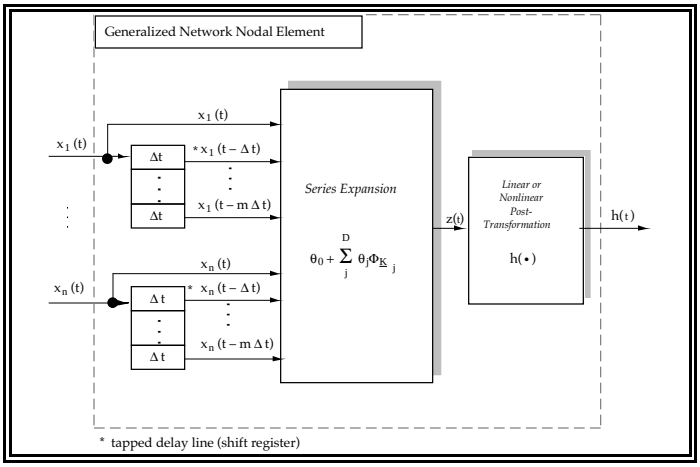


Figure C.2: Generalized Network Nodal Element

Basis Functions and Series Expansions

The series expansion of Figure C.2 is of the form

$$z(\underline{x}, \underline{\theta}) = \sum_{j=0}^{J} \theta_j \Phi(\underline{k}_j, \underline{x}) \quad (1)$$

where $\underline{\theta}$ is the vector of element coefficients, J is the total number of non-constant terms in the expansion, and $\underline{k}_j$ is a vector of integers. A bias term is ensured by requiring that $\Phi(0, \underline{x}) = 1$. The series expansion within a neural network element has the same form as traditional series-expansion techniques; however, with network function estimation, it is desirable that the total number of terms in any given element be kept as small as possible. This point will be elaborated on shortly.

The inclusion of $\underline{k}_j$, sometimes called the set of indices or multi-indices, allows the series expansion to handle both univariate and multivariate cases. For the multivariate case, each $\Phi(\underline{k}_j, \underline{x})$ is a product of functions of scalars. $\underline{k}_j$ is usually taken to be a vector of integers with each element of $\underline{k}_j$ corresponding to one of the variables in the $\underline{x}$ vector. Using this notation, the *j*th term in the series expansion may be written as:

$$\Phi(\underline{k}_j, \underline{x}) = \Phi(k_{j1}, x_1) \cdot \Phi(k_{j2}, x_2) \cdot \ldots \cdot \Phi(k_{jD}, x_D) \quad (2)$$

where D is the total number of inputs to the series expansion (i.e., the total number of outputs of the tapped delay lines).

The notation introduced above (and thus the nodal element) is sufficiently general to implement a variety of basis functions…

Polynomial:

$$\Phi(k, x) = x^k \quad (3)$$

Spline:

$$\Phi(k_{jd}, x) = \begin{cases} x^{k_i} & \text{if } k < r \\ (x-\alpha_{jd})_+^r & \text{if } k_i \geq r \end{cases} \quad (4)$$

Orthonormal Wavelet:

$$\Phi(k_{jd}, x) = \begin{cases} 2^{-k/2}\, \Psi(2^{-k} x - \alpha_{jd}) & \text{if } k > 0 \\ 1 & \text{if } k = 0 \end{cases} \quad (5)$$

Trigonometric:

$$\Phi(k, x) = \begin{cases} \sin\left(2\pi \frac{k+1}{2L} x\right) & \text{if k is odd} \\ \cos\left(2\pi \frac{k}{2L} x\right) & \text{if k is even} \end{cases} \quad (6)$$

For the polynomial basis function (3), the $\underline{k}_j$ vector is used to determine the powers to which each of the input variables is raised in the jth term of the expansion. The same is true for the spline basis function (4).

Note that in both the spline and the wavelet cases an additional set of multi-indices, α_{jd}, must be specified. The parameter α_{jd} in (4) and (5) is sometimes called the "knot" and is the value about which the approximation takes place. In the orthonormal wavelet basis function, $\Psi(\cdot)$ is termed the "mother wavelet" [Daubechies, 1992]. *GNOSIS* v2.6 uses only polynomial basis functions; however, future versions may offer additional choices.

From (1) - (6) it can be seen that the core expansion may be *fully specified* by (1) selecting a univariate basis function and (2) providing a $J \times D$ matrix, $\underline{\underline{K}}$, where each row of $\underline{\underline{K}}$ is the vector of integers $\underline{k}_j$ as defined above. The following example illustrates this point.

A *Full-Double* polynomial nodal element is shown below.

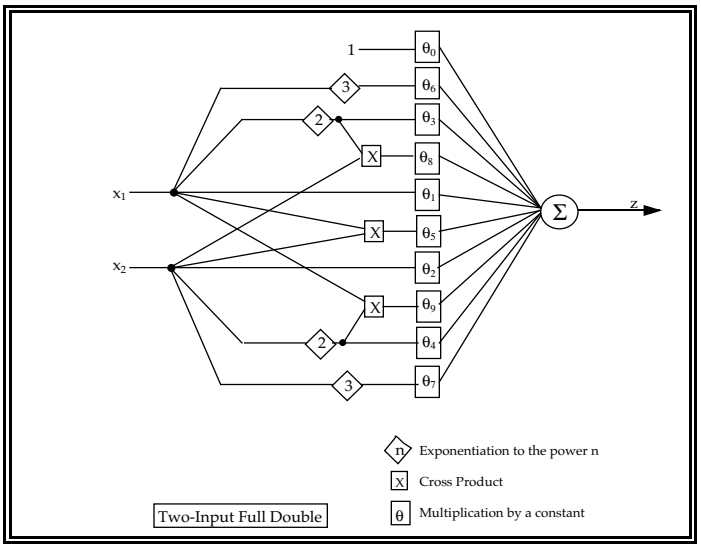


Figure C.3: *Full-Double* Polynomial Nodal Element

This nodal element has no input delays and no post-transformation $h(\cdot)$; therefore, it is completely specified by the following series expansion:

$$z(\bullet) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 + \theta_5 x_1 x_2 + \theta_6 x_1^3 + \theta_7 x_2^3 + \theta_8 x_1^2 x_2 + \theta_9 x_1 x_2^2 \quad (7)$$

Using the polynomial basis function (3), the $J \times D$ matrix $\underline{\underline{K}}$ that results in the transformation, (7), is

$$\underline{\underline{K}} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 2 & 0 \\ 0 & 2 \\ 1 & 1 \\ 3 & 0 \\ 0 & 3 \\ 2 & 1 \\ 1 & 2 \end{bmatrix} \tag{8}$$

Note that the θ_0 bias term is handled by a row of zeroes in the $\underline{\underline{K}}$ matrix.

Although the generalized nodal element is capable of implementing many commonly used series-expansion basis functions, neural network function estimation is fundamentally different from traditional series and nonparametric estimation techniques in the following ways:

- Each network element implements only a limited subset of the terms that would make up a complete series expansion; thus element complexity is kept low.
- Network interconnections allow a set of relatively simple network elements to be combined so that they can implement complex transformations; thus the network connections do a great deal of the "work" involved in the estimation problem.
- As the number of inputs to the function increases, the error bounds for network estimation can be shown to be more favorable than that of traditional function estimation techniques [A.R. Barron, 1991].

Network complexity is determined by:

1) **Number of Inputs (D):** Although the number of inputs to the network is largely determined by the application, it is possible to limit the number of inputs to *individual elements*, resulting in a less than fully interconnected network. The number of inputs corresponds to the number of columns in $\underline{\underline{K}}$.

2) **Maximum Degree (R):** The degree, R, of any given basis function is the maximum value of the sum of the elements in a row of $\underline{\underline{K}}$. For polynomial basis functions, the degree of a given term corresponds to the sum of the powers of the variables in the term.

3) **Maximum Coordinate Degree (P):** The coordinate degree of any given series expansion is the maximum value of *any* integer in $\underline{\underline{K}}$. For a polynomial basis function, this corresponds to limiting the power to which any given input may be raised.

4) **Maximum Interaction Order (Q):** In multivariate function estimation, the interaction order, Q, corresponds to the maximum number of different input variables that may appear at the same time in a given term and is equal to the number of non-zero elements in a row of $\underline{\underline{K}}$.

5) **Expansion Density:** Even after the number of inputs, degree, coordinate degree, and interaction order for a given series expansion are limited, one may choose to remove some terms to obtain a sparse or low-density expansion. This is often accomplished via carving or optimal brain damage [Elder, 1991; Hush, 1993].

Notice that, in all cases, the complexity of a generalized network element may be kept low by performing operations on the $\underline{\underline{K}}$ matrix.

Post-Transformations

In many neural network paradigms, the nonlinearities enter directly through the series expansion (e.g. Polynomial Neural Networks [Ward, 1992], Higher-Order Networks [Ward, 1993]). However, in many other common paradigms, the core transformation has few or no nonlinearities (e.g. MLPs, RBFs), and element nonlinearities are added using post processing. The linear or nonlinear fixed *post-transformation*, h(z), of Figure C.2 provides a means for incorporating these types of nonlinearities and allows the generalized nodal element specification to implement most nodal transformations currently in use. Figure C.4 shows the role of the post-transformation in the popular sigmoidal element used in multi-layer perceptron (MLP) neural networks.

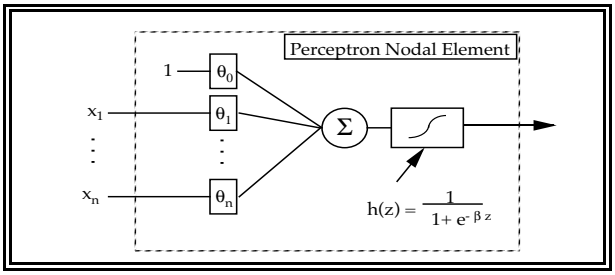


Figure C.4: An MLP Network Element

The element shown in Figure C.4 also has no time delays and implements a series expansion of the form

$$z(\bullet) = \theta_0 + \sum_{j=1}^{D} \theta_j x_j \qquad (9)$$

Following the same method outlined above, this series expansion can be represented by choosing the polynomial basis function of (3) and letting $\underline{\underline{K}}$ be a $D \times D$ identity matrix. Because $\underline{\underline{K}}$ contains only first-order interactions and has a maximum power of one, the number of terms in the series expansion is kept low.

The post-transformation, h(·), of Figure C.4 is a sigmoidal transformation and has the formula given in the figure. Due to the nonlinear post-transformation, the MLP nodal element is nonlinear in its parameters.

GNOSIS v2.6 and later implements linear, trigonometric, and sigmoidal post-transformations, and nonlinear *clamping* post transformations that clamp the output of the node at or within the

values obtained during training. This feature reduces the risk of incorrect extrapolations when a polynomial node is interrogated outside its training region.

CHOOSING AN APPROPRIATE LOSS FUNCTION

For a given network structure the "optimal" coefficients can be defined as those which minimize the sum of a loss function evaluated at every observation in a training database:

$$\min\left(\sum_{i=1}^{N} d(\underline{y}_i, \underline{s}_i)\right) \tag{10}$$

where, N is the number of observations in the training database, $\underline{y}_i$ is the *i*th output vector in the training database, $\underline{s}_i$ is the *i*th output vector of the network, and $d(\cdot)$ is the loss or *distortion* function.

While most ANN paradigms use a squared-error distortion function, it is desirable to allow the network to use different distortion functions depending on the network application. In general, most function estimation can fall into two broad categories: (1) system model estimation and (2) discriminant function estimation (Figure C.5).

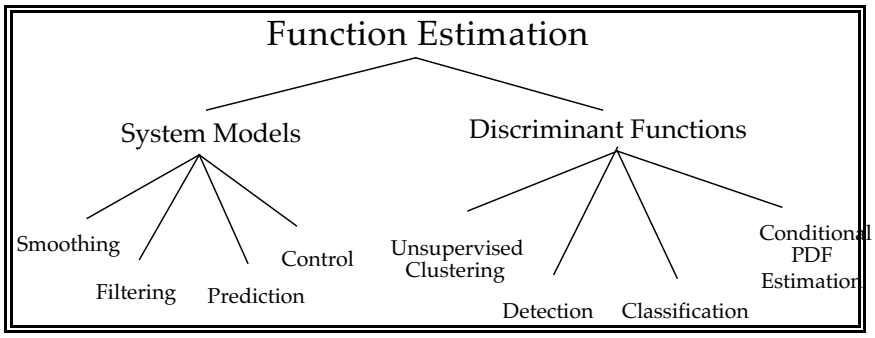


Figure C.5: Categories of Function Estimation

While a squared-error distortion function (or a modification thereof) is appropriate for system modeling, other distortion functions are more appropriate for the estimation discriminant functions. It is important that the generalized networks can be optimized using a variety of criteria; therefore, the only restrictions placed on the distortion function for the generalized network training are that it is convex and twice-differentiable.

Two of the most important distortion functions are discussed below; other loss functions are discussed in [Ward, 1993].

Squared-Error Loss Function

The squared-error loss function can be expressed as

$$d(\underline{y}_i, \underline{s}_i) = |\underline{y}_i - \underline{s}_i|^2 \tag{11}$$

In this case, the vector 2-norm $|\cdot|^2$ is defined as the sum of the squares of the differences between the coordinates of $\underline{y}_i$ and $\underline{s}_i$. One problem with the squared-error loss function is that data outliers tend to have a greater-than-desirable effect on the coefficient optimization. A number of *robust* loss functions have been suggested to reduce or nullify the effect of outlying data. One such function is Huber's loss function

$$d(\underline{y}_i,\underline{s}_i) = \begin{cases} |\underline{y}_i - \underline{s}_i|^2 & \text{if } |\underline{y}_i - \underline{s}_i|^2 \le A \\ 2A|\underline{y}_i - \underline{s}_i| - A^2 & \text{if } |\underline{y}_i - \underline{s}_i|^2 > A \end{cases} \tag{12}$$

where A is the distance at which outliers begin to have less effect. When $|\underline{y}_i - \underline{s}_i| > A$, $d(\cdot)$ becomes a 1-norm. This loss function has the advantages of a 1-norm; however, by using a 2-norm near the origin, the function is everywhere continuous in the first and second derivatives, which is not the case with a 1-norm loss function.

Logistic-Loss Function

Optimization of (10) using the squared-error distortion function of (11) corresponds to the maximum likelihood rule in the case of a Gaussian probability model for the distribution of the errors [Ljung, 1983]. However, for multi-class classification problems with categorical variables, a multinomial probability model in regular exponential form is more suitable than the Gaussian model [A.R. Barron, 1989]. In this case, the network functions should be used to model the log-odds associated with the conditional probability of each class given the observed inputs. In this setting, the maximum likelihood rule corresponds to the choice of the *logistic* loss function,

$$d(\underline{y}_i, \underline{s}_i) = -\underline{y}_i \cdot \underline{s}_i + \ln\left(\sum_{j=1}^{C} e^{s_{i,j}}\right) \tag{13}$$

where C is the number of outputs (or classes); $s_{i,j}$ is the *j*th element of the $\underline{s}_i$ vector; and $\underline{y}_i$ is a vector with the coordinate of the observed class equal to one, and all other coordinates equal to zero (i.e., the observed conditional probabilities given $\underline{x}_i$). In this context, the probability that an observation is a member of class k, given that the input state is $\underline{x}_i$, can be readily computed as follows:

$$p(k|\underline{x}_i) = \frac{e^{s_{i,k}}}{\sum_{j=1}^{C} e^{s_{i,j}}} \tag{14}$$

Both the squared-error and Logistic loss functions are available in *GNOSIS* v2.6 and later. A variety of other potential loss functions are discussed in Ward [1993].

Additional Penalty Terms

Additional penalty terms may be included to improve the ability of the network to interpolate between unseen data points. The most important of these is the complexity penalty, discussed below. However, there are a number of functions of the network coefficients that may be added

to any of the above loss functions to "smooth" the network output; these are often called *roughness* penalties.

In addition to improving the ability to interpolate, a roughness penalty can also improve network input-output stability, such that small variations in network input produce small variations in network output over the entire range of operating conditions. Any of the following, for example, may be used as a roughness penalty:

- Sum of squares of coefficient magnitudes
- Sum of squares of network gradients with respect to the inputs
- Minus the log of the prior density function of the network parameters

INFORMATION THEORETIC NETWORK OPTIMIZATION

The most important modification to any of the distortion functions above is the addition of an information-theoretic complexity penalty to prevent overfitting. A.R. Barron [1993] has given general conditions such that the minimum mean integrated squared error for an MLP neural network with one hidden layer will be bounded by

$$O\left(\frac{1}{n}\right) + O\left(\frac{nd}{N}\log N\right) \tag{15}$$

where O(·) represents "order of (·)," n is the number of elements, d is the dimensionality (number of coefficients per node), and N is the sample size (number of training exemplars). The first term in (15) bounds the approximation error, which *decreases* as network size increases. The second term in (15) bounds the estimation error, which represents the error that will be encountered on unseen data due to overfitting of the training database; it is caused by the error in estimating the coefficients. Estimation error, unlike approximation error, *increases* with network size.

Fully-connected, pre-structured networks, because they often have excessive internal degrees of freedom, are prone to overfit training data, resulting in poor performance on unseen data. Additionally, optimization of large pre-structured networks tends to be a slow and computationally intensive process. Without algorithms that learn the structure, the analyst often must resort to guesswork or trial and error if network complexity is to be reduced.

Improvements in network performance on unseen data can be made if one incorporates into the optimization algorithm modeling criteria that allow the network structure to grow to a just-sufficient level of complexity. Although this technique requires additional effort to search for an optimal structure, the overall network generation time is, in general, reduced due to the reduction in the number of coefficients.

Two decades of research have gone into this topic. In Ukraine, Ivakhnenko [1968] introduced the *Group Method of Data Handling* (GMDH). With GMDH, the loss function is squared-error, and overfitting is kept under control by means of cross-validation testing that employs independent subsets (groups) of the database for fitting and selection. GMDH is a satisfactory approach when sufficient data are available. In Japan, Akaike [1972] introduced an

information theoretic criterion (AIC) that uses all of the data and incorporates a penalty term for overfit control. Akaike's criterion is one of several that take the form

$$\frac{1}{N}\sum_{i=1}^{N} d(\underline{y}_i, \underline{s}_i) + C\frac{K}{N} \qquad (16)$$

where K, in this context, is the number of non-zero coefficients in the model, N is the number of data vectors in the database, and C is a constant.

Most forms of (16) depend on the structure of the candidate model, which can cause problems when learning the model structure. One form that does not is A.R. Barron's *predicted squared error* (PSE) criterion [1984]. The derivation of PSE results in a loss function of the form given in (16) with the constant

$$C = 2\sigma_p^2 \qquad (17)$$

where σ_p is an *a priori* estimate of the model error variance.

Once an appropriate constrained loss function has been identified, the learning algorithm may use a variety of techniques to search for a *statistically justified* level of complexity. Among the more successful structure-learning algorithms are GMDH [Ivakhnenko, 1968], Projection Pursuit [Friedman, 1981], CART and MARS [Friedman, 1991], and Evolutionary Programming (EP) approaches [Angeline, 1994]. Most of these algorithms, however, were originally designed to work with a specific model structure. However, these algorithms contain a number of proven heuristics that may be applied to the structure-learning problem for the generalized networks described here [Ward, 1993].

The Optional *ASPN-III* Structure Learning Algorithm for *GNOSIS*

The *ASPN-III* structure-learning option of *GNOSIS* uses the predicted squared error (PSE) to determine an optimal feedforward neural network structure. For each layer of the synthesized network, a succession of the various nodal functions, with different combinations of inputs, is fitted and scored with the PSE modeling criterion. Fitting consists of computing the optimum values for the candidate element parameters using a batch least-squares technique. The candidate element is fitted in such a way that it attempts to solve the entire input-output mapping problem by itself (except when the user wishes to use Projection Pursuit elements, as discussed below).

In the two fitting algorithms used by *ASPN-III*, only small subsets of network parameters are optimized at a given time; this reduces the dimensionality of the search space and improves the performance of the fitting algorithm. In most cases, it is sufficient to optimize only the parameters of a single element while holding all other elements fixed. This is the primary fitting algorithm in *ASPN-III*. Ivakhnenko [1968, 1971] was the first to propose this type of network construction. In Ivakhnenko's procedure, the parameters of *each element* are optimized in such a way that it attempts to solve the *entire* input-output mapping problem.

Ivakhnenko's fitting method is powerful, but has been improved. In particular, the *ASPN-III* projection-pursuit (secondary) fitting algorithm modifies the elements on a given layer *so that they work in linear combination with other elements in that layer* to minimize the objective

function. This is accomplished using a backfitting technique inspired by the projection-pursuit algorithm of Friedman *et al*. [1974, 1981, 1988, 1988]. In this strategy, an additional set of "dummy" parameters, β_1, ..., β_n, multiplies the outputs of the n elements on a given layer (Figure C.6).

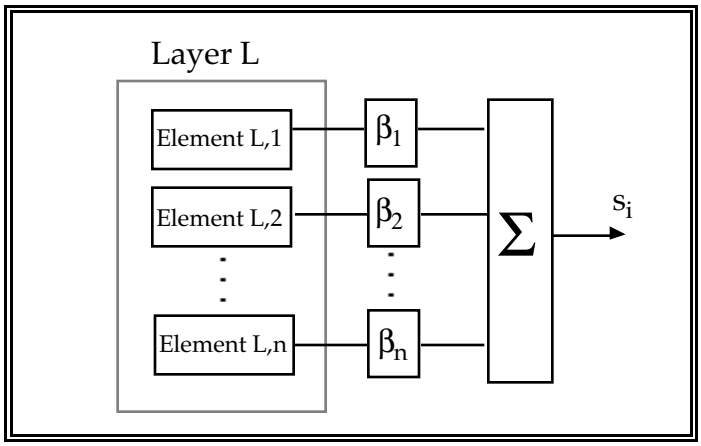


Figure C.6: Backfitting Strategy

The parameters of the node under consideration, along with the additional dummy parameters, are optimized together so that the weighted sum of element outputs minimizes the fitting squared error. This has the effect of training each new element to work well in combination with the existing elements of a given layer. Additional nodes are added to a layer only when their additional complexity is justified.

Entire layers may be optimized following a strategy originally used by A.R. Barron for Adaptronics, Inc. in *PNETTR* in the 1970s, and most recently suggested by Breiman and Friedman [1985]. In backfitting, each parameter subset is improved by iterating the search algorithm a few steps while holding the rest of the parameters fixed. This method is then repeated for another subset of network parameters, etc. For neural network based estimation, the nodal elements become the logical choice for the parameter subsets to be optimized, and a layer may be optimized by successively recursing through each nodal element, iterating the fitting algorithm a few times for each element. Breiman and Friedman showed that under appropriate conditions this method will yield the same parameter values as are obtained via a successful global optimization of the same structure. Practical implementation of the backfitting strategy has an advantage in that only a small set of linear equations needs to be solved at any given time.

An example of the way in which backfitting is applied can be illustrated using a network as defined in Figure C.6. Once the structure of the layer has been determined, the parameters of element L,1 and the dummy parameters, $\underline{\beta}$, are adjusted using one iteration of the fitting algorithm. Next the parameters of element L,2 and $\underline{\beta}$ are adjusted using one iteration of the fitting algorithm. This process continues n times until the parameters of element L,*n* have been adjusted. At this point, the process begins again with element L,1. The optimization routine continues until the optimization no longer improves performance significantly.

Another way that backfitting can be used is during the search for network structure. Elements may be backfitted each time a new element is added, and the new element can be scored based

on its performance in conjunction with the backfitted prior elements. In general, backfitting will increase training time, but it is a technique that can be used as often or as seldom as desired. Even when used to a small extent, backfitting can be a highly efficient way of optimizing larger sets of parameters so that they work well together.

Once the structure of a given layer is determined, subsequent layers have the option of combining the layer outputs linearly using the $\underline{\beta}$ coefficients chosen above, or they may go on and recombine the outputs in more complex ways if the improved performance justifies the additional complexity. Layers are added one at a time in this fashion until overall network growth stops when the modeling criterion (PSE) has reached a minimum.

ITERATIVE GAUSS-NEWTON OPTIMIZATION

At this point, the structure of the generalized networks has been completely specified in terms of nodal elements (Figure C.2), their interconnections (Figure C.1), and the objective function (10). Given this information, a regularized iterative least-squares (ILS) method for optimizing generalized nonlinear networks can be derived. The algorithm is iterative in the sense that multiple passes through the data are usually required to achieve convergence. It is a least-squares method in the sense that it minimizes a local quadratic approximation of the objective function; it does *not*, however, require that a squared-error distortion function be used or that the network equations be linear in the parameters. It is regularized in the sense that the condition of the pseudo-Hessian is monitored and adjusted to reduce numerical errors.

Let $\underline{\underline{\nabla f}}(\underline{x}_i, \underline{\theta}_0)$ be the gradient of the network output with respect to the element coefficients, $\underline{\theta}$, evaluated at $\underline{\theta}_0$ and abbreviated $\underline{\underline{\nabla f}}_{\underline{\theta}_0}$. Element i,j of the matrix $\underline{\underline{\nabla f}}_{\underline{\theta}_0}$ is the gradient of the *i*th network output with respect to the *j*th network coefficient.

Recall that each nodal element is the composition of a transformation h(z), with a series expansion, $z(\underline{x}, \underline{\theta})$, that is *linear in its coefficients*. Therefore, $\underline{\underline{\nabla f}}_{\underline{\theta}_0}$ may be computed via the chain rule as follows:

$$\frac{\partial f}{\partial \underline{\theta}} = \frac{dh}{dz}\left(\frac{\partial z}{\partial \underline{\theta}} + \frac{\partial z}{\partial \underline{x}}\frac{\partial \underline{x}}{\partial \underline{\theta}} \right) = \frac{dh}{dz}\left(\Phi_j(\underline{k}_j, \underline{x}) + \frac{\partial z}{\partial \underline{x}}\frac{\partial \underline{x}}{\partial \underline{\theta}} \right) \tag{18}$$

Note, that because the input to the node, $\underline{x}$, may be the output of other nodes (either feedforward or recurrent), (18) may also be used to compute $\partial \underline{x}/\partial \underline{\theta}$. If the network is recurrent, the gradients must be computed sequentially while stepping through the training database.

Once $\underline{\underline{\nabla f}}_{\underline{\theta}_0}$ has been computed, it may be used to make a local linear approximation of the network function about $\underline{\theta}_0$:

$$\underline{f}(\underline{x}_i, \underline{\theta}) = \underline{f}(\underline{x}_i, \underline{\theta}_0) + (\underline{\underline{\nabla f}}_{\underline{\theta}_0})^T(\underline{\theta} - \underline{\theta}_0) + \text{H.O.T.} \tag{19}$$

Because the general form of the method is iterative, we wish to find a $\Delta\underline{\theta}$ such that the iteration

$$\underline{\theta} = \underline{\theta}_0 + \mu\, \Delta\underline{\theta} \tag{20}$$

produces a minimum of the loss linearized about $\underline{\theta}_0$. If μ, the parameter that controls the step size, is taken to be unity and higher order terms are ignored, then (19) may be rewritten

$$\underline{f}(\underline{x}_i, \underline{\theta}) \cong \underline{f}(\underline{x}_i, \underline{\theta}_0) + (\underline{\underline{\nabla}} \underline{f}_{\underline{\theta}_0})^T(\Delta\underline{\theta}) \tag{21}$$

Now, let $\underline{\nabla} d(\underline{y}_i, \underline{f}_{i,0})$ and $\underline{\underline{\nabla}}^2 d(\underline{y}_i, \underline{f}_{i,0})$ be the $C \times 1$ gradient and $C \times C$ Hessian, respectively, of the distortion function with respect to the $C \times 1$ vector of network outputs, $\underline{f}_i$, at observation, i, and evaluated at $\underline{f}_i = \underline{f}_{i,0}$. These are abbreviated $\underline{\nabla} d_{\underline{f}_0}$ and $\underline{\underline{\nabla}}^2 d_{\underline{f}_0}$, respectively.

Because restrictions are put on the objective function such that it is everywhere twice-differentiable, the gradient and Hessian are known everywhere and can be used to make a local quadratic approximation of the loss function in the vicinity of the current network output, $\underline{f}_0$:

$$d(\underline{y}_i, \underline{f}_i) \cong d(\underline{y}_i, \underline{f}_0) + (\underline{\nabla} d_{\underline{f}_0})^T(\underline{f}_i - \underline{f}_0) + \frac{1}{2}(\underline{f}_i - \underline{f}_0)^T(\underline{\underline{\nabla}}^2 d_{\underline{f}_0})(\underline{f}_i - \underline{f}_0) \tag{22}$$

Using the local linear approximation of $\underline{f}_i$ given in (21), (22) becomes:

$$\begin{aligned} d(\underline{y}_i, \underline{f}_i) \cong\ & d(\underline{y}_i, \underline{f}_0) + (\nabla d_{\underline{f}_0})^T(\nabla \underline{f}_{\underline{\theta}_0})^T(\Delta\underline{\theta}) \\ & + \frac{1}{2}(\Delta\underline{\theta})^T\left((\nabla \underline{f}_{\underline{\theta}_0})(\nabla^2 d_{\underline{s}_0})(\nabla \underline{f}_{\underline{\theta}_0})^T\right)(\Delta\underline{\theta}) \end{aligned} \tag{23}$$

The total empirical loss, J, may then be calculated by summing the approximation of the distortion function over all observations:

$$J(\underline{\theta}) = J(\underline{\theta}_0) + \underline{b}^T(\Delta\underline{\theta}) + \frac{1}{2}(\Delta\underline{\theta})^T \underline{\underline{A}} (\Delta\underline{\theta}) \tag{24}$$

where

$$\underline{\underline{A}} = \frac{1}{N}\sum_{i=1}^{N} (\nabla \underline{f}_{\underline{\theta}})(\nabla^2 d_{\underline{f}})(\nabla \underline{f}_{\underline{\theta}})^T \tag{25}$$

and

$$\underline{b} = \frac{1}{N}\sum_{i=1}^{N} (\nabla \underline{f}_{\underline{\theta}})(\nabla d_{\underline{f}}) \tag{26}$$

It is now possible to calculate an approximate gradient of the empirical loss function with respect to the coefficient vector $\underline{\theta}$:

$$\nabla J_{\underline{\theta}} = \underline{b} + \underline{\underline{A}} (\Delta\underline{\theta}) \tag{27}$$

Because the loss function is required to be convex, the minimum is found at the point where the gradient is zero. Thus, (27) may be solved for $\Delta\underline{\theta}$ by the choice

$$(\Delta\underline{\theta}) = -\underline{\underline{A}}^{-1} \underline{b} \tag{28}$$

Thus

$$\underline{\theta}_{new} = \underline{\theta}_{old} - \mu \underline{\underline{A}}^{-1} \underline{b} \tag{29}$$

is the desired iteration.

To use the ILS optimization technique, the analyst must provide the following:

- An analytic form of the first and second partials of the objective function with respect to the network outputs, $\underline{\nabla} d_{\underline{s}}$ and $\underline{\underline{\nabla}}^2 d_{\underline{s}}$.
- An analytic form for the first derivative of the post-transformation h(z).
- an analytic form for the gradient of an element output with respect to its input vector, $\underline{\nabla} f_{\underline{x}}$.

Regularization

Experience has shown that, for non-quadratic objective functions, Newton methods may be unreliable, especially if the coefficients are initialized far from the minimum. This is because techniques for solving the system of equations in (28) break down when the pseudo-Hessian matrix, $\underline{\underline{A}}$, becomes singular or nearly singular. Regularization techniques are methods that can be used to ensure that $\underline{\underline{A}}$ is positive-definite. Many techniques can accomplish this and still provide an iteration that is only slightly different than the optimal Newton direction. One such technique is the Levenberg-Marquardt (LM) method [Marquardt, 1963; Press, 1986]. LM can be incorporated into the ILS algorithm in a straightforward fashion.

The matrix $\underline{\underline{A}}$, as defined by (29), is square and positive-semi-definite. One way of ensuring that $\underline{\underline{A}}$ is positive-definite is simply to add some small positive values to the diagonals. Thus, at each iteration, $\underline{\underline{A}}$ may be modified using one of the following methods:

$$\underline{\underline{A}}' = \underline{\underline{A}} + \lambda \underline{\underline{I}} \tag{30}$$

or

$$\underline{\underline{A}}' = \underline{\underline{A}} + \lambda \, \mathrm{diag}(\underline{\underline{A}}) \tag{31}$$

where λ is a positive constant, $\underline{\underline{I}}$ is the identity matrix, and $\mathrm{diag}(\underline{\underline{A}})$ denotes the matrix $\underline{\underline{A}}$ with all but its diagonal elements set to zero. When λ is large, the second term in the above equations dominates, and the iteration steps along the gradient (27); in this way, the algorithm becomes equivalent to the popular LMS backpropagation algorithm. When λ is small or zero, the first term in the above equations dominates, and the iteration becomes a Gauss-Newton iteration. There are a number of heuristic schemes for varying λ during the course of the search so that $\underline{\underline{A}}'$ remains positive-definite and the algorithm converges rapidly [Elder, 1991].

Because ILS is a variation of Gauss-Newton techniques, recursive forms can be derived that allow fast, efficient updating in a manner similar to recursive least squares and Kalman filters [Ward, 1993].

CONCLUSIONS

GNOSIS is built using both a generalized network structure and a generalized nonlinear learning algorithm capable of implementing many types of network transformations for a variety of purposes. By making specific choices concerning the nodal element structure, the distortion function, and the optimization algorithm parameters, one may use the algorithm to implement a variety of existing neural network paradigms, including polynomial neural networks (PNNs), multi-layer perceptrons (MLPs), and radial basis functions (RBFs). However, in addition to providing insight into existing artificial neural network paradigms by placing them in a broader context of generalized function estimation, the power of the technique lies in its ability to generalize and combine aspects of a variety of algorithms in new ways that may be uniquely appropriate for specific applications.

D

SYNTAX AND HELPSCREENS

Following is the syntax screen displayed for the -help function of *DB*:

```
************************************************
# DB Version 2.4
# Copyright 1989-1998, Barron Associates, Inc.
# All rights reserved
************************************************
# Cmdline: db -help
# The following options are available in DB's command line:
#
# -calc file name=expr [name=expr ...] [-a] -out file
# -clip file var min max [var min max ...] -out file
# -corr file [var ...]
# -delay file var d1...dN [var d1...dN] [-a] -out file
# -help [-option]
# -histo file [var bins/width ...] [bins/width]
# -merge file [file ...] [-obs] -out file
# -random [file] [-n num] [-name var] [-unif lo hi] [-gauss mean std] [-f]
      [-seed x] -out file
# -sequence [file] [-n num] var start step [var start step ...] [-block]
      -out file
# -select file [-del] [var ...] [b.o[-b.o] ...] [-range var lo hi] [-thin n]
      -out file
# -sort file var [var var] [-r] [-block] -out file
# -split file [-obs p] [-block p] [-seed x] -out filebase
# -stats file [var ...]
# -zero file var min max [var min max ...] [-r] -out file
#
# Multiple DB commands may be executed in sequence by recording
# them in the file db.cmd, using '#' as a comment character.
# Invoking DB without options executes the commands in db.cmd.
```

Following are the syntax screens displayed for each function of *DB*:

```
# -calc file name=expr [name=expr ...] [-a] -out file
#
# This option calculates new variables from existing ones. Use the -a
# option to append the new variables to the original database; otherwise
# the variables are written to the output alone.  Available operators are:
#
# Monadic Operators:
#   sqrt  abs  exp  sin  cos  tan  asin  acos  atan log log10 round
#
# Dyadic Operators:
#     +  -  *  /  ^ %
#
# NOTE: Numeric format e and unary minus are not implemented.
#       Trigonometric operators require radian arguments.
#       White space is not allowed within the name=expression string.
#
# Example: db -calc cannon.dat xdot=velocity*cos(gamma) -a -out cannon.out
# Append new variable xdot into cannon.out, computed from velocity and gamma
# in cannon.dat.
```

```
# -clip file var min max [var min max ...] -out file
#
# This option clips variable values. Values less than min are set to the min
# value, and variables greater than max are set to the max value specified.
# Min must be <= max; use '$' for either max or min to specify no min or max
# clipping respectively.
#
# Example: db -clip cannon.dat gamma 20.0 50.0 range 30000 $ -out cannon.out
# Limit gamma to the range 20..50; limit range to values at or above 30000.
# Copy all other variables without modification from cannon.dat to cannon.out.
```

```
# -corr file [var ...]
#
# This option determines the linear correlation between variables by calcu-
# lating a matrix of correlation coefficients.  If no vars are specified,
# the correlation for every permutation of two variables in the specified
# file will be calculated.  The linear correlation coefficient formula is:
#
#     r = summation(xy) / sqrt[summation(x^2) * summation(y^2)]
#     where x = X - mean of X, and y = Y - mean of Y
#
# Example: db -corr cannon.dat velocity gamma
# Display correlation between velocity and gamma of cannon.dat.
```

```
# -delay file var d1...dN [var d1...dN] [-a] -out file
#
# This option allows the user to produce delay/advance observations for one
# or more variables in a database.  Multiple non-zero positive or negative
# delays for each variable may be requested.  The delay outputs are labeled
# in order 'var_Dx' for each positive delay, 'var_Ax' for each negative delay
# (advance), where x = abs(delay).  Use the -a option to append the delay
# outputs to the original database; otherwise the variables are written to
# the output alone.
#
# NOTE: the first max_delay observations and last min_delay observations
# are deleted from each block of the output database, where max_delay is
# largest delay > 0 and min_delay is smallest delay (largest advance) < 0.
# If a block has insufficient observations, it is deleted entirely.
#
# Example: db -delay cannon.dat velocity -1 1 4 gamma -2 -a -out cannon.out
# Append delayed variables D1 and D4 for velocity, advance variables A1 for
# velocity and A2 for gamma into cannon.out.  First 4 and last 2 observations
# of every block in cannon.dat are not copied to cannon.out.  For all other
# observations, velocity_D1 is the value of velocity from the previous
# observation, velocity_A1 is velocity from the following observation, etc.




# -help [-option]
#
# This option displays a list of the available db options and the proper
# usage of each option.  If the suboption [-option] is specified, the
# usage and detailed description for the specified option is displayed.
#
# Example: db -help -calc
# Display help screen for calc function.




# -histo file [var bins/width ...] [bins/width]
#
# This option displays a histogram table of one or more variables.  The data
# for each variable is sorted and then separated into bins of width > 0
# specified for that variable.  If the specifier has a decimal point, it is
# used as bin width; otherwise the specifier is the number of bins > 0 and
# width is computed = (max-min)/bins.  If no variables are listed, histograms
# for all database variables are displayed using a single specified bin width.
#
# Example: db -histo cannon.dat velocity 5 range 10000.0
# Display number and percentage of observations in each of 5 bins for
# velocity, and in bins of width 10000 for range.
#
# Example: db -histo cannon.dat 10
# Display a 10-bin histogram table for each variable in cannon.dat.
```

```
# -merge file [file ...] [-obs] -out file
#
# This option merges two or more databases into a single database.  By default
# variables are merged column-wise. The number of blocks and observations must
# be the same for each input database and variable names should be unique.
# Use -obs to merge observations row-wise in separate blocks. The number of
# variables must be the same for each input database and variable names come
# from the first database.
#
# Example: db -merge r.dat s.dat t.dat -out merge.dat
# Merge the variables in r.dat, s.dat, and t.dat column-wise into merge.dat.
#
# Example: db -merge r.dat s.dat -obs -out merge.dat
# Merge the observation blocks in r.dat and s.dat row-wise into merge.dat.




# -random [file] [-n num] [-name var] [-unif lo hi] [-gauss mean std] [-f]
      [-seed x] -out file
#
# This option creates a column of pseudo-random numbers in a new file, or
# following other columns in an existing file.  Choose either uniform or
# gaussian distribution.  The suboptions available are:
#
#   -n num          number of observations > for a new file
#                   Not used if input file is given
#   -name var       variable name for the random numbers; default 'Random'
#   -unif low high  random number in uniform distribution between low and high
#   -gauss mean std random number in gaussian distribution with the given mean
#                   and standard deviation > 0
#   -f              random numbers are floats; default integers
#   -seed x         seed for the random number generator. 'x' is integer > 0
#                   or the string 'clock'
#
# Example: db -random -n 25 -gauss -10 10 -out cannon.out
# Output 25 random integers into cannon.out with mean = -10 and standard
# deviation = 10.  Random sequence is repeatable, due to fixed seed.
#
# Example: db -random cannon.dat -unif 0 100 -f -seed clock -out cannon.out
# Append variable Random into cannon.out with floating point values uniformly
# distributed between 0 and 100.  Repeated random sequences are unique, due
# to seeding by clock.




# -sequence [file] [-n num] var start step [var start step ...] [-block]
      -out file
#
# This option creates one or more columns of sequential numbers in a new file,
# or following other columns in an existing file.  Use the -n option to give
# the number of observations for a new file.  For each column, labeled 'var',
# the sequence begins at number 'start' and increments by number 'step'.  For
# existing files, 'start' may be a database variable; the sequence begins at
```

```
# the first observation value of the named variable.  The -block option for
# existing files causes the sequence to restart at every database block.
#
# Example: db -sequence -n 25 Obs 1 1 -out cannon.out
# Output variable Obs with 25 values {1, 2, .., 25} into cannon.out
#
# Example: db -sequence cannon.dat vel0 vel 0 test 0 -0.1 -block -out
      cannon.out
# Append variables vel0 and test into cannon.out.  For each block, vel0 is a
# constant = vel value in first observation of the block.  test is a linear
# sequence = {0.0, -0.1, -0.2, ...} which restarts at 0.0 for each block.




# -select file [-del] [var ...] [b.o[-b.o] ...] [-range var lo hi] [-thin n]
      -out file
#
# This option selects variables and/or observations from a database for
# deletion (-del option) or extraction (default).  If a variable is specified,
# the entire variable column is selected.  A number by itself, 'b.o', selects
# observation o of block b, counting from 1.  Observation ranges are specified
# with a hyphen, 'b.o-b.o', where b = block and o = observation number of the
# beginning and end of the range.  Use '$' for any b or o to specify the first
# or last block or observation in a range. For a single-block database, 'b.'
# may be omitted and is presumed to be 1.
#
# Other selection suboptions are:
#   -range var lo hi select based on value of one variable
#   -thin n          select 1st obs and every nth obs thereafter, n > 1
#
# Example: db -select cannon.dat -del gamma 1.10 5.5-$.$ -out cannon.out
# Delete gamma column, 10th observation in first block, and from observation 5
# of block 5 to the end of the database.  Remaining variables and observations
# are copied from cannon.dat to cannon.out.




# -sort file var [var var] [-r] [-block] -out file
#
# This option sorts a database by one to three variables. To start, the
# first variable is sorted.  If there are equal values, the second variable
# is sorted, etc.  Use the -r option to sort in reverse.  Use the -block
# option to sort the data within each block, retaining the block structure.
# Otherwise, a multi-block database is sorted into a single-block output.
#
# Example: db -sort cannon.dat range -r -out cannon.out
# Sort cannon.dat by descending range values into cannon.out.




# -split file [-obs p] [-block p] [-seed x] -out filebase
#
# This option splits a database into two files, either by blocks or by
# observations.  The splitting ratio 'p' gives the integer number of blocks
```

```
# or observations, or the fraction between 0.0 and 1.0 of the database to
# split into 'file.tra'.  The unselected part of the database is split into
# 'file.eva'.  The suboptions are:
#
#    -obs p     split by observations, ignoring block structure.
#               'p' is # obs >0 & <tot_obs or fraction >0.0 & <1.0
#    -block p   split by whole blocks; choose either -obs or -block.
#               'p' is # blocks >0 & <num_blocks or fraction >0.0 & <1.0
#    -seed x    seed for the random number generator. 'x' is integer > 0
#               or the string 'clock'
#    -out file  specify the basename of the two partial databases:
#               'file.tra' and 'file.eva'
#
# Example: db -split cannon.dat -obs 0.8 -out cannon
# Randomly split 80% of the observations in cannon.dat into cannon.tra. Split
# remaining 20% into cannon.eva.  Split is repeatable due to fixed seed.
#
# Example: db -split cannon.dat -block 10 -seed clock -out cannon
# Randomly split 10 of the blocks in cannon.dat into cannon.tra. Split
# remaining blocks into cannon.eva.  Repeated splits are unique, due to
# seeding by clock.




# -stats file [var ...]
#
# This option displays statistical quantities for one or more variables.
# The statistics include: min, max, mean, and median for both signed and
# absolute values, standard deviation and variance.  If no variables
# are specified, statistics for all are displayed.
#
# Example: db -stats cannon.dat gamma
# Display statistics for gamma from cannon.dat.




# -zero file var min max [var min max ...] [-r] -out file
#
# This option sets any values outside the specified min and max to zero.
# Min must be <= max; use '$' for either max or min to specify no min or max
# zeroing respectively.  Use option -r to reverse the operation and zero
# inside the specified range.
#
# Example: db -zero cannon.dat gamma 20.0 50.0 range 30000 $ -out cannon.out
# Set all gamma values outside the range 20..50 and ranges below 30000 to 0.
```