Pergamon

Computers & Geosciences Vol. 22, No. 9, pp. 1033–1049, 1996
Copyright © 1996 Elsevier Science Ltd
Printed in Great Britain. All rights reserved
0098-3004/96 $15.00 + 0.00

PII: S0098-3004(96)00042-8

A XERION-BASED PERL PROGRAM TO TRAIN A NEURAL NETWORK FOR GRID PATTERN RECOGNITION

JEHNG-JUNG KAO

Institute of Environmental Engineering, National Chiao Tung University, Hsinchu, Taiwan 30039, R.O.C.
e-mail: jjkao@green.ev.nctu.edu.tw

(Received 27 April 1995; revised 5 December 1995)

Abstract—Neural network training with the back propagation algorithm is an important artificial intelligence technique for grid pattern recognition. The training is time-consuming however and generally requires a trial-and-error procedure to configure the network. A Perl program executed with Xerion is presented to relieve the training burden. Statistical reports such as computation time, learning performance, and validation performance are generated automatically by the program. A case study applying the program for training networks to determine a drainage pattern from Digital Elevation Model data is demonstrated and discussed. Manually determining drainage patterns from topographical maps for a grid-based model is tedious and subjective. The neural network has a self-learning capability that can replace human judgment involved in the conventional approach. Copyright © 1996 Elsevier Science Ltd

Key Words: Neural network, Grid pattern, Digital Elevation Model, Drainage pattern.

INTRODUCTION

Human judgment to determine a grid pattern for a geographical feature from a topographical map, image, or aerial photograph, involves not only the attributes of a grid cell but also the spatial distribution of adjacent grid cells. Such a manual approach is time-consuming and generally subjective. A program in a conventional language (e.g. C or FORTRAN) or an expert system is used typically to replace such a manual determination. The spatial distribution and relationships among grid attributes, however, introduce complexity for a program or expert system to consider numerous variants of grid patterns. Such complexity may make the program or expert system inefficient compared to the conventional manual method. A neural network method thus is explored for its potential applicability.

Several difficulties may occur during implementation, even though the neural network method is now studied widely. Training a network to learn from a set of training data may require extensive computation before convergence to an optimum. Local optima instead of the global optimum may occur in determining a set of weights for network links during training. Another major difficulty is that no theory is available yet to construct an appropriate network configuration for a general problem, even though the optimum number of hidden nodes in the training cycle have been determined in previous research (e.g., Mirchandani and Cao, 1989). The optimal configuration of a neural network is still an important research issue. A trial-and-error procedure is employed generally by applying various learning methods and testing several different configurations. Such a process is however time-consuming and tedious. Preparation of training reports from numerous results obtained by applying different network training strategies and configurations are also cumbersome.

This paper presents a program written in Perl (Wall and Schwartz, 1992) to apply Xerion (Camp, Plate, and Hinton, 1993), a neural network package, for training neural networks with various methods and network configurations in a batch fashion. The user can leave the program as a background job and results are summarized automatically. Intermediate training progress is reported to the user through an electronic mailer. It is believed that a neural network study can be implemented effectively with the program. A case study applying the program to train and to find a neural network for determining a drainage pattern from Digital Elevation Model (DEM) data is demonstrated.

OVERVIEW OF BACK PROPAGATION NEURAL NETWORKS AND XERION

An artificial neural network is made up a number of layers of highly interconnected simple processors. A neural network generally is trained with a set of training patterns of which both input stimuli and corresponding output response are known. Once the network is trained completely, it is then applied as a transfer function to provide desired output patterns from given untrained external inputs. The advantages of a neural network include generalization, massive parallel processing, self-organization, etc. A good network trained from a proper set of training cases is capable of giving the appropriate response to a given input pattern, even if a certain degree of noise exists within the input pattern. A well-trained network can make generalizations from learned training patterns similar to an unfamiliar input pattern to produce a desired output response without user intervention or supervision. These advantages make it attractive to build a neural network for grid pattern recognition.

In this study, the popular three-layer network, as shown in Figure 1, with input, hidden, and output layers is used. Nodes are interconnected with numerous links whose strengths are expressed by numerical values named weights. Each node is activated from a function of the sum of the inputs received from other nodes through the weighted links. For a neural network to learn a set of training patterns involves modifying the weights via a learning algorithm. The back propagation approach applied frequently for this learning purpose was used in this study.

As this approach is described in detail by Rumelhart and others (1989), only a brief description follows. The implementation includes two stages. In the first stage, the output vectors obtained from a neural network according to the input vectors of a given training set are determined based on an initial weight set. Comparison of output vectors with the desired output ("target") vectors is made at the end of the first stage. A back propagation learning procedure, the second stage, is then initiated if differences (or "errors") result from the comparison. The differences are used as error signals to propagate backward through the network to correct the weights. This two-stage procedure is repeated until all the given training patterns are learned correctly.

The following summed square function is used to measure the error:

$$E = \sum_p E_p = \sum_p \sum_i \left(t_{pi} - o_{pi}\right)^2 \tag{1}$$

where p ranges over all input patterns, i ranges over output nodes, E_p is the error on pattern p, t_{pi} is the

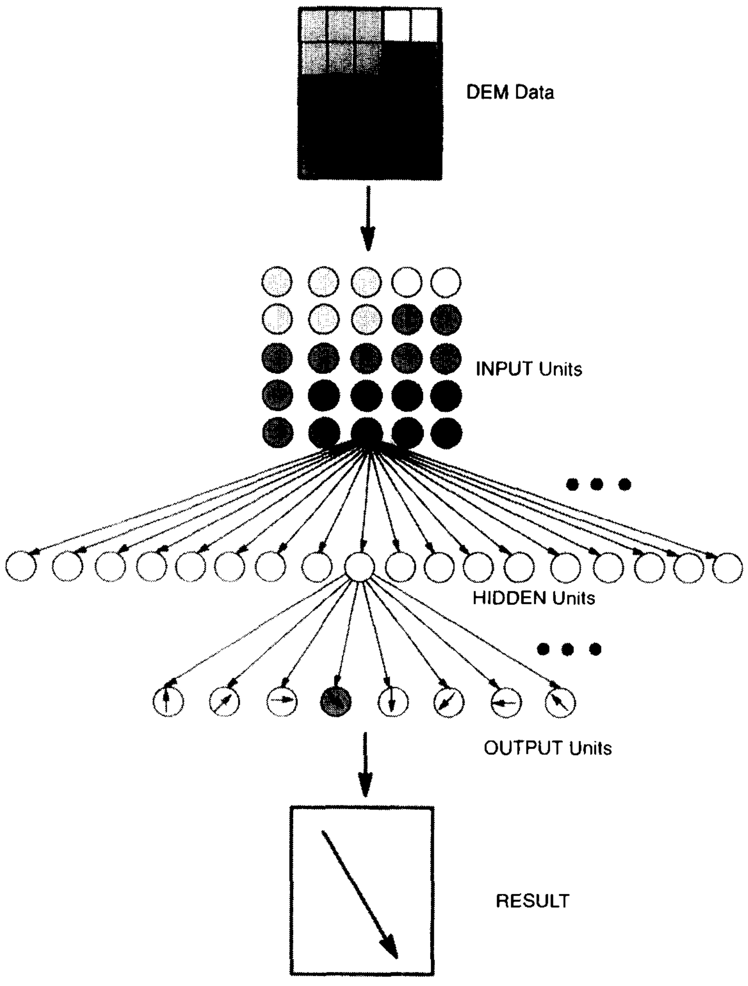


Figure 1. Neural network configuration and data processing flows.

target value of a node, and O_{pi} is the output (activation) value of a node. A direct method such as steepest descent then can be applied to find the set of weights that minimizes this function. For example, the weight on each link (between nodes i and j) can be modified on the basis of the error signal and the output value, $\Delta_p W_{ji} = \eta \delta_{pj} o_{pj}$, where δ is the step size factor, p is the input pattern number, and δ is the error. For an output node, δ is computed with $\delta_{pj} = (t_{pj} - o_{pj}) f_j'(net_{pj})$, where $f_j'(net_{pj})$ is the derivative of an activation function, such as the logistic equation shown subsequently in this section. For a hidden node, however, there is no desired target, and the error signal is determined from $\delta_{pj} = f_j'(net_{pj}) \Sigma_k \delta_{pk} w_{kj}$, where δ_{pk} are the error signals of the nodes to which the hidden node directly connects and w_{kj} are the weights of the connections. This procedure is based on the method of steepest descent with a fixed step size. This method, however, may not perform well and other algorithms for nonlinear optimization such as the conjugate gradient method are applicable also.

Xerion is a publicly accessible neural network package developed by Camp, Plate, and Hinton (1993). A collection of C libraries is provided to implement many neural network paradigms. Besides the back propagation network, the package provides also the Boltzmann machine, the mean-field-theory machine, hard and soft competitive learning, the Kohonen, cascade correlation, and recurrent back propagation networks. The user interface, which is built on X-windows, provides a friendly environment for either a novice or an experienced user to implement interactive training. The bp module of the Xerion (Camp, Plate, and Hinton, 1993) system implements the back propagation learning algorithm. Other than the steepest-descent method, the package also provides the methods of momentum descent, conjugate gradient, Rudi's conjugate gradient, and others, described by Camp, Plate, and Hinton (1993); only a brief description follows.

Momentum-descent method

This method sets the direction to be the steepest descent with a momentum term. The momentum term is the previous direction multiplied by a decay factor, α, as shown in Equation (2).

$$\Delta w_{ij}(n+1) = \eta(\delta_\pi o_{pj}) + \alpha \Delta w_{ij}(n). \qquad (2)$$

This method is intended to decrease the amount of wandering in ravines of the error surface.

Conjugate-gradient method

This method, based on the method described by Press and others (1992), is expressed according to Equation (3):

$$S_{new} = S \frac{(G_{new} - G)\cdot G_{new}}{G \cdot G} - G_{new}, \qquad (3)$$

where S_{new} is the new search direction, S is the previous search direction, G_{new} is the current gradient, and G is the previous gradient.

Rudi's conjugate-gradient method

This method modified by Rudi Mathon (Camp, Plate, and Hinton, 1993) from the previous method is expected to provide rapid convergence. The search direction is expressed as:

$$u_1 = \frac{S \cdot G_{new}}{(G_{new} - G)\cdot(G_{new} - G)}$$

$$u_2 = \frac{(G_{new} - G)\cdot G_{new}}{(G_{new} - G)\cdot(G_{new} - G)} - \frac{2S \cdot G_{new}}{(G_{new} - G)\cdot S}$$

$$u_3 = \frac{S \cdot (G_{new} - G)}{(G_{new} - G)\cdot(G_{new} - G)}$$

$$S_{new} = u_1 {*} (G_{new} - G) + u_2 {*} S - u_3 {*} G_{new} \qquad (4)$$

Step methods

Once the search direction is determined, the step size should then be determined. Two types of step methods were provided by Xerion: a fixed step and a line search. The former uses a fixed size of step. A line search method is intended to find the minimum value of a function along a given search direction. For example, if W is the current weight set (starting position) and S is the search direction vector, the line search should find η that minimizes the error function $E(W + \eta\ S)$.

THE PERL SCRIPT PROGRAM

Perl (Wall and Schwartz, 1992) is a freely available script language to manipulate text, files, and computational processes that were implemented previously with complex programming in C, AWK (Aho, Kerninghan, and Weinberger, 1988), or a shell script language. The script program is written in Perl to operate with Xerion in a batch fashion. Although the interactive display provided with Xerion is useful for learning and monitoring training strategies, it is tedious to manipulate inputs and reports for massive training strategies using various learning methods, network configurations, and numbers of nodes in each layer. The pseudocode listed in Table 1 describes the execution flow of the script program listed in the Appendices. The user must provide a file including all training cases and a validation file including all testing cases to verify the results obtained from a trained network. Two other files must be provided also by the user to specify formats for reporting the result and validation outputs, shown in Appendices 2 and 3. The script program frequently is executed as a background job on a UNIX workstation. A typical command to run the script is listed next, although other commands such as `crontab` can be used also.

```
nohup perl doNet demNnet dem.train
dem.valid >& /dev/null
```

where `doNet` is the

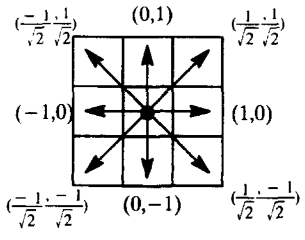
Table 1. Pseudocode of Perl script program, **doNet**

```
Usage: doNet netName trainingFileName validationFileName [#_of_input_nodes #_of_output_nodes]
main program:
    Set user-defined variables (or use default values): #_of_hidden_nodes, email address, etc.
    Initialization for this run, netName:
            create a directory named by netName for keeping information for this run.
            link files (use 'ln' instead of 'cp' to save space).
    Disable the Xerion X-window display.
    If the user provides a user specified reporting file, 'report.sub, include it into the script program.
    for each specified #_of_hidden_nodes do
    {
            1.set learning method and related parameters.
            2.call function doNet to implement a training and validation path.
            3. repeat 1. and 2. until all desired learning strategies are implemented.
    }
Function doNet:
    Set a unique identification for current training and validation path.
    Set input and output file names.
    Create a tag for monitoring the training progress.
    If the user's email address is provided, the tag is sent to the user.
    Implement the training stage:
    Write Xerion bp input command files by calling function writeNetIn.
    Implement the training by Xerion bp module in batch mode.
            Computation time is logged to a time report file.
    Implement the validation stage:
    Write Xerion bp input command files by calling function writeNetIn.
    Implement the validation stage by Xerion bp module in batch mode with weights obtained in the training stage. Computation time
is also logged to a time report file.
    Implement the user specified reporting function (reportsub) to generate statistical summary report.
Function reportsub:
    General steps:
    Extract computation time summary from the time report file created by doNet.
    Extract the first iteration and last iteration information from Xerion bp output.
    Problem dependent steps for the case study: (implemented by function compare)
    Report the summary of validation result for the training set.
    Report the summary of validation result for the validation set.
```

script file name, `demNnet` is the name of the current run, `dem.train` is the name of a training file, and `dem.valid` is the name of a validation file. Several such commands can be executed simultaneously for separate training or validation sets. A tag file is sent automatically to the user's e-mail address, if provided, to report training progress. Several files are created at each stage of training and validation including a summary report for CPU-time usage and learning performance.

A CASE STUDY

A case study to apply the program for training neural networks to determine drainage patterns from DEM data is described. The example is used primarily to demonstrate how the program works; for the complete results see Kao, 1992a. The drainage pattern of a watershed is an important parameter in non-point-source water quality modeling to determine hydrological runoff paths within a watershed. Manual preparation of this pattern from topographic maps is tedious and subjective. DEM data are obtained normally from stereoscopic aerial photographs, although a ground survey or radar scanning can provide alternative data sources. The altitude matrix (Burrough, 1986) of the point model is used in this research. The use of DEM data to determine micro-drainage patterns has been explored previously by other researchers (e.g. Band, 1986; O'Callaghan and Mark, 1984). These algorithms, although demonstrated successfully for generating a micro-drainage network from a set of DEM data, are unsuitable for generating drainage directions for a grid-based model such as AGNPS (a non-point source pollution model, described by Young and others, 1994), because approximating macro-scale trends from micro-variation data is difficult. For example, the DEM data used here have a pixel resolution 40 m $\times$ 40 m. Grid-based models, such as those used for water quality and regional planning, do not require such detailed topographical information. A typical cell size for AGNPS is 10 acres, which includes roughly 5×5 DEM pixels. The drainage direction of a grid cell is illustrated in Figure 2 is restricted to one of eight directions. Each direction points to an adjacent grid cell. The DEM data for micro-scale areas (pixels) are processed for approximating drainage trends of larger area, model grid cells. Unlike the research undertaken for a micro-scale drainage network, the micro-scale DEM



$(-\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}})$
(0,1)
$(\frac{1}{\sqrt{2}},\frac{1}{\sqrt{2}})$
(−1,0)
(1,0)
$(-\frac{1}{\sqrt{2}},-\frac{1}{\sqrt{2}})$
(0,−1)
$(\frac{1}{\sqrt{2}},-\frac{1}{\sqrt{2}})$

Figure 2. Possible drainage directions.

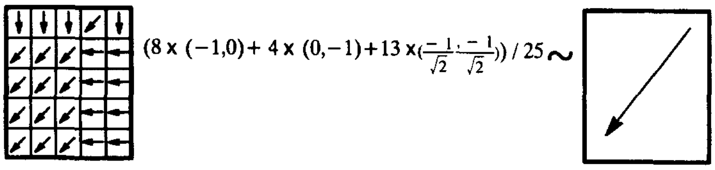

$(8 \times (-1,0) + 4 \times (0,-1) + 13 \times (\frac{-1}{\sqrt{2}}, \frac{-1}{\sqrt{2}}))/25 \sim$

Figure 3. Micro-scale variation vs. macro-scale trend (vectorial average method).

data in this research are converted into macro approximations of drainage trends. An example of applying the vectorial average method (Kao, 1992b) is provided in Figure 3, in which a grid drainage direction was approximated from 25 pixel directions. Some difficulties may be encountered during such approximations. For example, Figure 4 shows two grid cells with the same vectorial average but with different spatial distributions of micro-scale pixel directions. The drainage directions of the two grid cells, if manually determined, obviously differ. Methods developed previously for determining micro-scale drainage network cannot be used directly for macro approximations because of such difficulties. The detail of micro-variation vs. macro-scale trend is discussed by Kao (1992a, and 1992b). Human judgment involved in determining the drainage directions from a topographic map is difficult to model mathematically or with a computer code. Therefore the neural network method with self-learning capability was explored.

Six methods were developed by Kao (1992a, and 1992b) to approximate a drainage pattern with a FORTRAN program. The drainage network method is demonstrated to be superior to other methods. The following discussion is focused therefore on comparison of results obtained from the manual, drainage network and neural network methods. A subwatershed within Chi-Mei Creek in North Taiwan serves as the test area. The area is divided into 1962 200 m × 200 m grid cells. All DEM pixels within the test area are divided into groups of 5 × 5 pixels; each group represents a typical model grid cell used for AGNPS.

Manual method

A manual method for creating a drainage pattern from topographical maps based on visual inspection was first implemented before testing other methods.

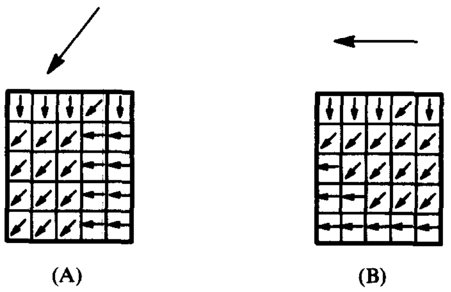


Figure 4. Two grid cells with same vectorial average.

The manual method was performed using two approaches. One was carried out by placing grid cell boundaries on the maps, and grid cells were examined sequentially so as to determine their drainage directions manually. This process was, however, time consuming. In the second method a three-dimensional display, a contour figure, and an elevation-based figure were provided, and for each grid cell the major drainage direction was determined from the local grid cells. Inherent bias exists during the application of this method (see discussion in Kao 1992a, and 1992b). The drainage pattern so determined is illustrated in Figure 5. The darker area in the figure indicates lower ground, and vice versa. The three dark grid cells are outlets of the subwatershed. The subwatershed is encircled with a darkest line, which was digitized from topographical maps.

Drainage-network method

Each grid cell, in using this method, is treated as a small watershed. The first step is to determine the drainage direction of each DEM pixel. Allowing the drainage direction to drain towards the adjacent pixel with the lowest elevation is the simplest way to determine the pixel drainage direction. The outlet pixel is then determined on the basis of all pixel drainage directions within a composite grid cell. The drainage direction of the outlet pixel is then used as the drainage direction of the whole grid cell. If no outlet is found due to a depression pixel existing at a pixel within the grid cell, the pixel is then filled by re-setting its elevation to be slightly greater than that of the adjacent pixel with minimal elevation. This process is repeated until an outlet is found. This process was adopted from the Apparent Elevation method proposed by Yuan and Vanderpool (1986). The outlet pixel, however, may not be unique; having more than one outlet pixel within a grid cell is possible. If such a problem arises, the outlet pixel into which the greatest number of upstream pixels drain is chosen as the outlet. Discussion of the differences among the results obtained from this method and others are provided later.

NEURAL NETWORK DEMONSTRATION BY THE PROGRAM

Network configuration and hidden nodes

The neural network configuration tested (Fig. 1) is constructed of three layers: 25 input nodes, several hidden nodes, and eight output nodes. Each input

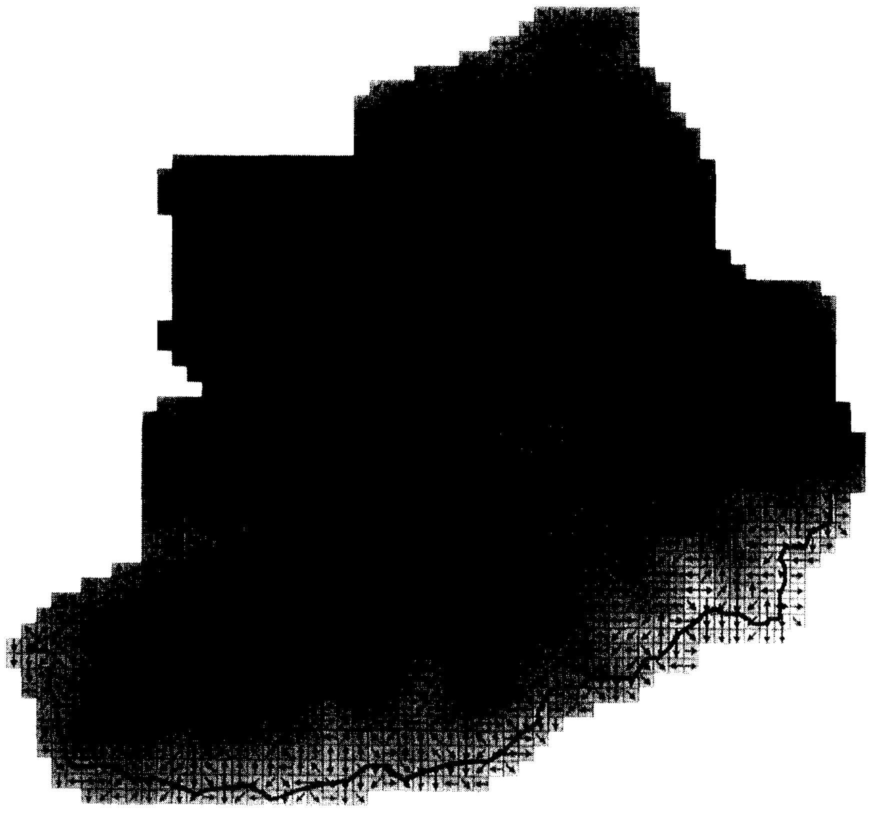


Figure 5. Manually determined drainage pattern.

node is associated with a DEM pixel of a grid cell, and each output node is associated with one of eight drainage directions shown in Figure 2. Networks with 2, 3, 5, 8, 10, 13, 15, 18, 20, 23, 26, 29, 32, 35, 38, 41, and 44 hidden nodes were tested, as defined by the Perl array `@hiddenSet` in the program.

Preparation of training sets

The selection of an appropriate training set is important for learning convergence and neural network performance. Of 1959 manually determined grid patterns, 160 grid patterns (less than 10%) were selected as a training set for the network. Training patterns were prepared in two sets as follows:

1. the entire grid cell set was divided into eight pools; the manually determined drainage direction of each grid cell in the same pool is the same;
2. grid cells are deleted from each pool if their drainage directions determined according to the manual and drainage network methods are not the same;
3. to form training set D1, 20 grid cells were selected randomly from each pool;
4. repeat step (3) to form training set D2.

The output node values lie between zero and unity. Only one output node of each training grid pattern is set to unity; all others are set to zero. The node with a nonzero output is associated with the drainage direction assigned for the input pattern of the training grid pattern. Other training sets such as those generated randomly from the entire grid cell set were tested also, but are not demonstrated here.

Normalization of input patterns

The normalization function shown in Equation (5) was used to compute the input value for each input node based on the associated DEM pixel elevation. Input values are between zero and unity.

$$I_i = \frac{n_i - n_{\min}}{n_{\max} - n_{\min}} *0.9 + 0.1 \qquad (5)$$

where I_i is the final input value to an input node, $n_i = (e_i - e^g_{\min})/(e^g_{\max} - e^g_{\min})*0.9 + 0.1$, e_i is the elevation of the current DEM pixel, $e^g_{\min}$ is the minimum elevation of all pixels within the tested area, $e^g_{\max}$ is the maximum elevation of all pixels within the area, $n^g_{\min}$ is the minimum of n_i and $n^g_{\max}$ is the maximum of n_i's of the 25 pixels within the current grid cell. The network was sometimes not completely trained with the normalized value n_i. Several obvious patterns, but with a small elevation, were observed to not be learned during many learning cycles. The magnitude of the elevation appeared to be learned by the network. The drainage pattern is governed primarily by the pixel elevation variation rather than by the magnitude. The normalized function (Eq. 5) is thus proposed to enlarge the variation of a flat area.

Direction and step methods

Direction and step methods in various combinations were tested (see the end of the program in Appendix 1 after START JOBS). The steepest descent method was tested for a step size equal to 0.1, 0.3, or 0.5 and the momentum-descent method was tested with a step size 0.1, 0.3, or 0.5, and decay rate 0.5, 0.7, or 0.9. In total 442 (2 training sets $\times$ 17 numbers of hidden nodes $\times$ 13) cases were tested according to steepest- and momentum-descent methods with fixed step sizes. Because these methods failed to learn the training set completely within 2000 iterations, the results are not reported here. These two methods were not tested further, because finding a good fixed step size is arbitrary, requires an extensive iterative procedure, and because the optimal solution may be missed. Only the methods of steepest descent (SD), conjugate gradient (CG), and Rudi's conjugate (RC) gradient with line search generated satisfactory results.

Network training and results

Each input value is computed by the normalized function (Eq. 5). All training grid patterns are fed through the network once, and the link weights are modified at the end of each epoch. The values of weights are not constrained. The network is trained in at most 2000 iterations (training stage). The 1959 grid drainage patterns of the test area are then determined by the trained network (validation stage). The drainage direction of each grid cell is set to be that associated with the output node with the maximum output value. All neural network training for this study was implemented with Xerion on a Sun Sparc workstation with the following two commands.

```
nohup perl doNet demNnet dem.trainDl
dem.valid >& /dev/null &nohup perl doNet
demNnetl dem.trainD2 dem.valid >& /dev/
null &
```

where `dem.trainDl` and `dem.trainD2` are names of files for the two sets of training cases, and `dem.valid` is the file name for the validation set. Several intermediate and summary files are created for each training and validation stage. The files include an input file to operate the Xerion bp module, a Xerion bp output file, a file to store link weight values, validation reporting files for training and validation sets, and a summary file. Table 2 shows an example of the Xerion bp input file created by the program for both training and validation stages. Table 3 shows a sample of the summary file for learning performance, validation performance, and CPU time usage. The summary file is created mainly

Table 2. Sample Xerion bp input file created by program

```
# sample input for demNnetD1__12__cg.in used for the training stage.
addNet "demNnetD1"
useNet "demNnetD1"
addGroup -type INPUT input 25
addGroup -type HIDDEN hidden 12
addGroup -type OUTPUT output 8
disconnectGroups "Bias" "input"
connectGroups input hidden
connectGroups hidden output
addExamples -type TRAINING dem.trainD1
seed 120
randomize 1
set currentNet.extension.zeroErrorRadius = 0
minimize -tolerance 1.0e-10 -iter 5 -cg -epsilon 0.3
saveWeights wt/demNnetD1__12__cg.wt
exit
# sample input for demNnetD1__12__cg.inV used for the validation stage.
open demNnetD1 > validate/demNnetD1__12__cg.validate
format UnitRec.target "%7.4f"
format UnitRec.output "%7.4f"
addTrace doExamples "print "
(trace.fmt listed in Appendix 3 is included here.)
@demNnetD1"
addNet "demNnetD1"
useNet "demNnetD1"
addGroup -type INPUT input 25
addGroup -type HIDDEN hidden 12
addGroup -type OUTPUT output 8
disconnectGroups "Bias" "input"
connectGroups input hidden
connectGroups hidden output
addExamples -type TRAINING dem.trainD1
loadWeights wt/demNnetD1__12__cg.wt
addExamples -type VALIDATION dem.test
doExamples -t VALIDATION
close demNnetD1
deleteStream -quiet demNnetD1
open demNnetD1 > validate/demNnetD1__12__cg.train
doExamples -type TRAINING
close demNnetD1
exit
```

Table 3. Sample list of summary file created by program

```
Report for demNnetD1__02__steepest__Ray on 941025232212
    TIME user = 188.3 sys = 2.3 real = 211.9
    TIMEV user = 65.4 sys = 7.2 real = 78.1
    iter =  0 nFE =  1 f =  380.65738 |g| =  1.2e + 02 d =   - 1.33e + 04 dr =  1
    iter =  2000 nFE =  2061 f =  65.805695 |g| =  0.77 d =   - 0.741 dr =   - 0.23
    TDiff 0:124 45:35 90:1 135:0 180:0 - >  160
    VDiff 0:1032 45:785 90:106 135:32 180:4 - >  1959...OK?...
Report for demNnetD1__02__cg__Ray on 941025232520
    TIME user = 82.2 sys = 0.8 real = 88.5
    TIMEV user = 65.4 sys = 5.9 real = 79.5
    iter =  0 nFE =  1 f =  380.65738 |g| =  1.2e + 02 d =   - 1.33e + 04 dr =  1
    iter =  525 nFE =  893 f =  61.628765 |g| =  0.32 d =   - 0.00542 dr =  1
    TDiff 0:122 45:35 90:3 135:0 180:0 - >  160
    VDiff 0:998 45:751 90:144 135:47 180:19 - >  1959...OK?...
...
Report for demORG__05__cgrudi__Ray on 941026000227
    TIME user = 99.4 sys = 1.4 real = 107.3
    TIMEV user = 64.8 sys = 5.8 real = 77.5
    iter =  0 nFE =  1 f =  315.37115 |g| =  1.3e + 02 d =   - 1.62e + 04 dr =  1
    iter =  471 nFE =  689 f =  2.0000029 |g| =  3.5e - 05 d =   - 9.3e - 07 dr =  1
    TDiff 0:158 45:2 90:0 135:0 180:0 - >  160 Hmmm...Try further...
    VDiff 0:1180 45:590 90:109 135:50 180:30 - >  1959...OK?...
...
```

by a user provided function call `reportsub`. A sample of `reportsub` for the case study is listed in Appendix 2 and its pseudocode is shown at the end of Table 1. The function `compare` called by `reportsub` is a problem-dependent function and must be modified if the problem is altered. This function compares the result from the validation stage with the desired one based on a comparison method provided by the user. The function appears complicated for this example because comparison of the results is not straightforward and requires data conversion from a numeric value to an angular value (unit: degree). For most problems that require direct comparison of numeric values, the function can be simplified significantly.

In total 102 (2 training sets $\times$ 3 methods (SD, CG, and RC) $\times$ 17 numbers of hidden nodes) cases, were trained. Table 4 shows all completely trained cases for each training set and method. A completely trained case is one that learns all the given training patterns within 2000 iterations. Cases with 2, 3, or 5 hidden nodes were unable to learn any of the tested training sets completely, consistent with the concept that more hidden nodes are required in learning a complex problem. The computation time and final errors for training the completely trained cases are summarized in Table 4. Comparisons are made on the basis of the difference between the computed and manual results reported according to the difference in the number of grid directions matching or 45°, 90°, 135°, or 180°. 1293 grid cells exactly match the manually determined ones, 573 grid cells have a 45° difference, 73 grid cells have a 90° difference, 40 grid cells have a 135° difference, and sixteen grid cells have a 180° difference. Grid cells at a 45° difference from the manually determined results may arise because during the manual process one or two alternative adjacent directions are observed frequently. Grid cells with a 45° difference from manual ones therefore are considered acceptable. There are 36 cases in which the computed and manual methods were different. However, this number was reduced greatly if directions off by 45° were considered acceptable.

CONCLUSION

Researchers seek automated methods for implementing conventional modeling tasks. A neural network with self-learning and self-organization capability is a promising technique to replace tasks that involve human judgment. The Perl program developed in this work is intended to relieve the burden on an analyst from implementing the conventional iterative neural network training procedure and research result reporting. A complex test such as the study demonstrated here can be implemented with a small number of nohup commands.

The result obtained from the neural network method applied to a case study is satisfactory in comparison to the manually determined drainage pattern, and is superior to other previously developed numerical methods. The selection of appropriately balanced training sets, sufficient hidden nodes, a suitable normalization method, and the best direction methods are the major factors in building an efficient neural network. These factors are explored readily with the program which can be executed in a background mode. With slight modification the program can be applied to explore neural network research of other types. All programs, including several data and report processing scripts developed in this work are available for public accesses. Information for obtaining them can be requested by e-mail addressed to environ@ev004.ev.nctu.edu.tw., or by anonymous FTP from IAMG.ORG.

Table 4. Completely trained cases

Training set	# of hidden units	Direction method	CPU time	# of iterations	SSE	# of different grids Difference in degrees					Good
						0	45	90	135	180	
For pattern determined by the drainage network method ⇒						1293	537	73	40	16	
D1	8	CG	1	414	3.26E − 05	1272	573	66	36	12	*
D1	10	SD	3.1	2000	0.304	1311	532	69	33	14	*
D1	10	CG	0.9	401	0.000117	1293	536	77	43	10	
D1	10	RC	1.3	308	5.35E − 06	1232	566	100	40	21	
D1	13	SD	4	2000	0.0345	1330	509	65	34	21	*
D1	13	CG	1.7	278	8.26E − 06	1288	536	73	36	26	
D1	13	RC	1.1	205	1.62E − 05	1287	555	63	42	12	*
D1	18	SD	4.5	2000	0.148	1328	511	65	37	18	*
D1	18	CG	1.2	356	2.99E − 05	1319	521	61	41	17	*
D1	18	RC	0.8	212	4.31E − 06	1334	517	57	32	19	*
D1	20	SD	7.6	2000	0.0963	1323	517	74	32	13	*
D1	20	CG	4.1	455	9.79E − 06	1302	540	79	29	9	*
D1	20	RC	2.2	363	8.00E − 06	1275	557	76	38	13	*
D1	23	SD	2.3	2000	0.342	1364	492	63	26	14	*
D1	23	CG	1.1	357	3.86E − 05	1335	509	65	35	15	*
D1	23	RC	0.9	253	4.20E − 06	1336	513	63	31	16	*
D1	26	SD	3.1	2000	0.0265	1351	488	65	35	20	*
D1	26	CG	0.9	416	1.93E − 05	1353	494	58	39	15	*
D1	26	RC	0.6	213	3.90E − 06	1363	477	63	43	13	*
D1	29	SD	3.2	2000	0.136	1364	486	57	31	21	*
D1	29	CG	1.1	453	3.48E − 05	1349	487	65	34	24	*
D1	29	RC	0.7	258	8.60E − 07	1326	519	61	31	22	*
D1	32	CG	2	580	1.96E − 05	1354	484	65	30	26	*
D1	32	RC	1	279	3.97E − 06	1364	494	54	33	14	*
D1	35	SD	2.1	2000	0.22	1377	481	53	28	20	*
D1	35	RC	0.8	265	4.52E − 06	1344	509	56	33	17	*
D1	38	SD	2.5	2000	1.98	1398	453	59	31	18	*
D1	38	RC	0.8	296	1.79E − 06	1344	495	68	34	18	*
D1	41	SD	2.2	2000	0.404	1364	483	62	28	22	*
D1	41	CG	1.1	534	5.02E − 05	1360	484	58	33	24	*
D1	41	RC	0.7	254	1.70E − 06	1360	489	55	31	24	*
D1	44	SD	2.8	2000	0.00609	1357	500	55	25	22	*
D1	44	CG	1.1	346	1.83E − 05	1350	507	58	24	20	*
D1	44	RC	1.4	221	4.70E − 06	1341	510	56	29	23	*
D2	8	SD	4.1	2000	0.863	1283	528	80	38	30	
D2	10	CG	1.5	465	7.98E − 05	1338	471	75	47	28	
D2	13	RC	1.4	297	8.65E − 07	1387	430	67	37	38	
D2	18	CG	2.4	476	2.70E − 05	1350	448	81	47	33	
D2	23	CG	0.9	421	4.87E − 05	1377	440	73	34	35	
D2	26	SD	2.8	2000	0.169	1397	428	57	40	37	
D2	26	CG	1.1	487	4.75E − 05	1373	440	68	39	39	
D2	26	RC	1.2	233	1.26E − 06	1367	473	58	33	28	*
D2	29	CG	3.5	480	2.20E − 05	1390	434	68	33	34	
D2	29	RC	1.9	303	9.62E − 07	1396	437	75	22	29	*
D2	32	CG	2.5	553	2.82E − 05	1387	445	59	42	26	*
D2	32	RC	1.5	275	9.66E − 06	1356	463	74	40	26	
D2	35	SD	6.1	2000	0.411	1407	425	60	33	34	*
D2	35	RC	1.4	234	1.77E − 06	1414	418	55	34	38	*
D2	38	SD	6	2000	0.0311	1393	426	59	40	41	
D2	38	RC	1.4	324	5.28E − 06	1375	443	64	41	36	
D2	41	SD	5.8	2000	0.379	1377	428	67	42	45	
D2	41	CG	2.8	418	1.93E − 05	1392	417	66	41	43	
D2	41	RC	1.7	217	1	1355	449	75	33	47	
D2	44	SD	7.8	2000	0.21	1381	435	64	39	40	
D2	44	CG	2.2	372	8.47E − 05	1376	436	62	40	45	

Acknowledgments—The author thanks Mr C. H. Tsai and Mr S. F. Bau for assistance in determining drainage pattern from DEM data manually, the National Science Council of Taiwan R. O. C. for providing partial financial support (NSC 82-0410-E-009-18), and to Camp, Plate, and Hinton (1993) for providing Xerion; without it, complicated research such as the case study could not be achieved with only two simple **nohup** commands.

REFERENCES

Aho, A. V., Kerninghan, P. J., and Weinberger, P. J., 1988, The AWK programming language: Addison-Wesley, Massachusetts, 210 p.

Band, L. E., 1986, Topographic partition of watersheds with digital elevation models: Water Resources Research, v. 22, no. 1, p. 15–24.

Burrough, P. A., 1986, Principles of geographical information systems for land resources assessment: Clarendon Press, Oxford, 194 p.

Camp, D., Plate, T., and Hinton, G., 1993, The Xerion neural network simulator version 3.1: Department of Computer Science, Univ. Toronto, 43 p.

Kao, J.-J., 1992a, Determining drainage pattern using DEM data for nonpoint source water quality modeling: Final Report to National Science Council, Taiwan, R. O. C. (NSC 80-0410-E-009-18), 84 p.

Kao, J.-J., 1992b, Determining drainage pattern using DEM data for nonpoint source water quality modeling:

Water Science and Technology, v. 26, no. 5–6, p. 1431–1438.
Mirchandani, G., and Cao, W., 1989, On the hidden nodes for neural nets: IEEE Trans. Circuits and Systems, v. 36, no. 5, p. 661–664.
O'Callaghan, J. F., and Mark, D. M., 1984, The extraction of drainage networks from digital elevation data: Computer Vision, Graphics, and Image Processing, v. 28, p. 324–344.
Press, W. H., Teukolsky, S. A., Vettlerling, W. T. and Flannery, B., 1992, Numerical recipes in C: Cambridge University Press, New York, 994 p.
Rumelhart, D. E., McClelland, J. L., and the PDP research group, 1989, Parallel distributed processing: MIT Press, Cambridge, Massachusetts, 547 p.
Wall, L., and Schwartz, R. L., 1992, Programming Perl: O'Reilly and Associates, Inc., California, 465 p.
Yuan, L. P., and Vanderpool, N. L., 1986, Drainage network simulation: Computers & Geosciences, v. 12, no. 5, p. 653–665.
Young, R. A., Onstad, C. A., Bosch, D. D., and Anderson, W. P., 1994, Agricultural non-point source pollution model, version 4.03, AGNPS user's guide: US Dept. Agriculture-ARS North Central Soil Conservation Research Laboratory, Morris, Minnesota, 112 p.

APPENDIX 1

Program doNet

```
#!/usr/local/bin/perl
# Script name: doNet
# Purpose:  Implement batch Neural Network trainings using Xerion/bp.
#           Computation time is computed by Unix command 'time.'
#       Tested successfully on SunOS4.1.x and HPUX 9.02.
# Author:   Jehng-Jung Kao jjkao@green.ev.nctu.edu.tw
# ftp site: ftp.edu.tw:/misc/environment/NCTU_EV/nnet/doNet
#           ev009.ev.nctu.edu.tw:/nnet/doNet
# Last updated on 03/10/1995.

# Variable sets to be predefined by the user.
  @hiddenSet=("02","05","08","11","14","17","20","23","26","29","32","35","38");
    # number of nodes in the hidden layer.
  $email_address="jjkao@ev001.ev.nctu.edu.tw";

# Default values for optional user-specified variables.
  $traceFMTfile="trace.fmt";
  $reportsub="report.sub"; # user defined subroutine for producing reports
  $reportFile="report/Summary";

# Default values for problem dependent variables.
  $inUnitNum=49;
  $outUnitNum=8;
  # $rep="1";    # if you need information for every $rep iterations.
  $iter="2000";
  $UnitRectarget="%7.4f";
  $UnitRecoutput="%7.4f";

# Problem independent variables(well! you can change them also.)
  $seed=120;
  $randomize=1.0;
  #$weightCost=$1; #if set, minimize error with network cost errors
  $zeroErrorRadius=0;
  $doExamples="VALIDATION";
  $doMinimize="yes";

# Command path which may be system-dependent;
  $MVBIN="/bin/mv";
  $MKDIRBIN="/bin/mkdir";
  $TIMEBIN="time";
  $XERION_BPBIN="bp";
  $MAILCMD="/usr/ucb/Mail";
    # for hpux, $MAILCMD="/usr/bin/mailx";
  $RMBIN="/bin/rm";
  $LNBIN="/bin/ln";

# Tag character for validation path.
  $validateTAG="V";

$usage="Usage: donet netName trainFile validateFile [in# out#]";
  if($#ARGV != 2 && $#ARGV != 4) { print "$usage\n"; exit;}
  $netName=$ARGV[0];
  $trainFile=$ARGV[1];
  $validateFile=$ARGV[2];
  if ($#ARGV == 4) {
    $inUnitNum=$ARGV[3];
    $outUnitNum=$ARGV[4];
  }

# Check file and directory existence.
  if (! -s $trainFile) {
     print "Error: Training Set file $trainFile does not exist.\n$usage\n";
     exit;
  }
  if (! -s $validateFile) {
     print "Error: Validation Set file $validateFile does not exist.\n$usage\n";
     exit;
  }

  if (! -d $netName) {
     print "Creating directory $netName/ ...\n";
     system("$MKDIRBIN $netName");
  }

  chdir($netName);

  if(! -s "$trainFile"){
    print "Linking $trainFile to $netName/ ...\n";
    system("$LNBIN -s ../$trainFile $trainFile");
  }

  if(! -s "$validateFile"){
    print "Linking $validateFile to $netName/ ...\n";
    system("$LNBIN -s ../$validateFile $validateFile");
  }

  if(! -s "../$traceFMTfile") {
     $traceFMTfile="";
     print "Warning: no trace format file. No trace done.\n";
  } elsif (! -s "$traceFMTfile"){
    print "Linking $traceFMTfile to $netName/ ...\n";
    system("$LNBIN -s ../$traceFMTfile $traceFMTfile");
  }

# user-provided reporting file.
  if(-s "../$reportsub"){
          #  if you want to keep different version of the report file, pls.
          #     uncomment the following 5 lines.
          #    if(-s $reportFile) {
          #          $thistime=`date +%y%m%d%H%M%S`;
          #          chop($thistime);
          #          system("$MVBIN $reportFile $reportFile.$thistime");
          #    }
    if(! -s "$reportsub") {
        print "Linking $reportsub to $netName/ ...\n";
        system("$LNBIN -s ../$reportsub $reportsub");
    }
   }

# I donot want to see display in batch training,
  #    but it is useful for interactive training.
  $ENV{'DISPLAY'}="/dev/null";

# Create a TAG file for which run is being implemented.
  $home=$ENV{'HOME'};
  $hostname=`hostname`; chop($hostname);
  if ( ! -d "$home/tmp" ) { system("$MKDIRBIN $home/tmp"); }
  $tagFile="$home/tmp/donet$user$netName$hostname";

# create required directories.
  @needDirs=("in","wt","validate","out","report");
  foreach $d (@needDirs){
    if (! -d $d) {
        system("mkdir $d");
    }
  }

if(-s $reportsub) { require $reportsub;}

# This function execute a training and validation process.
```

```
sub doNet{
  # set name=$netName_$hiddenUnitNum_{$cg|$method}[_$epsilon[_$alpha]][_$ls]
  $thisRunName="${netName}_$hiddenUnitNum";
  if($method eq "") { # non-momentum and non-steepest-decent method
     $thisRunName="${thisRunName}_$cg";
  } else {
     $thisRunName="${thisRunName}_$method";
     if($epsilon ne "")    { $thisRunName="${thisRunName}_$epsilon"; }
     if($alpha ne "")  { $thisRunName="${thisRunName}_$alpha"; }
  }
  if($ls ne ""){$thisRunName="${thisRunName}_$ls"};

   # set input/output file names.
  $weightFile="wt/$thisRunName.wt";                        # weight
  $loadWeights=$weightFile;                                # re-load weight
  $validateOutputFile="validate/$thisRunName.validate"; # validation
  $trainOutputFile="validate/$thisRunName.train";        # training
  $bpInFile="in/$thisRunName.in";                          # bp input
  $bpOutputFile="out/$thisRunName.bp";                     # bp output
  $timeReportFile="report/$thisRunName.time";            # time report

   # create a TAG to keep track where it is, when running background.
  system("echo I am doing this case:$thisRunName now. >$tagFile");

   # notify the user; NOTE: you may receive too many message.
     # if you want to reduce, you may move this line after doNet.
     system("$MAILCMD -s \"$thisRunName$hostname\" $email_address <$tagFile");

   # training
  $validateIt="";
  &writeNetIn($bpInFile,"");
  system("$TIMEBIN $XERION_BPBIN <$bpInFile$validateIt 1>$bpOutputFile$validateIt
2>$timeReportFile$validateIt");

   # validation
  $validateIt=$validateTAG;
  &writeNetIn($bpInFile,$validateIt);
  system(
     "$XERION_BPBIN <$bpInFile$validateIt 1>$bpOutputFile$validateIt 2>/dev/null");
  # In most case, validation is quick and no need to report.
  # system("$TIMEBIN $XERION_BPBIN <$bpInFile$validateIt 1>$bpOutputFile$validateIt
2>$timeReportFile$validateIt");

   # execute user-specified reporting, if provided.
  if(-s $reportsub) {

  &reportsub($thisRunName,$bpOutputFile,$timeReportFile,$trainOutputFile,$validateOut
putFile,$reportFile);
   }
 }

# This function write a XERION input file.  You may consult XERION manual to
  # modify it to meet your needs.
  sub writeNetIn{
   local($netInFile,$validateIt)=@_;
   #print "Writing $netInFile, please wait...\n";
   open(NETIN,">$netInFile$validateIt") ||
               die "Error:canot open $netInFile$validateIt $!\n";

   if($validateIt eq $validateTAG){  # validation phase
      if($validateOutputFile ne "") {print NETIN
           "open $netName > $validateOutputFile\n";}
      if($UnitRectarget ne "") {print NETIN
           "format UnitRec.target \"$UnitRectarget\"\n";}
      if($UnitRecoutput ne "") {print NETIN
           "format UnitRec.output \"$UnitRecoutput\"\n";}
      if($traceFMTfile ne ""){
          $count{$diffDegree}++;

        $test_validEntry=<TEST_VALID>; # ignore blank lines
    }
    close(TEST_VALID);
    # report total # of cases, if desired. should be = $matchcount
    #   print "TOtal=$totalcount\n";
    $matchcount=0;
    foreach $d (@degreeList){ $matchcount += $count{$d}; }

    # set flag for success
    $success="";

    # best result;
    if($count{'0'} == $totalcount){$success=" Yu...Hoo...";}

    # for training set, < 5 untrained case; marginally trained.
    elsif(($totalcount-$count{'0'}) < 5) {$success=" Hmmm...Try further...";}

    # for validation set, 0+45 > 1350 cases is acceptable.
    elsif(($count{'0'}+$count{'45'}) > 1350){$success=" ...OK?...";}

    # report the summary.
    print REPORT "  TDiff\t0:$count{'0'} 45:$count{'45'} 90:$count{'90'}
135:$count{'135'} 180:$count{'180'}  -> $matchcount$success\n";

    close(REPORT);
    if($success eq "") { return "D";}  # bad case
    return "";
  }

# This function extract time information from Unix time command output.
  sub timeOSdep {
    local($OSNAME,$timeReportFile)=@_;
    local($systime,$realtime,$usertime);
    open(TIME,"$timeReportFile");
        # SunOS's and HP-UX's 'time' commands produce slightly different output.
    while (<TIME>){
        if ($OSNAME eq "SunOS") {

    if(/^[ \t]+([\w.]*)[ \t]+real[ \t]+([\w.]*)[ \t]+user[ \t]+([\w.]*)[ \t]+sys/) {
                $realtime=$1;
                $usertime=$2;
                $systime=$3;
            }
        } elsif ($OSNAME eq "HP-UX") {
            if(/^sys[ \t]+([\w.]*)[ \t]*/) {$systime=$1}
            if(/^real[ \t]+([\w.]*)[ \t]*/) {$realtime=$1}
            if(/^user[ \t]+([\w.]*)[ \t]*/) {$usertime=$1}
        }
        else {
            print "Hmmm... I donot know your OS: $OSNAME\n";
            return (-1,-1,-1);
        }
        # you may add other OS dependent code here, if applicable.
    }
    close(TIME);
    system("$RMBIN -f $timeReport"); #NOTE $RMBIN is defined in doNet
    return ($systime,$realtime,$usertime);
}

# Function called by doNet for user-specified reporting.
  sub reportsub{
    local($entryName,$bpOut,$timeReport,$trainOut,$validateOut,$reportFile)=@_;

     # get OS name for 'time' command output format.
    if ($uname eq ""){
```

```
    $uname=`uname -a`; chop($uname);
    if ($uname =~/^([\w-]*)[ \t]+/) { $thisOS=$1}
}

open(REPORT,">>$reportFile");

 # get system date/time.
$thistime=`date +%y%m%d%H%M%S`; chop($thistime);
print REPORT "Report for $entryName on $thistime\n";

 # if $timeReport file exists, extract time and add it into the summary.
if(-s "$timeReport"){
   ($systime,$realtime,$usertime)=&timeOSdep($thisOS,$timeReport);
   print REPORT "  TIME\tuser=$usertime\tsys=$systime\treal=$realtime\n";
}

 # if time report file for validation phase exists, summary it also.
    #In most case, validation is quick, and no need to report it. see doNet.
    #NOTE: $validateTAG is defined in doNet.
if(-s "$timeReport$validateTAG"){
   ($systime,$realtime,$usertime)=&timeOSdep($thisOS,"$timeReport$validateTAG");
   print REPORT "  TIMEV\tuser=$usertime\tsys=$systime\treal=$realtime\n";
}

 # try to get 1st and last Iter lines.
    # You may edit function getIter for this purpose.
if( -s $bpOut) {
        &getIter($entryName,$bpOut);
    $firstIter=`head -1 $bpOut`;
    if($firstIter ne "") { print REPORT "  $firstIter";}
    $lastIter=`tail -1 $bpOut`;
    if($lastIter ne "") { print REPORT "  $lastIter";}
    system("$RMBIN -f $bpOut"); #NOTE: $RMBIN is defined in doNet.
    system("$RMBIN -f $bpOut$validateTAG");
}

close(REPORT);
# ----------------Specific steps for DEM case -------------
  # check training results.
  # if the Nnet configuration produce acceptable result, return "";
    # otherwise $lastFlag will return "D"
    $lastFlag=&compare($trainOut,$reportFile);
    $lastFlag1="D"; # kill all files, anyway, to save space
    if($lastFlag == "D"){
            system("$RMBIN -f $trainOut");
            system("$RMBIN -f in/$entryName.in");
            system("$RMBIN -f in/$entryName.in$validateTAG");
            system("$RMBIN -f wt/$entryName.wt");
    }
  # check validation results.
  # if the Nnet configuration produce acceptable result, return "";
    # otherwise $lastFlag will return "D"
    $lastFlag1=&compare($validateOut,$reportFile);
    $lastFlag1="D"; # kill all files, anyway, to save space
    if($lastFlag1 == "D"){
            system("$RMBIN -f $validateOut");
            if( $lastFlag eq "") {
                    system("$RMBIN -f $trainOut");
                    system("$RMBIN -f in/$entryName.in");
                    system("$RMBIN -f in/$entryName.in$validateTAG");
                    system("$RMBIN -f wt/$entryName.wt");
            }
    }


---------- END of report.sub included (use require) into doNet-----------
```

APPENDIX 2

Sample report.sub

```
# ---------- START of report.sub included by doNet-----------
# This is a sample file for report.sub for running doNet.
# Case description: DEM Drainage Network training.
# Author: Jehng-Jung Kao jjkao@ev001.ev.nctu.edu.tw
# ftp site: ftp.edu.tw:/misc/environment/NCTU_EV/nnet/donet
# Last updated on 03/10/95.

# This function keep only lines with ^iter= from $thefile, XERION/bp output.
  sub getIter{
        local($entryName,$thefile)=@_;
        local($tmpfile)="/tmp/junk$entryName";
        open(THEFILE,"$thefile");
        open(TMPFILE,">$tmpfile");
        while (<THEFILE>) { if (/^iter= /) {print TMPFILE $_;}}
        close(TMPFILE); close(THEFILE);
        system("$MVBIN $tmpfile $thefile"); #NOTE $MVBIN is defined in doNet.
  }

# List of degrees to be reported for differences.
  @degreeList=('0','45','90','135','180');

# This function compare training or validating DEM drainage pattern results.
  sub compare{
    local($test_validOutFile,$reportFile)=@_;
    if(! -s $test_validOutFile) {return;}

    # Initialization.
    open(REPORT,">>$reportFile");
    open(TEST_VALID,$test_validOutFile);
    $totalcount=0;    # count total # of cases.
    foreach $d (@degreeList){ $count{$d}=0; }

    # Do comparison for reporting difference.
    while (<TEST_VALID>) {
        $totalcount++;
        # delete leading blanks
        $test_validEntry=$_; chop($test_validEntry); $test_validEntry =~ s/^[ \t]*//;

        # split into a List
        @targetList=split(/[ \t]+/,$test_validEntry);

        # find the target direction, with maximal value.
        $max=0.0;
        for($i=0; $i < 8; $i++){
            if($targetList[$i] > $max) {$max=$targetList[$i];
                $targetDir=$i+1;}
        }
        $targetDegree=($targetDir-1)*45;

        # find the Nnet determined direction, with maximal value.
        $test_validEntry=<TEST_VALID>; chop($test_validEntry);
        $test_validEntry =~ s/^[ \t]*//;  # delete leading blanks.
        @test_validList=split(/[ \t]+/,$test_validEntry);
        $max=0.0;
        for($i=0; $i < 8; $i++){
            if($test_validList[$i] > $max) {$max=$test_validList[$i];
                $test_validDir=$i+1;}
        }
        $test_validDegree=($test_validDir-1)*45;

        # compute the difference and count
            $diffDegree=$targetDegree-$test_validDegree;
            if($diffDegree < 0) {$diffDegree=-$diffDegree;}
            if($diffDegree > 180) {$diffDegree=360-$diffDegree;}
        if(! -s $traceFMTfile) {print "Error: $traceFMTfile is empty.\n";exit;}
        print NETIN "addTrace doExamples \"print \'\'";
        open(TRACEFMTFILE,"$traceFMTfile");
        while (<TRACEFMTFILE>){print NETIN "\t$_";}
        close(TRACEFMTFILE);
        print NETIN "@$netName\"\n";
       }
    }
    print NETIN "addNet \"$netName\"\n";
    print NETIN "useNet \"$netName\"\n";
    if($inUnitNum eq "") { print "Error: no inUnitNum.\n"; exit}
    print NETIN "addGroup -type INPUT input $inUnitNum\n";
    if($hiddenUnitNum eq "") { print "Error: no hiddenUnitNum.\n"; exit}
    print NETIN "addGroup -type HIDDEN hidden $hiddenUnitNum\n";
    if($outUnitNum eq "") { print "Error: no outUnitNum.\n"; exit}
    print NETIN "addGroup -type OUTPUT output $outUnitNum\n";
    print NETIN "disconnectGroups \"Bias\" \"input\"\n";
    print NETIN "connectGroups input hidden\n";
    print NETIN "connectGroups hidden output\n";
    if($trainFile ne "") {print NETIN "addExamples -type TRAINING $trainFile\n";}
        if($doMinimize eq "yes" && $validateIt ne $validateTAG) { # training phase
        if($seed ne "") {  print NETIN "seed $seed\n";}
        if($randomize ne "") {  print NETIN "randomize $randomize\n";}
        if($weightCost ne "") {  print NETIN "set currentNet.weightCost = $weightCost\n";}
        if($zeroErrorRadius ne "") {  print NETIN
            "set currentNet.extension.zeroErrorRadius = $zeroErrorRadius\n";}
        print NETIN "minimize -tolerance 1.0e-10";
        if($rep ne "") {  print NETIN " -rep $rep";}
        if($iter ne "") {  print NETIN " -iter $iter";}
        if($method ne "") {  print NETIN " -$method";
            if($method eq "quickProp" && $ls eq ""){print NETIN " -qpEpsilon 0.1";}
        }
        if($cg ne "") {  print NETIN " -$cg";}
        if($epsilon ne "") {  print NETIN " -epsilon $epsilon";}
        if($alpha ne "") {  print NETIN " -alpha $alpha";}
        if($ls ne "") {  print NETIN " -ls$ls";}
        print NETIN "\n";
        if($weightFile ne "") {  print NETIN "saveWeights $weightFile\n";}
    }
    if($validateIt eq $validateTAG){
       if($loadWeights ne "") {  print NETIN "loadWeights $loadWeights\n";}
       if($validateFile ne "") {print NETIN
           "addExamples -type VALIDATION $validateFile\n";}
       if($doExamples ne ""){print NETIN "doExamples -t $doExamples\n";}
       print NETIN "close $netName\n";
       print NETIN "deleteStream -quiet $netName\n";
       if($trainOutputFile ne "") {print NETIN "open $netName > $trainOutputFile\n";}
       if($trainFile ne "") {print NETIN "doExamples -type TRAINING \n";}
       print NETIN "close $netName\n";
    }
    print NETIN "exit\n";
    close(NETIN);
  }

#-----------START JOBS-------------------------------
  foreach $h (@hiddenSet) {
    # repeat for number of hidden nodes = $h.
    $hiddenUnitNum=$h;
    # START OF USER SPECIFIED TRAINING STRATEGIES
      # steepest method -epsilon [0.3|0.5] and -lsRay
        $ls="";   # line search method
        $cg="";
        $method="steepest"; # only one can be non-null string
        $epsilon="0.3";       # for step size ()
        &doNet();
        $epsilon="0.5";       # for step size ()
        &doNet();
  $epsilon="";       # for step size ()
  $ls="Ray";
  &doNet();

# momentum method -epsilon [0.1|0.3]  -alpha [0.9|0.7]
  $ls=""; $method="momentum";
  $alpha="0.9";      # for momentum method
  $epsilon="0.1";        # for step size
  &doNet();
  $epsilon="0.3";        # for step size
  &doNet();
  $alpha="0.7";      # for momentum method
  $epsilon="0.1";        # for step size
  &doNet();
  $epsilon="0.3";        # for step size
  &doNet();

# Conguate Gradient method
  $cg="cg"; $method=""; # only one can be non-null string
  &doNet();

# Rudi's Conguate Gradient method
  $cg="cg"; $method=""; # only one can be non-null string
  $cg="cgrudi";
  &doNet();

  $cg="cgRestart";
  &doNet();
  system("$RMBIN -f $tagFile");
```

APPENDIX 3
Sample trace.fmt

```
currentNet.group[3].unit[0].target \
currentNet.group[3].unit[1].target \
currentNet.group[3].unit[2].target \
currentNet.group[3].unit[3].target \
currentNet.group[3].unit[4].target \
currentNet.group[3].unit[5].target \
currentNet.group[3].unit[6].target \
currentNet.group[3].unit[7].target  \"\n\" currentNet.group[3].unit[0].output \
currentNet.group[3].unit[1].output \
currentNet.group[3].unit[2].output \
currentNet.group[3].unit[3].output \
currentNet.group[3].unit[4].output \
currentNet.group[3].unit[5].output \
currentNet.group[3].unit[6].output \
currentNet.group[3].unit[7].output \
\"\n\"\
```