INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Downloaded from SHAREOK. Persistent Handle URL: https://hdl.handle.net/11244/5550

UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

PRIMAL-DUAL TECHNIQUES FOR NONLINEAR PROGRAMMING AND APPLICATIONS TO ARTIFICIAL NEURAL NETWORK TRAINING

A Dissertation

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

By

NICOLAS COUELLAN

Norman, Oklahoma

1997

UMI Number: 9808408

UMI Microform 9808408
Copyright 1997, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized copying under Title 17, United States Code.

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

PRIMAL-DUAL TECHNIQUES FOR NONLINEAR PROGRAMMING

AND

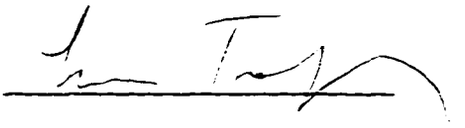
APPLICATIONS TO ARTIFICIAL NEURAL NETWORK TRAINING

A Dissertation

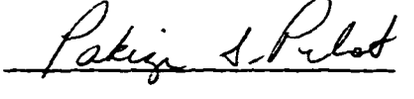
APPROVED FOR THE

SCHOOL OF INDUSTRIAL ENGINEERING

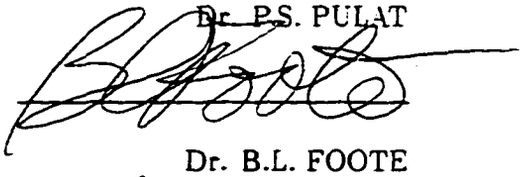
BY



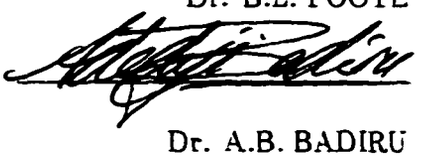
Dr. T.B. TRAFALIS



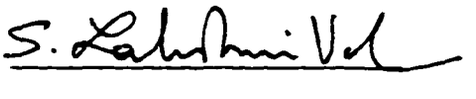
Dr. P.S. PULAT



Dr. B.L. FOOTE



Dr. A.B. BADIRU



Dr. S. LAKSHMIVARAHAN

©Copyright by Nicolas P. Couellan 1997

All Rights Reserved.

Acknowledgements

I would like to thank Dr. T.B. Trafalis. He has been my advisor for nearly five years now. His expertise and guidance were of great importance for this research. He has always been very involved in my work and I have learned a lot from him. His input and advice have greatly improved my technical writing and this resulted in several joint publications in journals and conferences.

I also thank my committee members Dr. P.S. Pulat, Dr. B.L. Foote, Dr. S. Lakshmivarahan and Dr. A.B. Badiru for getting involved in my work. Their input throughout the completion of this work and during the proposal defense has contributed to the quality of this report. I also thank them for everything I have learned attending the various courses they have offered.

I sincerely thank my parents for their moral support and their encouragement.

Contents

Acknowledgements iv

Abstract xi

1 Introduction 1

1.1 Overview 1

1.2 Research Objectives 3

1.3 Scope of the Dissertation 4

2 Literature Review 6

2.1 Linear and Quadratic Programming 6

2.2 General Nonlinear Programming 7

3 A Nonlinear Primal-Dual Algorithm 11

3.1 Introduction 11

3.2 The Nonlinear Primal-Dual Algorithm 12

3.3 Implementation Issues 17

3.3.1 Ensuring the Second Order Optimality Conditions 17

3.3.2 Factorizing the Jacobian Matrix 20
3.3.3 The Update of μ 22
3.3.4 The Linesearches 23
3.3.5 The Stopping Criterion 24
3.4 Computational Results with General Nonlinear Programs 25

4 A Stochastic Nonlinear Primal-Dual Algorithm 28
4.1 Introduction 28
4.2 The Stochastic Algorithm 29
4.3 Computational Experiments with SNLPD 33

5 An Incremental Primal-Dual Algorithm for Problems with Special Structure 41
5.1 Introduction 41
5.2 Example 43
5.3 The Incremental Algorithm 45
5.4 Local Convergence of INCNLPD 49

6 Application to Artificial Neural Network Training 55
6.1 Artificial Neural Networks 55
6.2 Problem Statement and Training Algorithms 57
6.2.1 Notations 59
6.2.2 The Error Minimization Problem 60
6.2.3 A Variant Approach to the Training Problem 61

6.3 Computational Experiments with Odd-Parity Problems 65

7 Application to Financial Forecasting **70**

7.1 Introduction 70

7.2 Experiments 71

8 Application to Medical Diagnosis and Classification Problems **85**

9 Summary, Conclusions and Recommendations **90**

9.1 Summary 90

9.2 Recommendations for Future Research 92

A Convergence of Algorithms **105**

B Source Codes **107**

List of Tables

3.1 Experiments with Hock and Schittkowski's Nonlinear Problems 26

3.2 (Cont'd) Experiments with Hock and Schittkowski's Nonlinear Problems 27

4.1 Comparisons of NLPD and SNLPD for Function 1 36

4.2 Comparisons of NLPD and SNLPD using for Function 2 37

4.3 Comparisons of NLPD and SNLPD for Function 3 38

4.4 Comparisons of NLPD and SNLPD for Function 4 39

6.1 XOR2 Problem 65

6.2 ANN Training with XOR2 Problem using TBB. 66

6.3 ANN Training with XOR2 Problem using TOB. 67

6.4 ANN Training with XOR2 Problem using TOSN. 68

6.5 Comparisons of Training Algorithm for XOR2 Problem 68

7.1 Training Results for Boeing Stock Price Return 74

7.2 Training Results for Swiss Franc Exchange Rate 74

List of Figures

4.1 Schubert's Function Restricted to [-20;20]. 35

4.2 Schubert's Function Restricted to [-5;5]. 35

5.1 Incremental Moves for an Unconstrained Example 44

6.1 Multilayer Artificial Neural Network 58

6.2 Example of Simple ANN Architecture 62

6.3 Example of Error Surface. 62

6.4 Hyperbolic Tangent Activation Function. 63

7.1 Artificial Neural Network Architecture for Financial Forecasting 71

7.2 Training (Time<40) and Testing (Time>40) for Boeing stock price return 75

7.3 Training (Time<40) and Testing (Time>40) for Boeing stock price return 76

7.4 Training (Time<40) and Testing (Time>40) for Boeing stock price return 77

7.5 Training (Time<40) and Testing (Time>40) for Boeing stock price return 78

7.6 Training (Time<40) and Testing (Time>40) for Boeing stock price return 79

7.7 Training (Time<100) and Testing (Time>100) for Swiss Franc exchange rate 80

7.8 Training (Time<100) and Testing (Time>100) for Swiss Franc exchange rate 81

7.9 Training (Time<100) and Testing (Time>100) for Swiss Franc exchange rate 82

7.10 Training (Time<100) and Testing (Time>100) for Swiss Franc exchange rate 83

7.11 Training (Time<100) and Testing (Time>100) for Swiss Franc exchange rate 84

8.1 Training results for the Breast Cancer Database 86

8.2 Testing results for the Breast Cancer Database 87

8.3 Training results for the Iris Plants Database 88

8.4 Testing results for the Iris Plants Database 89

Abstract

In this work, new developments in primal-dual techniques for general constrained nonlinear programming problems are proposed. We first implement a modified version of the general nonlinear primal-dual algorithm that was published by El-Bakry et al. [21]. We use the algorithm as a backbone of a new stochastic hybrid technique for solving general constrained nonlinear programming problems. The idea is to combine a fast local optimization strategy and a global search technique. The technique is a modified nonlinear primal-dual technique that uses concepts from simulated annealing to increase the probability of converging to the global minima of the objective function. At each iteration, the algorithm solves the Karush-Kuhn-Tucker optimality conditions to find the next iterate. A random noise is added to the resulting direction of move in order to escape local minima. The noise is gradually removed throughout the iteration process. We show that for complicated problems that possess numerous local minima and global minima, the proposed algorithm outperforms the deterministic approach. We also develop a new class of incremental nonlinear primal-dual techniques for solving optimization problems with special decomposition properties. Specifically, the objective functions of the problems are sums of independent nonconvex differentiable terms min-

imized subject to a set of nonlinear constraints for each term. The technique performs successive primal-dual increments for each decomposition term of the objective function. The method is particularly beneficial for online applications and problems that have a large amount of data. We show that the technique can be nicely applied to artificial neural training and provide experimental results for financial forecasting problems.

Chapter 1

Introduction

1.1 Overview

For more than ten years there has been a great deal of research in the area of interior point methods (IPMs). Karmarkar's work [35] on projective algorithms for linear programming (LP) is at the origin of IPMs. Since then, many other LP solver algorithms have been developed. The primal-dual IPM [43, 75] is now considered the most effective technique for solving large scale LP problems. With the success of IPMs for LP, extensions to the convex quadratic cases have been investigated. For example, the primal-dual method for convex quadratic programming is available as a software package [69]. Other IPMs such as the trust region techniques [79] have also been successful for solving convex quadratic programming problems.

With a better understanding of the theory of IPMs, researchers in the field are now trying to extend the algorithms to the next level of complexity: the general nonlinear nonconvex programming problem. The first conceptual papers along this line appeared

in 1992. Yamashita [77] and El-Bakry et al. [21] provided both algorithms and proofs of convergence of the nonlinear primal-dual technique. Since then. other work has been published, see for example [78, 41, 8].

The range of applications of the nonlinear primal-dual method is broad. Except differentiability, no particular assumption on the objective function and the constraints are required. Typically, the problem that is solved is as follows

$$\begin{aligned} min \quad & f(x) \\ s.t. & \\ h(x) \quad &= \quad 0 \\ g(x) \quad &\geq \quad 0 \end{aligned} \tag{1.1}$$

where $f: \Re^n \rightarrow \Re$, $h: \Re^n \rightarrow \Re^m$, and $g: \Re^n \rightarrow \Re^p$ are differentiable.

The range of applications of nonlinear programming is wide. Nonlinear programming techniques have been used to solve optimal control problems [7, 11], structural and mechanical design problems [12, 26], electrical power network problems [1, 76], water resource management problems [29], financial portfolio problems [45] and many other problems.

The Artificial Neural Network (ANN) training problem can also be seen as a general nonlinear programming problem. The training problem is an error minimization problem. Constraints on the variables arise naturally from the design of the neural network. The error surface involved is known to be highly nonlinear and nonconvex. In the neural network literature, much work has been done in developing new efficient network

architectures. however little work has been done in applying the new advancements in optimization for the training problem. The backpropagation (BP) training algorithm [71] is commonly used in commercial neural network software packages. BP is a gradient descent technique that exhibits limitations as the network size or the complexity of the error surface increases. The algorithm becomes slow or gets trapped in a local minimum of the error surface leading to incorrect training of the data. To overcome some of these problems, there are two ways to go. One can modify the network architecture or one can use more sophisticated optimization tools to train the network. In this research, we propose to develop a new class of primal-dual techniques that is particularly suitable to solving the error minimization problem of neural network training.

1.2 Research Objectives

We propose to develop new path following interior point methods for solving the general nonlinear programming problem. We will first implement a generic primal-dual algorithm. Using ideas from linear programming, the algorithm keeps primal and dual information by solving the Karush-Kuhn-Tucker conditions of the nonlinear problem. A merit function will be used to measure the progress along the direction of move. Improvements of the techniques will then be investigated.

Among those improvements, we use a Quasi-Newton approximation of the Hessian of the Lagrangian of the objective function for applications where the function is non-convex and also where the calculation of the true Hessian is too complicated and time consuming.

We also develop a stochastic version of algorithm. Borrowing ideas from simulated annealing, we introduce a random noise in the objective function and slowly remove it as we approach the optimal solution. This increases the chance of reaching a global optimum. This is the first time a stochastic perturbation has been embedded in a general nonlinear primal-dual framework. The main motivation of our approach comes from the idea of combining efficient local optimization techniques with stochastic optimization strategies for global optimization problems.

With the artificial neural network training application in mind, we will also develop a new class of incremental nonlinear primal-dual techniques. The idea of incremental techniques has been applied to gradient descent techniques before [67, 4, 5] but never to the primal-dual framework. We will show that this incremental version of the algorithm can be nicely applied to the online training of artificial neural networks. The algorithm will be used to solve financial forecasting problems with neural networks.

1.3 Scope of the Dissertation

Chapter 2 is a literature review of algorithms for linear, quadratic, and nonlinear programming. Chapter 3 presents the generic nonlinear primal-dual method and its implementation. Chapter 4 describes a stochastic modification of the algorithm and shows some computational experiments to validate the use of such modification. Chapter 5 introduces the incremental version of the algorithm and provides convergence results. Chapter 6 described how the technique developed can be used to solve artificial neu-

ral network training problems. Chapter 7 reports computational results for financial forecasting problems. Chapter 8 concludes the dissertation and proposes some future research.

Chapter 2

Literature Review

2.1 Linear and Quadratic Programming

Ideas such as the ellipsoid method [37] or barrier methods [23] from the 60's and 70's are at the origin of the interior point methods for linear programming. However, before 1984, there was no practical implementation of these ideas. In 1984, Karmarkar presented a projective algorithm [35] for linear programming. The algorithm outperformed the simplex method for large scale problems. Soon, variants such as the affine scaling algorithm [69] and the method of analytical centers [60] were developed and lead to even better computational complexity than the original projective algorithm. The primal-dual method for linear programming [50, 43] was developed in 1986. The primal-dual method is now considered as the most effective interior point method for linear programming. It leads to very nice complexity results and it also shows a very elegant and attractive structure that is at the origin of what is now known as the path-following methods. From a theoretical point of view, researchers realized that there was a strong

connection between these algorithms and the barrier methods [23].

Later, extensions of interior point methods to the convex quadratic programming problem were investigated. Algorithms such as the trust region technique [61, 24] or the primal-dual method for convex quadratic problems [53] were proposed. A trust region technique for nonconvex quadratic programming problem was later developed [79]. It solves a ball constrained convex quadratic program using an affine scaling based method at each iteration.

In the past few years, extensive work has been done in the area of positive semi-definite programming (PDP) [68]. The problem is to minimize a linear function with respect to the variable x subject to the constraint that a matrix $F(x)$ is positive semi-definite. It can be shown that linear and quadratic programming are special cases of PDP. Many problems that arise in engineering can be written as PDP problems. Because PDP unifies linear, quadratic and convex optimization problems in general, it has become a very appealing research direction in the area of optimization. The reader should refer to [68] for an extensive review of PDP.

2.2 General Nonlinear Programming

We have seen that for the specific cases above such as linear, convex quadratic or general quadratic programming, many algorithms have flourished. However, for the general case where the objective function and the constraints are nonconvex differentiable functions, only a handful of algorithms have been proposed. These algorithms follow a common

framework. They all attempt to solve the Karush-Kuhn-Tucker (KKT)systems of optimality conditions using a Newton based strategy [6]. To obtain a globally convergent algorithm, the optimality conditions are parameterized. The parameterized optimality conditions have the nice property that their solution coincides with the solution of the KKT conditions for the associated logarithmic barrier function problem.

Yamashita [77] proposed the first algorithm of this type. A proof of global convergence is given as well as numerical results with large scale linear problems and dense nonlinear programs. Later Yamashita and Tanabe [78] published a trust region variant of the algorithm. Instead of using a linesearch approach as before, a trust region technique is used to minimize the barrier penalty function. A proof of global convergence is also given.

El-Bakry et al. [21] developed a variant of Yamashita's algorithm. Both equality and inequality constraints are included in the formulation. Global convergence of the algorithm is demonstrated and preliminary results are promising. The paper also discusses the relationship between the perturbed KKT and the regular KKT conditions for the associated logarithmic barrier problem. They claim that the two systems are not equivalent but that they share the same optimal solution and that by defining some auxiliary variable one can avoid ill-conditioning. This discussion gives some insights into the formulation of the KKT conditions of the general nonlinear programming problem. The choice of a more general formulation of the problem that includes inequality constraints gives more practicality to the algorithm as compared to Yamashita's algorithm.

Lasdon et al. [41] have also proposed a variant of the nonlinear programming algorithm. The idea is fundamentally the same. The algorithm attempts to solve the system of

perturbed KKT conditions using Newton's method. The formulation of the problem sets upper and lower bounds on the variables and treat them separately. The nonlinear constraints are taken as equality constraints. The derivation of the KKT optimality conditions varies from earlier authors because of the separate treatment on the bounds. Different versions of the algorithm are investigated. In the primal-dual version, no linesearches are performed. The algorithm takes full Newton's steps. In the trust region version, the linesearches are implicitly avoided by the trust region technique. The third version is the primal version obtained by replacing the slack variables for the bound by the differences of the variables and their respective bounds. The resulting algorithm is similar to the primal-dual algorithm except that the dual variables do not appear in the formulation. Computational results are given for the three versions. The primal-dual and the primal version performed much better than the trust region variant. Descriptions are given for some of the implementation issues such as exploiting the sparsity of the matrices and the use of forward differences to compute the derivatives.

Breitfeld and Shanno [8] have considered a different approach for the general nonlinear programming problem. They proposed to transform the problem into an unconstrained problem by embedding the constraints into the objective function by borrowing ideas from the augmented Lagrangian and logarithmic barrier methods. Equality, inequality constraints, as well as bounds on the variables are included in the original formulation. Update formulas are derived for the barrier parameter and the Lagrange multipliers. The resulting code performs really well on the Hock and Schittkowski's nonlinear programming problem database [32]. Numerical performance is reported for different updates of the barrier parameter and the Lagrange multipliers.

Recently. Forsgren and Gill [25] have proposed a variant of the generic primal-dual algorithm. Their methods differs from earlier algorithms by the choice of the merit function to be used in solving the unconstrained minimization problem. Instead of using the square norm of the KKT conditions as it is usually implemented, they proposed to use an augmented penalty-barrier function. The authors show that this choice of merit function has the potential to give points that satisfy the second order necessary optimality conditions whereas the usual merit function only guarantees the first order optimality conditions.

Issues such as computational errors and ill-conditioning in nonlinear primal-dual methods are now being investigated and some results are already published [73].

In parallel to primal-dual techniques, trust region methods for nonlinear programming have been developed. The algorithm uses ideas from the primal-dual methods and sequential quadratic programming to solve a sequence of barrier problems. Trust regions are used for each sequential barrier problem. Coleman and Li [13], Dennis et al. [18],Byrd et al. [10], and recently Das [15], published such nonlinear trust region algorithms. It is interesting to note that when the quadratic subproblems are strongly convex, there is an equivalence between the sequential quadratic programming and Newton's method applied to the KKT system of the original nonlinear programming problem.

Chapter 3

A Nonlinear Primal-Dual Algorithm

3.1 Introduction

Interior Point Methods (IPMs) have been very successful in solving large scale problems in linear programming. Numerous algorithms have been developed for this purpose. The primal-dual algorithm [43] and its variants are among the most efficient techniques for solving linear programming problems. In the general nonlinear case, only preliminary work has been done and the area is still at its early development. However, among the algorithms that have been proposed, a common general framework is always found. Researchers have naturally tried to borrow ideas from the linear programming primal-dual technique and adapt them to the nonlinear case. The main steps of the nonlinear primal-dual algorithm (NLPD) are similar to the steps of the IPM linear programming primal-dual technique. NLPD first transforms the constrained problem into

an unconstrained problem by setting the KKT optimality conditions. The KKT system of nonlinear equations is linearized and solved using Newton's method. Step lengths are chosen on the resulting directions of move in the primal and dual spaces. These steps are repeated until some optimality criterion is satisfied.

The difficulty arises when the nonlinear problems are nonconvex. In the nonconvex case, the Hessian of the Lagrangian becomes indefinite and the Jacobian matrix involved in the Newton step can become singular. To avoid such difficulties, we propose to approximate the Hessian matrix by a positive definite matrix and to solve the approximated system of equations using Newton's method. This can be seen as a Quasi-Newton method approach. Next, we give the details of the algorithm.

3.2 The Nonlinear Primal-Dual Algorithm

Consider the following general nonlinear programming problem

$$\begin{aligned} min \quad & f(x) \\ s.t. \quad & \\ h(x) \quad &= \quad 0 \\ g(x) \quad &\geq \quad 0 \end{aligned} \tag{3.1}$$

where $f: \Re^n \rightarrow \Re$, $h: \Re^n \rightarrow \Re^m$, and $g: \Re^n \rightarrow \Re^p$. The Lagrangian associated with Problem 3.1 is

$$L(x, y, z) = f(x) + y^T h(x) - z^T g(x) \tag{3.2}$$

where $y \in \Re^m$ and $z \in \Re^p$ are the vectors of Lagrange multipliers for the equalities h and the inequalities g.

The Karush-Kuhn-Tucker optimality conditions can be written as follows

$$\begin{aligned} \nabla_x L(x,y,z) &= 0 \\ h(x) &= 0 \\ g(x) &\geq 0 \\ Zg(x) &= 0 \\ z &\geq 0 \end{aligned} \tag{3.3}$$

where $\nabla_x L(x,y,z) = \nabla f(x) + \nabla h(x)y - \nabla g(x)z$ and Z is a diagonal matrix formed with the coordinates of the vector z ($Z = diag\{z\}$).

We introduce slack variables for the inequality constraints. Let s be the vector of slack variables and S be the associated diagonal matrix. We also define e as the unit vector ($e \in \Re^p$, $e(i) = 1, i = 1, \ldots, p$). The KKT system of equations 3.3 can be reduced to the following compact form

$$F_\mu(x,y,s,z) = \begin{pmatrix} \nabla_x L(x,y,z) \\ h(x) \\ g(x) - s \\ ZSe \end{pmatrix} = 0 \ (s,z) \geq 0 \tag{3.4}$$

When Newton's method is applied to the system 5.24, the complementary slackness conditions are linearized as follows

$$Z\Delta s + S\Delta z = -SZe \tag{3.5}$$

It is important to notice that if the coordinate s_i^k of the current iterate becomes zero, it will remain zero in the subsequent iterations. From Equation 3.5, we have $(Z\Delta s)_i = 0$

because $s_i^k = 0$. therefore $\Delta s = 0$ and the s^{th} coordinate of s will remain at zero. Similar conclusions can be drawn about z in the case where one of its component vanishes. In other words if the current point approaches the boundary, it gets trapped by that boundary and this might results in small steps and possible nonconvergence of the algorithm. Global convergence is not achieved. To avoid this problem, we perturb the complementary slackness condition as follows

$$SZe = \mu e \tag{3.6}$$

where $\mu > 0$ is the perturbation parameter and will be decreased at each iteration. The update of μ will be defined later.

The perturbation of the KKT conditions can be seen as a way to ensure adherence to the central path in order to achieve global convergence.

In the following, we will use the perturbed KKT system 3.7 instead of the original KKT system 5.24. It can be written in the form

$$F_\mu(x, y, s, z) = \begin{pmatrix} \nabla_x L(x, y, z) \\ h(x) \\ g(x) - s \\ ZSe - \mu e \end{pmatrix} = 0 \ (s, z) \geq 0 \tag{3.7}$$

Next, we described the application of Newton's method for solving the nonlinear system of equations 3.7.

Let $v_k = (x_k, y_k, s_k, z_k)$ be a current Newton iterate. The Newton correction $\Delta v_k = (\Delta x_k, \Delta y_k, \Delta s_k, \Delta z_k)$ is the solution of the following linearized system

$$J(v_k)\Delta v_k = -F_\mu(v_k) \tag{3.8}$$

where $J(v_k) = F'_\mu(v_k)$ and $J(v_k)$ is given by

$$J(v_k) = \begin{pmatrix} \nabla_x^2 L(x,y,z) & \nabla h(x) & 0 & -\nabla g(x) \\ \nabla h(x)^T & 0 & 0 & 0 \\ \nabla g(x)^T & 0 & -I & 0 \\ 0 & 0 & Z & S \end{pmatrix} \tag{3.9}$$

At this point, we assume that $J(v_k)$ satisfies all the necessary conditions in order for the system 3.8 to have a unique solution v^*. We will see later that in the general case the matrix $J(v_k)$ is indefinite.

We perform linesearches in each Newton direction. Let $\alpha_k = (\alpha_x, \alpha_y, \alpha_s, \alpha_z)$ be the vector of resulting optimal steplengths. We can summarize the move in the primal and dual spaces by the following update equation

$$v_{k+1} = v_k + \Lambda_k \Delta v_k \tag{3.10}$$

where $\Lambda_k = diag\{\alpha_k\}$ is the diagonal matrix composed of the coordinates of α_k.

The algorithm performs multiple moves as decribed above until a stopping criterion is met. The stopping criterion is chosen to be the following

$$\|F_\mu(v_k)\| < \epsilon \tag{3.11}$$

This means that we will find an ϵ-approximate solution of the KKT optimality conditions. In practice, this approximate solution is usually satisfactory. The choice of ϵ depends on the accuracy needed for the practical application. For good accuracy, one should expect a larger amount of computational effort than for an average level of accuracy.

Next, we give a generic primal-dual algorithm that emphasizes and summarizes the major steps that we have elaborated above. In the next section, we give details on implementation issues such as the implementation of the linesearch, the update of the parameter μ, how to handle indefiniteness of the hessian of the Lagrangian and also particular choices of ϵ.

Nonlinear Primal-Dual Algorithm (NLPD)

Step 1 Set k=0.

Step 2 Start with $v_k = (x_k, y_k, s_k, z_k)$ where $(s_k, z_k) > 0$ and initialize $\mu_k > 0$.

Step 3 Check stopping criterion 3.11

- if Condition 3.11 is satisfied go to Step 8.
- if Condition 3.11 is not satisfied go to Step 4.

Step 4 Solve the system of linear equations 3.8 for Δv_k.

Step 5 Perform the linesearches to determine α_k.

Step 6 Move to next point $v_{k+1} = v_k + \Lambda_k \Delta v_k$.

Step 7 Set $k = k + 1$, update μ_k and go to Step 3.

Step 8 Stop with the optimal point $v^* = v_k$.

3.3 Implementation Issues

3.3.1 Ensuring the Second Order Optimality Conditions

We first recall the second order optimality conditions for which a KKT point is guaranteed to be a local minimizer of the original function f. These conditions can be summarized in the following theorem

Theorem 3.1 *(Second Order Optimality Conditions)[48]*

Assume that

1. *The functions f, g and h are twice differentiable.*

2. *There exists a (x^*, y^*, z^*) that solves the KKT conditions 3.3.*

3. *For all $w \in \Re^n$ such that if*

   - $\nabla g_j(x^*)w = 0$ *for* $j \in \{j/g_j(x^*) = 0, z_j^* > 0\}$,
   - $\nabla g_j(x^*)w \geq 0$ *for* $j \in \{j/g_j(x^*) = 0, z_j^* = 0\}$,
   - $\nabla h_k(x^*)w = 0$ *for* $k = 1, \ldots, m$,

   $$w^T \nabla_x^2 L(x, y, z) w \ > \ 0$$

Then x^* is a local minimizer for Problem 3.1.

It can be shown that if the second order optimality conditions are satisfied and if the gradients of the binding constraints at x^* are linearly independent, the above theorem

is equivalent with saying that the Jacobian matrix J is non-singular. A proof of this is well detailed in [48] for the case where the constraints are equality constraints El-Bakry et al. [21] described how it can be adapted to the more general case where inequality constraints are involved.

In the general case, the matrix J is indefinite and the non-singularity of J is not always insured. If singularity happens, it is due to the fact that the hessian of the Lagrangian $\nabla_x^2 L(x, y, z)$ is not positive definite. For problems that do not generate singularity, we will calculate $\nabla_x^2 L(x, y, z)$ by central differences. For problems that are nonconvex, $\nabla_x^2 L(x, y, z)$ will be approximated by a positive definite matrix using a Quasi-Newton update formula. This approximation of the hessian not only helps in convexifying the problem but also in improving computational complexity. The central differences involved in calculating the hessian are rather tedious and time consuming because a large amount of function evaluations is required. However, Quasi-Newton techniques only require first derivatives, therefore the number of function evaluations is considerably reduced.

Next, we describe the Quasi-Newton update used in approximating $\nabla_x^2 L(x, y, z)$.

Let $H^{(k)}$ denote the approximation of $\nabla_x^2 L(x, y, z)$ at iteration k. We will use the following approximation

$$H^{(k+1)} = \lambda H^{(k)} + \nabla_x L(x, y, z)(\nabla_x L(x, y, z))^T \tag{3.12}$$

Using the Sherman-Morrisson-Woodbury formula [19], the inverse of $H^{(k+1)}$ denoted by $P^{(k+1)}$ can be expressed as follows

$$P^{(k+1)} = \frac{P^{(k)} - P^{(k)} \nabla_x L \left((\nabla_x L)^T P^{(k)} \nabla_x L + \lambda I \right)^{-1} (\nabla_x L)^T P^{(k)}}{\lambda} \tag{3.13}$$

where $\nabla_x L = \nabla_x L(x, y, z)$.

This update of the Hessian is based on the Recursive Prediction Error Method (RPEM) [59]. It is similar to the Quasi-Newton update of Davidon [16]. In numerical analysis, the BFGS update is often used to approximate the Hessian. However, the RPEM method only requires the storage of $P^{(k)}$ at each iteration whereas the BFGS method requires additional storage. For this reason, we will use RPEM update instead of a classical BFGS approach.

When the problem is convex, one can use central difference calculations to evaluate the hessian. The central difference derivative of a function f of one variable is given by

$$f'(x) \simeq \frac{f(x+h) - f(x-h)}{2h} \tag{3.14}$$

where h is small.

The choice of the parameter h is critical. Bad choices of h can lead to significant round-off errors [19].

In our case, we are more interested in computing gradients of functions of several variables. To compute the gradients, we will take each component of the function as a function of a single variable itself. In our implementation, h is a parameter of the code and can be changed at any time. We use Ridders' idea of extrapolating the finite differences calculation to higher order as h is decreased from its original value [55]. The extrapolation follows the Neville's algorithm [55]. The degree of interpolation determines the number of function evaluations. Typically, we use a degree of interpolation of 2 which leads to a maximum of 6 function evaluations to calculate one derivative. The algorithm is described in [55].

For more details on Quasi-Newton updates of Hessian and on Hessian calculations through central differences, one should refer to: [19], [48], [55].

3.3.2 Factorizing the Jacobian Matrix

Recall that the main step of the primal-dual algorithm is to solve the following linear system of equations

$$J(v_k)\Delta v_k = -F_\mu(v_k) \tag{3.15}$$

where $J(v_k) = F'_\mu(v_k)$ and $J(v_k)$ is given by

$$J(v_k) = \begin{pmatrix} \nabla^2_x L(x,y,z) & \nabla h(x) & 0 & -\nabla g(x) \\ \nabla h(x)^T & 0 & 0 & 0 \\ \nabla g(x)^T & 0 & -I & 0 \\ 0 & 0 & Z & S \end{pmatrix} \tag{3.16}$$

By susbstituting some terms in 3.15, we will see that we can reduce the dimension of the system to be inverted from $(n+m+2p)\times(n+m+2p)$ to $(n+m)\times(n+m)$ and that the reduced system is symmetric. The symmetry of the system is an important property. It permits the use of factorization techniques such as the Cholesky factorization or the Bunch and Parlett symmetric indefinite factorization [74, 9]. The reduction of 3.15 to a smaller symmetric system is described next.

3.15 can be written as

$$\begin{cases} \nabla_x^2 L(x,y,z)dx & +\nabla h(x)dy & & -\nabla g(x)dz & = & -f_1 \\ \nabla h(x)^T dx & & & & = & -f_2 \\ \nabla g(x)^T dx & & -ds & & = & -f_3 \\ & & Zds & +Sdz & = - & f_4 \end{cases} \tag{3.17}$$

where

$$F_\mu(x,y,s,z) = \begin{pmatrix} \nabla_x L(x,y,z) \\ h(x) \\ g(x) - s \\ ZSe - \mu e \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix} \tag{3.18}$$

From 3.17, we have

$$dz = -S^{-1} f_4 - S^{-1} Z ds \tag{3.19}$$

and

$$ds = \nabla g(x)^T dx + f_3 \tag{3.20}$$

therefore,

$$dz = -S^{-1} f_4 - S^{-1} Z \nabla g(x)^T dx \tag{3.21}$$

Substituting the expression of dz back in the first equation of 3.17, we obtain

$$\nabla_x^2(x,y,z)dx + \nabla h(x)dy + \nabla g(x) S^{-1} f_4 + \nabla g(x) S^{-1} Z \nabla g(x)^T dx = -f_1 \tag{3.22}$$

Hence 3.17 is equivalent to

$$\begin{cases} \left[\nabla_x^2 L(x,y,z) + \nabla g(x) S^{-1} Z \nabla g(x)^T\right] dx & +\nabla h(x)dy & = & -f_1 - \nabla g(x) S^{-1} f_4 \\ \nabla h(x)^T dx & & = & -f_2 \end{cases} \tag{3.23}$$

and

$$ds = \nabla g(x)^T dx + f_3 \tag{3.24}$$

$$dz = -S^{-1} f_4 - S^{-1} Z \nabla g(x)^T dx \tag{3.25}$$

As we can see, the system 3.23 is symmetric and of dimension $(n+m) \times (n+m)$. To gain memory space and to speed up the algorithm, we use fast factorization techniques for inverting 3.23.

3.3.3 The Update of μ

The update of μ remains one of the most critical issue in primal-dual techniques. It has great effect on the behavior of the algorithm. It is well understood that it corresponds to the concept of adherence to the central path and therefore to the convergence of the algorithm to the optimal point. Even though the role of μ is clear, the choice of the update formula for μ remains uncertain. In the linear programming case, μ is usually chosen as proportional to the duality gap so that reducing μ ensures a reduction in the duality gap at each iteration and eventually it becomes zero at optimality. Experiments have shown that this update is sucessful. In the nonlinear case, a similar update is used. The justification for this choice is not as straight forward as in linear programming. Attempts to explain it have already been proposed [21, 77].

We use the following update formula

$$\mu_k = \sigma_k \frac{s_k^T z_k}{p} \tag{3.26}$$

where σ_k is a parameter of the algorithm. El-Bakry et al. [21] have proposed the following values for σ_k

$$\sigma_k = 100 s_k^T z_k \quad \text{if} \quad 100 s_k^T z_k < 0.2$$

$$\sigma_k = 0.2 \quad \text{if} \quad 100 s_k^T z_k > 0.2$$

3.3.4 The Linesearches

Recall the following notations

$$v = \begin{pmatrix} x \\ y \\ s \\ z \end{pmatrix} \quad dv = \begin{pmatrix} d_x \\ d_y \\ d_s \\ d_z \end{pmatrix}$$

We start by taking a full Newton step as follows

$$\begin{aligned} x_{k+1} &= x_k + \alpha_p d_x \\ y_{k+1} &= y_k + \alpha_d d_y \\ s_{k+1} &= s_k + \alpha_s d_s \\ z_{k+1} &= z_k + \alpha_z d_z \end{aligned} \tag{3.27}$$

where

$$\begin{aligned} \alpha_p &= 1 \\ \alpha_d &= 1 \\ \alpha_s &= \frac{-1}{\min_k\{\frac{d_{s_k}}{s_k}, -1\}} \\ \alpha_z &= \frac{-1}{\min_k\{\frac{d_{z_k}}{z_k}, -1\}} \end{aligned} \tag{3.28}$$

α_s and α_z are chosen so as to ensure that s_{k+1} and z_{k+1} remain positive after the move. We define Φ as follows

$$\Phi(v) = \|F(v)\|^2 = F(v)^T F(v) \tag{3.29}$$

to be used as the merit function for the linesearch. We follow the Armijo rule for backtracking. The backtracking avoids taking steps that lead to small decrease in Φ. Refer to Dennis and Schnabel's book [19] or McCormick's book [48] for more details on backtracking and step taking rules. The Armijo condition can be expressed as follows Find the smallest integer i such that

$$\Phi(v + \overline{\alpha}2^{-i}dv) - \Phi(v) \leq \overline{\alpha}2^{-i}dv^T \nabla\Phi(v) \tag{3.30}$$

where $\overline{\alpha} = \min\{\gamma\alpha_{\max}, 1\}$, $\alpha_{\max} = \min\{\alpha_p, \alpha_d, \alpha_s, \alpha_z\}$ and γ is a parameter of the algorithm. A typical value for γ is 10^{-4}.

3.3.5 The Stopping Criterion

As mentioned before, the algorithm stops when the KKT conditions are satisfied with an accuracy of ϵ. Algebraically, the stopping rule is as follows

$$\|F_\mu(v_k)\| < \epsilon \tag{3.31}$$

In our implementation, we will rather use a normalized version of the criterion as shown below

$$\frac{\|F_\mu(v_k)\|_2^2}{1 + \|v_k\|_2^2} < \epsilon \tag{3.32}$$

ϵ is problem dependent and is given as a parameter of the algorithm.

3.4 Computational Results with General Nonlinear Programs

In this section, we report the computational results for a subset of the Hock and Schittkowski's database of nonlinear programs [32]. The dimension of these problems is small, however they are famous for their complexity. Most of these problems are highly nonconvex and ill-conditioned. We show comparisons in terms of number of iterations (Newton's steps) with the modified barrier algorithm proposed by Breitfeld and Shanno [8]. Tables 3.1 and 3.2 summarize the results. The problem numbers refer to the problems defined in the original book [32]. The columns *Iterations* and *Obj. value* under *NLPD* report the number of iterations and the optimal objective value achieved by our nonlinear primal-dual algorithm. The *Mod. Barrier Iterations* column reports the number of iterations required by Breitfeld and Shanno's code for the same problem and the *Hock and Schittkowski Obj. value* gives the theoretical optimal objective value of the problem.

In 60 percent of the cases, our algorithm required a smaller number of iterations to reach the optimal value. For other problems, the modified barrier was more efficient. We can conclude that on the average our algorithm performs as well as the modified barrier. Note that the modified barrier uses the true expression of the hessian of the objective function for each problem while our algorithm uses the central difference approximations. Despite the approximation of the hessian, the performance remains similar and the optimal objective value is reached with extremely good accuracy. These results show very good promise and motivate us to investigate improvements of the algorithm

Table 3.1: Experiments with Hock and Schittkowski's Nonlinear Problems

		NLPD		Mod. Barrier	Hock and Schittkowski
Problem	Variables	Iterations	Obj. value	Iterations	Obj. value
hs1	2	3	1.5240e-04	28	0
hs2	2	16	5.0424e-02	18	5.042e-02
hs3	2	3	1.5242e-04	16	0
hs4	2	6	2.66667	26	2.66667
hs5	2	6	-1.91322	9	-1.91269
hs6	2	43	1.658e-28	32	0
hs7	2	12	-1.73205	20	-1.73205
hs8	2	60	-1.0	7	-1.0
hs9	2	3	-0.5	3	-0.5
hs10	2	23	-1.0	26	-1.0
hs11	2	72	-8.4987	25	-8.49846
hs12	2	11	-30.0	26	-30.0
hs14	2	59	1.39337	23	1.39346
hs15	2	107	306.499	60	306.5
hs16	2	78	0.25	34	0.25
hs17	2	78	1.0	34	1.0
hs18	2	10	5.0002	36	5.0
hs19	2	27	-7950.94	40	-6961.814
hs20	2	40	40.1988	35	38.1987

Table 3.2: (Cont'd) Experiments with Hock and Schittkowski's Nonlinear Problems

		NLPD		Mod. Barrier	Hock and Schittkowski
Problem	Variables	Iterations	Obj. value	Iterations	Obj. value
hs21	2	15	-99.9599	16	-99.96
hs22	2	24	1.00023	22	1.0
hs32	3	59	1.0	45	1.0
hs47	5	49	-3.6593e-08	58	0
hs71	4	11	17.01401	48	17.01401

as we will discuss next.

Chapter 4

A Stochastic Nonlinear Primal-Dual Algorithm

4.1 Introduction

From its design, the primal-dual algorithm stops when it finds a KKT point of the nonlinear programming problem. Practically, a good local minimum is often satisfying and the cost involved in finding the global minimum is too high when compared to the corresponding gain in the objective function value. However, in applications that lead to function having a large number of local minima and few global optima such as artificial neural network training where the error surface can be very complicated, there is a need for finding the global optimum.

To help in escaping local minima, we propose to add a random noise to the objective function. These stochastic perturbations can also be seen as perturbations on the Newton's directions to visit neighborhoods of the current point and seek better solutions. As

implemented in simulated annealing approaches [38]. we introduce a schedule of temperature that controls the amount of perturbation throughout the iteration process. The perturbation should be vanishing as we approach the global optimum so as to ensure convergence of the algorithm. We also allow bad moves during the process, this means that with some probability, we will accept to move to points that lead to worse objective values. This feature gives the chance to the algorithm to escape a local minimum.

This modification of the algorithm is a heuristic and one can not ensure that the global minimum will be reached. However, simulated annealing based approaches have been very successful in practice and we expect to gain improvement from its use. Trafalis and Tutunji [65] has developed an hybrid simulated annealing/logarithmic barrier function method for minimizing the error function of artificial neural network training and significant improvements were obtained. Next, we describe the idea in more mathematical details.

4.2 The Stochastic Algorithm

Addind the random noise to the ojective function leads to a new perturbed function $\tilde{f}$ given by

$$\tilde{f}(x_k) = f(x_k) + c^k \sum_{i=1}^{n} x_i N_i^k \tag{4.1}$$

where c^k is the correction on the perturbation due to the temperature T^k. A practical choice of c^k is $c^k = \sqrt{2T^k}$. N_i^k represents the additive random noise on the i^{th} coordinate of x. N_i^k is taken as a uniform random number between -1 and 1. One can think of other choices of probability distribution such as the Gaussian distribution.

At each iteration k. the temperature T^k must be decreased. The cooling schedule plays an important role in simulated annealing and finding the right schedule is still an open problem. We found that $T^k = \rho T^{k-1}$ where $\rho = 0.9, 0.95$ or 0.99 works well in practice. The starting temperature T^0 is left as a parameter of the algorithm because its choice is often problem dependent.

From Equation 4.1, we derive the corresponding perturbation on the Newton's direction by setting $\frac{\partial \tilde{f}(x_k)}{\partial x_k} = 0$, which gives

$$\tilde{x_k} = x_k + \alpha_x d_x + \alpha_x \sqrt{2T^k} N^k \tag{4.2}$$

We will introduce a probability P of accepting the above move in the case where it leads to a worse objective value. P should be decreasing as the iteration counter k increases. A common choice in simulated annealing for P is

$$P = e^{\frac{-\Delta f}{T^k}} \tag{4.3}$$

where $\Delta f = f(\tilde{x_k}) - f(x_k)$. Note that the move 4.2 might lead to a point that is infeasible and violates one or both types of constraints h and g. The primal-dual technique does not require that the current point remains feasible. Infeasibility will be corrected by bringing the point in the neighborhood of the central trajectory. In practice, we have noticed that when starting with an infeasible point, much effort is spent in bringing the point in the feasible region. Experiments show that the algorithm performs many small steps from the infeasible point to a feasible point. In the case of linear constraints as in the error minimization problem of artificial neural network training, we propose to project the perturbed direction onto the null space of the constraint matrix. This ensures that the current point remains feasible at all times assuming that the algorithm

started from a feasible point. Note also that this feature is not necessary in the case where the constraints are of full dimension such as box constraints because the perturbed direction will always be feasible. We recall that the projection matrix Q onto the null space of a feasible region defined by a constraint matrix A is

$$Q = I - A^T(AA^T)^{-1}A \tag{4.4}$$

The stochastic nonlinear primal-dual algorithm is given next.

Stochastic Nonlinear Primal-Dual Algorithm (SNLPD)

Step 1 Set k=0, $T^k = T^0$.

Step 2 Start with $v_k = (x_k, y_k, s_k, z_k)$ where $(s_k, z_k) > 0$ and initialize $\mu_k > 0$.

Step 3 Check stopping criterion 3.11

- if Condition 3.11 is satisfied go to Step 8.
- if Condition 3.11 is not satisfied go to Step 4.

Step 4 Solve the system of linear equations 3.8 for Δv.

Step 5 Perform the linesearches to determine α_k.

Step 6 Calculate $v^{(2)} = v_k + \Lambda_k \Delta v = (x^{(1)}, y^{(1)}, s^{(1)}, z^{(1)})$ and $x^{(2)} = x^{(1)} + \alpha_x \sqrt{2T^k} N$. Compute $\Delta f = f(x^{(2)}) - f(x^{(1)})$,

If $\Delta f < 0$, $v_{k+1} = (x^{(2)}, y^{(1)}, s^{(1)}, z^{(1)})$

Else $P = e^{-\frac{\Delta f}{T^k}}$ and,

if $r < P$, $v_{k+1} = (x^{(2)}, y^{(1)}, s^{(1)}, z^{(1)})$

else $v_{k+1} = v^{(1)}$.

Step 7 $k = k + 1$, update μ_k, $T^k = \rho T^{k-1}$, go to Step 3.

Step 8 Stop with the optimal point $v^* = v_k$.

4.3 Computational Experiments with SNLPD

We test SNLPD algorithm with functions for which the global minimization problem is known to be difficult to solve [2]. Tables 4.1 to 4.4 report 10 runs of NLPD and SNLPD for 4 functions. Each run used the same starting point for both algorithms. T^0 was set to 100 and $\rho = 0.99$ for SNLPD. The functions are as follows

Function 1. Schubert Function. It has more than 760 local minima and more than 18 global minima. The minimization problem is as follows

$$min\ f(x) = \left\{\sum_{i=1}^{5} i\ \cos[(i+1)x_1 + i]\right\}\left\{\sum_{i=1}^{5} i\ \cos[(i+1)x_2 + i]\right\}$$

s.t.

$$-20 \leq x_1, x_2 \leq 20$$

The optimal objective value is -186.7309. Schubert's function is shown on Figures 4.1 and 4.2.

Function 2. Camel-back function. It has 6 local minima and 2 global minima and we solve the following minimization problem

$$min\ f(x) = (4 - 2.1x_1^2 + x_1^4/3)x_1^2 + x_1x_2 + (-4 + 4x_2^2)x_2^2$$

s.t.

$$-5 \leq x_1, x_2 \leq 5$$

The optimal objective value is -1.032

Function 3. This function has 4 local minima and one global minima. The minimization problem is as follows

$$min\ f(x) = \frac{1}{2}\left[(x_1^4 - 16x_1^2 + 5x_1) + (x_2^4 - 16x_2^2 + 5x_2)\right]$$

s.t.

$$-20 \leq x_1, x_2 \leq 20$$

The optimal objective value is -78.332.

Function 4. This function has 8 local minima and one global minima. The minimization problem is as follows

$$min\ f(x) = \frac{1}{2}\left[(x_1^4 - 16x_1^2 + 5x_1) + (x_2^4 - 16x_2^2 + 5x_2) + (x_3^4 - 16x_3^2 + 5x_3)\right]$$

s.t.

$$-20 \leq x_1, x_2 \leq 20$$

The optimal objective value is -117.498.

From Table 4.1, we see that the optimal objective value for Function 1 is never reached, this is due to the complexity of the function (see Figure 4.1). However, the average objective value obtained by SNLPD is much lower than the one from NLPD. For the Camel-back optimization problem (Function 2), the optimal objective value is -1.032. NLPD never reached the optimal value (see Table 4.2). SNLPS reached the optimal value in 40 % of the cases. For Function 3, SNLPD always reached the optimal value

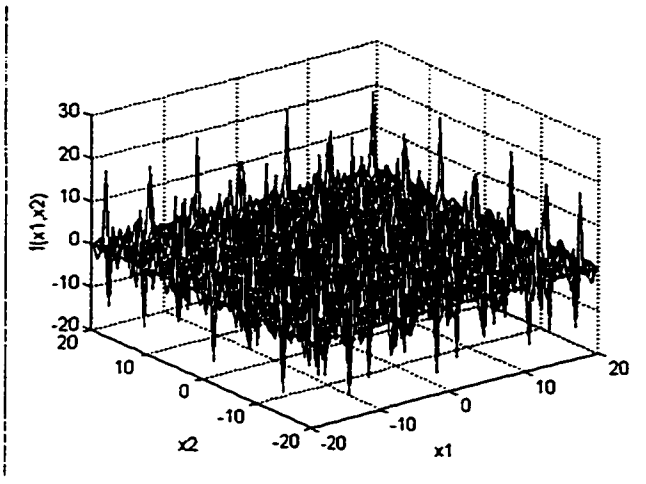


Figure 4.1: Schubert's Function Restricted to [-20;20].

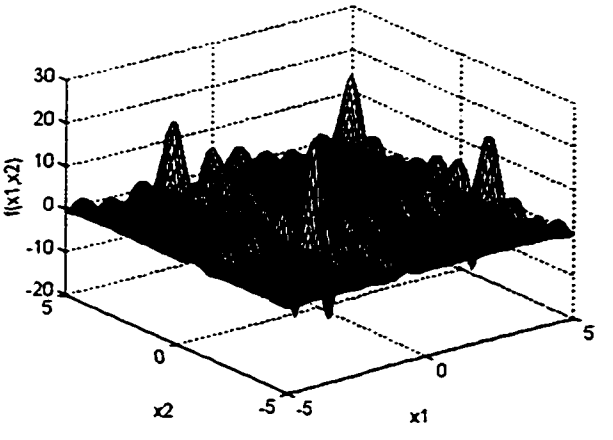


Figure 4.2: Schubert's Function Restricted to [-5;5].

Table 4.1: Comparisons of NLPD and SNLPD for Function 1

		NLPD		SNLPD	
Run #	Starting point	Objective value	# of iterations	Objective value	# of iterations
1	(1,1)	8.1955e-09	32	-64.2169	63
2	(0,0)	1.46677e-09	24	-123.577	149
3	(-5,5)	-37.6811	39	-123.577	81
4	(-9,-8)	3.07096e-09	43	-10.2636	204
5	(3.5,6.7)	4.84299e-09	40	-60.0222	68
6	(19,-19)	5.85442e-10	60	-39.5887	87
7	(1,-1)	-2.54264e-09	32	-16.8359	68
8	(-5,-19)	-1.3178e-09	50	-123.577	58
9	(17,18)	-1.78365e-09	55	-123.577	86
10	(0,-10)	-1.78365e-09	42	-23.1367	151
Mean		-3.76811	41.7	-70.8368	101.5

Table 4.2: Comparisons of NLPD and SNLPD using for Function 2

		NLPD		SNLPD	
Run #	Starting point	Objective value	# of iterations	Objective value	# of iterations
1	(1,1)	0.543718	43	-1.03163	54
2	(0,0)	-0.215464	42	-0.215464	53
3	(-4.5,4.5)	2.10425	44	-0.215464	13
4	(-4.5,-4.5)	2.10425	45	-0.215464	51
5	(3.5,4.5)	2.10425	24	-1.03163	22
6	(4.5,-4.5)	-0.215464	48	-0.215464	14
7	(1,-1)	2.10425	40	-1.03163	13
8	(-2,-3)	2.10425	43	2.10425	9
9	(2,3)	2.22947	38	-1.03163	36
10	(-0.5,1.5)	-0.215464	41	-0.215464	41
Mean		1.2648047	40.8	-0.309959	30.6

Table 4.3: Comparisons of NLPD and SNLPD for Function 3

	NLPD		SNLPD	
Run #	Objective value	# of iterations	Objective value	# of iterations
1	-78.3304	71	-50.0589	4
2	-78.3258	4	-64.1956	7
3	-78.1829	65	-64.1956	8
4	-78.294	118	0.391225	4
5	-78.3067	57	-64.1956	9
6	-78.1823	19	-50.0589	8
7	-78.3269	23	-78.3323	8
8	-78.1879	45	-64.1956	6
9	-78.3268	9	-50.0589	10
10	-78.3309	109	-38.9706	9
Mean	-78.2795	52	-52.3870	7.3

Table 4.4: Comparisons of NLPD and SNLPD for Function 4

	NLPD		SNLPD	
Run #	Objective value	# of iterations	Objective value	# of iterations
1	-116.833	109	-75.0883	6
2	-89.2251	150	0.586837	5
3	-116.602	47	-75.0883	5
4	-112.085	94	-78.1367	8
5	-113.17	75	-75.0883	7
6	-75.0883	150	-75.0883	6
7	-114.565	60	0.586837	4
8	-89.2251	150	-89.2251	6
9	-78.1367	150	-75.0883	7
10	-89.2251	150	-89.2251	6
Mean	-99.4155	113.5	-63.32	6

while NLPD reached it only once. For Function 4. SNLPD reached a close neighborhood of the optimal value in 50 % of the cases. NLPD never reached the optimal value. Clearly, from these tests on difficult functions, the stochastic version of the primal-dual algorithm gives significant improvements in the average objective value. Note that for the Camel-back problem, on the average SNLPD even required less number of iterations to reach the optimal point.

Chapter 5

An Incremental Primal-Dual Algorithm for Problems with Special Structure

5.1 Introduction

We propose to modify the generic primal-dual technique which was developed in Chapter 3 for problems that have the following form

$$
\begin{array}{lllll}
min & f(x) = \sum_{l=1}^{L} f_l(x) & & & (5.1) \\
s.t. & & & & \\
h_l(x) & = & 0 & & \\
g_l(x) & \geq & 0 & \text{for l=1,...,L} &
\end{array}
$$

This type of problems is very common in engineering applications. For example, the least square problems are one of the special instances of Problem 5.1. Our interest is the artificial neural network training problem that has also this specific structure. In this section, we develop the incremental primal-dual technique for the general problem 5.1 and in the next chapter we describe how it is applied to the neural network training problem. In the neural network community, the incremental update of the variables is known as *online training*. In general, the problems for which incremental methods are suitable are problems that deal with a large amount of data and the objective function f is a sum of given measures between the data points and parameters of the specific problem. If the given problem has L data points, the incremental technique decomposes the problem into L subproblems. The update of the variables is performed by increments. Each increment corresponds to a data point.

One benefit from using an incremental idea rather than a classical method is a memory space gain. The memory space required by the method depends only on the size of the subproblems whereas in the traditional methods one must store information from all the data points simultaneously, which can lead to memory overflows when dealing with large scale problems. Another benefit from using an incremental technique is that the data are fed several times to the system and in the case of neural network training for example, cycles of data feeds increases the chance to find a global optimum to the error function. Also, in online applications, where all the data are not available at one time but only on a one-by-one basis, incremental methods are needed.

Incremental gradient techniques have been investigated recently [4, 5, 67]. Here, we propose to use for the first time these ideas with interior point methods and more

specifically with the primal-dual algorithm previously studied.

5.2 Example

To illustrate the concept of incremental technique, we propose to study a simple example. For clarity purposes, we will take the case of a one-dimensional unconstrained convex minimization problem. We will see later how the technique changes in the case of constrained optimization.

Consider the following minimization problem

$$min f(x) = f_1(x) + f_2(x) + f_3(x) \tag{5.2}$$

where $x \in \Re$ and,

$$f_1(x) = x^2 \tag{5.3}$$

$$f_2(x) = \left(\frac{3}{4}x + 5\right)^2 \tag{5.4}$$

$$f_3(x) = \left(\frac{3}{2}x - 5\right)^2 \tag{5.5}$$

Figure 5.1 shows the plot of the functions f_i, $i = 1, 2, 3$ and the plot of the original sum function f. The incremental technique starts by computing a descent direction for the first data block (i.e. f_1). This direction can be taken as a step along the negative gradient as follows

$$d_1 = -\alpha_1 f_1'(x) \tag{5.6}$$

Starting from the point $(-10, 100)$ on the figure, the algorithm moves along d_1 so as to decrease the value of f_1. From the new point, the algorithm will now work on decreasing

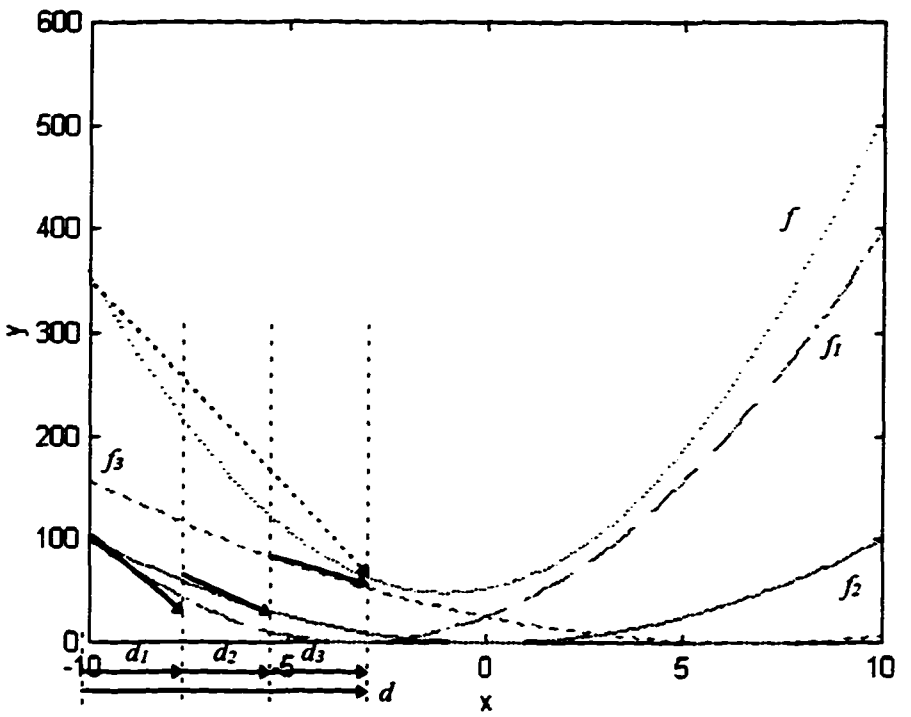


Figure 5.1: Incremental Moves for an Unconstrained Example

f_2 by moving along the following direction

$$d_2 = -\alpha_2 f_2'(x) \tag{5.7}$$

and finally, the first cycle through the data blocks is completed by moving along $d_3 = -\alpha_3 f_3'(x)$.

We see how a single cycle operates. For multiple cycles, the process is repeated starting from the ending point of the previous cycle. This constitutes the basic concept of the incremental technique. One can also see on Figure 5.1 that the resulting direction for the original function after a single cycle, corresponds to the sum of the directions of each increments, which can be written as $d = d_1 + d_2 + d_3$.

In this example, the functions are one-dimensional convex functions and the problem is unconstrained. When the functions are nonconvex, the directions defined above might not be descent directions. The addition of constraints in the problems makes also the problem much more complicated. In the following, we propose new ideas to overcome some of these problems. We consider nonconvex constrained problems and compute incremental directions of move by solving a set of KKT optimality conditions for each subproblem associated with a data block.

5.3 The Incremental Algorithm

The incremental primal-dual algorithm starts from a point $v_1^0 \in \Re^{n+m+2p}$ and generates the sequence $(v_1^t, \ldots, v_{L+1}^t)_{t=0,1,\ldots}$ where

$$\begin{aligned} v_{l+1}^t &= v_l^t + \Lambda_l^t \Delta v_l^t \qquad\qquad l=1,\ldots,L \\ v_1^{t+1} &= v_{L+1}^t \end{aligned}$$

and Δv_l^t is calculated by performing one Newton's step towards the solution of the KKT conditions for the following subproblem

$$min \quad f_l(x)$$

$$s.t.$$

$$h_l(x) = 0$$

$$g_l(x) \geq 0$$

The parameter Λ_l^t is the step length along Newton's direction.

Specifically, Δv_l^t is the solution of the linearized system of the KKT conditions

$$J(v_l^t)\Delta v_l^t \;=\; -F_\mu^l(v_l^t) \tag{5.8}$$

where $v_l^t = (x_l^t, y_l^t, s_l^t, z_l^t)$ and,

$$F_\mu^l(v_l^t) = \begin{pmatrix} \nabla_x L_l(x_l^t, y_l^t, z_l^t) \\ h_l(x_l^t) \\ g_l(x_l^t) - s_l^t \\ ZSe - \mu e \end{pmatrix} = 0 \;\; (s_l^t, z_l^t) \geq 0 \tag{5.9}$$

$$L_l(x_l^t, y_l^t, z_l^t) = f_l(x_l^t) + (y_l^t)^T h_l(x_l^t) - (z_l^t)^T g_l(x_l^t) \tag{5.10}$$

and

$$J(v_l^t) = \begin{pmatrix} \nabla_x^2 L_l(x_l^t, y_l^t, z_l^t) & \nabla h_l(x_l^t) & 0 & -\nabla g_l(x_l^t) \\ \nabla h_l(x_l^t)^T & 0 & 0 & 0 \\ \nabla g_l(x_l^t)^T & 0 & -I & 0 \\ 0 & 0 & Z & S \end{pmatrix} \tag{5.11}$$

Δv_l^t is the vector of step lengths calculated through linesearches. The update of μ is only performed after all the increments of one phase have been executed.

This method of incrementing the update of the variables can be seen as performing a Newton's step that is the geometric sum of all the Newton's directions corresponding to each one of the data points l, $l = 1, \ldots, L$. Mathematically, it can be seen as follows

$$v_1^t = v_{start} \tag{5.12}$$

$$v_2^t = v_1^t + \Lambda_1^t \Delta v_1^t \tag{5.13}$$

$$v_3^t = v_2^t + \Lambda_2^t \Delta v_2^t \tag{5.14}$$

$$v_4^t = v_3^t + \Lambda_3^t \Delta v_3^t \tag{5.15}$$

$$\vdots \tag{5.16}$$

$$v_L^t = v_{L-1}^t + \Lambda_{L-1}^t \Delta v_{L-1}^t \tag{5.17}$$

$$v_{L+1}^t = v_L^t + \Lambda_L^t \Delta v_L^t \tag{5.18}$$

Summing right and left hand sides in Equations 5.12 through 5.18 and cancelling identical terms on both sides, we obtain

$$v_{L+1}^t = v_{start} + d^t \tag{5.19}$$

where $d^t = \sum_{l=1}^{t} \Lambda_l^t \Delta v_l^t$ and v_{start} is some arbitrary starting value.

d^t represents the geometric sum of all the Newton's directions generated by each of the data point individually.

Next, we give the main steps of the incremental primal-dual algorithm for nonlinear programming.

Nonlinear Incremental Primal-Dual Algorithm (INCNLPD)

Step 1 Set $t = 0$, $l = 1$.

Step 2 Start with $v_l^t = (x_l^t, y_l^t, s_l^t, z_l^t)$ where $(s_l^t, z_l^t) > 0$ and initialize $\mu > 0$.

Step 3 If stopping criterion is satisfied, go to Step 10, otherwise go to Step 4.

Step 4 while $(l \leq L + 1)$ do Step 5, 6, 7 and 8, otherwise go to Step 9.

Step 5 Solve the system of linear equations 5.8 for Δv_l^t.

Step 6 Perform the linesearches to determine Λ_l^t.

Step 7 Move to next point $v_{l+1}^t = v_l^t + \Lambda_l^t \Delta v_l^t$.

Step 8 Set $l = l + 1$ and go to Step 4.

Step 9 Set $t = t + 1$, $v_1^t = v_{L+1}^{t-1}$, update μ and go to Step 3.

Step 10 Stop with the optimal point $v^* = v_l^t$.

Note that in Step 3 the stopping criterion can not be defined as in the general NLPD algorithm because the norm of the KKT conditions is different for each one of the increment l. Usually, in practice the stopping criterion used is a measure of accuracy

for the given application. For example in the case of artificial neural network training. the algorithm stops when the training error becomes smaller than a preset threshold. For computational experiments with the INCNLPD algorithm, we refer the reader to the chapter on application to neural network training where we use the incremental technique to train the network.

5.4 Local Convergence of INCNLPD

In this section, we prove the local convergence of the incremental nonlinear primal-dual algorithm. In other words, we show that starting from a neighborhood of the optimal solution, the sequence of iterates generated by INCNLPD converges q-linearly to that solution (See Appendix A for more information on the convergence of algorithms). Before stating the convergence theorem, we recall the fundamental equations of the incremental primal-dual technique and we introduce new notations that we will use in the proof of the theorem.

The problem to solve has the following structure

$$\begin{aligned} min \quad & f(x) = \textstyle\sum_{l=1}^{L} f_l(x) && (5.20)\\ s.t. & \\ h_l(x) \quad & = \quad 0 & \\ g_l(x) \quad & \geq \quad 0 \qquad \text{for l=1,...,L} \end{aligned}$$

The update of the variable follows the following rule

$$v^k{}_{l+1} = v_l^k + \Lambda_l^k \Delta v_l^k \qquad \text{l=1,...,L}$$

$$v_1^{k+1} = v_{L+1}^k$$

Here, we are interested in the sequence $\{v_{L+1}^0, v_{L+1}^1, v_{L+1}^2, \ldots\}$. It records the values of the variable v after each cycle through all L increments. We will show that this sequence converges to the optimal solution v^*. Note that this sequence can also be written as $\{v_1^1, v_1^2, v_1^3, \ldots\}$.

The direction Δv_l^k is calculated as follows

$$\Delta v_l^k = -J_l(v_l^k)^{-1} F_{\mu_l}^l(v_l^k) \tag{5.21}$$

$$= -J_l^{-1}\left[F^l(v_l^k) - \mu_l \hat{p}\right] \tag{5.22}$$

$$\tag{5.23}$$

where

$$F^l(v_l^k) = \begin{pmatrix} \nabla_x L \\ h_l(x_l^k) \\ g(x_l^k) - s_l^k \\ ZSe \end{pmatrix} = 0 \ (s, z) \geq 0 \tag{5.24}$$

and

$$\hat{p} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ e \end{pmatrix} \tag{5.25}$$

where e is the vector of ones, $J_l = J_l(v_l^k)$, and $\mu_k = \sigma_k min(S_k Z_k e)$ with $\sigma_k > 0$.

Λ_l^k is calculated using a line search procedure. We consider equal step lengths for all

variables. e.g. $\Lambda_l^k = diag\{\tilde{\alpha}_k\}$ with $\tilde{\alpha}_k = (\alpha_k, \alpha_k, \alpha_k, \alpha_k)$.

α_k is chosen as follows

$$\alpha_k = min(1, \tau_k \alpha_s, \tau_k \alpha_z) \tag{5.26}$$

where $\alpha_s = \frac{-1}{min_j\{\frac{ds_j}{s_j}, -1\}}$ and $\alpha_z = \frac{-1}{min_j\{\frac{dz_j}{z_j}, -1\}}$

Next, we state the assumptions on the problem in order to ensure the local convergence of the algorithm.

Assumptions 5.1 *There exists* $v^* = (x^*, y^*, s^*, z^*)$ *such that*

1. *The KKT conditions 3.3 are satisfied.*

2. $J_l(v^*)$ *is non-singular for* $l = 1, \ldots, L$.

3. J_l *is locally Lipschitz continuous at* v^*, $l = 1, \ldots, L$.

Theorem 5.1 *Consider the problem 5.20 and a solution* v^* *satisfying the assumptions 5.1. There exists a neighborhood* $D(v^*)$ *of* v^*, *and a constant* $\hat{\sigma}$ *such that for any starting point in* $D(v^*)$ *and for any* $\tau_k \in [\hat{\tau}, 1]$ *(*$\hat{\tau} \in (0, 1)$*) and* $\sigma_k \in (0, \hat{\sigma}]$, *the sequence* $v_1^1, v_1^2, v_1^3, \ldots$ *converges q-linearly to* v^*.

Proof: In order to prove the above theorem, we first need to state some preliminary results.

Lemma 5.1

$$\|v_{l+1}^k - v^*\| \leq (1 - \alpha_k)\|v_l^k - v^*\| + \mu_k \|J_l^{-1}\hat{p}\| + O(\|v_l^k - v^*\|^2) \tag{5.27}$$

proof of Lemma 5.1:

We have

$$v_{l+1}^k - v^* = v_l^k - v^* - \alpha_l J_l^{-1}\left[F^l(v_l^k) - \mu_k \hat{p}\right] \tag{5.28}$$

therefore,

$$\|v_{l+1}^k - v^*\| \leq \|v_l^k - v^* - \alpha_l J_l^{-1} F^l(v_l^k)\| + \mu_k \alpha_l \|J_l^{-1}\hat{p}\|. \tag{5.29}$$

The first term of the right hand side can be written as follows Since

$$\begin{aligned} v_l^k - v^* - \alpha_l J_l^{-1} F^l(v_l^k) &= (1-\alpha_l)(v_l^k - v^*) + \alpha_l\left[v_l^k - v^* - J_l^{-1}[F^l(v_l^k) - F^l(v^*)]\right] \\ &= (1-\alpha_l)(v_l^k - v^*) + \alpha_l J_l^{-1}\left[F^l(v^*) - F^l(v_l^k) - J_l(v^* - v_l^k)]\right] \end{aligned}$$

The first term of the right hand side can be bounded as follows

$$\|v_l^k - v^* - \alpha_l J_l^{-1} F^l(v_l^k)\| \leq (1-\alpha_l)\|v_l^k - v^*\| + \alpha_l \|J_l^{-1}\| \|F^l(v^*) - F^l(v_l^k) - J_l(v^* - v_l^k)]\| \tag{5.30}$$

From the local Lipschitz continuity of J_l, we can write

$$\|F^l(v^*) - F^l(v_l^k) - J_l(v^* - v_l^k)]\| \leq \frac{\gamma}{2}\|v^* - v_l^k\|^2 \tag{5.31}$$

where γ is a positive constant. See for example [19] for a proof of this result.

This leads to

$$\|v_{l+1}^k - v^*\| \leq (1-\alpha_l)\|v_l^k - v^*\| + \mu_k \alpha_l \|J_l^{-1}\hat{p}\| + \frac{M\gamma}{2}\alpha_l \|v^* - v_l^k\|^2 \tag{5.32}$$

where M is a positive constant such that $\|J_l^{-1}\| \leq M$ (the existence of M is ensured by the nonsingularity of J_l).

Therefore,

$$\|v_{l+1}^k - v^*\| \leq (1-\alpha_l)\|v_l^k - v^*\| + \mu_k \alpha_l \|J_l^{-1}\hat{p}\| + O(\|v^* - v_l^k\|^2) \tag{5.33}$$

Q.E.D.

Lemma 5.2

$$\alpha_l = min(1, \tau_l + O(\sigma_l) + O(\|F^l(v_l^k)\|) \tag{5.34}$$

proof of Lemma 5.2:

This result has been derived for the non-incremental nonlinear primal-dual method. It does not involve any particular difficulties. See [21] for a proof of Lemma 5.2 . **Q.E.D.**

We are now ready to prove the local convergence theorem. The idea is to apply Lemma 5.1 at each increment level and write the recursive rule between increments so as to conclude on the asymptotic behavior of two iterates of the sequence $\{v_1^1, v_1^2, v_1^3, \ldots\}$.

From Lemma 5.1, we have

$$\begin{aligned}
\|v_2^k - v^*\| &\leq (1-\alpha_1)\|v_1^k - v^*\| + \mu_k\alpha_1\|J_1^{-1}\hat{p}\| + O(\|v^* - v_1^k\|^2) \\
\|v_3^k - v^*\| &\leq (1-\alpha_2)\|v_2^k - v^*\| + \mu_k\alpha_2\|J_2^{-1}\hat{p}\| + O(\|v^* - v_2^k\|^2) \\
\|v_4^k - v^*\| &\leq (1-\alpha_3)\|v_3^k - v^*\| + \mu_k\alpha_3\|J_3^{-1}\hat{p}\| + O(\|v^* - v_3^k\|^2) \\
\vdots &\leq \vdots \\
\|v_{L+1}^k - v^*\| &\leq (1-\alpha_L)\|v_L^k - v^*\| + \mu_k\alpha_L\|J_L^{-1}\hat{p}\| + O(\|v^* - v_L^k\|^2)
\end{aligned}$$

Substituting each equation in the right hand side of the next equation recursively. one would obtain the following

$$\begin{aligned}
\|v_{L+1}^k - v^*\| &= \left[\prod_{l=1}^{L}(1-\alpha_l)\right]\|v_1^k - v^*\| \\
&+ \mu_k\sum_{l=1}^{L}\left(\prod_{j=1}^{L-l-1}(1-\alpha_{L-j})\right)\|J_l^{-1}\hat{p}\|
\end{aligned}$$

$$+ \sum_{l=1}^{L} \left(\prod_{j=1}^{L-l-1} (1 - \alpha_{L-j}) \right) O(\|v_l^k - v^*\|^2)$$

Using Lemma 5.2, we can write

$$\begin{aligned}
\|v_{L+1}^k - v^*\| \leq & \left[\prod_{l=1}^{L} (1 - \tau_l + O(\sigma_l) + O(\|v_l^k - v^*\|) \right] \|v_1^k - v^*\| \\
& + \mu_k \sum_{l=1}^{L} \left(\prod_{j=1}^{L-l-1} (1 - \tau_{L-l} + O(\sigma_{L-l}) + O(\|v_{L-l}^k - v^*\|)) \right) \|J_l^{-1} \hat{p}\| \\
& + \sum_{l=1}^{L} \left(\prod_{j=1}^{L-l-1} (1 - \tau_{L-l} + O(\sigma_{L-l}) + O(\|v_{L-l}^k - v^*\|)) \right) O(\|v_l^k - v^*\|^2)
\end{aligned}$$

We also have

$$\mu_k = \sigma_1 O(\|F(v_1^k)\|) = \sigma_1 O(\|v_1^k - v^*\|) \tag{5.35}$$

therefore,

$$\|v_{L+1}^k - v^*\| \leq \left[\prod_{l=1}^{L} (1 - \tau_l + O(\sigma_l) + O(\|v_l^k - v^*\|)) \right] \|v_1^k - v^*\| \tag{5.36}$$

or, there exists a positive constant $c < 1$, such that

$$\|v_{L+1}^k - v^*\| \leq c \|v_1^k - v^*\| \tag{5.37}$$

or,

$$\|v_1^{k+1} - v^*\| \leq c \|v_1^k - v^*\| \tag{5.38}$$

This shows that the sequence $\{v_1^k\}$ converges q-linearly to v^*. **Q.E.D.**

Chapter 6

Application to Artificial Neural Network Training

6.1 Artificial Neural Networks

The earliest form of Artificial Neural Networks (ANNs) was developed in the 40's. It was introduced by McCulloch and Pitts [49] as network of simple logical units. Since then, many forms of ANNs have been elaborated and their capabilities have greatly improved. Hopfield and Tank [33] have shown that ANNs can even be used to solve problems as hard as the Traveling Salesman Problem. Today, ANNs are applied in many different fields. They have been applied to pattern and speech recognition [42, 39], medical diagnosis [44], business [30], neurocontrol [51], meteorology [46] and many other applications.

ANNs are often thought as black-box systems, but they can actually be represented as mathematical quantities. Hecht-Nielsen [31] have proven the Kolmogorov's mapping

neural network existence theorem which is an application of the original Kolmogorov theorem on continuous functions [40]. The theorem states that every continuous function can be represented as the sum of single variable functions that do not depend on the original function. Hecht-Nielsen applied the theorem to ANN to claim that for every continuous function, there exists an ANN that can represent it. Specifically, the Kolmogorov's theorem was restated for the representation of arbitrary continuous functions from the n-dimensional cube $[0,1]^n$ to the real numbers in terms of functions of only one variable. It was shown that given any continuous function f, f: $[0,1]^n \rightarrow \Re^m$. f can be implemented exactly by a 3 layers feedforward neural network with $2n+1$ hidden nodes. This result is of great importance for the application of ANN to function approximation problems. However, it is an existence theorem and it does not give any constructive algorithm to find the appropriate architecture.

The training of ANNs is a difficult task. The backpropagation algorithm (BP) is the most fundamental technique for training feedforward ANNs. It was first developed by Werbos [71] and later studied by Rumelhart et al. [58, 47]. The idea is to backpropagate the output errors to the hidden layers by using what is known as the generalized delta rule. The algorithm was able to give a correct mapping for inputs that are not linearly separable such as the famous exclusive OR problem (XOR) used by Minsky and Papert [52] as a counterexample to show that single layer ANNs were not able to map correctly nonlinearly separable sets of inputs.

Since the backpropagation algorithm, a great deal of effort has been given to increase the speed of training. Successfully, algorithms based on the BP framework were proposed. For instance Battiti [3] describes second-order methods that use the BP scheme.

Instead of using the on-line updating rule for the connection weights, these methods follow Newton steps allowing a quadratic convergence to the desired output. Today, one of the new direction is to use interior point methods to solve the training error minimization problem. Some work has already given successful results [62, 65, 63, 64, 14]. This research is a new contribution to training techniques with interior point methods.

6.2 Problem Statement and Training Algorithms

In this research we will only consider feedforward artificial neural networks. All the connections between neurons are forward and there are no connections between neurons belonging to the same layer. We are interested in the case of multilayer ANNs. Typically, the ANN will have one input layer, one hidden layer and one output layer. The training methods that we will develop can be easily adapted to the case where multiple hidden layers are present. Furthermore, we will assume that the ANN is fully connected. If for some reasons some connections must be deleted between neurons, this can be achieved by adding extra constraints in the error minimization problem to be solved. Figure 6.1 illustrates the ANN architecture used in this research.

The problem to be solved is known as supervised ANN training. Given a set of input patterns and their corresponding desired outputs for the networks, the optimal set of connection weights between neurons that achieved a minimum error between the desired outputs and the actual outputs of the neurons is to be found. The network is said to be trained when the optimal connection weights are found.

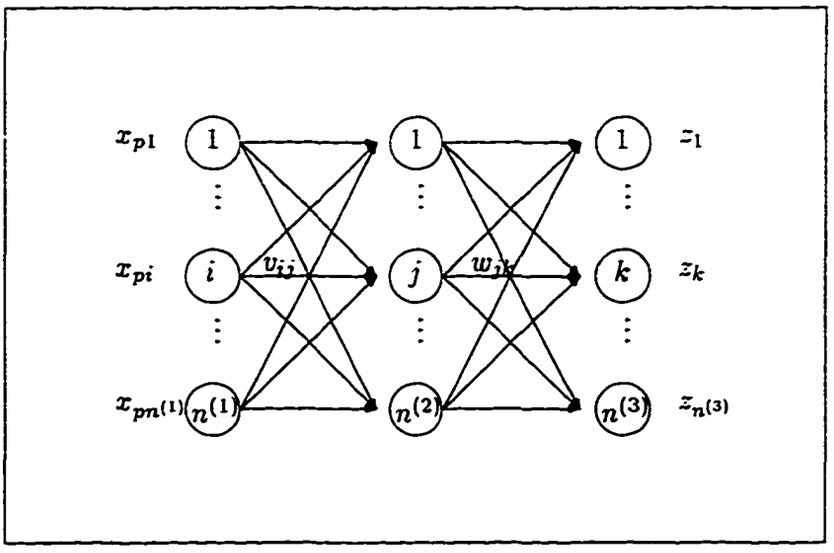


Figure 6.1: Multilayer Artificial Neural Network

The classical methodologies to train such ANNs are based on the backpropagation concept first presented by Werbos [71]. The idea is to iteratively update the weights, starting from the output layer where complete information is given, and backpropagate the information to the layers that precede it. The update of the weights is derived from the gradient descent technique. Variants that used Quasi-Newton techniques have also been proposed [3]. The backpropagation algorithm (BP) has given very good results and it is still used as a benchmark algorithm for its robustness. However, BP leads to long training times and its use in online applications that require fast training phases is therefore inappropriate.

We propose to develop a completely new approach for supervised training of ANNs. We avoid the use of a backpropagation scheme and instead we write the training problem as a more general problem that can benefit from state of the art techniques of optimization. Instead of considering the ANN as a succession of layers and train each layer

at a time, we assimilate the ANN as a mathematical function as a whole and write the training problem as an error minimization problem. By constraining the weights between bounds, we avoid very large weights that can lead to what is known as network paralysis [70]. Large weights can create very large output values and the neurons have more difficulties to fire. Then the ANN becomes paralyzed.

The training problem is now seen as a general nonconvex constrained minimization problem with linear constraints. We will use the algorithms presented in the previous chapters to solve this problem. Next, we give a mathematical formulation of the problem and show how it can be solved.

6.2.1 Notations

The following notations are to be used next

- P - Number of input patterns.
- $n^{(1)}$ - Number of input neurons.
- $n^{(2)}$ - Number of hidden neurons.
- $n^{(3)}$ - Number of output neurons.
- f_a - Common activation function for all the neurons.
- $X_p = (x_{p1}, x_{p2}, \ldots, x_{pn^{(1)}})^T$ - Input vector corresponding to pattern p.
- $D_p = (d_{p1}, d_{p2}, \ldots, d_{pn^{(3)}})^T$ - Desired output vector corresponding to pattern p.
- v_{ij} - Weight on the connection from i^{th} input neuron to j^{th} hidden neuron.

- w_{jk} - Weight on the connection from j^{th} hidden neuron to k^{th} output neuron.
- $y = (y_1, y_2, \ldots, y_{n^{(2)}})^T$ - Output vector of the hidden layer.
- $z = (z_1, z_2, \ldots, z_{n^{(3)}})^T$ - Output vector of output layer.

From the previous notations, we can already write the following

$$y_j = f_a(\sum_{i=1}^{n^{(1)}} v_{ij} x_{pi}) \text{ for all } j = 1, \ldots, n^{(2)}. \tag{6.1}$$

and,

$$z_k = f_a(\sum_{j=1}^{n^{(2)}} w_{jk} y_j) \text{ for all } k = 1, \ldots, n^{(3)}. \tag{6.2}$$

The activation functions for the neurons are assumed to be all equal. The case where not all the neurons have the same activation function can be easily handled by our methodology. A typical choice for the activation function is the hyperbolic tangent defined as

$$f_a(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{6.3}$$

6.2.2 The Error Minimization Problem

With the use of the L_2-norm as an error measure, the error minimization problem can be written as follows

$$\begin{aligned} &min\ E(v, w) \\ &s.t. \\ &-M \leq\ v\ \leq\ +M \\ &-M \leq\ w\ \leq\ +M \end{aligned} \tag{6.4}$$

where

$$E(v, w) = \frac{1}{2} \sum_{p=1}^{P} \sum_{k=1}^{n^{(3)}} (z_k - d_{pk})^2 \tag{6.5}$$

The error function $E(v, w)$ is calculated at the output of the ANN. therefore the activation functions f_a's are nested under the term z_k. This makes the function highly nonlinear and also highly nonlinear. We will use the nonlinear primal-dual algorithms developed in the previous chapter to find a minimizer of $E(v, w)$.

Next, we give an example of a simple ANN architecture and we plot the error surface (See Figure 6.3). Some of the weights have been fixed to 1 so that the dimension of the weight space is equal to 2. Figure 6.2 shows the ANN. The error function to be minimized is as follows

$$
\begin{aligned}
E(w_1, w_2) = & \left[0 - \frac{1}{1+e^{-1}}\right]^2 \\
& + \left[1 - \frac{1}{1+e^{-\left(\frac{1}{1+e^{-w_2}} + \frac{1}{1+e^{-w_1}}\right)}}\right]^2 \\
& + \left[1 - \frac{1}{1+e^{-\left(\frac{1}{1+e^{-w_1}} + \frac{1}{1+e^{-1}}\right)}}\right]^2 \\
& + \left[1 - \frac{1}{1+e^{-\left(\frac{1}{1+e^{-(w_1+1)}} + \frac{1}{1+e^{-(w_2+1)}}\right)}}\right]^2
\end{aligned}
\tag{6.6}
$$

Figure 6.3 shows that for this simple ANN, the error surface has two local minima and one global minimum. One can imagine that as the number of weights increases, the function becomes more complicated and the number of local minima increases. It is possible that the training algorithm gets trapped in a local minimum and never reaches the optimal solution.

6.2.3 A Variant Approach to the Training Problem

Figure 6.4 recalls the shape of the activation function f_a. To ensure that the neurons are not saturated and that they are able to fire at all times, we would like to remain

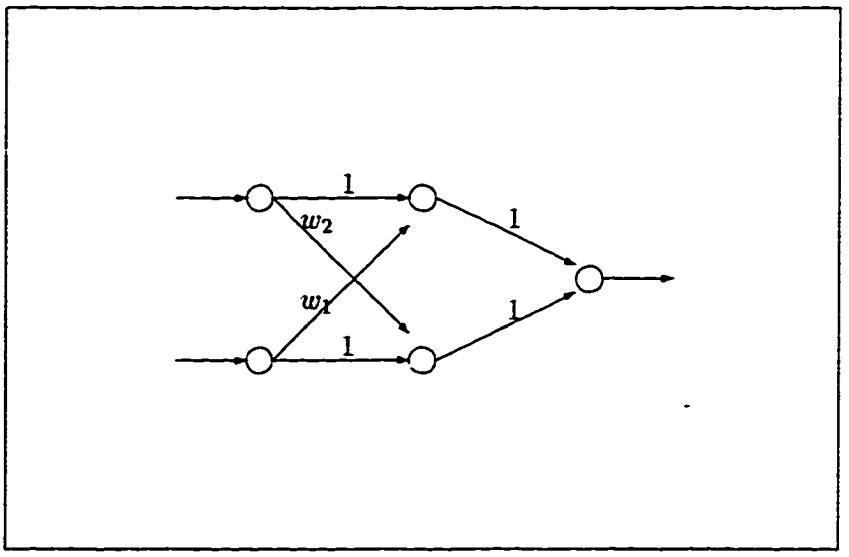


Figure 6.2: Example of Simple ANN Architecture

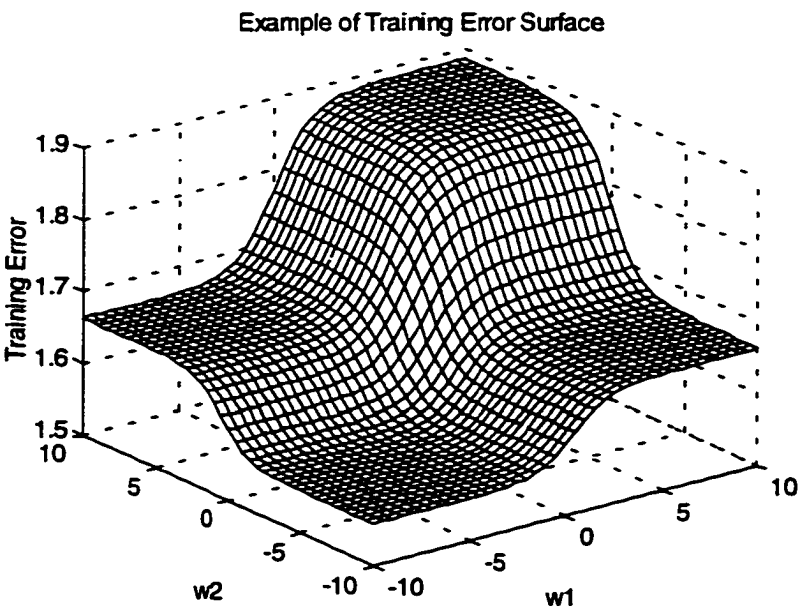


Figure 6.3: Example of Error Surface.

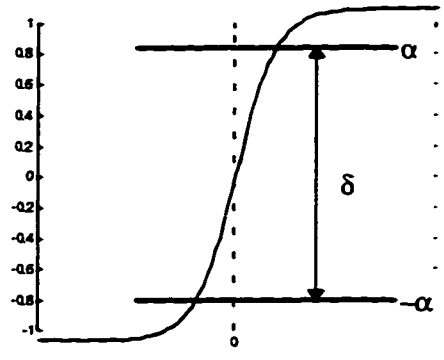


Figure 6.4: Hyperbolic Tangent Activation Function.

in the region δ (see Figure 6.4) of the curve where it has a linear behavior. With this in mind, rather than imposing box constraints on the weights, we will constrain the weights between the input and hidden layers as follows

$$-\overline{\alpha} \leq f_a(\sum_{i=1}^{n^{(1)}} v_{ij}x_{pi}) \leq \overline{\alpha} \; for \; j = 1, \ldots, n^{(2)} \tag{6.7}$$

f_a being symmetric around 0 and one-to-one mapping in the region δ, the above inequality is equivalent to

$$-f_a^{-1}(\overline{\alpha}) \leq \sum_{i=1}^{n^{(1)}} v_{ij}x_{pi} \leq f_a^{-1}(\overline{\alpha}) \; for \; j = 1, \ldots, n^{(2)} \tag{6.8}$$

where $\overline{\alpha}$ is a preset value, typically $0.7 \leq \overline{\alpha} \leq 0.9$.

Therefore, the training problem can be written as the following new error minimiza-

tion problem

$$
\begin{aligned}
&min && E(v, w) \\
&s.t. && \\
& && for\ p = 1, \ldots, P \\
& && -f_a^{-1}(\overline{\alpha}) \leq \textstyle\sum_{i=1}^{n^{(1)}} v_{i1} x_{pi} \leq f_a^{-1}(\overline{\alpha}) \\
& && -f_a^{-1}(\overline{\alpha}) \leq \textstyle\sum_{i=1}^{n^{(1)}} v_{i2} x_{pi} \leq f_a^{-1}(\overline{\alpha}) \\
& && \vdots \\
& && -f_a^{-1}(\overline{\alpha}) \leq \textstyle\sum_{i=1}^{n^{(1)}} v_{in^{(2)}} x_{pi} \leq f_a^{-1}(\overline{\alpha})
\end{aligned}
$$

where $E(v, w) = \frac{1}{2} \sum_{p=1}^{P} \sum_{k=1}^{n^{(3)}} (z_k - d_{pk})^2$. We see that this formulation possesses the special structure required by the incremental primal-dual technique developed in the previous chapter. By using this scheme, the number of constraints is greatly reduced. We had $2[n^{(2)}(n^{(1)}n^{(3)})]$ box constraints whereas now we have $2n^{(2)}$ weighted-sum constraints. Note that at each step of the incremental algorithm, we only consider the constraints corresponding to the current pattern. The information from the previous constraints is stored in the incremental direction of move. This type of training is known as online training of the data. One pattern is fed to the system at a time. In the next section, we show some computational comparisons between the box constraints version and this version of the training algorithm.

Table 6.1: XOR2 Problem

Inputs		Output
x1	x2	d
0	0	0
0	1	1
1	0	1
1	1	0

6.3 Computational Experiments with Odd-Parity Problems

We train a 3 layer ANN with the XOR2 problem data. The data are shown on Table 6.1. We use 4 hidden nodes. The ANN is trained using 3 versions of the algorithm. The notation for the different codes is as follows

TBB Batch Training Algorithm with Box Constraints

TOB Online Training Algorithm with Box Constraints (Using Incremental Primal-Dual Technique)

TOSN Online Training Algorithm with Weighted-Sum Constraints and Noise (Using Hybrid Stochastic/Incremental Primal-Dual Technique)

TBB and TOB constrain the weights into box contraints. TBB takes all the patterns at one time whereas TOB takes the patterns one by one in an incremental fashion.

Table 6.2: ANN Training with XOR2 Problem using TBB.

TBB Algorithm			
Run	Error	CPU time (s)	Iterations
1	0.0966	2	14
2	0.0944	3	17
3	0.0783	2	16
4	0.0702	3	16
5	0.0672	3	16
6	0.0966	3	18
7	0.0976	3	15
8	0.0832	3	14
9	0.0712	2	13
10	0.973	3	18
Mean	0.0853	2.7	15.7

TOSN is an hybrid algorithm, it is an incremental method and also uses the stochastic modification of the primal-dual technique. The starting cooling temperature T^0 is taken as 0.01 and the schedule parameter ρ is 0.95. Tables 6.2,6.3 and 6.4 report 10 runs of each of the algorithms. Table 6.5 is a comparison of the mean values of the training errors, CPU times and numbers of iterations.

TOB is much slower than TBB and TOSN. This shows that the box constraints are not well suitable to online training. There are as many constraints as in the batch

Table 6.3: ANN Training with XOR2 Problem using TOB.

TOB Algorithm			
Run	Error	CPU time (s)	Iterations
1	0.0970	31	56
2	0.0949	24	43
3	0.0360	24	43
4	0.0884	23	43
5	0.0541	20	35
6	0.0399	21	38
7	0.0884	24	43
8	0.0685	25	43
9	0.0947	15	27
10	0.762	14	26
Mean	0.0738	22.1	39.7

Table 6.4: ANN Training with XOR2 Problem using TOSN.

TOSN Algorithm			
Run	Error	CPU time (s)	Iterations
1	0.01638	2	11
2	0.08360	3	15
3	0.01550	3	15
4	0.01023	3	20
5	0.00259	5	29
6	0.06498	1	8
7	0.06613	4	22
8	0.04802	2	8
9	0.00139	3	16
10	0.01348	5	23
Mean	0.03223	3.1	16.7

Table 6.5: Comparisons of Training Algorithm for XOR2 Problem

Algorithm	Mean Error	Mean CPU time (s)	Mean Iterations
TBB	0.0853	2.7	15.7
TOB	0.0738	22.1	39.7
TOSN	0.03223	3.1	16.7

case but the KKT conditions must be solved many more times because of the successive increments. However, TOSN is much faster because the weighted-sum constraints are added one at a time to the system and the dimension of the KKT system is kept relatively small in comparison with the box constraint case. The stochastic perturbation helps also in achieving small training error. For larger problems, we expect TOSN to perform better than TBB. For large problems, TBB will have to store all the patterns at the same time and the KKT system to be solved will be very large and memory space overflows might be encountered.

Chapter 7

Application to Financial Forecasting

7.1 Introduction

In finance, one would like to predict the behavior of some market indicators so as to increase the return on investment. The problem of financial forecasting is complicated. Its complexity is due to the number of factors that can influence the behavior of the market. Usually, experts isolate a number of significant factors that are either derived from historical data or estimated subjectively from experience. Recently, a different approach has been shown to be very successful. The idea is to use machine learning to detect patterns in the financial history of a specific market and try to identify these patterns in the present and future data. Neural networks have been shown to be very accurate for this specific application [72, 56, 57, 17].

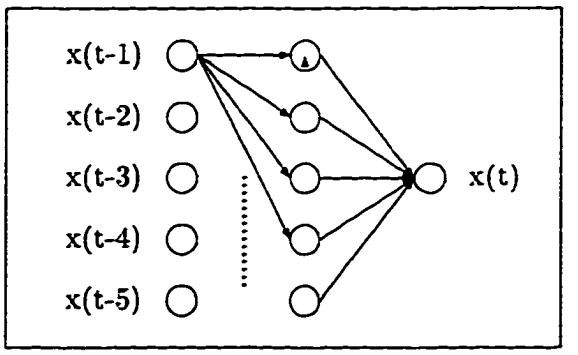


Figure 7.1: Artificial Neural Network Architecture for Financial Forecasting

7.2 Experiments

We present computational results for two types of financial data. The first database contains the values of return rate for the BOEING stock price over time (days). The second database is a record of the exchange rate from the Swiss Franc to US dollars. It has been recorded every minute. We propose to use a 3 layers neural network architecture to learn the data. We use data from the 5 previous time steps as input, and the next data point as output. Therefore the input layer has 5 input nodes, the output layer has one output node and we use 5 nodes for the hidden layer. Our experiments have shown that 5 hidden nodes gave the best architecture for this application. Figure 7.1 describes the precise ANN architecture.

We train the ANN using 5 different methodologies as listed below

- A backpropagation algorithm. It is a gradient descent technique that uses information of the output layer to compute gradients at the hidden layer.

- A Recursive Prediction Error Method (RPEM)[65]. It is Quasi-Newton modification of the backpropagation algorithm where the update of the approximation of the hessian matrix follows the RPEM update [59].

- A logarithmic barrier method (Log. Barrier)[65, 66]. The algorithm embeds the box constraints on the weights as a logarithmic penalty term into the objective function. It is an interior point method that uses the backpropagation framework to compute the gradients.

- A Stochastic logarithmic barrier method (Stoc. Log. Barrier)[66]. It is a stochastic version of the previous algorithm. It inflicts random perturbation on the search directions to seek better local minima and hopefully global minima.

- The incremental nonlinear primal-dual algorithm (INCNLPD) developed above.

Tables 7.1 and 7.2 report the means and variances of the training error for ten runs of each algorithm. Figures 7.2 to 7.11 show the training and testing results for each algorithm for the two databases. For the BOEING stock database, the x axis represents days and the y axis is the stock return. In Figures 7.2 to 7.6, the plain line is the real output of the ANN while the '+' line is the desired output. In Figures 7.7 to 7.11, the plain line is also the real output and the dotted line is the desired output. The training is performed over 40 days, and prediction is performed from day 40 to day 58. For the Swiss Franc Exchange database, the x axis represents minutes and the y axis is the exchange rate. The training is performed over 100 patterns and the next 100 patterns are used to evaluate the prediction. From the tables and figures, it is clear that INCNLPD outperforms the backpropagation algorithm. The backpropagation technique

stops with local minima that often are not good enough to train the data. This is due to the fact that the backpropagation is a gradient descent technique that only uses local information on the error surface while INCNLPD uses primal and dual information as well as perturbation on the KKT conditions to seek good local minima. INCNLPD performs as good as the Stoc. Log. Barrier method. The testing phases are very similar for both algorithms. This is very encouraging because the Stoc. Log. Barrier algorithm was shown to be very effective for function approximation [65]. INCNLPD seems to be more robust than RPEM or Log. Barrier methods. It gives more consistent results. For the Swiss Franc database, the Log. Barrier was not able to train the data. We have also train the same ANN architecture with the *NevProp* implementation of the fast backpropagation algorithm [54]. For the BOEING database, the mean training error achieved for 10 runs was of 0.31555 with zero variance and for the Swiss Franc database, the mean training error was of 0.00326 with also zero variance. Comparing these results with those of Tables 7.1 and 7.2, INCNLPD is also much better.

Table 7.1: Training Results for Boeing Stock Price Return

Algorithm	Mean of Training Error	Variance of Training Error
Backpropagation	0.131106	0.000312879
RPEM	0.605977	0.495492
Log. Barrier	3.6606	63.9709
Stoc. Log. Barrier	0.032735	3.07687E-05
INCNLPD	0.05999418	6.13E-12

Table 7.2: Training Results for Swiss Franc Exchange Rate

Algorithm	Mean of Training Error	Variance of Training Error
Backpropagation	0.000821972	3.84808E-12
RPEM	7.45659E-05	6.56031E-10
Log. Barrier	9.54193	11.2404
Stoc. Log. Barrier	0.000189679	4.17968E-09
INCNLPD	0.00010937982	3.9536324E-11

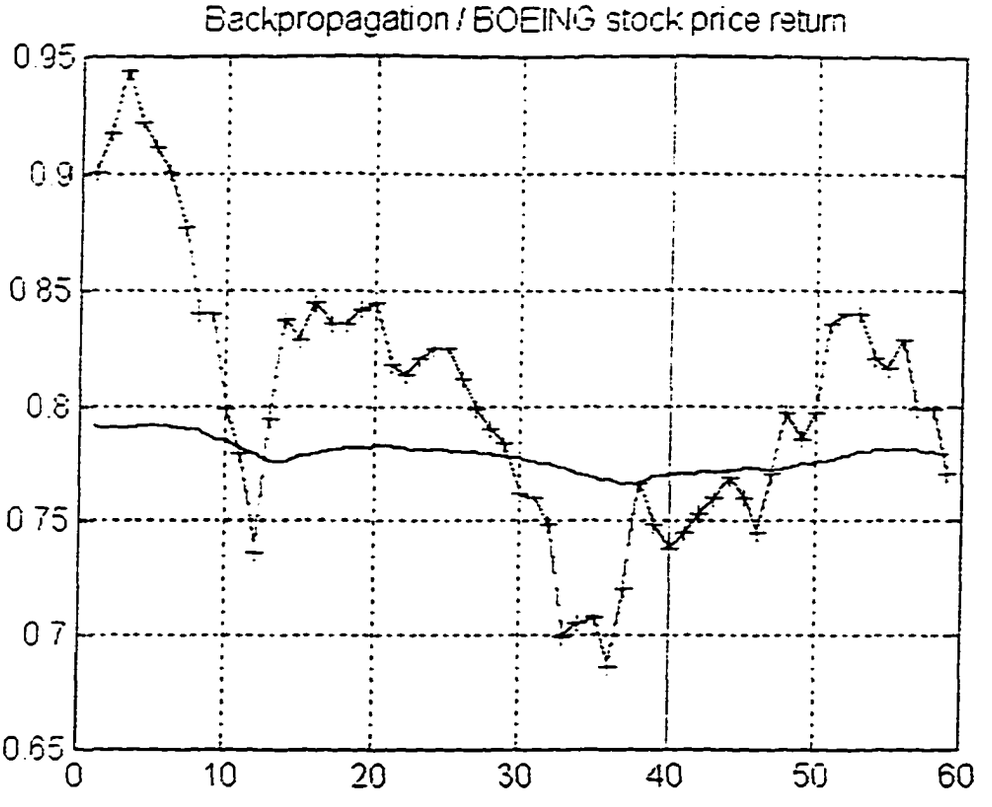


Figure 7.2: Training (Time<40) and Testing (Time>40) for Boeing stock price return

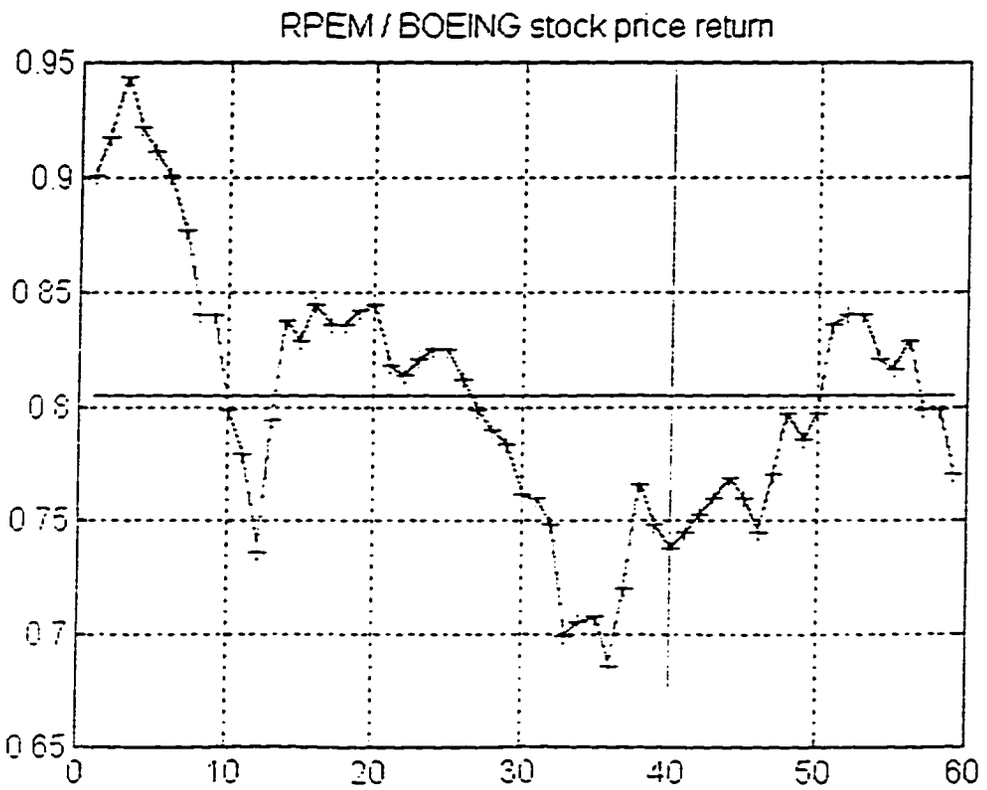


Figure 7.3: Training (Time<40) and Testing (Time>40) for Boeing stock price return

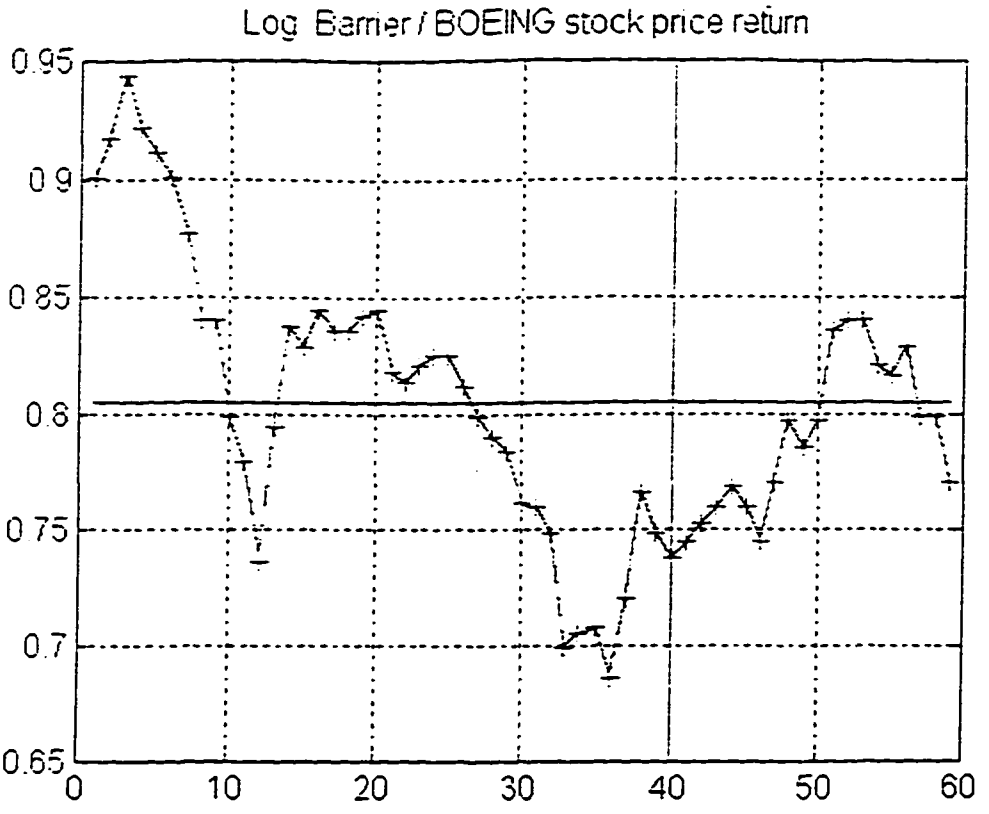


Figure 7.4: Training (Time<40) and Testing (Time>40) for Boeing stock price return

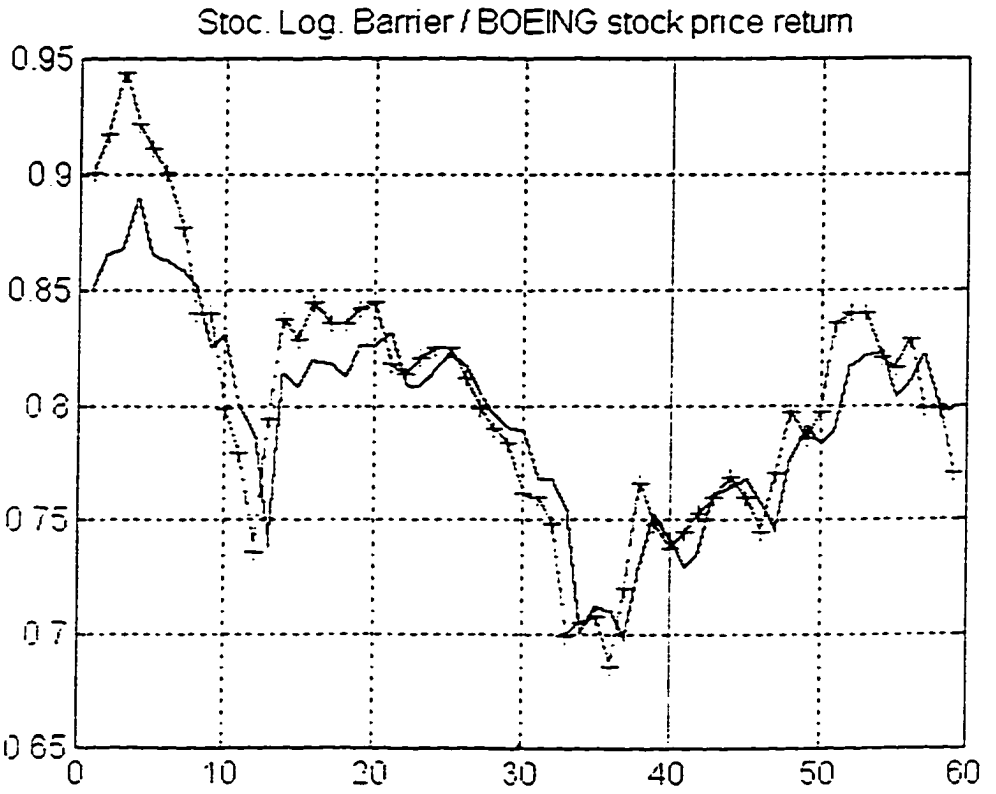


Figure 7.5: Training (Time<40) and Testing (Time>40) for Boeing stock price return

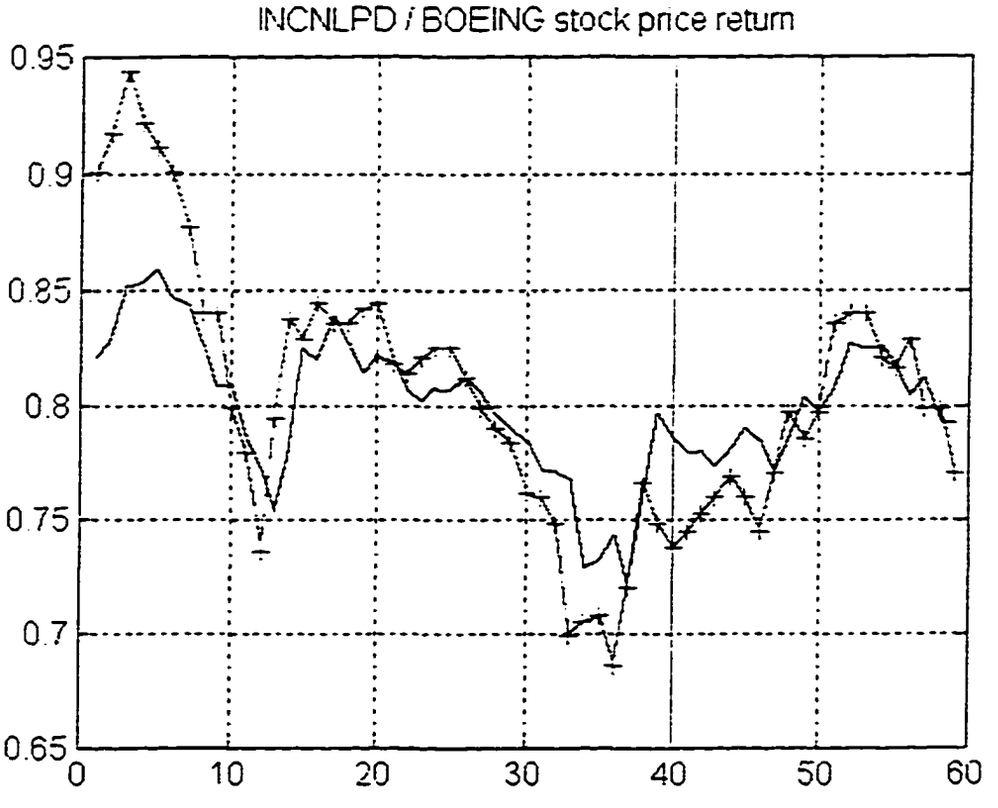


Figure 7.6: Training (Time<40) and Testing (Time>40) for Boeing stock price return

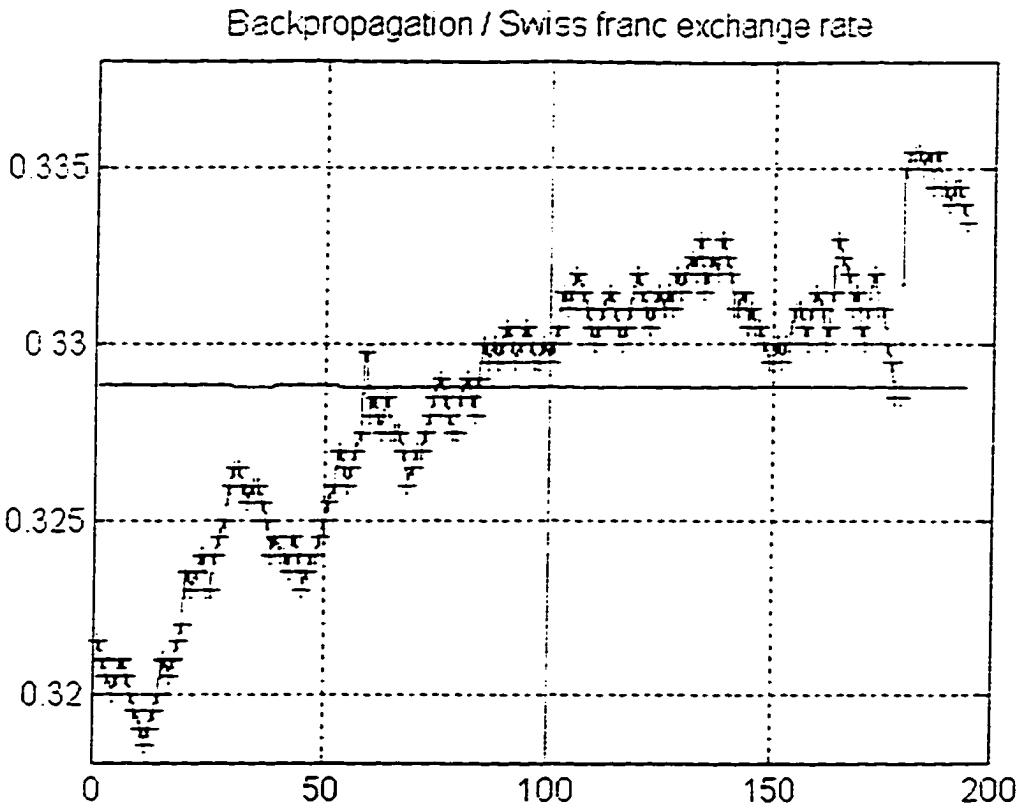


Figure 7.7: Training (Time<100) and Testing (Time>100) for Swiss Franc exchange rate

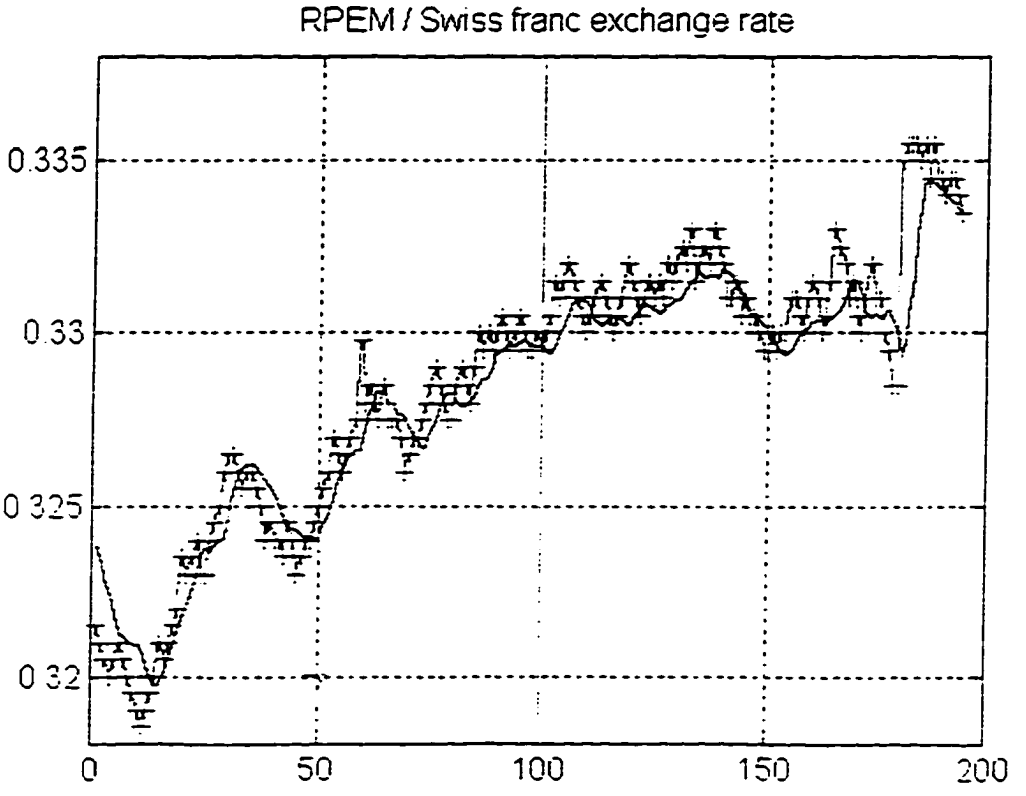


Figure 7.8: Training (Time<100) and Testing (Time>100) for Swiss Franc exchange rate

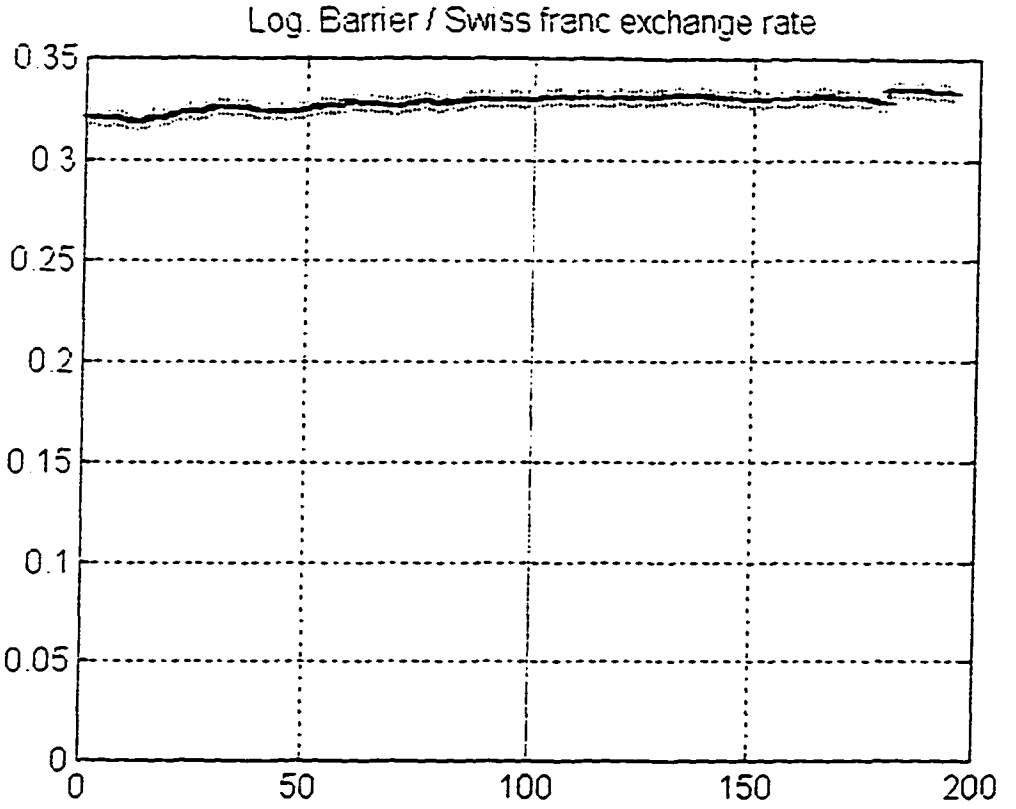


Figure 7.9: Training (Time<100) and Testing (Time>100) for Swiss Franc exchange rate

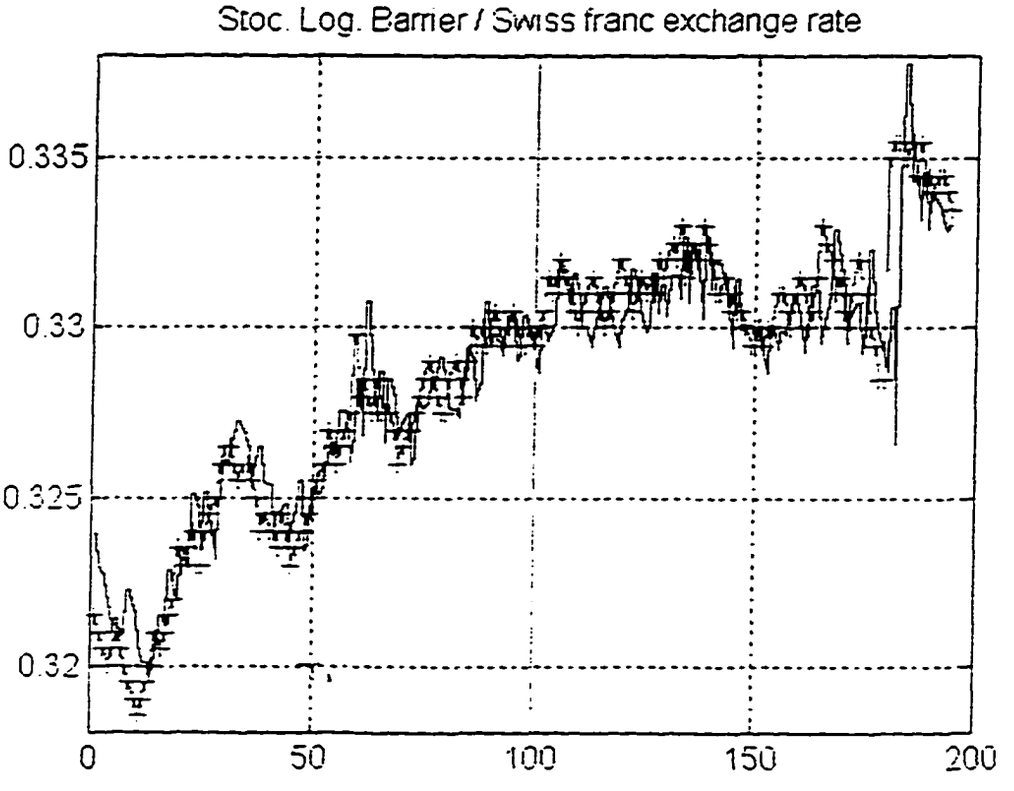


Figure 7.10: Training (Time<100) and Testing (Time>100) for Swiss Franc exchange rate

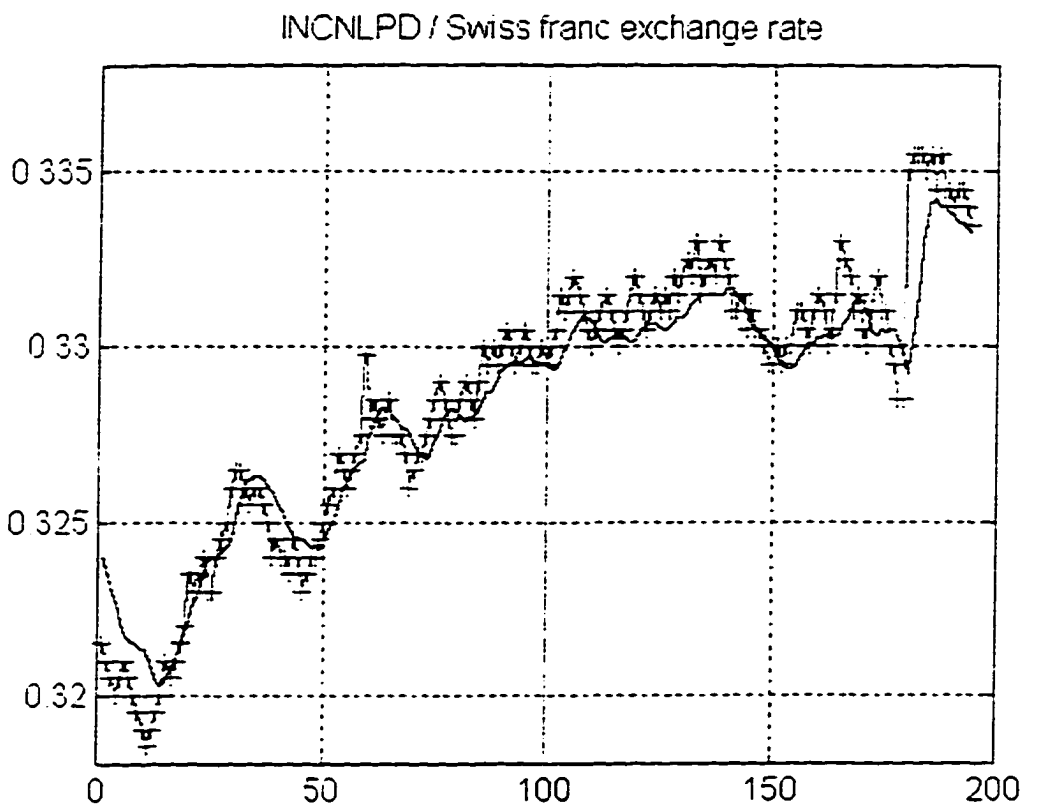


Figure 7.11: Training (Time<100) and Testing (Time>100) for Swiss Franc exchange rate

Chapter 8

Application to Medical Diagnosis and Classification Problems

In this chapter, we provide additional computational results. We train a 3 layer ANN with a subset of the Breast Cancer Database [44] and the Iris Plant Database [20]. The databases are split into two subsets, a training subset and a testing subset. We use 9 input nodes, 10 hidden nodes and 1 output node for the cancer database and 4 input nodes, 5 hidden nodes and 1 output node for the Iris database. Figures 8.1 and 8.2 show the training and testing results for the cancer database. The x-axis represents the patient number and the y-axis is the diagnosis. An output '1' corresponds to the diagnosis 'malignant' and an output '0' corresponds to the diagnosis 'begnin'. The ANN is trained to perform a diagnosis. Figures 8.3 and 8.4 show the training and testing results for the Iris database. There are 3 classes of plants, the class represented as '-1', the class '0' and the class '1'. The ANN is trained to identify the classes. From

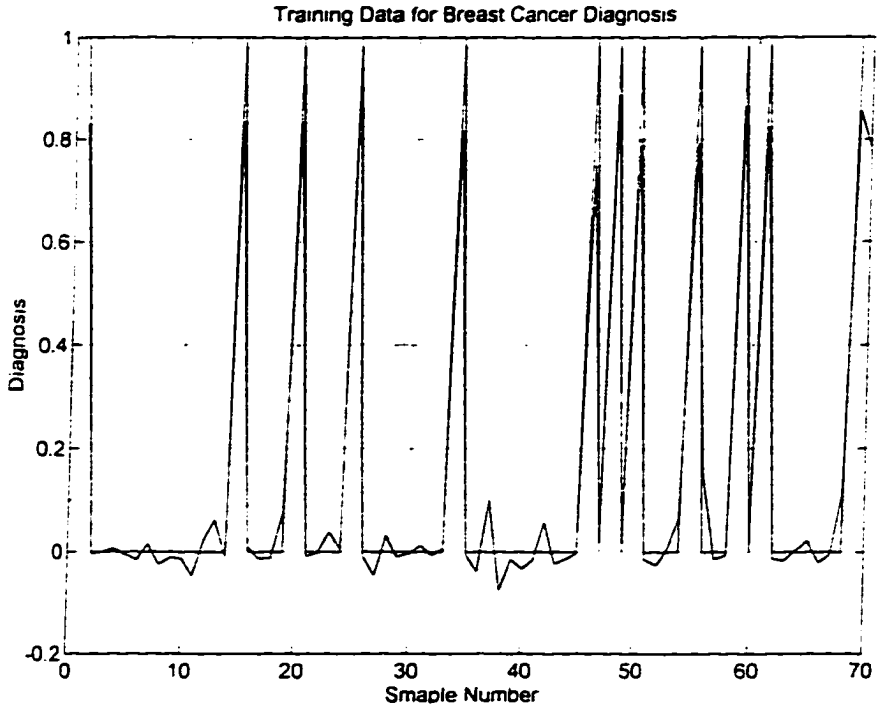


Figure 8.1: Training results for the Breast Cancer Database

the figures, one can see that INCNLPD was able to achieve the correct mapping between inputs and outputs for both applications.

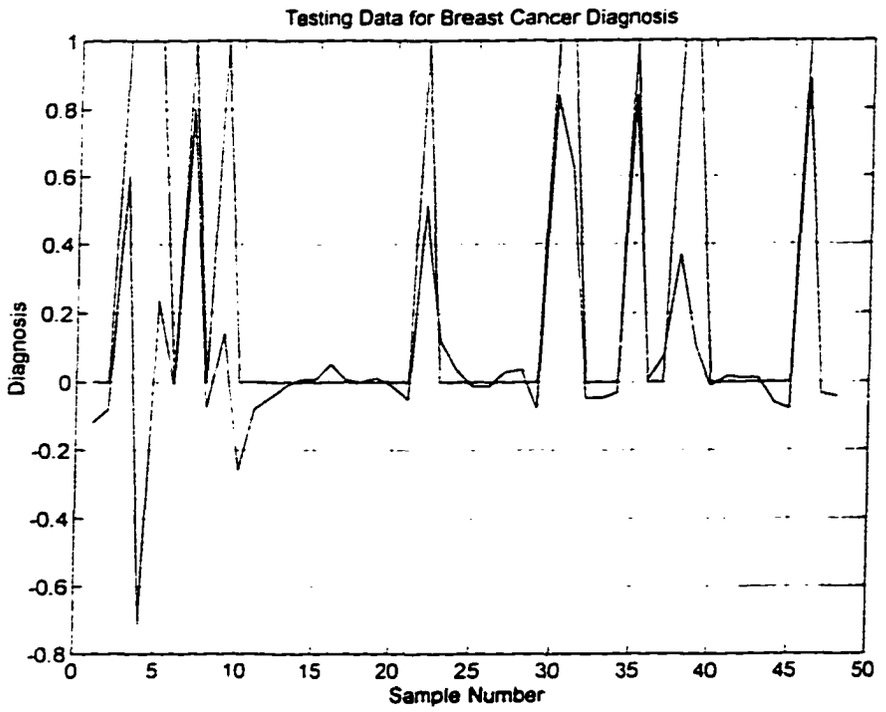


Figure 8.2: Testing results for the Breast Cancer Database

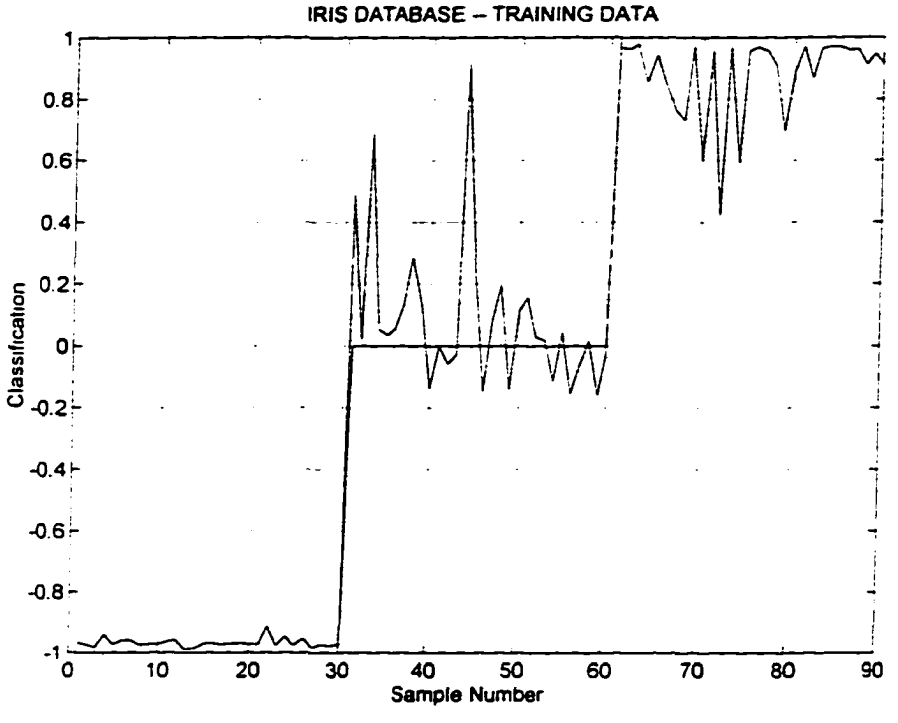


Figure 8.3: Training results for the Iris Plants Database

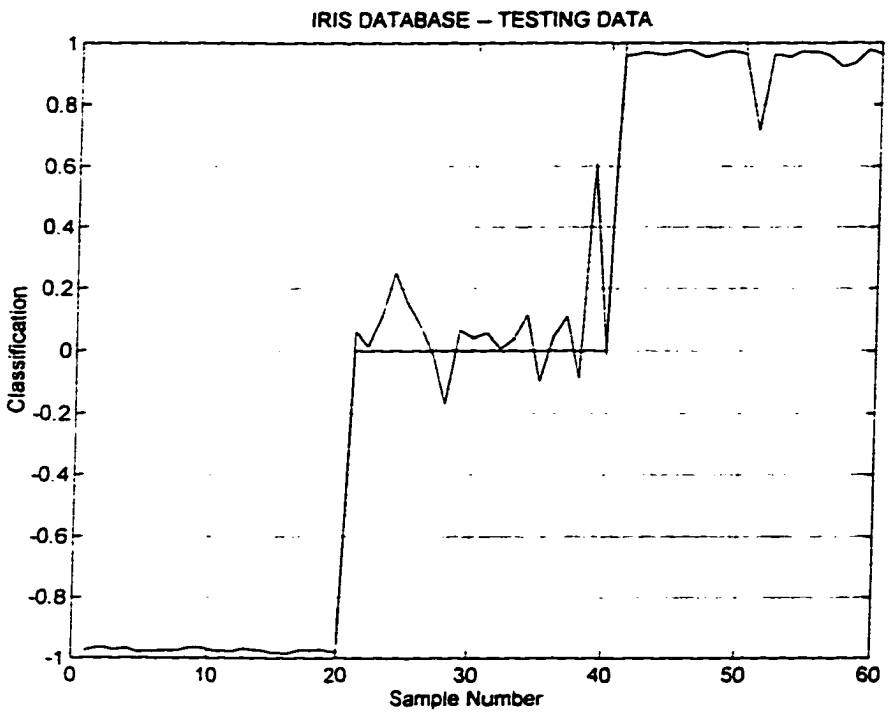


Figure 8.4: Testing results for the Iris Plants Database

Chapter 9

Summary, Conclusions and Recommendations

9.1 Summary

In this work, motivated by the artificial neural network training application, we have developed new algorithms for solving the general nonlinear programming problem. We provide a new implementation of the nonlinear primal-dual algorithm and validate its efficiency through experiments with benchmarking problems. The implementation uses a fast factorization technique to solve the KKT system. For nonconvex problems, it also approximates the Hessian matrix by a recursive prediction error method.

We have proposed a stochastic variant of the technique. By introducing random perturbations on the direction of move, we increase the chance of reaching a global minimum. Computational results show that indeed the results were improved and the stochastic algorithm reaches either the global minimum or a better local minimum than in the

deterministic case.

For problems that have special decomposition properties, we have developed an incremental version of the nonlinear primal-dual method. The technique incrementally updates the variables with respect to each data point of the problem. The data is fed several times to the algorithm. This increases the approximation capabilities of the algorithm. The use of the incremental framework reduces the memory space requirements and for large scale problems it can also help in reducing computing time. We provide convergence results for the incremental nonlinear primal-dual method. The range of applications of the algorithm is wide since it can be used to solve general nonlinear least square problems.

We show that the incremental nonlinear primal-dual technique can be nicely applied to the training problem of artificial neural networks. Such problems possess decomposition properties with respect to the training patterns. For the financial forecasting application, we show that a neural network trained with the incremental nonlinear primal-dual method can achieve very good predictions. We also show that our algorithm outperforms existing training techniques such as the backpropagation algorithm.

This work is a contribution to the field of constrained nonlinear optimization and more specifically to interior point methods. It provides new optimization tools that combine a state of the art technique such as the nonlinear primal-dual method with other strategies such as simulated annealing and incremental update of variables. This research is also bringing new developments in the area of artficial neural network training. A new effective training strategy has been presented. Together with other work

along this line [14, 65, 66, 62], the results that are reported here, show that interior point methods can be effectively used as training approaches for artificial neural networks. From the mathematical programming point of view, the training problem is a general nonconvex constrained minimization problem that we solve using state of the art optimization tools. Experiments have shown that this is a very successful approach and that it outperforms classical methodologies such as the backpropagation algorithm still widely used in commercial softwares.

9.2 Recommendations for Future Research

The primal-dual algorithm can be seen as a type of nonlinear least square technique. The problem is to minimize the merit function Φ that is the square of the norm of the KKT conditions. To solve the problem, we use Newton's method as an unconstrained minimization technique. Bertsekas [4, 5] has described how the Extended Kalman Filter techniques for solving nonlinear least square problems can be seen as an incremental version of the Gauss-Newton's method. There is a strong connection between Kalman Filtering and the incremental version of the primal-dual algorithm that we have developed. Actually, if instead of approximating the Hessian of the Lagrangian using a recursive formula, one would approximate the entire Jacobian matrix of the KKT conditions, the incremental primal-dual method equations and the Kalman filter equations would be the same and the two approaches would be identical. If indeed the primal-dual technique is a special type of Kalman Filters, then this is a completely new look at primal-dual methods. There is a need for further investigation along this line.

From another viewpoint, interior point methods are related to some existing techniques of the field of physics. With some particular settings of parameters, interior points algorithms lead to systems of equations that are identical to the ones found when deriving the *Mean Field Equations* of statistical physics. Actually, it is possible that the KKT conditions are special cases of the *Mean Field Equations*. Some work is already being done in relating interior point methods and the *Mean Field Equations* of statistical physics [36].

For the neural network application, one is concerned with the choice of number of hidden nodes. It is believed that for each neural network application, there is an optimal number of hidden nodes that achieves the correct training and the best generalization properties at the least possible cost. One strategy is to embed the training algorithm within a genetic search procedure. The idea is to use the global search strategy of genetic algorithms to determine the optimal number of hidden nodes. For each intermediate architecture, it would be beneficial to use an effective training technique such as our incremental nonlinear primal-dual method. In real life applications, neural networks tend to have a large amount of connection weights and the number of hidden nodes become critical, such hybrid genetic/INCNLPD strategies could be very effective in achieving fast and accurate training of the data.

Bibliography

[1] Abou-Taleb, N., Megahed, I., Moussa, A. and Zaky, A. (1974), "A New Approach to the Solution of Economic Dispatch Problems", *Winter Power Meeting*, New York.

[2] Al-Harkan, I.M. and Trafalis, T.B. (1996), "A Hybrid Scatter Genetic Tabu Approach for Continuous Global Optimization", *Technical Report*, School of Industrial Engineering, University of Oklahoma, Norman, OK.

[3] Battiti R. (1992), "First and Second-Order Methods for Learning Between Steepest Descent and Newton's Method", *Neural Computation*, Vol. 4, 141–166.

[4] Bertsekas, D. P. (1995), "Incremental Least Squares Methods and the Extended Kalman Filter", *Technical Report*, Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, MA.

[5] Bertsekas, D. P. (1996), "A New Class of Incremental Gradient Methods for Least Squares Problems", *Technical Report*, Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, MA.

[6] Bertsekas, D.P. (1982), *Constrained Optimization and Lagrange Multiplier Methods.* Academic Press, San Diego, CA.

[7] Bracken, J. and McCormick, G.P. (1968), *Selected Applications of Nonlinear Programming*, John Wiley and Sons, New York.

[8] Breitfeld, M. and Shanno, D. (1994), " Preliminary Computational Experience with Modified Log-Barrier Functions for Large-Scale Nonlinear programming ", *Large Scale Optimization State of the Art.* Hager, Hearn, and Pardalos, Editors. Kluwer Academic Publishers, 45-67.

[9] Bunch, J. R. and Parlett, B. N. (1971), "Direct Methods for Solving Symmetric Indefinite Systems of Linear Equations", *SIAM Journal on Numerical Analysis*, Vol. 8, 639-655.

[10] Byrd, R. H., Gilbert, J. C. and Nocedal, J. (1996), "A Trust Region Method Based on Interior Point Techniques for Nonlinear Programming", *Technical Report*, INRIA Rocquencourt, Le Chesnay, France.

[11] Canon, M.D., Cullum, C. and Polak, E. (1970), *Theory of Optimal Control and Mathematical Programming*, McGraw-Hill, New York.

[12] Cohn, M.Z. (1969), *An Introduction to Structural Optimization*, University of Waterloo Press.

[13] Coleman, T. F. and Li, Y. (1993), "An Interior Trust Region Approach for Nonlinear Minimization Subject to Bounds", *Technical Report*, TR93-1342, Department of Computer Science, Cornell University.

[14] Couëllan, N. P. (1995), *Artificial Neural Network Training Via Interior Point Methods*, Masters Thesis, School of Industrial Engineering, University of Oklahoma. Norman. OK.

[15] Das, I. (1997), "An Interior Point Algorithm for the General Nonlinear Programming Problem with Trust Region Globalization", *Technical Report*, Department of Computational and Applied Mathematics, Rice University, Houston, TX.

[16] Davidon, W. C. (1976), "New Least-Square Algorithms", *Journal of Optimization Theory and Applications*, Vol. 18, no. 2, 187-197.

[17] Deboeck, G. J. (1994), *Trading On the Edge*, John Wiley and Sons, New York.

[18] Dennis, J.E., Heinkenschloss, M. and Vicente, L. N. (1994), "Trust-Region Interior Point SQP Algorithms for a Class of Nonlinear Programming Problems", *Technical Report*, TR94-45, revised Nov. 1995 and Dec. 1996, Department of Computational and Applied Mathematics, Rice University, Houston, TX.

[19] Dennis, J. E. and Schnabel, R. B. (1996), *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Classics in Applied Mathematics, SIAM, Philadelphia.

[20] Duda, R.O. and Hart, P.E. (1973), *Pattern Classification and Scene Analysis*, John Wiley and Sons.

[21] El-Bakry, A. S., Tapia, R. A., Tsuchiya, T. and Zhang, Y. (1996),"On the Formulation and Theory of the Primal-Dual Newton Interior Point Method for Nonlinear Programming", *Journal of Optimization Theory and Applications*, 89, 507-541.

[22] Fang. S. and Puthenpura. S. (1993). *Linear Optimization and Extensions: Theory and Algorithms*. Prentice Hall.

[23] Fiacco, A.V., and McCormick, G.P. (1968), *Nonlinear Programming: Sequential Unconstrained Minimization Techniques*, John Wiley, New York.

[24] Flippo, O.E., and Jansen, B. (1992), "Duality and Sensitivity in Quadratic Optimization Over a Sphere", Technical Report 92-65. Falculty of Technical Mathematics and Informatics, Delft University of Technology.

[25] Forsgren, A. and Gill, P. E. (1996), "Primal-Dual Interior Methods for Nonconvex Nonlinear Programming", *Technical Report NA-3*, UCSD Department of Mathematics, Royal Institute of Technology, Stockholm, Sweden.

[26] Fox. R.L. (1971), *Optimization Methods for Engineering Design*, Addison-Wesley. MA.

[27] Gilbert, J. C., Le Vey, and Masse, J. (1991), "La Différentiation Automatique de Fonctions Représentées par des Programmes", *INRIA Research Report 1557.*

[28] Griewank, A. (1989), "Mathematical Programming: Recent Developments and Applications", *On automatic differentiation*, eds. Iri, M. and Tanabe, K., Kluwer Academic Publishers, 83-108.

[29] Haimes, Y.Y. (1977), *Hierarchical Analyses of Water Resources Systems: Modeling and Optimization of Large-Scale Systems*, McGraw-Hill, New York.

[30] Harston, C. T. (1990). "Business with Neural Networks". *Handbook of Neural Computing Applications*, eds. Maren, A. J., Harston, C. T. and Pap, R. M., Academic Press, San Diego, CA, 391-400.

[31] Hecht-Nielsen, R. (1987), "Combinatorial Hypercompression", *Proceedings of the IEEE First International Conference on Neural Networks*, San Diego, CA, Vol. II, 455-462.

[32] Hock, W. and Schittkowski, K. (1981), "Test Examples for Nonlinear Programming Codes", *Lecture Notes in Economics and Mathematical Systems*, Vol. 187, Springer-Verlag, Berlin.

[33] Hopfield, J. J. and Tank, D. W. (1985), "Neural Computation of Decisions in Optimization Problems", *Biological Cybernetics*, Vol. 52, 141-152.

[34] Jerosolimski, M. and Levacher, L. (1994), "A New Method for Fast Calculation of Jacobian Matrices: Automatic Differentiation for Power System Simulation". *IEEE Transactions on Power Systems*, Vol. 9, no. 2, 700-706.

[35] Karmarkar, N. (1984), "A New Polynomial-Time Algorithm for Linear Programming", *Combinatorica*, Vol. 4 (4), 373-395.

[36] Kasap, S. (1997), *title not yet available*, Ph.D. Dissertation in progress, School of Industrial Engineering, University of Oklahoma, Norman, OK.

[37] Khachyian, L. G. (1979), "A Polynomial Algorithm in Linear Programming" (in Russian), *Doklady Akademiia Nauk USSR*, Vol. 224, 1093-1096.

[38] Kirkpatrick, S., Gelatt, C.D. and Vecchi. M.P. (1983), "Optimization by Simulated Annealing", *Science*, Vol. 220, 671-680.

[39] Kohonen, T. (1988), "The Neural Phonetic Typewriter", *Computer*, Vol. 21, 11-22.

[40] Kolmogorov, A.N. (1957), "On the Representation of Continuous Functions of Many Variables by Superposition of Continuous Functions of One Variable and Addition", *Doklady Akademii Nauk SSSR*, Vol. 144, 679-681, (American Mathematical Society Translation, 28, 55-59).

[41] Lasdon, L. S., Plummer, J. and Yu, G. (1995), "Primal-Dual and Primal Interior Point Algorithms for General Nonlinear Programs", *ORSA Journal on Computing*, Vol. 7 (3), 321-332.

[42] Le Cun, Y., Denker. J. S. and Solla, S. A., (1990), "Optimal Brain Damage", *Advances in Neural Information Processing Systems II (Denver 1989)*, Morgan Kaufmann, San Mateo, CA, 598-605.

[43] Lustig, I. J., Marsten, R. E., and Shanno, D. F. (1994), "Interior Point Methods for Linear Programming: Computational State of the Art", *ORSA Journal on Computing*, Vol. 6 (1), Winter 1994, 1-14.

[44] Mangasarian, O.L., Setiono, R. and Wolberg, W.H. (1990), "Pattern Recognition via Linear Programming: Theory and Application to Medical Diagnosis", In Coleman, T.F., and Li. Y. (Eds.)*Large -scale numerical optimization*, Philadelphia: SIAM, 22-30.

[45] Markowitz, H.M. (1952), "Portfolio Selection", *Journal of Finance*, 7, 77-91.

[46] Marzban, C. and Stumpf, G. J. (1996), "A Neural Network for Tornado Prediction Based on Doppler Radar-Derived Attributes", *Journal of Applied Meteorology*. Vol. 35 (5), 617-626.

[47] McClelland, J. L. and Rumelhart, D. E. (1988), *Explorations in Parallel Distributed Processing*, Cambridge, MA, MIT Press.

[48] McCormick, G.P. (1983), *Nonlinear Programming*, John Wiley and Sons. New York.

[49] McCulloch, W.S. and Pitts, W. (1943), "A Logical Calculus of the Ideas Immanent in Nervous Activity", *Bulletin of Mathematical Biophysics*. Vol. 5, 155-163.

[50] Megiddo, N. (1986), "Pathways to the Optimal Set in Linear Programming", *Progress in Mathematical Programming*, ed. N. Megiddo. Springer-Verlag, New-York, 131-158.

[51] Miller, W. T., Sutton, R. S., and Werbos, P. J., (1990), *Neural Networks for Control*, MIT Press, Cambridge, MA.

[52] Minsky, M. L. and Papert, S. A. (1988), *Perceptrons*, Expanded Edition, Cambridge, MA, MIT Press, original edition 1969.

[53] Monteiro, R. C. and Adler, I. (1989), "Interior Path Following Priaml-Dual Algorithms, Part II: Convex Quadratic Programming", *Mathematical Programming*, Vol. 44, 43-66.

[54] *NevProp*, University of Nevada at Reno, FTP://unssun.scs.unr.edu/pub/goodman/nevpropdir

[55] *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press.

[56] Refenes, A. P. (1995), *Neural Networks in the Capital Markets*, John Wiley and Sons, New York.

[57] Refenes, A. P., Abu-Mostafa, Y., Moody, J. and Weigend, A. (1996), "Neural Networks in Financial Engineering", *Series: Progress in Neural Processing*, Vol. 2, *Proceedings of the 3rd International Conference in Neural Networks in Capital Markets*, World Scientific Publishing Company, NJ.

[58] Rumelhart, D. E., Hinton, G. E. and Williams, R. (1986), "Learning Internal Representations by Error Propagation", *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, eds. Rumelhart, D. E. and McClelland, J. L., Vol. 1, Foundations, MIT Press.

[59] Söderstrom, T. and Stoica, P. (1989), *System Identification* . Prentice Hall International (UK), Englewood Cliffs, NJ.

[60] Sonnevend, G. (1985), "An Analytical Center for Polyhedrons and then New Classes of Global Algorithms for Linear (smooth,convex) Programming", *Proceedings of the 12th IFIP Conference on System Modeling and Optimizations*, Budapest, *Lectures Notes in Control Information Sciences*, Springer-Verlag, New-York, Vol. 84, 866-876.

[61] Sorensen, D. C. (1982), "Newton's Method with a Model Trust Region Modification", *SIAM Journal on Numerical Analysis*, Vol. 19, 409-426.

[62] Trafalis, T. B. and Sieger, D. B. (1993), "Training of Multilayer Feedforward Artificial Neural Networks by a Logarithmic Barrier Function Adaptation Scheme". In C.H. Dagli, L.I. Burke, B.R. Fernandez and J. Ghosh (Eds.),*Intelligent Engineering Systems Through Artificial Neural Networks*, Vol. 3, 167-173.

[63] Trafalis T.B. and Couëllan, N.P. (1994), "Neural Networks Training via Primal-Dual Interior Point Method for Linear Programming", *Proceedings of WCNN*, Vol. II, 798-803.

[64] Trafalis T.B. and Couëllan, N.P. (1996), "Neural Networks Training via an Affine Scaling Quadratic Optimization Algorithm", *Neural Networks*, 9:3, 475-481.

[65] Trafalis, T.B. and Tutunji, T. (1994), "A Quasi-Newton Barrier Function Algorithm for Artificial Network Training with Bounded Weights", In C.H. Dagli, L.I. Burke, B.R. Fernandez and J. Ghosh (Eds.), *Intelligent engineering systems through artificial neural networks*, Vol. 4, 161-173.

[66] Trafalis, T.B. and Tutunji, T. (1997), "Barrier and Stochastic Barrier Newton-Type Methods for Training Feedforward Neural Networks with Bounded Weights", *International Journal of Smart Engineering System Design*, in press.

[67] Tseng, P. (1995), "Incremental Gradient(-Projection) Method with Momentum Term and Adaptive Stepsize Rule", *Technical Report*, Department of Mathematics, University of Washington, Seattle, WA.

[68] Vandenberghe, L. and Boyd, S. (1994), "Positive Definite Programming", *Technical Report*, Information Systems Laboratory, Department of Electrical Engineering, Stanford University, Stanford, CA.

[69] Vanderbei, R. J., Meketon, M. S. and Freedman, B. A. (1986), "A Modification of Karmarkar's Linear Programming Algorithm", *Algorithmica*, Vol. 1, 395-407.

[70] Wasserman, P.D., (1989), *Neural Computing: Theory and Practice*, NY: Van Nostrand Reinhold.

[71] Werbos, P., (1974), Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD. Thesis, Committee on Applied Mathematics, Harvard University, Cambridge, MA. Reprinted in P. Werbos, *The Roots of Backpropagation:From Ordered Derivatives to Neural Networks and Political Forecasting*, NY: Wiley (1993).

[72] White, H., (1993), "Economic Prediction Using Neural Networks: The Case of IBM Daily Stock Returns", in Trippi, R. R. and Turban E. (Eds.), *Neural Networks in Finance and Investing*, pp. 315-328.

[73] Wright, M.H., (1997), "Ill-Conditioning and Computational Error in Interior Point Methods for Nonlinear Programming", *Technical Report*, 97-4-04, Computing Sciences Research Center, Bell Laboratories, Murray Hill, New Jersey 07974.

[74] Wright, S., (1996), "Modified Cholesky Factorizations in Interior-Point Algorithms for Linear Programming", *Technical Report MCS-P600-0596*, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL.

[75] Wright. S.. (1996). *Primal Dual Interior Point Methods*. SIAM.

[76] Wu, Y.C, Debs, A.S. and Marsten, R.E. (1994), "A Direct Nonlinear Predictor-Corrector Primal-Dual Interior Point Algorithm for Optimal Power Flows", Transactions on Power Systems, Vol. 9, No. 2, 876-883.

[77] Yamashita, H. (1992), "A Globally Convergent Primal-Dual Interior Point Method for Constrained Optimization", *Technical Report*, Mathematical System Institute, Inc., Shinjuku, Tokyo, Japan.

[78] Yamashita, H. (1994), "A Primal-Dual Interior Point Trust Region Method for Large Scale Constrained Optimization", *Technical Report*, Mathematical System Institute, Inc., Shinjuku, Tokyo, Japan.

[79] Ye Y., (1992), "On Affine Scaling Algorithms for Nonconvex Quadratic Programming", *Mathematical Programming*, Vol. 56, 285-300.

Appendix A

Convergence of Algorithms

In this appendix, we recall some definitions and concepts relative to the convergence of algorithms.

Optimization algorithms can be seen as iterative processes that generate sequences of vector iterates $\{x_1, x_2, x_3, \ldots\}$. When we say an algorithm converges to a solution, we mean that the sequence of iterates $\{x_k\}$ converges to a solution point x^*. An algorithm is locally convergent if starting from a neighborhood of the solution, the algorithm converges to that solution. An algorithm is globally convergent if it converges from almost any feasible starting point (Note: The term *global* here does not refer to seeking a global minimizer). Usually, instead of saying $\{x_k\}$ converges to x^*, we say the sequence $\{\|x_k - x^*\|\}$ converges to zero, which can also be written as $\{\|x_k - x^*\|\} \to 0$ or $lim_{k\to\infty}\|x_k - x^*\| = 0$.

The rate of convergence is another important concept. It gives an estimate of how fast the algorithm converges to the solution. The different types of convergence are classified according to the rate of convergence. Next, we define the classification.

Definitions of Order of Convergence [19]

- If there exists a constant $c \in [0, 1)$ and an integer $\hat{k}$ such that for all $k \geq \hat{k}$, $\|x_{k+1} - x_k\| \leq c\|x_k - x^*\|$, then $\{x_k\}$ is said to be *q-linearly convergent* to x^*.

- If for some sequence $\{c_k\}$ that converges to 0, $\|x_{k+1} - x_k\| \leq c_k\|x_k - x^*\|$, then $\{x_k\}$ is said to be *q-superlinearly convergent* to x^*.

- If there exist constants $p > 1$, $c \geq 0$, and $\hat{k} \geq 0$ such that $\{x_k\}$ converges to x^* and for all $k \geq \hat{k}$, $\|x_{k+1} - x_k\| \leq c\|x_k - x^*\|^p$, then $\{x_k\}$ is said to converge to x^* with *q-order at least p*. If $p = 2$ then convergence is said to be *q-quadratic*, if $p = 3$, convergence is said *q-cubic*.

- If $\{x_k\}$ converges to x^*, and for some sequence $\{c_k\}$ that converges to 0 and for some fixed integer j, $\|x_{k+j} - x_k\| \leq c_k\|x_k - x^*\|$, then $\{x_k\}$ is said to be *j-step q-superlinearly convergent* to x^*.

- If $\{x_k\}$ converges to x^*, and there exist constants $p > 1$, $c \geq 0$, and $\hat{k} \geq 0$ such that $\{x_k\}$ converges to x^* and for all $k \geq \hat{k}$, $\|x_{k+j} - x_k\| \leq c\|x_k - x^*\|^p$ for some fixed integer j, then $\{x_k\}$ is said to converge to x^* with *j-step q-order at least p*.

Appendix B

Source Codes

```
-------------------------------
Makefile for INCNLPD code
-------------------------------


LDIR=/a/ranger/home/resrch0/intptmnc/clapack/F2CLIBS
IDIR=/a/ranger/home/resrch0/intptmnc/clapack/F2CLIBS

OPTIONS=
incnlpd: incnlpd.o dspsv.o ann.o funcann.o gradient.o vector.o matrix.o multiply.o
        g++ $(OPTIONS) -o incnlpd -O3 -I$(IDIR) incnlpd.o dspsv.o ann.o funcann.o
gradient.o vector.o matrix.o multiply.o  -L$(LDIR) -lF77  -lI77 -lsmallblas

incnlpd.o: incnlpd.h incnlpd.C
        g++ $(OPTIONS) -w -c -O3 -I$(IDIR) incnlpd.C

dspsv.o: dspsv.c
        cc $(OPTIONS) -c -w -O3 -I$(IDIR) dspsv.c

ann.o:  ann.h ann.C
        g++ $(OPTIONS) -w -O3 -c ann.C

funcann.o: funcann.h funcann.C
        g++ $(OPTIONS) -w -O3 -c funcann.C

gradient.o: gradient.h gradient.C
        g++ $(OPTIONS) -w -O3 -c gradient.C

vector.o: vector.h vector.C
        g++ $(OPTIONS) -w -O3 -c vector.C

matrix.o: matrix.h matrix.C
        g++ $(OPTIONS) -w -O3 -c matrix.C

multiply.o: vector.h matrix.h multiply.h multiply.C
        g++ $(OPTIONS) -w -O3 -c multiply.C

sparse.o: matrix.h sparse.h sparse.C
        g++ $(OPTIONS) -w -O3 -c sparse.C

clean:
        rm *.o
```

```
//-----------------------------------
// File: incnlpd.h
// Main subroutines declarations
//-----------------------------------

class vector;
class matrix;
class ann;

vector fkkt(vector&, vector&, vector&, vector&, double, vector&,int );
vector gradL(vector&, matrix&,int pat);
double phi(vector&,vector&,int);
```

```
//-----------------------------------------
// File: incnlpd.C
// Main File
// Implementation of INCNLPD Algorithm
//-----------------------------------------

#include <stdlib.h>
#include <iostream.h>
#include "incnlpd.h"
#include <f2c.h>
#include "vector.h"
#include "matrix.h"
#include "multiply.h"
#include "ann.h"
#include "funcann.h"
#include "gradient.h"
#include <time.h>

// global variables
ann network;
unsigned int n,p;
double mu=1;
vector z;
int fiter=0;
time_t t1,t2;

vector fkkt(vector& x, vector& s, vector& z, double mu, vector& vgradL,int pat)
{
  vector result(n+p+p);

  //vector vgradL=gradL(x);
  vector vh=h(x);
  vector vg=g(x,pat);

  // gradL(x)
  for (register i=0;i<n;i++)
    result[i]=vgradL[i];

  // h(x)
  for (register i=n;i<n;i++)
    result[i]=vh[i-n];

  // g(x) - s
  for (register i=n;i<n+p;i++)
    result[i]=vg[i-n]-s[i-n];
```

```
  // Z S e - mu e
  for (register i=n+p;i<n+p+p;i++)
    result[i]=z[i-n-p]*s[i-n-p]-mu;

  return (result);
}

vector gradL(vector& x, matrix& gg,int pat)
{
  time_t t1,t2;
  vector gradgz(n);
  for (register i=0;i<n;i++){
   double sprod=0;
   for (register j=0;j<p;j++) sprod+=gg.get_val(i,j)*z.get_val(j);
   gradgz.set_val(i,sprod);//=mpy(gg,z);
  }
  vector result;
  t1=time(NULL);
  result=grad(&f,x,pat,0.01);
  t2=time(NULL);
//  cout << "CPU TIME for grad(f) = " << difftime(t2,t1) <<endl;
  for (register i=0;i<n;i++) result.set_val(i,result.get_val(i)-gradgz.get_val(i));
  return (result);
}

double phi(vector& vec,vector& vgradL, int pat)
{
  double result=0;
  vector vx(n),vs(p),vz(p),vecfkkt(n+p+p);

  for (register i=0;i<n;i++) vx[i]=vec[i];
  for (register i=n;i<n+p;i++) vs[i-n]=vec[i];
  for (register i=n+p;i<n+p+p;i++) vz[i-n-p]=vec[i];
  vecfkkt=fkkt(vx,vs,vz,mu,vgradL,pat);
  for(i=0;i<n+p+p;i++) result+=vecfkkt.get_val(i)*vecfkkt.get_val(i);
  return (result);
}


main()
{
  network.info();

  n=network.get_weight().get_dim();
  p=2*network.nb_neurons[1];
```

```
cout << "size of vector v: " << (n+p+p) << endl;
vector x(n);

int seed;
cout << "enter seed" << endl;
cin >> seed;
srand(seed);
double random=0;
double coef;
cout << "Coef. for the starting weights:" << endl;
cin >> coef;
// random starting point
for (register i=0;i<n;i++) {
  //random=((double)(rand())/RAND_MAX-0.5)*2;
  random=coef*(2.0*rand()/(RAND_MAX+1.0)-1.0);
  x[i]=random;
  //if (random>0){x[i]=0.7*pow(network.nb_neurons[1],1/network.nb_neurons[0]);}
  //else {x[i]=-0.7*pow(network.nb_neurons[1],1/network.nb_neurons[0]);}
}

cout << "starting point:" << endl;
x.print();

vector s(p);
vector dv(n);
vector v(n+p+p),new_v(n+p+p);
vector vgradL(n);
vector ds(p),dz(p);
double sigma,sigma2,f1,f2;
double lambda;
double eps;
double tau,tau1,beta,alfap;
double gphiTdv,alfas,alfaz,new_crit;
double armleft,armright,wolleft,wolright;
unsigned int armtrue,woltrue;

double exit;
cout << "enter exit criterion" << endl;
cin >> exit;
cout << "enter forgetting factor"<<endl;
cin >> lambda;
double m0;
cout << "M0:"<<endl;
cin >>m0;
cout << "enter eps for steps ?"<<endl;
cin >> eps;
```

```
vector xs(n);

double crit;
double tmp;
unsigned int iter=0;

// used to set the sizes of z
z=g(x,0);
s=z;

matrix H(n,n);

//f2c declarations
doublereal ap[n*(n+1)/2],brhs[n];
integer ipiv[n],info,nrhs,n2,ldb;
char uplo;

H.init(0);
double h0;
cout <<"h0 ? "<<endl;cin>>h0;

for (register i=0;i<n;i++)
  H.set_val(i,i,h0);

crit=100;
t1=time(NULL);

while (crit>exit) {
  sigma=0;
  for (register i=0;i<p;i++) sigma+=s.get_val(i)*z.get_val(i);
   mu=sigma/m0;

 for(register pat=0;pat<network.np;pat++){

  matrix gg(n,p),gg2(p,n),gg3(n,n);

  gg.init(0);

  for(register i=0;i<network.nb_neurons[0];i++) for(register
j=0;j<network.nb_neurons[1];j++){
      gg.set_val(i*network.nb_neurons[1],2*j,-network.get_pat_vec(pat,i));
      gg.set_val(i*network.nb_neurons[1],2*j+1,network.get_pat_vec(pat,i));}

  vgradL=gradL(x,gg,pat);

  for (register i=0;i<n;i++) for (register j=0;j<n;j++){
```

```
    H.set_val(i,j,lambda*H.get_val(i,j)+vgradL[i]*vgradL[j]);}

 for (register j=0;j<p;j++) for (register i=0;i<n;i++)
    gg2.set_val(j,i,z[j]/s[j]*gg.get_val(i,j));

 for (register i=0;i<n;i++) for (register j=0;j<n;j++){
    double sprod=0;
    for (register k=0;k<p;k++) sprod+=gg.get_val(i,k)*gg2.get_val(k,j);
    gg3.set_val(i,j,sprod);
    }

 gg2.~matrix();

 // H
 register k=0;
 for (register j=0;j<n;j++)
   for (register i=0;i<=j;i++){
     ap[k]=H.get_val(i,j)+gg3.get_val(i,j);
     k++;}

 gg3.~matrix();

 // solve the linear system: J dv = - Fkkt
 vector vfkkt(n+p+p);
 vfkkt=fkkt(x,s,z,mu,vgradL,pat);
time_t ts,te;
ts=time(NULL);
 for (register i=0;i<n;i++){
    f1=0;
    for (register j=0;j<p;j++)
       f1+=gg.get_val(i,j)/s[j]*vfkkt[n+p+j];
    brhs[i]=-vfkkt[i]-f1;
 }


 uplo='U';
 n2=n;
 nrhs=1;
 ldb=n;
 dspsv_(&uplo,&n2,&nrhs,ap,ipiv,brhs,&ldb,&info);

 for (register i=0;i<n;i++) {
   v[i]=x[i];
   dv[i]=brhs[i];
 }
te=time(NULL);
```

```
//    cout <<"Inversion time:"<<difftime(te,ts)<<endl;
     for (register j=0;j<p;j++) {
       f2=0;
       for (register i=0;i<n;i++)
          f2+=gg.get_val(i,j)*dv[i];
       ds[j]=f2+vfkkt[n+j];
       v[n+j]=s[j];
       dz[j]=-vfkkt[n+p+j]/s[j]-f2/s[j]*z[j];
       v[n+j+p]=z[j];
     }
     vfkkt.~vector();
     gg.~matrix();

     // backtracking

     tau=0.9995;
     tau1=0.99;
     beta=0.0001;
     alfap=1;
     gphiTdv=2*(-phi(v,vgradL,pat));
   do{
     alfas=-1;
     alfaz=-1;

     for (register i=0;i<p;i++) {
       if ((tmp=ds[i]/s[i])<alfas) alfas=tmp;
       if ((tmp=dz[i]/z[i])<alfaz) alfaz=tmp;
     }
     alfas=-1/alfas;
     alfaz=-1/alfaz;
     alfas=alfap*tau*alfas;
     alfaz=alfap*tau*alfaz;
     alfas=(alfas<1?alfas:1);
     alfaz=(alfaz<1?alfaz:1);

     for (register i=0;i<n;i++)
       new_v[i]=v[i]+alfap*dv[i];
     for (register i=n;i<n+p;i++)
       new_v[i]=v[i]+alfas*ds[i-n];
     for (register i=n+p;i<n+p+p;i++)
       new_v[i]=v[i]+alfaz*dz[i-n-p];

     new_crit=phi(new_v,vgradL,pat);
     armleft=new_crit-phi(v,vgradL,pat);
     armright=beta*alfap*gphiTdv;
```

```
    wolleft=-2*new_crit;
    wolright=tau1*gphiTdv;

    armtrue=(armleft<=armright);
    woltrue=(wolleft>=wolright);

    alfap*=.5;
  } while (((!armtrue)||(!woltrue))&&(alfap>eps));

  v=new_v;

    // update x,y,s,z
    for (register i=0;i<n;i++)
      x[i]=v[i];
    for (register i=n;i<n+p;i++)
      s[i-n]=v[i];
    for (register i=n+p;i<n+p+p;i++)
      z[i-n-p]=v[i];
        }
    crit=ft(x);
    iter++;
    cout << "E=" << crit << endl;
    fiter=0;
  } // another iteration of the algorithm
  t2=time(NULL);
  cout << "weights=";
  x.print();
  cout << endl;
  cout << "END" << endl;
  cout << "number of feeds:";
  cout << iter << endl;
  cout << "CPU time:";
  cout << difftime(t2,t1)<<endl;
  cout << endl;
  unsigned int resp=1;
  while (resp==1){
    network.test();
    cout <<"Another test ? (0/1)";
    cin >> resp;
    cout <<endl;
  }
}
```

```
//-----------------------
// File: vector.h
// Vector Class
//-----------------------

#ifndef _VECTOR_
#define _VECTOR_

#include <iostream.h>

class matrix;
class sparse;

class vector {
  unsigned int dim;
  double*     ptr;
  unsigned int transpose;

  friend vector mpy(const vector&, const matrix&);
  friend vector mpy(const matrix&, const vector&);
  friend vector mpy(const sparse&, const vector&);
  friend matrix mpy(const vector&, const vector&);
  friend vector trans(const vector&);
  friend vector sum(const vector&, const vector&);
  friend vector dif(const vector&, const vector&);
  friend vector mpy(const double&, const vector&);
  friend double dot(const vector&, const vector&);

public:

  vector();
  vector(unsigned int);
  vector(const vector&);
 ~vector();
  init(const double);

  int get_dim(){return (dim);}

  vector& operator=(const vector&);
  double& operator[](unsigned int);
  double get_val(unsigned int i){return ptr[i];}
       set_val(unsigned int i,double val){ptr[i]=val;}

  set_transpose();
  print();
```

```
};

#endif
```

```
//-----------------------
// File: vector.C
// Vector Class Body
//-----------------------

#include "vector.h"
#include <iostream.h>

vector::vector()              // default constructor
{
  dim=0;
  transpose=0;
}

vector::vector(unsigned int size) // constructs a 'size'-long vector
{
  ptr=new double[dim=size];
  transpose=0;
}

vector::vector(const vector& vec) // copy constructor
{
  ptr=new double[dim=vec.dim];
  for (register i=0;i<dim;i++)
    ptr[i]=vec.ptr[i];
  transpose=vec.transpose;
}

vector::~vector()
{
  delete [] ptr;
}

vector::init(const double val)   // initialize vector with 'val'
{
  for (register i=0;i<dim;i++)
    ptr[i]=val;
}

vector& vector::operator=(const vector& v)
{
  if (!(dim)) ptr=new double[dim=v.dim];
  if (dim!=v.dim) {cerr<<"vect-aff: sizes don't match"<<endl;exit(1);}
  for (register i=0;i<dim;i++)
    ptr[i]=v.ptr[i];
  transpose=v.transpose;
```

```
  return(*this);
}

double& vector::operator[](unsigned int i)
{
  if (i<=dim)
    return (ptr[i]);
  else {
    cerr << "subscript out of range in vector" << endl;
    exit(1);
  }
}

vector::set_transpose()
{
  transpose=1-transpose;
}

vector::print()
{
  cout << "( ";
  for (register i=0;i<dim;i++)
    cout << ptr[i] << " ";
  cout << ")";
  if (!transpose) cout << "T";
  cout << "\n";
}
```

```
//-------------------------
// File: matrix.h
// Matrix Class
//-------------------------

#ifndef _MATRIX_
#define _MATRIX_
#include <iostream.h>
#include <math.h>

class vector;

class matrix {
  unsigned int row,col;
     double** mat;

  friend vector mpy(const vector&, const matrix&);
  friend vector mpy(const matrix&, const vector&);
  friend matrix mpy(const vector&, const vector&);
  friend matrix trans(const matrix&);
  friend matrix sum(const matrix&, const matrix&);
  friend matrix dif(const matrix&, const matrix&);
  friend matrix mpy(const double&, const matrix&);
  friend matrix mpy(const matrix&, const matrix&);

public:
  matrix();
  matrix(unsigned int, unsigned int);
  matrix(const matrix&);
 ~matrix();

  unsigned int get_row(){ return row; }
  unsigned int get_col(){ return col; }
  double&  get_val(unsigned int i,unsigned int j) { return mat[i][j];}
        set_val(unsigned int i,unsigned int j,double val) {mat[i][j]=val;}


  matrix& operator=(const matrix&);

  init(double);
  print();
  inv();
  vector row2vec(unsigned int);
  vector col2vec(unsigned int);

};
```

```
#endif
```

```
//-------------------------
// File: matrix.C
// Matrix Class Body
//-------------------------

#include "matrix.h"
#include "vector.h"
#include <iostream.h>
#include <stdio.h>

matrix::matrix()
{
  row=col=0;
}

matrix::matrix(unsigned int nb_row, unsigned int nb_col)
{
  row=nb_row;
  col=nb_col;
  mat=new double*[row];
  mat[0]=new double[row*col];
  for (register i=1;i<row;i++)
    mat[i]=mat[i-1]+col;
}

matrix::matrix(const matrix& m)
{
  row=m.row;
  col=m.col;
  mat=new double*[row];
  mat[0]=new double[row*col];
  for (register i=1;i<row;i++)
    mat[i]=mat[i-1]+col;
  for (register i=0;i<row;i++)
    for (register j=0;j<col;j++)
        mat[i][j]=m.mat[i][j];
}

matrix::~matrix()
{
//  for (register i=0;i<row;i++)
//    delete mat[i];
  delete [] mat[0];
//  delete mat;
}
```

```
matrix& matrix::operator=(const matrix& m)
{
  if (!(row)||!(col)) {
    row=m.row;
    col=m.col;
    mat=new double*[row];
    mat[0]=new double[row*col];
    for (register i=1;i<row;i++)
        mat[i]=mat[i-1]+m.col;
  }
  if ((row!=m.row)||(col!=m.col)) {cerr<<"aff: sizes don't match"<<endl;exit(1);}
  for (register i=0;i<row;i++)
    for (register j=0;j<col;j++)
        mat[i][j]=m.mat[i][j];
  return (*this);
}

matrix::init(double val)
{
  for (register i=0;i<row;i++)
    for (register j=0;j<col;j++)
        mat[i][j]=val;
}

matrix::print()
{
  for (register i=0;i<row;i++)
  {
    for (register j=0;j<col;j++)
        printf("%7.3f ",mat[i][j]);
    cout << "\n";
  }
}

matrix::inv()
{
  matrix lu(row,row);
  vector y(row),b(row);
  int p,temp;
  double m,s;
  int r[row];

  for (register i=0;i<row;i++)
    for (register j=0;j<row;j++)
        lu.mat[i][j]=mat[i][j];
```

```
// LU decomposition

for (register k=0;k<row;k++)
  r[k]=k;
for (register k=0;k<row-1;k++) {
  p=k;
  for (register i=k+1;i<row;i++)
      if (fabs(lu.mat[r[i]][k])>fabs(lu.mat[r[p]][k])) p=i;
  if (!lu.mat[r[p]][k]) { cerr << "singular matrix" << endl; exit(1);}
  if (p!=k) {
      temp=r[k];
      r[k]=r[p];
      r[p]=temp;
  }
  for (register i=k+1;i<row;i++) {
      m=lu.mat[r[i]][k]/lu.mat[r[k]][k];
      for (register j=k+1;j<row;j++)
        lu.mat[r[i]][j]=lu.mat[r[i]][j]-m*lu.mat[r[k]][j];
      lu.mat[r[i]][k]=m;
  }
}

// backsubstitution

for (register jj=0;jj<row;jj++) {
  b.init(0.0);
  b[jj]=1.0;

  y[0]=b[r[0]];
  for (register i=1;i<row;i++) {
      s=b[r[i]];
      for (register j=0;j<i;j++)
        s-=lu.mat[r[i]][j]*y[j];
      y[i]=s;
  }

  mat[row-1][jj]=y[row-1]/lu.mat[r[row-1]][row-1];
  for (register i=row-2;i>=0;i--) {
      s=y[i];
      for (register j=i+1;j<row;j++)
        s-=lu.mat[r[i]][j]*mat[j][jj];
      mat[i][jj]=s/lu.mat[r[i]][i];
  }
 }
}
```

```
vector matrix::row2vec(unsigned int line)
{
  vector result(col);
  for (register i=0;i<col;i++)
    result[i]=mat[line][i];

  return (result);
}

vector matrix::col2vec(unsigned int column)
{
  vector result(row);
  for (register j=0;j<row;j++)
    result[j]=mat[j][column];

  return (result);
}
```

```
//-------------------------------------
// File: multiply.h
// Vector and Matrices Operations
//-------------------------------------

#include "vector.h"
#include "matrix.h"
#include "sparse.h"

vector mpy(const vector&, const matrix&);
vector mpy(const matrix&, const vector&);
vector mpy(const sparse&, const vector&);
vector mpy(const double&, const vector&);
matrix mpy(const vector&, const vector&);
vector trans(const vector&);
double dot(const vector&, const vector&);

matrix trans(const matrix&);
vector sum(const vector&, const vector&);
vector dif(const vector&, const vector&);

matrix sum(const matrix&, const matrix&);
matrix dif(const matrix&, const matrix&);
matrix mpy(const double&, const matrix&);
matrix mpy(const matrix&, const matrix&);
```

```
//-----------------------------------------
// File: multiply.C
// Vectors and Matrices Operations Body
//-----------------------------------------

#include "multiply.h"
#include <iostream.h>

vector mpy(const vector& v_T, const matrix& m)
{
  //if (!v_T.transpose) { cerr << "vector must be transposed" << endl; exit(1); }
  if (v_T.dim!=m.row) { cerr << "sizes don't match" << endl; exit(1); }

  vector result(m.col);
  for (register i=0;i<m.col;i++) {
    result[i]=0;
    for (register j=0;j<m.row;j++)
      result[i]+=v_T.ptr[j]*m.mat[j][i];
  }
  return (result);
}

vector mpy(const matrix& m, const vector& v)
{
  //if (v.transpose) exit(1);
  if (v.dim!=m.col) exit(1);

  vector result(m.row);
  for (register i=0;i<m.row;i++) {
    result[i]=0;
    for (register j=0;j<m.col;j++)
      result[i]+=v.ptr[j]*m.mat[i][j];
  }
  return (result);
}

vector mpy(const sparse& m, const vector& v)
{
  cell* curr;
  if (v.dim!=m.col) exit(1);

  vector result(m.row);
  for (register i=0;i<m.row;i++) {
    result[i]=0;
    curr=m.row_elts[i];
    while (curr) {
```

```
        result[i]+=v.ptr[curr->colpos]*curr->val;
        curr=curr->next_in_row;
      }
    }
    return (result);
}

matrix mpy(const vector& v, const vector& w_t)
{
    matrix result(v.dim,w_t.dim);
    for (register i=0;i<v.dim;i++)
      for (register j=0;j<w_t.dim;j++)
        result.mat[i][j]=v.ptr[i]*w_t.ptr[j];

    return (result);
}

vector mpy(const double& d, const vector& v)
{
    vector result(v.dim);
    for (register i=0;i<v.dim;i++)
      result.ptr[i]=d*v.ptr[i];
    return (result);
}

double dot(const vector& v, const vector& w)
{
    if (v.dim!=w.dim) {cerr<<"vec-dot: size pbm"<<endl; exit(1);}
    double result=0;
    for (register i=0;i<v.dim;i++)
      result+=v.ptr[i]*w.ptr[i];
    return (result);
}

vector trans(const vector& v)
{
    vector result(v.dim);
    result=v;
    result.set_transpose();
    return (result);
}

matrix trans(const matrix& m)
{
    matrix result(m.col, m.row);
    for (register i=0;i<m.col;i++)
```

```
    for (register j=0;j<m.row;j++)
      result.set_val(i,j,m.mat[j][i]);
  return (result);
}

vector sum(const vector& v, const vector& w)
{
  if (v.dim!=w.dim) {cerr << "sizes don't match" << endl; exit(1);}

  vector result(v.dim);
  for (register i=0;i<v.dim;i++)
    result.ptr[i]=v.ptr[i]+w.ptr[i];
  return (result);
}

vector dif(const vector& v, const vector& w)
{
  if (v.dim!=w.dim) {cerr << "sizes don't match" << endl; exit(1);}

  vector result(v.dim);
  for (register i=0;i<v.dim;i++)
    result.ptr[i]=v.ptr[i]-w.ptr[i];
  return (result);
}

matrix sum(const matrix& a, const matrix& b)
{
  if ((a.row!=b.row)||(a.col!=b.col)) {cerr << "mat-sum: sizes don't match" << endl; exit(1);}

  matrix result(a.row,a.col);
  for (register i=0;i<a.row;i++)
    for (register j=0;j<a.col;j++)
      result.mat[i][j]=a.mat[i][j]+b.mat[i][j];
  return (result);
}

matrix dif(const matrix& a, const matrix& b)
{
  if ((a.row!=b.row)||(a.col!=b.col)) {cerr << "mat-dif: sizes don't match" << endl; exit(1);}

  matrix result(a.row,a.col);
  for (register i=0;i<a.row;i++)
    for (register j=0;j<a.col;j++)
      result.mat[i][j]=a.mat[i][j]-b.mat[i][j];
  return(result);
}
```

```
matrix mpy(const double& d, const matrix& m)
{
  matrix result(m.row,m.col);
  for (register i=0;i<m.row;i++)
    for (register j=0;j<m.col;j++)
      result.mat[i][j]=d*m.mat[i][j];
  return (result);
}

matrix mpy(const matrix& a, const matrix& b)
{
  if (a.col!=b.row) {cerr<<"mat-mpy: sizes pbm" << endl; exit(1);}

  matrix result(a.row,b.col);
  for (register i=0;i<a.row;i++)
    for (register j=0;j<b.col;j++) {
      result.mat[i][j]=0;
      for (register k=0;k<a.col;k++)
        result.mat[i][j]+=a.mat[i][k]*b.mat[k][j];
    }
  return (result);
}
```

```
//-------------------------
// File: gradient.h
// Gradient Subroutines
//-------------------------

static double maxarg1,maxarg2;
#define max(a,b) (maxarg1=(a),maxarg2=(b),(maxarg1)>(maxarg2)? \
(maxarg1):(maxarg2))

class vector;
class matrix;

vector grad(double (*func)(vector&),vector& x,double h);
vector grad(double (*func)(vector&,int),vector& x,int pat,double h);
matrix grad(vector (*func)(vector&,matrix&,int),vector& x,matrix& gg, int pat, double h);
```

```
//-----------------------
// File: gradient.C
// Gradient Subroutines Body
//-----------------------

#include <math.h>
#include "gradient.h"
#include "funcann.h"
#include "vector.h"
#include "matrix.h"
#include "multiply.h"

#define CON 1.4
#define CON2 (CON*CON)
#define BIG 1.0e20

// NTAB=10 (default)
#define NTAB 2
#define SAFE 2.0

vector grad(double (*func)(vector&),vector& x,double h)
{
  int i,j;
  double err,errt,fac;
  matrix a(NTAB,NTAB);
  vector hh(x.get_dim());
  vector result(x.get_dim());

  for (int register gc=0;gc<x.get_dim();gc++)
  {
    if (h==0.0) cerr << "h must be nonzero in gradient." << endl;
    a.init(0);
    hh.init(0);
    hh[gc]=h;

    a.set_val(0,0,((*func)(sum(x,hh))-(*func)(dif(x,hh)))/(2.0*hh[gc]));
    err=BIG;
    for (i=1;i<NTAB;i++) { /* successive columns in the Neville tableau will */
                           /* go to smaller stepsizes and higher orders of  */
                           /* extrapolation                                 */
      hh[gc]/=CON;
      a.set_val(0,i,((*func)(sum(x,hh))-(*func)(dif(x,hh)))/(2.0*hh[gc])); /* try smaller step-
size */
      fac=CON2;
      for (j=1;j<=i;j++) { /* compute extrapolations of various orders, */
                           /* requiring no new function evaluations     */
```

```
            a.set_val(j,i,(a.get_val(j-1,i)*fac-a.get_val(j-1,i-1))/(fac-1.0));
            errt=max(fabs(a.get_val(j,i)-a.get_val(j-1,i)),fabs(a.get_val(j,i)-a.get_val(j-1,i-1)));
            /* the error strategy is to compare each new extrapolation to one */
            /* order lower, both at the present stepsize and the previous one */
            if (errt<=err) { /* if error is decreased, save the improved answer */
              err=errt;
              result[gc]=a.get_val(j,i);
            }
          }
          if (fabs(a.get_val(i,i)-a.get_val(i-1,i-1))>=SAFE*(err)) {
            /* if higher order is worse by a significant factor SAFE, then quit */
            /* early.                                                          */
            break;
          }
      }
    }
    return (result);
}

matrix grad(vector (*func)(vector&,matrix&,int),vector& x,matrix& gg, int pat, double h)
{
    int i,j;
    double err,errt,fac;
    matrix a(NTAB,NTAB);
    unsigned int row=x.get_dim();
    unsigned int col=(*func)(x,gg,pat).get_dim();

    vector hh(row);
    matrix result(row,col);

    for (int register gcol=0;gcol<col;gcol++)
    {

      for (int register grow=0;grow<row;grow++)
      {
          if (h==0.0) cerr << "h must be nonzero in gradient." << endl;
          a.init(0);
          hh.init(0);
          hh[grow]=h;

          a.set_val(0,0,((*func)(sum(x,hh),gg,pat)[gcol]-
(*func)(dif(x,hh),gg,pat)[gcol])/(2.0*hh[grow]));
          err=BIG;
          for (i=1;i<NTAB;i++) { /* successive columns in the Neville tableau will */
                                 /* go to smaller stepsizes and higher orders of  */
                                 /* extrapolation                                 */
```

```
            hh[grow]/=CON;
            a.set_val(0,i,((*func)(sum(x,hh),gg,pat)[gcol]-
(*func)(dif(x,hh),gg,pat)[gcol])/(2.0*hh[grow])); /* try smaller step-size */
            fac=CON2;
            for (j=1;j<=i;j++) { /* compute extrapolations of various orders, */
                                 /* requiring no new function evaluations     */
              a.set_val(j,i,(a.get_val(j-1,i)*fac-a.get_val(j-1,i-1))/(fac-1.0));
              errt=max(fabs(a.get_val(j,i)-a.get_val(j-1,i)),fabs(a.get_val(j,i)-a.get_val(j-1,i-1)));
              /* the error strategy is to compare each new extrapolation to one */
              /* order lower, both at the present stepsize and the previous one */
              if (errt<=err) { /* if error is decreased, save the improved answer */
                   err=errt;
                   result.set_val(grow,gcol,a.get_val(j,i));
              }
            }
            if (fabs(a.get_val(i,i)-a.get_val(i-1,i-1))>=SAFE*(err)) {
              /* if higher order is worse by a significant factor SAFE, then quit */
              /* early.                                                          */
              break;
            }
          }
    }
  }
  return (result);
}

vector grad(double (*func)(vector&,int pat),vector& x,int pat,double h)
{
  int i,j;
  double err,errt,fac;
  matrix a(NTAB,NTAB);
  vector hh(x.get_dim());
  vector result(x.get_dim());

  for (int register gc=0;gc<x.get_dim();gc++)
  {
    if (h==0.0) cerr << "h must be nonzero in gradient." << endl;
    a.init(0);
    hh.init(0);
    hh[gc]=h;

    a.set_val(0,0,((*func)(sum(x,hh),pat)-(*func)(dif(x,hh),pat))/(2.0*hh[gc]));
    err=BIG;
    for (i=1;i<NTAB;i++) { /* successive columns in the Neville tableau will */
                           /* go to smaller stepsizes and higher orders of   */
                           /* extrapolation                                  */
```

```
        hh[gc]/=CON;
        a.set_val(0,i,((*func)(sum(x,hh),pat)-(*func)(dif(x,hh),pat))/(2.0*hh[gc])); /* try
smaller step-size */
        fac=CON2;
        for (j=1;j<=i;j++) { /* compute extrapolations of various orders, */
                             /* requiring no new function evaluations     */
          a.set_val(j,i,(a.get_val(j-1,i)*fac-a.get_val(j-1,i-1))/(fac-1.0));
          errt=max(fabs(a.get_val(j,i)-a.get_val(j-1,i)),fabs(a.get_val(j,i)-a.get_val(j-1,i-1)));
          /* the error strategy is to compare each new extrapolation to one */
          /* order lower, both at the present stepsize and the previous one */
          if (errt<=err) { /* if error is decreased, save the improved answer */
            err=errt;
            result[gc]=a.get_val(j,i);
          }
        }
        if (fabs(a.get_val(i,i)-a.get_val(i-1,i-1))>=SAFE*(err)) {
          /* if higher order is worse by a significant factor SAFE, then quit */
          /* early.                                                           */
          break;
        }
    }
  }
  return (result);
}
```

```
//---------------------------------------
// File: funcann.h
// Objective and Constraints Functions
//---------------------------------------

double f(vector&,int);
double ft(vector&);
vector h(vector&);
vector g(vector&,int);
```

```
//-------------------------------------------
// File: funcann.C
// Objective and Constraints functions Body
//-------------------------------------------

#include "ann.h"
#include "vector.h"

extern ann network;
extern unsigned int n,m,p;
extern int fiter;

double f(vector& w,int pat)
{
  network.update_weight_matrix(w);
  fiter++;
  return (network.E(pat));
}

double ft(vector& w)
{
  network.update_weight_matrix(w);
  fiter++;
  return (network.E());
}

vector h(vector& w)
{
  vector vh(0);

  return (vh);
}

vector g(vector& w,int pat)
{
  vector vg(2*network.nb_neurons[1]);
  network.update_weight_matrix(w);
  for (register i=0;i<network.nb_neurons[1];i++) {
    vg[2*i]=-network.weighted_sum(i,pat)+network.weight_bound;
    vg[2*i+1]=network.weighted_sum(i,pat)+network.weight_bound;
  }
  return (vg);
}
```

```
//-------------------------
// File: ann.h
// ANN Class
//-------------------------

#include <stdio.h>
#include <iostream.h>

class vector;
class matrix;
//class sparse;
struct cell;

class ann {

private:
  char net_filename[10];
  char out_filename[10];
  FILE *net_fp;
  FILE *out_fp;
  unsigned int nb_layers;
  unsigned int tot_neurons;
  unsigned int nb_input;
  unsigned int nb_output;
  unsigned int nb_patterns;
  vector* connexion_vector;
  //sparse* weight_matrix;
  matrix* weight_matrix;
  matrix* pattern;
  vector* input;

public:
  unsigned int np;
  unsigned int* nb_neurons;
  double weight_bound;
  vector* weight_vector;
       ann();
      ~ann();
       info();
  vector get_weight();
       update_weight_matrix(vector&);
  double activation(double);
  vector output(vector&);
  double weighted_sum(int,int);
  double get_pat_vec(int,int);
  double E();
```

```
  double E(int);
  double E_henon();
        set_input(unsigned int, double);
  double verif();
        load_weight();
        write_out();

  test();
};
```

```
//--------------------
// File: ann.C
// ANN Class Body
//--------------------

#include <iostream.h>
#include <math.h>
#include "ann.h"
#include "vector.h"
#include "matrix.h"
//#include "sparse.h"
#include "multiply.h"

ann::ann()
{
  cout << "enter net filename:";
  cin >> net_filename;
  if (!(net_fp=fopen(net_filename,"r"))) {cerr << "unable to open file" << endl; exit(1);}
  //cout << "enter outputs filename:";
  //cin >> out_filename;
  //if (!(out_fp=fopen(out_filename,"w"))) {cerr << "unable to open file" << endl; exit(1);}

  fscanf(net_fp,"%i",&nb_layers);

  unsigned int dim=0;
  tot_neurons=0;

  nb_neurons=new unsigned int[nb_layers];

  for (register i=0;i<nb_layers;i++) {
    fscanf(net_fp,"%i",nb_neurons+i);
    if (i) dim+=nb_neurons[i-1]*nb_neurons[i];
    tot_neurons+=nb_neurons[i];
  }
  nb_input=nb_neurons[0];

  input=new vector(nb_input);

  nb_output=nb_neurons[nb_layers-1];

  connexion_vector=new vector(tot_neurons);
  weight_vector=new vector(dim);

  //weight_matrix=new sparse(tot_neurons,tot_neurons);
  weight_matrix=new matrix(tot_neurons,tot_neurons);
```

```
  fscanf(net_fp,"%lf",&weight_bound);

  fscanf(net_fp,"%i",&nb_patterns);
  np=nb_patterns;
  pattern=new matrix(nb_patterns,nb_input+nb_output);

  double val=0;

  for (register i=0;i<nb_patterns;i++)
    for (register j=0;j<nb_input+nb_output;j++) {
      fscanf(net_fp,"%lf",&val);
      pattern->set_val(i,j,val);
    }

  fclose(net_fp);
}

ann::load_weight()
{
  char weight_filename[20];
  FILE* weight_fp;
  double val;

  cout << "enter weight filename:";
  cin >> weight_filename;

  if (!(weight_fp=fopen(weight_filename,"r")))
    { cerr << "unable to open file" << endl;
     exit (1);
    }

  for (register i=0;i<weight_vector->get_dim();i++) {
    fscanf(weight_fp,"%lf",&val);
    weight_vector->set_val(i,val);
  }

  fclose(weight_fp);
  cout << "weights:";
  weight_vector->print();
  update_weight_matrix(*weight_vector);
}

ann::~ann()
{
  //fclose(out_fp);
}
```

```
ann::info()
{
  cout << "NETWORK FILE: " << net_filename << endl;
  cout << "number of inputs: " << nb_input << endl;
  cout << "number of outputs: " << nb_output << endl;
  cout << "number of hidden layers: " << (nb_layers-2) << endl;
  for (register i=0;i<nb_layers-2;i++)
    cout << "hidden layer # " << (i+1) << ": " << nb_neurons[i+1] << " neurons" << endl;
  cout << "weight boundary: ";
  cout << weight_bound << endl;
  cout << endl;
  cout << "training pattern:" << endl;
  pattern->print();
}

vector ann::get_weight()
{
  vector result(weight_vector->get_dim());
  for (register i=0;i<result.get_dim();i++)
    result[i]=weight_vector->get_val(i);

  return (result);
}

ann::update_weight_matrix(vector& w)
{
  unsigned int col_indx,row_indx;
  unsigned int cpt=0;
  weight_matrix->init(0);

  col_indx=nb_neurons[0];
  row_indx=0;
  for (register i=0;i<nb_layers-1;i++) {
    for (register entry=0;entry<nb_neurons[i];entry++) {
      for (register j=0;j<nb_neurons[i+1];j++)
        weight_matrix->set_val(row_indx+entry,col_indx+j,w.get_val(cpt++));
    }
    col_indx+=nb_neurons[i+1];
    row_indx+=nb_neurons[i];
  }
}

double ann::activation(double in_neuron)
{
  double result;
```

```
//  result=1./(1+exp(-in_neuron)); // binary sigmoid
  result=(1-exp(-in_neuron))/(1+exp(-in_neuron)); // bipolar sigmoid
  return (result);
}

vector ann::output(vector& input)
{
  //cell* curr;
  //double sum;
  vector result(nb_output);

  for (register i=nb_neurons[0];i<tot_neurons;i++)
    input[i]=0;

  for (register i=nb_neurons[0];i<tot_neurons;i++)
    input[i]=activation(dot(weight_matrix->col2vec(i),input));

    // sparse matrix version
    //curr=weight_matrix->col_elts[i];
    //sum=0;
    //while (curr) {
      //sum=sum+input[curr->rowpos]*curr->val;
      //curr=curr->next_in_col;
    //}
    //input[i]=activation(sum);
  //}

  for (register i=0;i<nb_output;i++)
    result[i]=input[i+tot_neurons-nb_output];

  return (result);
}

double ann::weighted_sum(int j,int pat)
{
  vector vec(tot_neurons);

  for (register i=0;i<nb_input;i++)
     vec.set_val(i,pattern->get_val(pat,i));
  for (register i=nb_input;i<tot_neurons;i++) vec.set_val(i,0);
  return (dot(weight_matrix->col2vec(j),vec));
}

double ann::get_pat_vec(int pat,int j)
{
  return pattern->get_val(pat,j);
```

```
}

double ann::E()
{
  double result=0;
  vector out_vec(nb_output);

  for (register i=0;i<nb_patterns;i++) {
    for (register j=0;j<nb_input;j++) {
      connexion_vector->set_val(j,pattern->get_val(i,j));}
      out_vec=output(*connexion_vector);
    for (register j=0;j<nb_output;j++)
      out_vec[j]-=pattern->get_val(i,j+nb_input);
    result+=dot(out_vec,out_vec);
  }
  return (result);
}

double ann::E(int pat)
{
  double result=0;
  vector out_vec(nb_output);

  for(register j=0;j<nb_input;j++) {
    connexion_vector->set_val(j,pattern->get_val(pat,j));}
    out_vec=output(*connexion_vector);
  for (register j=0;j<nb_output;j++)
     out_vec[j]-=pattern->get_val(pat,j+nb_input);
  result+=dot(out_vec,out_vec);
  return (result);
}

double ann::E_henon()
{
  double a=1.4;
  double b=0.3;

  //cout << "zk-1=" << input->get_val(1) << " zk-2=" << input->get_val(2) << endl;

  double result=0;
  vector out_vec(nb_output);

  for (register i=0;i<3;i++)
    connexion_vector->set_val(i,input->get_val(i));
  out_vec=output(*connexion_vector);
  result=out_vec[0];
```

```
  result-=1-a*pow(input->get_val(1),2)+b*input->get_val(2);
  result*=result;
  return (result);
}

ann::set_input(unsigned int i, double val)
{
  input->set_val(i,val);
}

ann::test()
{
  vector input(tot_neurons);
  cout << "enter input vector:" << endl;
  for (register i=0;i<nb_input;i++) {
    cout << "input[" << i << "]=";
    cin >> input[i];
  }
  cout << "output=";
  output(input).print();
  cout << endl;
}

double ann::verif()
{
  vector out_vec;
  vector in_vec(tot_neurons);
  int correct=0;
  double percent=0;

  double threshold=0.5;
  for (register i=0;i<nb_patterns;i++) {
   for (register j=0;j<nb_input;j++) {
      in_vec.set_val(j,pattern->get_val(i,j));
    }
    out_vec=output(in_vec);
    //fprintf(out_fp,"%g,",out_vec[0]);
    if (out_vec[0]<threshold) out_vec[0]=0;
    else out_vec[0]=1;
    if (out_vec[0]==pattern->get_val(i,nb_input)) correct++;
  }
  //fprintf(out_fp,"\n");
  cout << "correct outputs " << correct << endl;
  percent=(double)correct/nb_patterns;
  cout << "efficiency " << percent << endl;
```

```
  return (percent);
}

ann::write_out()
{
  vector net_out(nb_output);

  for (register p=0;p<nb_patterns;p++) {
    for (register i=0;i<nb_input;i++) {
      connexion_vector->set_val(i,pattern->get_val(p,i));
      net_out=output(*connexion_vector);
      fprintf(out_fp,"%g ",net_out[0]);
    }
    fprintf(out_fp,"\n");
  }
}
```