APPROVAL SHEET

Title of Dissertation: Analysis of Low-Energy Electron Diffraction IV Spectra Using Artificial Neural Networks

Name of Candidate: David Grant Simpson
Doctor of Philosophy, 1999

Dissertation and Abstract Approved:

Dr. Philip Rous
Associate Professor of Physics
Department of Physics

Date Approved: 7th Odober 1999

Name: David Grant Simpson.

Permanent Address: 140 Woodridge Place,
Laurel, Maryland 20724-1802.

Degree and date to be conferred: Ph.D., 1999.

Date of Birth: January 17, 1961.

Place of Birth: Charleston, West Virginia.

Secondary Education: South Charleston High School,
South Charleston, West Virginia.
Graduation date: June 7, 1979.

Collegiate institutions attended:

1994–1999	University of Maryland, Baltimore County, Baltimore, Maryland. Ph.D., Applied Physics, 1999.
1990–1993	Johns Hopkins University, Baltimore, Maryland. M.S., Applied Mathematics, 1993.
1984–1990	Johns Hopkins University, Baltimore, Maryland. M.S., Applied Physics, 1990.
1979–1984	Virginia Polytechnic Institute and State University, Blacksburg, Virginia. B.S., Physics, 1984 (minor in mathematics).

Major: Applied Physics.

Professional publications:

"An Alternative Lunar Ephemeris Model for On-Board Flight Software Use," *Proceedings of the 1999 NASA Goddard Flight Mechanics Symposium, NASA Goddard Space Flight Center, Greenbelt, Maryland.*

"Spacecraft Attitude Determination Using the Earth's Magnetic Field," *Proceedings of the 1989 NASA Goddard Flight Mechanics / Estimation Theory Symposium, NASA Goddard Space Flight Center, Greenbelt, Maryland.*

Professional positions held:

1991– *NASA Goddard Space Flight Center, Greenbelt, Maryland.*
Flight Software Senior Designer.
Currently the project manager for the Hubble Space Telescope's DF-224 on-board attitude control computer flight software project.

1991–1994 *Prince George's Community College, Largo, Maryland.*
Adjunct Associate Professor of Physics.
Taught sophomore-level calculus-based physics to science and engineering majors.

1985–1991 *OAO Corporation, Greenbelt, Maryland.*
Flight Software Technical Manager.
Developed and led flight software efforts for the on-board computers of NASA's International Ultraviolet Explorer, Solar Maximum Mission, and Extreme Ultraviolet Explorer spacecraft.

1980–1985 *COMSAT Laboratories, Clarksburg, Maryland.*
Member of Technical Staff.
Performed testing and computer analysis of satellite power systems for several communications satellites.

ABSTRACT

Title of Dissertation: Analysis of Low-Energy Electron Diffraction I-V Spectra Using Artificial Neural Networks

David Grant Simpson, Doctor of Philosophy, 1999

Dissertation directed by: Dr. Philip Rous, Associate Professor of Physics

Low-energy electron diffraction (LEED) has proven to be a very successful method for determining the structure of surfaces. However, the calculations involve the use of a global search algorithm, which can require substantial amounts of computer time. This Dissertation investigates the use of artificial neural networks as a method for increasing the efficiency of this search. Using the $Ni_{50}Pd_{50}(100)$ surface as an example, it is shown that once a neural network is trained on I-V curves produced by a LEED full dynamical calculation, it can successfully recognize the surface structure parameters from an experimental I-V curve.

ANALYSIS OF LOW-ENERGY ELECTRON DIFFRACTION *I-V* SPECTRA USING ARTIFICIAL NEURAL NETWORKS

by

David Grant Simpson, M.S.

Dissertation submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1999

Preface

As we approach the dawn of the twenty-first century, it is natural to pause and reflect on the scientific advancements that have transformed the world during the past hundred years, and to wonder about what the coming century will bring. One hundred years ago, many physicists assumed that our understanding of Nature was essentially complete, and that twentieth-century physics would consist largely of performing experiments which would yield measurements with increasing degrees of precision.

This view did not hold for long. Einstein's publication of the theories of special relativity in 1905 and general relativity in 1916 were followed shortly by the development of the quantum theory in the 1920s. These theories have revolutionized our understanding of Nature, and have given birth to whole new branches of physics which are still being explored today. Our understanding of these new fields has resulted in astonishing technological advancements which have changed the lives of people everywhere. These technological advancements have, in turn, made possible the experimental and computational techniques that have allowed us to advance our understanding of Nature even further.

The development of the quantum theory has made possible our modern understanding of the theory of solids. A sub-field of solid-state physics, *surface physics*, is concerned

with the ways in which the atomic composition and structure of a solid is changed in the vicinity of a surface (that is, at its interface with a surrounding vacuum or gas). This study is important for a thorough understanding of many processes of great practical importance, such as corrosion, semiconductor physics, and catalysis.

Experimental surface physics is based upon techniques that are sensitive to the structure and composition of the atoms of the solid near the surface, while being relatively insensitive to the atoms in the bulk of the material. The atoms of interest are those in the top few (usually around three) atomic layers near the vacuum interface, since it is generally found that those are the layers whose structure differs appreciably from that of the bulk solid.

Since the earliest days of surface physics, one of the most fundamental and useful of these experimental techniques has been *low-energy electron diffraction*, or *LEED*. LEED is an experimental technique used for surface crystallography in which one illuminates a solid surface with low-energy electrons and observes the resulting diffraction pattern. The resulting data allows one to determine the atomic positions of the surface layers, typically to within about 0.1 angstrom.

Only in the past few decades have advances in vacuum technology made it possible to keep surfaces clean for enough time to allow LEED techniques to investigate their surface properties. Technological advancements over these same decades have seen the development of digital computers of increasing speed and power, which have made possible the theoretical calculations with which to properly interpret LEED experimental data for an accurate understanding of the structure and composition of crystalline surfaces.

This Dissertation is a description of a new development in this field: the application of artificial neural networks to the interpretation of LEED experimental data. We begin with

a overview of the physics of surfaces and LEED experimental techniques in Chapter 1. Chapter 2 gives a brief description of the theoretical calculations used to predict the outcome of a LEED experiment. These calculations are used to create a database that is used by the computational technique described in Chapter 3, that of artificial neural networks. Chapter 4 gives a description of the research with which these techniques were developed, and Chapters 5 and 6 describe the results of the research and the applicability of the artificial neural network techniques to the analysis of experimental LEED data.

Dedication

For my family and friends, without whose constant encouragement this work would not have been possible; and for Hazel Childress, my high school math teacher, who taught me to appreciate the beauty of mathematics.

Acknowledgement

The author wishes to express his deepest thanks and appreciation for the guidance and support of Dr. Philip Rous for his guidance and advice in preparing this work.

Contents

Preface ii

Acknowledgements vi

1 Surface Physics and Low-Energy Electron Diffraction 1

1.1 Physics of Surfaces 3

1.2 Low-Energy Electron Diffraction 6

1.2.1 Multiple Scattering 11

1.2.2 Thermal Effects 11

1.3 Experimental Apparatus 12

1.3.1 Instrument Response Function 16

1.3.2 Preparation of Samples 17

2 LEED Dynamical Calculations and Structure Analysis 18

2.1 LEED Dynamical Calculations 19

2.1.1 The Inner Potential 20

2.1.2 Inelastic Processes 22

2.1.3 Intra-atomic Scattering 24

2.1.4 Interatomic Scattering 26
2.2 LEED Structure Analysis 29
2.2.1 Reliability Factors 30
2.2.2 Exhaustive Global Search 32
2.2.3 Steepest Descent Method 33
2.2.4 Simulated Annealing 35
2.2.5 Genetic Algorithms 36

3 Artificial Neural Networks **40**
3.1 Introduction 40
3.2 Biological Neural Networks 41
3.3 Artificial Neural Networks 45
3.3.1 Examples of Artificial Neural Networks 47
3.4 Backpropagation Networks 48
3.5 Scaling 50
3.6 Learning 52
3.6.1 Error Derivatives for Hidden-to-Output Weights 53
3.6.2 Error Derivatives for Input-to-Hidden Weights 55
3.7 Learning Algorithms 57
3.7.1 Constant Learning Rate 58
3.7.2 Momentum 59
3.7.3 Adaptive Learning Rates 59
3.8 Weight Initialization 61
3.9 Batch Learning 61

4 Development of Artificial Neural Networks for LEED Surface Structure Determination 63

4.1 Network Program Design 64

4.2 Initial Testing 65

4.2.1 Adjustment of Network Parameters 68

4.3 Training with Calculated I-V Data 74

4.4 Tests of Data Set Reduction 80

4.4.1 Reduction of Number of Training Structures 80

4.4.2 Reduction of Number of Points Per Training Structure 83

4.5 Conclusions 85

5 Application of Artificial Neural Networks to Low-Energy Electron Diffraction 86

5.1 Introduction 86

5.2 Description of Experimental Sample 87

5.3 Theoretical Calculations 88

5.4 Initial Results 89

5.5 Correction for Interlayer Spacings 97

5.6 Test with Wider Training Range and Fewer Examples 99

5.7 The Pendry Y-function 103

5.8 Inner Potential Energy Shift 106

5.9 Testing With the Y-function 106

6 LEED Surface Structure Determination Using Artificial Neural Networks 108

6.1 Determination of Interlayer Spacings 109

6.2 Determination of Compositions Over the Full Training Range 110
6.3 Simultaneous Determination of Compositions and Interlayer Spacings . . . 111
6.3.1 Calculated Data 112
6.3.2 Experimental Data (Restricted Range) 113
6.3.3 Experimental Data (Full Range) 113
6.4 Further Improvements 117
6.5 Energy Shift 121
6.6 Transferability 123
6.7 Discussion 124
6.8 Concluding Remarks 126
6.8.1 Future Directions 126
6.8.2 Summary 127

A LEEDNET User's Guide **128**
A.1 Running LEEDNET 129
A.2 Environment Variables 130
A.3 File `LEEDNET.INI` 130
A.4 Formatting the Data 131
A.5 Training the Network 132
A.6 Using the Trained Network 134
A.7 Command Reference 135
A.8 Environment Variable Reference 141

B Listing of Program `LEEDNET.C` **143**

C Listing of Program FORMAT01.C **178**

D Listing of File LEEDNET.H **183**

E Listing of File LEEDNET.HLP **184**

List of Tables

2.1 Genetic algorithm results for maximizing $y = f(x)$ (initial population). . . 37

2.2 Genetic algorithm results for maximizing $y = f(x)$ (second generation). . 38

4.1 Network recognition of $y = \sin kx$ for $k = 1, 2$. 65

4.2 Network recognition of $y = A \sin x$ for $A = 1, 2$. 66

4.3 Network recognition of $y = \sin kx$ for $k = 1, 2, 3, 4, 5$. 67

4.4 Network recognition of $y = A \sin x$ for $A = 1, 2, 3$. 68

4.5 Network recognition of $y = \sin kx$ for $A = 1, 2, 3$ and $k = 1, 2, 3, 4, 5$. . . 69

4.6 Network recognition of $y = 3 \sin 2x$ with different numbers of hidden nodes. 73

4.7 Network recognition of $Ni_{50}Pd_{50}(100)$ I-V curves for six different top layer compositions. 76

4.8 Network recognition of $Ni_{50}Pd_{50}(100)$ I-V curves for six different top layer compositions, having trained on five. 78

4.9 Network recognition of $Ni_{50}Pd_{50}(100)$ I-V curves for six different second layer compositions. 78

4.10 Network recognition of $Ni_{50}Pd_{50}(100)$ I-V curves for six different second layer compositions, having trained on five. 79

4.11 Network recognition of $Ni_{50}Pd_{50}(100)$ I-V curves for 180 combinations of compositions in the top three atomic layers. 80
4.12 Network test results after training on 15 selected structures. 81
4.13 Network test results after training on every 17th structure. 83
4.14 Network test results for reducing number of points in training sets. 84

5.1 Network test results for network shown experimental I-V data. 97
5.2 Network test results for network shown experimental I-V data, corrected for interlayer spacings. 98
5.3 Network test results for network shown experimental I-V data, using reduced training set. 100
5.4 Network test results for network shown Y-function of experimental I-V data. 107

6.1 Network test results for determination of interlayer spacings. 110
6.2 Network test results for determination of atomic layer compositions (full range of training data). 111
6.3 Network test results for simultaneous determination of compositions and interlayer spacings, with network shown calculated data. 112
6.4 Network test results for simultaneous determination of compositions and interlayer spacings (restricted range of training data). 114
6.5 Initial network test results for simultaneous determination of compositions and interlayer spacings (full range of training data). 115
6.6 Improved network test results for simultaneous determination of compositions and interlayer spacings with $\phi = 0.85$ (full range of training data). 116

6.7 Network test results for determination of atomic compositions, comparing two methods for ensuring network convergence. 122
6.8 Network test results for a variety of inner potential energy shifts. 123
6.9 Network test results for simultaneous determination of compositions and interlayer spacings for $Cu_{50}Pd_{50}(100)$. 124

List of Figures

1.1 Surface relaxation and reconstruction. 4

1.2 Surface relaxation. 5

1.3 Universal curve of electron mean free path vs. incident electron energy. . . 8

1.4 Schematic diagram of a simple LEED apparatus. 13

1.5 Typical LEED I-V curve. 15

2.1 Muffin-tin potential, for two spatial two dimensions. 21

3.1 Schematic figure of a typical vertebrate neuron. 42

3.2 Details of nerve cell body. 43

3.3 Encoding of letters for character recognition. 48

3.4 An artificial neural network. 49

3.5 Activation function $f(x) = 1/(1 + e^{-x})$. 51

4.1 Network error vs. training epoch, with and without adaptive learning rate. 71

4.2 Network error vs. training epoch, with and without momentum term. . . . 72

4.3 Input of I-V data to a backpropagation network. 75

5.1 Bulk termination of $Ni_{50}Pd_{50}$ in the (100) plane. 90

5.2 Experimental and theoretical I-V spectra for $Ni_{50}Pd_{50}(100)$. 91
5.3 Theoretical I-V spectra for $Ni_{50}Pd_{50}(100)$ for several top-layer compositions. 92
5.4 Theoretical I-V spectra for $Ni_{50}Pd_{50}(100)$ for several interlayer spacings. . 93
5.5 Network training error for $Ni_{50}Pd_{50}(100)$ vs. number of training epochs. . 101
5.6 Network independent test error for $Ni_{50}Pd_{50}(100)$ vs. number of training epochs. 102
5.7 Network training error for $Ni_{50}Pd_{50}(100)$ vs. number of training epochs, using $Y(E)$. 105

6.1 Theoretical I-V curves for $Ni_{50}Pd_{50}(100)$ (with "swapped" layer spacings). 118

Chapter 1

Surface Physics and Low-Energy Electron Diffraction

"Natura inest in mentibus nostris insatiabilis quaedam cupiditas veri videndi."

— Cicero, *Tusculanae Disputationes, I, ii, 44*

ONE of the newest branches of solid-state physics is *surface physics*, the study of the physical processes that occur at the surface of a solid—that is to say, in the few atomic layers near its interface with a surrounding gas or vacuum. The study of the physics at solid surfaces is of great importance for understanding many processes of practical interest. For example, corrosion and catalysis involve chemical processes that take place on a material surface, as does much of the interesting physics that takes place in semiconductor devices (for example, at the junction between p-type and n-type semiconductors, or between a metal oxide and a semiconductor in MOSFETs and similar devices [1]).

A greater understanding of the physics and chemistry that takes place at surfaces may yield many practical benefits. For example, it may result in ways to prevent unwanted

corrosion or the development of new materials and devices with many interesting and useful properties.

Surface physics is an emerging discipline that is still very much in its infancy. Only in recent years has experimental and computational technology advanced to a point that has allowed physicists to investigate in detail the ways in which atoms arrange themselves in the vicinity of a material surface and the chemical processes that take place on a surface. At the moment, most studies are confined to investigations of fairly simple surfaces—usually crystalline metals or semiconductors whose surfaces have been thoroughly cleaned. Typically these surfaces are studied in an ultra-high vacuum (UHV) environment to prevent complicating contaminations from distorting the data. Only in coming years, after we have gained a thorough understanding of the physics of clean crystalline surfaces, will we be able to begin investigating more complicated systems, such as highly contaminated surfaces at atmospheric pressure or the physics at the surfaces of amorphous solids. Such advances will undoubtedly depend on further developments in experimental and computational technology.

In this Dissertation I will review the experimental, theoretical, and computational background behind one of the most important experimental techniques of the surface physicist: *low-energy electron diffraction,* or *LEED.* This will be followed by a description of my recent research in the use of computer artificial intelligence in helping to solve a difficult computational problem in LEED: that of extracting information about the surface structure and composition from LEED diffraction patterns.

1.1 Physics of Surfaces

When a solid material is cut, the atoms in the vicinity of the newly-formed surface will generally rearrange themselves into a pattern that differs from that of the bulk material. This is because atoms near the surface of a material lack coordination compared to atoms deep in the bulk, causing the surface atoms rearrange themselves into a minimum-energy configuration that differs from that of the bulk.

Experimental studies have shown that, in metals, the presence of a surface interface generally results in a *relaxation* of the surface, in which the spacings between atomic layers parallel to the surface are changed relative to the bulk, while the crystal structure remains essentially unchanged [1, 2]. The region over which this relaxation takes place, called the *selvedge,* generally extends to just a few atomic layers into the bulk of the crystal (Fig. 1.1).

Relaxation of the top layer of surface atoms tends towards the bulk so that the inter-atomic layer spacing at the selvedge is contracted relative to the bulk. The change in the layer spacings deeper in the crystal is often oscillatory; this may be understood by referring to Fig. 1.2. The figure shows Wigner-Seitz cells drawn around each of the core atoms in the surface, where each cell represents a portion of the mobile conduction electrons surrounding that atom. At the surface, the conduction electrons will redistribute themselves into the smooth, lower-energy configuration shown. This net shifting of electrons toward the interior of the material will also shift the ion cores' electrostatic equilibrium position in the same direction, resulting in a net shift of the ion cores into the bulk and a general contraction of the atomic layers near the surface.

The experimental study of the behavior of the atoms near the solid surface is hampered

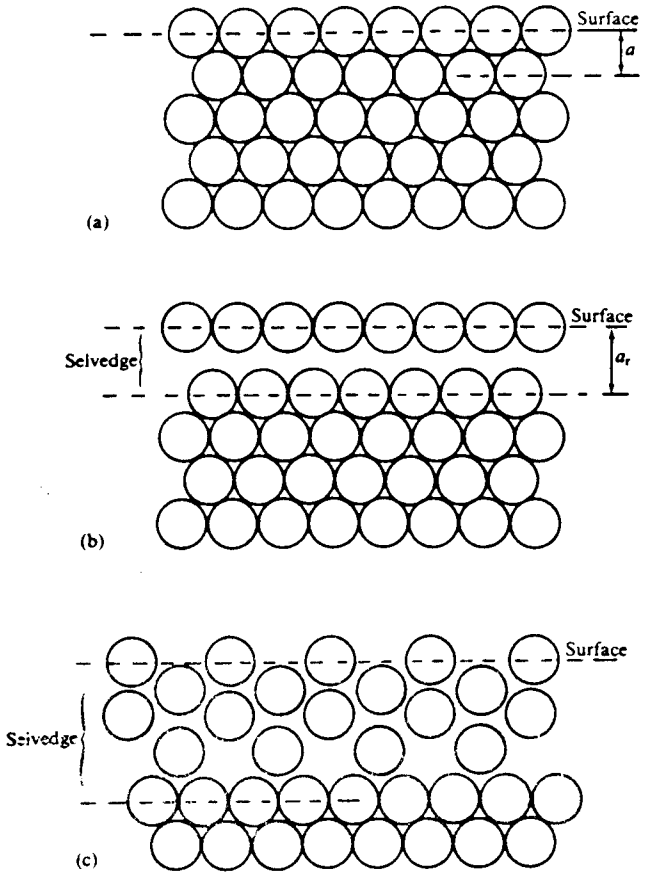

Figure 1.1: Surface relaxation and reconstruction.
(a) Atomic arrangement of bulk solid. (b) Relaxation of surface atoms. (c) Reconstruction of surface atoms. (From Prutton, 1994 [1].)

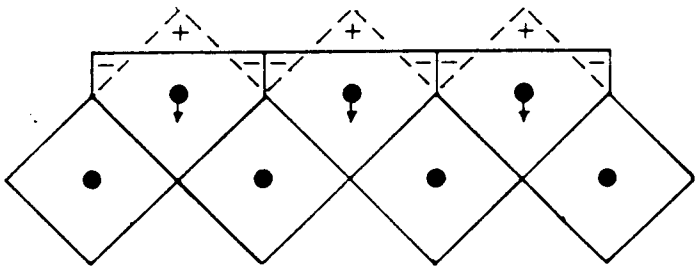

Figure 1.2: Surface relaxation.
(From Finnis and Heine, 1974 [3].)

by the presence of contaminating atoms which are easily adsorbed onto the surface. In order to simplify the study of the surface atoms, it is necessary to observe the surface in an ultra-high vacuum, where the density of contaminating atoms is sufficiently low to allow the pure surface to be studied at length without significant interference from contaminants. Only in recent years have advances in vacuum technology made it possible to create vacuums of sufficiently low pressure ($\sim 10^{-10}$ torr) to allow crystalline surfaces to be studied before they become excessively contaminated by atoms from the surrounding gases.

1.2 Low-Energy Electron Diffraction

Many experimental methods have been devised for studying the atoms at a crystalline surface. Each method is required to be surface sensitive in some way, so that it returns information about the surface atoms while ignoring the atoms in the bulk material. One may, for example, bombard a surface with high-energy electrons or x rays at grazing incidence so that they never enter the bulk, as is done in reflection high-energy electron diffraction (RHEED) and grazing-incidence x-ray diffraction.

One of the most important surface-sensitive experimental methods of the surface physicist is *low-energy electron diffraction,* or *LEED.* LEED has its origins in the famous 1927 experiment of Davisson and Germer [4], in which a crystal of nickel was bombarded by a beam of low-energy (54 eV) electrons. Measurement of the scattering angle of the scattered electrons allowed Davisson and Germer to make the first experimental measurement of the electron wavelength—a vivid experimental confirmation of the wavelike nature of the electron, for which Davisson and Thomson were awarded the 1937 Nobel prize in

physics.

The surface sensitivity of low-energy electrons is best illustrated with the so-called "universal curve," shown in Fig. 1.3. The universal curve shows the electron mean free path versus incident electron kinetic energy and shows two competing effects. The first effect is a general *decrease* in the electron mean free path through the material with increasing electron energy. This is caused by greater availability of energy levels for inelastic scattering in the material as the electron energy is increased. The second effect shown in the universal curve is a general *increase* in the electron mean free path through the material with increasing electron energy. This is just an artifact of the electron's greater velocity at higher energy, which allows it to travel farther before being inelastically scattered.

As shown in the figure, the first of these effects dominates at lower energies, while the second dominates at higher energies. Around 50–100 eV, the two effects "balance," producing a broad minimum in the mean free path around these energies. By a happy coincidence, this minimum occurs at electron energies for which the de Broglie wavelength is short enough to return useful information on the surface structure. By another coincidence, the magnitude of this minimum has a mean free path of a few angstroms, just the right magnitude to be sensitive to the first few atomic layers of the material.

In modern LEED experiments, a crystalline surface is illuminated with a beam of low-energy ($<$ 1000 eV) electrons to produce a diffraction pattern. The positions of the diffraction spots give an indication of the symmetry of the crystal lattice. Because LEED is a surface-sensitive technique, however, the positions of the diffraction spots really only give an indication of the positions of the atoms in the top atomic layer, and perhaps the positions of adsorbed atoms. In order to determine the type of crystal lattice in the bulk

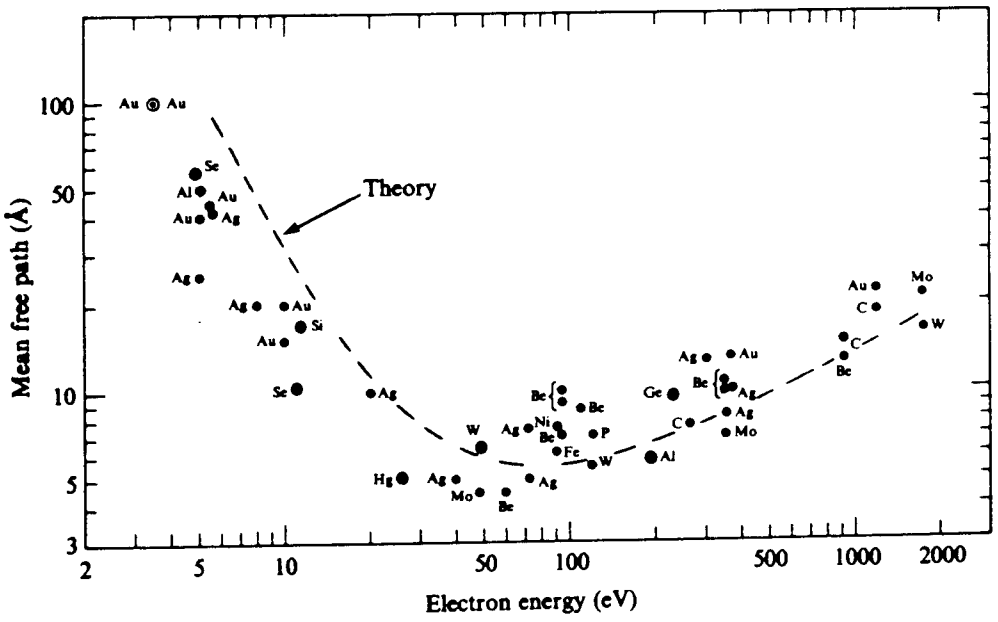

Figure 1.3: Universal curve of electron mean free path vs. incident electron energy. (From Zangwill, 1988 [2].)

material, a bulk-sensitive technique such as x-ray diffraction must be used [5].

Besides examining the positions of the diffracted beams, additional information may be extracted from the experiment by observing changes in the intensity of the diffraction spots as the sample is rotated or the energy of the incident electrons is changed. Specifically, the following types of data may be collected [6]:

I-V curves. These are plots of the diffracted electron beam intensities I as a function of the incident electron energy (or, equivalently, the potential V through which the incident electrons are accelerated). In this way, the changes in the diffracted electron intensity may be observed as the incident electron wavelength is changed. Care must be taken to ensure that the incident electron energies are low enough that they will be sensitive to scattering by atoms near the surface, rather than the bulk. The incident electron energies must also be high enough so that the incident electron wavelength is not longer than the scale of the atomic distances being studied.

I-V curves are by far the most common type of data collected in LEED experiments, and will be the data set used for the research with artificial neural networks to be discussed later. The data is collected for a particular diffracted electron beam, for fixed incidence angles θ (polar angle) and φ (azimuthal angle).

I-θ curves. In this type of data, one measures the diffracted electron intensity for a particular diffracted beam, varying the polar incidence angle θ and keeping the azimuthal angle φ and incident electron energy fixed. This is the type data collected by Davisson and Germer in their 1927 experiment to measure the wavelength of the electron.

I-φ curves. These are produced by measuring the diffracted electron intensity for a particular diffracted beam while varying the azimuthal incidence angle φ. The polar incidence angle θ and electron energy are held constant.

For any of these types of curves, the data is generally collected for each of several different diffraction beams. The data for beams with symmetrically equivalent Miller indices (such as (10), $(\bar{1}0)$, (01), and $(0\bar{1})$) are generally averaged to yield the final data set. Also, the units in which the intensities are measured are not easily related to theoretical calculations, so intensity units are taken be arbitrary and are usually normalized in some way. One may, for example, arbitrarily set the maximum intensity in a data set to be unity, and scale all other intensities accordingly.

It is believed that there is roughly an equivalent amount of information about the surface structure contained in each of these three types of data sets [6]. In the case of I-θ and I-φ curves, the information content is constrained by how much the incidence angles can be varied ($0 \leq \theta \leq 90°$, $0 \leq \varphi \leq 180°$). For I-V data, there is a practical constraint on the range of incident electron energies, which also constrains the information content: the electron wavelength must be short enough to contain information about the distance scales being studied, and long enough to be surface-sensitive. The low-energy electrons used cannot be of arbitrarily low energy; the electron de Broglie wavelength must be less that the interatomic distances being studied (~ 2 Å) in order for the method to return any useful information on the surface structure. This places a lower energy limit on LEED electrons of about

$$E = \frac{h^2}{2m_e\lambda^2} \sim 40 \quad \text{eV} \ , \tag{1.1}$$

where h is Planck's constant, m_e is the electron mass, and λ is the electron de Broglie wavelength.

1.2.1 Multiple Scattering

The surface atoms of a material have a large scattering cross-section (comparable with the physical cross-section) with respect to incident low-energy electrons. This means that the scattering probability for low-energy electrons in a close-packed solid is very high, and it is this property that makes LEED surface-sensitive. This same property also adds a complication to the analysis of LEED data: electrons entering the material will generally be scattered multiple times on their way out of the material. This *multiple scattering* is fairly difficult to deal with computationally, and only in recent years has computer technology advanced to a point that has allowed accurate LEED predictions to be made which adequately allow for it.

Chapter 2 reviews several methods which are used to perform the so-called LEED *dynamical calculations* which allow for this multiple scattering of electrons.

1.2.2 Thermal Effects

The behavior of the diffracted electron beams will generally depend upon the temperature of the sample, since increasing temperatures cause the atoms in the crystal to vibrate with increasing amplitude and frequency about their equilibrium positions. The three most important effects that are observed when the temperature of a sample is increased are a broadening of the diffraction spots, a general decrease in the intensity of the spots, and an increase in the intensity of the background illumination (between the spots) [6].

The broadening of the diffraction spots can be attributed to momentum transfer between the electrons and the crystal lattice via phonon excitation. The decrease in spot intensity with increasing temperature is due to interference effects: as the temperature is increased,

the atoms in the sample vibrate about their equilibrium positions, partially disrupting the periodicity of the crystal lattice. This decrease in intensity may described by [7]

$$I = I_0 \exp(-\tfrac{1}{3}\langle u^2 \rangle G^2) \ , \tag{1.2}$$

where I is the diffracted electron intensity, I_0 is the intensity that would be observed if the electrons were scattered from a rigid lattice, and $\langle u^2 \rangle$ is the mean squared amplitude of vibration in the direction of **G**, the reciprocal lattice vector associated with the electron beam. The exponential factor in Eq. (1.2) is called the *Debye-Waller factor,* and is often written as $\exp(-2M)$.

The third thermal effect, the increase in background intensity with increasing temperature, is essentially due to the decrease in spot intensity. The electrons that would otherwise have been scattered into the beams are instead scattered into the background when they no longer satisfy the Bragg scattering conditions due to thermal vibrations of the crystal lattice.

1.3 Experimental Apparatus

A simplified version of the apparatus used in a typical LEED experiment is shown schematically in Fig. 1.4. The entire apparatus is contained within an ultra-high vacuum (UHV) chamber, which maintains a vacuum of 10^{-10} torr or better. The ultra-high vacuum is necessary to reduce the rate of accumulation of contaminants on the surface, so that the surface can be kept relatively clean during the experiment.

A beam of electrons is emitted by a heated tungsten filament inside the electron gun. Other components of the electron gun serve to accelerate, collimate, and focus the emitted

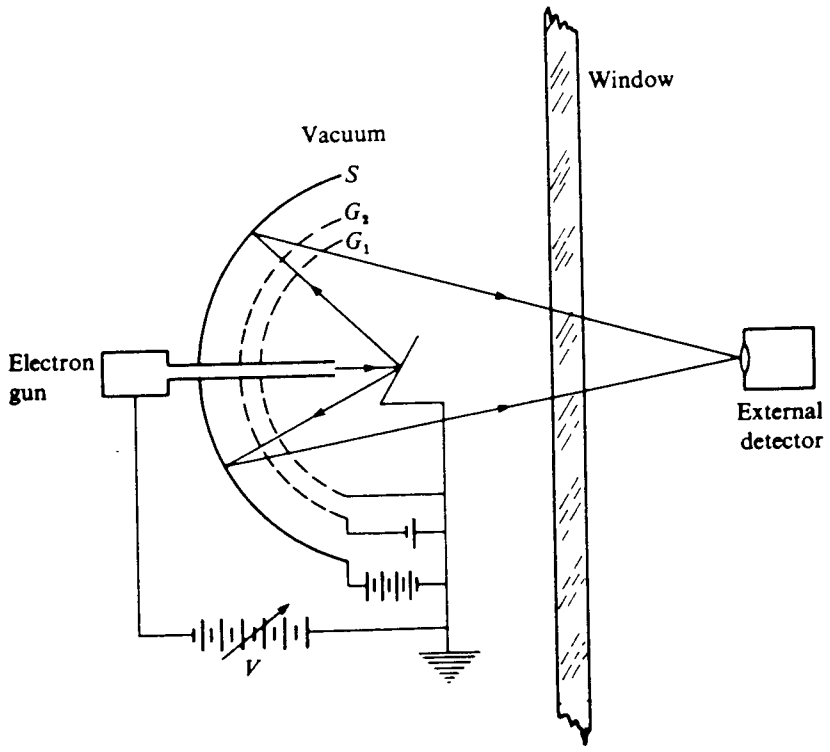

Figure 1.4: Schematic diagram of a simple LEED apparatus. (From Zangwill, 1988 [2].)

electrons in the direction of the sample, which is mounted on a movable stage that can be rotated to hold the sample at any desired angle with respect to the incident electron beam. The emitted electron beam typically has a current of $\sim 10^{-9}$ A, corresponding to about 10^{10} electrons per second. The beam typically has a diameter of less than 1 mm.

Electrons scattered by the sample are reflected back toward grid G_1, which is held at the same potential as the sample (earth ground) to provide field-free region through which the electrons can travel before reaching the grids. This field-free region is necessary in order to avoid unwanted deflection of the scattered electrons. Grid G_2, called a *suppressor grid,* is held at a negative potential designed to reject inelastically scattered electrons so that only elastically scattered electrons pass through to a fluorescent screen S. Screen S is held at a large positive potential that serves to accelerate electrons toward the screen; the extra kinetic energy helps to excite the phosphors in the screen so that the electron impacts are more visible. The fluorescent screen may be viewed through a window in the vacuum chamber. More modern LEED systems may include a charge-coupled devide (CCD) camera in place of the phosphorescent screen for more accurate intensity measurements.

Other grids may be present, in addition to those shown in the figure. For example, there may be two suppressor grids that allow a range of electron energies to pass through to the screen. There may also be an additional grounded grid placed between the suppressor grids and the fluorescent screen to help prevent the large electric field from the fluorescent screen from affecting the suppressor grids. Helmholtz coils may also be present to cancel the geomagnetic field present in the laboratory.

Figure (1.5) is a typical LEED I-V curve, which shows the intensity of a LEED diffraction spot as a function of incident electron energy. There will be one such curve for each diffraction spot, with symmetrically equivalent beams being theoretically identical.

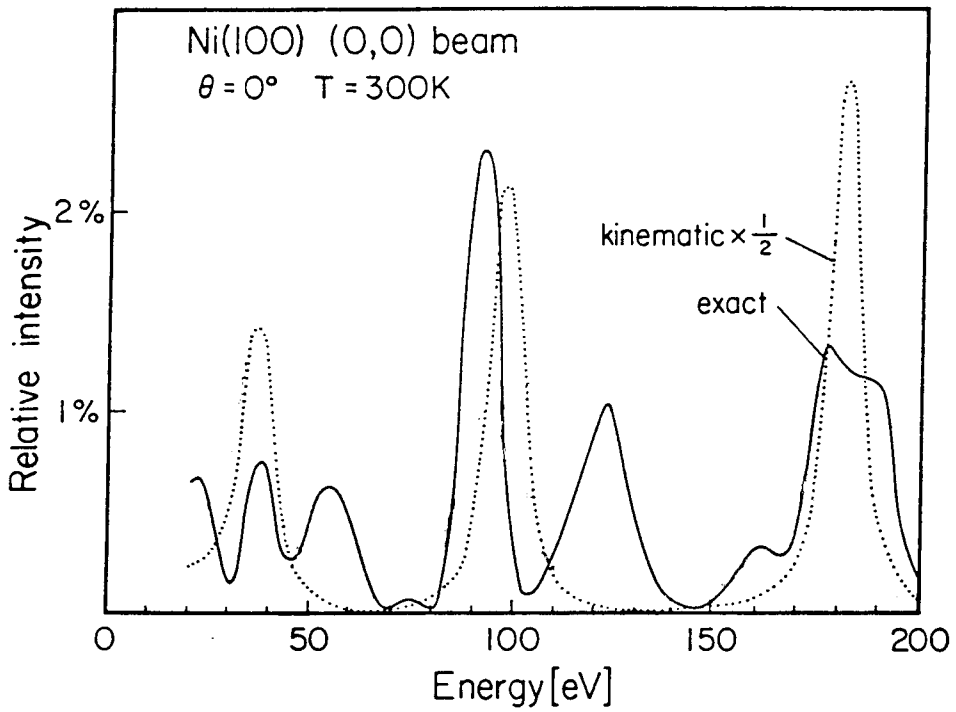

Figure 1.5: Typical LEED I-V curve.
The curve labeled "exact" is the product of a full dynamical LEED calculation for Ni(100). The curve label "kinematic" does not include the multiple scattering effects included in the dynamical calculation. Note the shift in peak positions and appearance of additional peaks in the dynamical calculation. (After Van Hove, Weinberg, and Chan, 1986 [6].)

These curves contain information about the surface structure, since changes in the structure produce changes in the shape of the curves. It is the goal of LEED surface structure analysis to extract information about the surface from these curves.

To extract surface structure information from a LEED I-V curve, one generally calculates theoretical I-V curves for a number of plausible surface structures, then compares each of these with the experimental curve to determine which theoretical curve best matches the experimental data. One then assumes that this best match represents the actual surface structure. It is this comparison between the LEED theoretical and experimental I-V curves that is the focus of this Dissertation.

1.3.1 Instrument Response Function

In any real laboratory measurement of LEED diffraction data, the observed data will differ from their theoretical ideal because of imperfections in the crystal lattice and instrument distortion. The effect of instrument distortion can be modeled using an *instrument response function* which corrects for such effects as the energy spread in the incident electron beam and the finite aperture width of the instrument.

Let $I(\mathbf{s})$ be the intensity of the diffracted electron beam measured by an ideal instrument, and let $J(\mathbf{s})$ be the beam intensity measured by a real instrument. Here $\mathbf{s}$ represents the momentum transfer from the incident to the diffracted electron beam,

$$\mathbf{s} = \mathbf{k}_{\text{out}} - \mathbf{k}_0 \ , \tag{1.3}$$

where $\mathbf{k}_0$ is the wave vector of the incident electron beam and $\mathbf{k}_{\text{out}}$ is wave vector of the diffracted electron beam. Both $I(\mathbf{s})$ and $J(\mathbf{s})$ are assumed to be beam intensities measured by diffraction from a perfect crystal lattice, so that they differ only by the inclusion of instrument distortion in $J(\mathbf{s})$. Then from linear system theory, the intensity $J(\mathbf{s})$ measured by the real instrument may be written as the convolution [8, 9, 10]

$$J(\mathbf{s}) = F\{t(\mathbf{r})\} * I(\mathbf{s}) \ , \tag{1.4}$$

where $t(\mathbf{r})$ is the *transfer function* linking I and J, $\mathbf{r}$ is a displacement vector connecting a pair of lattice points, and F denotes the Fourier transform of $t(\mathbf{r})$. The Fourier transform of the transfer function is known as the *instrument response function*

$$T(\mathbf{s}) = F\{t(\mathbf{r})\} = \int_{-\infty}^{\infty} t(\mathbf{r}) e^{i\mathbf{r}\cdot\mathbf{s}} d\mathbf{s} \ . \tag{1.5}$$

Eq. (1.4) may then be written

$$J(\mathbf{s}) = T(\mathbf{s}) * I(\mathbf{s}) \ . \tag{1.6}$$

The instrument response function may be written as the convolution product of several individual response functions, each of which describes a particular contribution:

$$T(\mathbf{s}) = T_{\rm es}(\mathbf{s}) * T_{\rm aw}(\mathbf{s}) * T_{\rm se}(\mathbf{s}) * T_{\rm bd}(\mathbf{s}) \ . \tag{1.7}$$

Here $T_{\rm es}(\mathbf{s})$ is the contribution due to the energy spread in the incident beam, $T_{\rm aw}(\mathbf{s})$ is due to the finite aperature width of the detector, $T_{\rm sc}(\mathbf{s})$ is due to the finite source extent, and $T_{\rm bd}(\mathbf{s})$ is due to the finite beam diameter. Each of these contributions to the instrument response function may be determined from measurable properties of a diffracted electron beam for a particular LEED system [6].

1.3.2 Preparation of Samples

Materials to be studied in a LEED experiment must first be prepared to ensure that a well-ordered, cleaned crystal face is available [6]. A large number of procedures for preparing a sample have been developed over the years, and each material has its own procedure for effective preparation. Typically one begins by cleaving or cutting the sample along the desired crystallographic direction. One then begins cleaning the surface in vacuum by any of several methods. The sample may, for example, be heated to near its melting point for several hours (a process called *annealing*) to help desorb contaminants from the sample into the surrounding vacuum. One may also perform chemical cleaning of the sample, or bombard the sample with an ion beam in an attempt to knock contaminants from the surface (called *sputtering*). Cleaning by sputtering will generally be followed by additional annealing in order to repair the damage done by ion beam to the surface crystal structure. Analysis of the gases from the vacuum chamber gives an indication of the effectiveness of the cleaning process.

Chapter 2

LEED Dynamical Calculations and Structure Analysis

"Tum consummatum habet plenumque bonum sortis humanae cum calcato omni malo petit altum et in interiorem naturae sinum venit."

— Seneca, *Naturales quaestiones, I, praef., 7*

THE calculation of predicted LEED diffraction spot intensity vs. electron energy (I-V) curves is a subject of great interest in surface physics. An I-V curve may be thought of as a sort of "fingerprint" of the surface being studied; changes in the surface parameters (e.g. compositions and interlayer spacings) will produce changes in the I-V curves, so that a careful analysis of the I-V curves can yield information about the surface structure.

The calculation of theoretical I-V curves is complicated by the presence of *multiple scattering* of electrons by the surface atoms. This multiple scattering may be attributed to the large scattering cross-section of low-energy electrons by the atoms. Despite this complication, methods have been developed to allow for this multiple scattering and to

calculate LEED I-V curves for a given set of surface parameters with reasonable accuracy [6, 11]. Because of the complexity of the calculations, however, it has not been possible to directly "invert" the LEED I-V calculations; that is, to solve for the surface parameters for a given I-V curve. Instead, one is forced to calculate theoretical I-V curves for a variety of plausible surface structures, then use some means to decide which calculated structure best fits the given curve.

This chapter will briefly review the background behind the LEED theoretical calculations, including the corrections for multiple scattering of electrons. This will be followed by a description of several of the methods which have been devised to find the best fit within a set of calculated I-V curves to a given (experimental) curve. Chapter 3 will introduce a new method for performing this search that will be the focus of this Dissertation: the use of artificial neural networks.

2.1 LEED Dynamical Calculations

As mentioned in Chapter 1, the most common data set collected in a LEED experiment is a plot of diffraction spot intensity vs. incident electron energy, or I-V curve. The calculation of a predicted I-V curve from theory is relatively complicated compared to an analogous calculation in x-ray diffraction due to the presence of strong *multiple scattering* of electrons by the surface atoms. This multiple scattering must be allowed for in any LEED calculation, because it introduces features into the I-V curves comparable in magnitude to the features that would be found in a purely kinematic calculation (one not including multiple scattering). The calculation of a LEED I-V curve that includes multiple scattering is termed a *dynamical* calculation, the fundamentals of which will be

reviewed in this section.

Intra-atomic scattering is internal to the atoms of the crystal surface. It may be understood as an acceleration of the electron while it is in the vicinity of the atom. This causes the electron to emerge on the other side of the atom with a phase difference in its wave function relative to what the phase would have been in the absence of the atom. *Interatomic scattering* is multiple scattering that takes place between atoms of the crystal, both among atoms within the same atomic layer and between atomic layers. In addition to these two types of multiple scattering, there is a potential energy step at the crystal surface due to its *inner potential* which must be considered during the calculations. Each of these effects will be described below.

The presence of multiple scattering introduces several features into the dynamically calculated I-V curves that differentiates them from their kinematically calculated counterparts. Intra-atomic scattering causes peaks in the I-V curve to shift positions relative to their kinematically expected positions, largely toward lower energies. Interatomic multiple scattering also shifts these peak positions, and also introduces additional peaks into the I-V curve, beyond those that would be expected in a kinematic model. These extra peaks are a consequence of the additional scattering paths introduced by multiple scattering; each peak corresponds to a chain of scatterings that satisfies the Bragg condition [6].

2.1.1 The Inner Potential

The potential within a crystal is usually described by a so-called *muffin-tin* model, as shown schematically in Fig. 2.1. (Due to the difficulty of drawing four-dimensional diagrams, the figure shows a version of this potential with just two spatial dimensions;

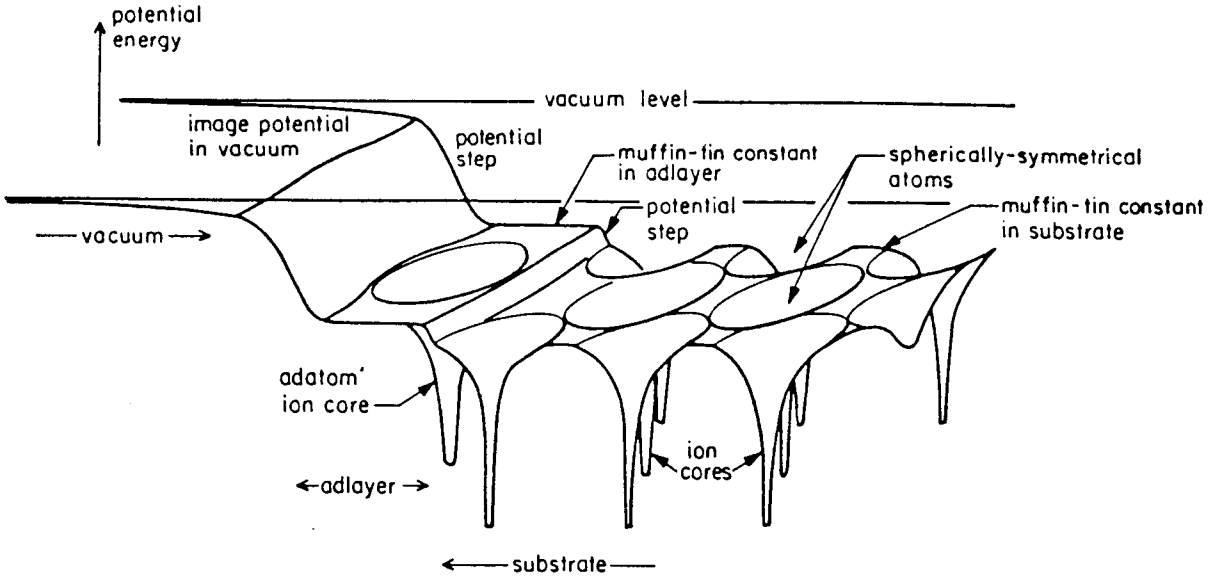

Figure 2.1: Muffin-tin potential, for two spatial two dimensions.
(From Van Hove, Weinberg, and Chan, 1986 [6].)

the figure's third dimension shows the potential energy.) Each atom in the crystal is envisaged as being surrounded by a spherically symmetrical potential out to a radius R_{MT} (the *muffin-tin radius*) from the center of the atom, beyond which the potential is assumed to be a constant (called the *muffin-tin constant*). The *inner potential* of the crystal, which is the kinetic energy that an electron gains upon entering the crystal surface, includes both the muffin-tin constant and a contribution due to the dipole layer at the surface barrier [6].

The presence of this inner potential step at the crystal surface has several effects. First, it will cause a rigid shift of the I-V curve along the energy axis. This is because an I-V curve is plotted against the incident electron energy; if the incident electrons gain kinetic energy at the surface from the inner potential, each feature in the I-V curve will

appear at a lower incident energy than it would have in the absence of the inner potential. Second, the incident electron waves are *refracted* at the surface. The component of the electron momentum parallel to the surface is conserved across the surface boundary, while the component of the momentum perpendicular to the surface is not. Third, there will be a *reflection* of the electron wave at the surface. In most LEED calculations, the incident electrons energies are tens or hundreds of electron volts, while the inner potential step is just a few electron volts; consequently, this reflection effect is small and is often neglected. Fourth, if the inner potential step has structure, there will be a *diffraction* of the electron wave by the inner potential. Experience has shown that this inner potential diffraction can be safely ignored in most cases [6].

2.1.2 Inelastic Processes

As described in Chapter 1, the LEED experimental apparatus contains a set of suppressor grids which allow only elastically scattered electrons to reach the screen. In performing a dynamical LEED calculation of an I-V curve, it is therefore important to identify the various electron scattering mechanisms so that inelastic processes are excluded from the calculation of the elastic intensity.

The dominant inelastic scattering processes in LEED are through excitation of bulk and surface plasmons (charge density waves in the electron gas) and single-electron excitations [6, 11]. Excitation of phonons (elastic waves in the crystal lattice) is a borderline case; while this is an inelastic process, only a small amount of energy and momentum is transfered between electrons and the crystal lattice through phonons and may therefore be termed "quasi-elastic." Phonon lossses are typically on the order of meV and are therefore

collected with the elastic electrons unless one is using a very high-resolution detector.

Inelastic processes are all accounted for in the LEED dynamical calculations through the introduction of a mean free path for the incident electrons. This mean free path may be modeled by simply adding an imaginary component to the potential energy inside the crystal. This may be seen by writing the electron wave function ψ inside the crystal as

$$\psi \sim e^{ik_r x} e^{-k_i x} \quad , \tag{2.1}$$

where the first factor represents the traveling wave, and the second factor models a damping envelope due to inelastic scattering. Here k_r is the wave number of the electron, and k_i is a damping constant related to the electron mean free path λ_e by

$$\lambda_e = \frac{1}{k_i} \quad . \tag{2.2}$$

Eq. (2.1) may be written

$$\begin{aligned} \psi &\sim e^{i(k_r + ik_i)x} \\ &= e^{ikx} \quad , \end{aligned} \tag{2.3}$$

where k is complex:

$$k = k_r + ik_i \quad . \tag{2.4}$$

Since the electron energy within the crystal may be written as

$$E + V_0 = \frac{\hbar^2 k^2}{2m_e} \tag{2.5}$$

and k is complex, this implies that V_0 is also complex:

$$V_0 = V_{0r} + i\, V_{0i} \quad . \tag{2.6}$$

Often the imaginary component of V_0 is taken to be a fixed value of -4 eV. Using Eqs. (2.4) and (2.6) to equate real and imaginary parts of Eq. (2.5) shows that the imaginary part of the potential V_0 may be related to the reciprocal mean free path k_i by

$$V_{0i} = \frac{\hbar^2}{m_e} k_r k_i \ . \tag{2.7}$$

The real part V_{0r} of the potential is just the inner potential of the crystal described earlier.

2.1.3 Intra-atomic Scattering

Just as the variety of inelastic electron scattering processes is modeled by a single number (the mean free path or imaginary component of the inner potential), so the complicated intra-atomic scattering of an electron within an atom is modeled by a set of quantities called the *phase shifts*. Physically, the phase shift represents the difference in phase of the angular momentum components of the electron wave function due to the presence of the scattering atom.

The phase shifts are determined by solving the Schrödinger equation for the system of electrons interacting with the atom. The solution to the Schrödinger equation is the product of the spherical harmonic $Y_{lm}(\theta, \varphi)$ and a radial function $R_l(r)$, where $R_l(r)$ is the solution to the radial differential equation [6]

$$-\frac{\hbar^2}{2m_e}\left(\frac{1}{r^2}\right)\frac{d}{dr}\left[r^2\frac{dR_l(r)}{dr}\right] + \frac{\hbar^2 l(l+1)}{2m_e r^2}R_l(r)$$

$$+\left[-\frac{Ze^2}{r} + V_{sc}(r) + V_{ex}(r)\right]R_l(r) = E\,R_l(r) \ . \tag{2.8}$$

Here l is the angular momentum quantum number, Z is the atomic number of the atoms

in the crystal, $V_{sc}(r)$ is the screening potential, $V_{ex}(r)$ is the exchange potential, and E is the kinetic energy of the incident electrons.

For the case of a constant potential, Eq. (2.8) has solution

$$j_l(kr) = \tfrac{1}{2}\left[h_l^{(1)}(kr) + h_l^{(2)}(kr)\right] \quad , \tag{2.9}$$

where j_l is the spherical Bessel function, and $h_l^{(1)}$ and $h_l^{(2)}$ are the spherical Hankel functions of the first and second kind, respectively [12]. The presence of the atomic potential inside the muffin-tin radius simply introduces a phase shift δ_l relative to the constant-potential case, so that the solution to Eq. (2.8) then becomes

$$\tfrac{1}{2}\left[\exp(2i\delta_l)h_l^{(1)}(kr) + h_l^{(2)}(kr)\right] \quad . \tag{2.10}$$

The phase shifts are found by solving the radial differential equation (2.8) for the muffin-tin potential. This is done by numerically integrating Eq. (2.8) from 0 to the muffin-tin radius R_{MT} to find the solution inside R_{MT}. Eq. (2.10) is used as the solution outside R_{MT}, and logarithmic derivatives of the solutions in the two regions are equated at the boundary R_{MT}. The result is [6]

$$\exp(2i\delta_l) = \frac{L_l h_l^{(2)}(kR_{MT}) - h_l^{(2)'}(kR_{MT})}{h_l^{(1)'}(kR_{MT}) - L_l h_l^{(1)}(kR_{MT})} \quad , \tag{2.11}$$

where

$$L_l = \frac{R_l'(R_{MT})}{R_l(R_{MT})} \quad . \tag{2.12}$$

The phase shifts δ_l are dependent on the angular momentum of the electron wave (through the quantum number l) and on the electron energy. In calculating LEED I-V spectra, one specifies a set of phase shifts for $l = 0, 1, 2, \ldots, l_{max}$ for various electron energies, then interpolates between those energies to find the phase shift for any desired energy and angular momentum.

2.1.4 Interatomic Scattering

To calculate the effect of the multiple scattering that takes place between the atoms of a crystal, we begin by considering the scattering between two atoms. We will generalize this result to calculate the scattering among atoms in a plane, and finally the scattering between planes of atoms. The discussion here follows that given in Reference [6].

To begin, consider two atoms, labeled 1 and 2, at positions $\vec{r}_1$ and $\vec{r}_2$ and with atom 1 having angular momentum L_1. Assume that atom 1 emits a spherical electron wave of angular momentum L' (described by quantum numbers $|l', m'\rangle$). Assume also that this electron wave arrives at atom 2 as a spherical wave of angular momentum L (described by quantum numbers $|l, m\rangle$). Then the propagation of the electron wave from atom 1 to atom 2 is given by the Green's function

$$\begin{aligned} \overline{G}^{21}_{LL'} &= -4\pi i \frac{2m_e}{\hbar^2} k \sum_{L_1} i^{l_1} a\left(L, L', L_1\right) h^{(1)}_{l_1}\left(k\left|\vec{r}_2 - \vec{r}_1\right|\right) \\ &\quad \times Y_{L_1}\left(\vec{r}_2 - \vec{r}_1\right) \exp\left[-i\,\vec{k}_{\mathrm{in}} \cdot \left(\vec{r}_2 - \vec{r}_1\right)\right] \quad , \end{aligned} \tag{2.13}$$

which describes the amplitude of the wave arriving at atom 2. Here k is the wave number for the beam being calculated, $\vec{k}_{\mathrm{in}}$ is the wave vector of the incident plane wave, and the coefficients $a\left(L, L', L_1\right)$ are defined by

$$a\left(L, L', L_1\right) = \int_{4\pi\ \mathrm{sr}} Y^*_L(\Omega)\, Y_{L'}(\Omega)\, Y^*_{L_1}(\Omega)\, d\Omega \quad . \tag{2.14}$$

The summation in Eq. (2.13) is over all values of l_1 and m_1 such that $|l - l'| \leq l_1 \leq l + l'$ and $m_1 = m + m'$. The wave number k is complex; its imaginary component models the finite electron mean free path due to inelastic scattering, as described earlier.

While Eq. (2.13) describes the propagation of an electron wave from atom 1 to atom 2, the scattering of an electron is described in terms of the phase shifts δ_l by the t-matrix

element

$$t_l = -\frac{\hbar^2}{2m_e}\left(\frac{1}{k}\right)\sin\delta_l\,\exp(i\delta_l)\ . \tag{2.15}$$

Using Eq. (2.15) to describe the scattering and Eq. (2.13) to describe the propagation of the wave, we can form various products of t_l and $\overline{G}^{21}_{LL'}$ to describe the different combinations of multiple scattering. In particular, all scattering paths that terminate at atom 1 are described by the summation (dropping angular momentum subscripts)

$$T^1 = t^1 + t^1\overline{G}^{12}t^2 + t^1\overline{G}^{12}t^2\overline{G}^{21}t^1 + t^1\overline{G}^{12}t^2\overline{G}^{21}t^1\overline{G}^{12}t^2 + \cdots\ , \tag{2.16}$$

while those that terminate on atom 2 are described by

$$T^2 = t^2 + t^2\overline{G}^{21}t^1 + t^2\overline{G}^{21}t^1\overline{G}^{12}t^2 + t^2\overline{G}^{21}t^1\overline{G}^{12}t^2\overline{G}^{21}t^1 + \cdots\ . \tag{2.17}$$

It has been shown [13] that T^1 and T^2 are simply related by

$$T^1 = t^1 + t^1\overline{G}^{12}T^2 \tag{2.18}$$

$$T^2 = t^2 + t^2\overline{G}^{21}T^1\ , \tag{2.19}$$

or in matrix notation,

$$\begin{bmatrix} T^1 \\ T^2 \end{bmatrix} = \begin{bmatrix} I & -t^1\overline{G}^{12} \\ -t^2\overline{G}^{21} & I \end{bmatrix}^{-1} \begin{bmatrix} t^1 \\ t^2 \end{bmatrix}\ . \tag{2.20}$$

This result may be generalized to describe the scattering among N atoms as

$$\begin{bmatrix} T^1 \\ T^2 \\ T^3 \\ \vdots \\ T^N \end{bmatrix} = \begin{bmatrix} I & -t^1\overline{G}^{12} & -t^1\overline{G}^{13} & \cdots & -t^1\overline{G}^{1N} \\ -t^2\overline{G}^{21} & I & -t^2\overline{G}^{23} & \cdots & -t^2\overline{G}^{2N} \\ -t^3\overline{G}^{31} & -t^2\overline{G}^{32} & I & \cdots & -t^3\overline{G}^{3N} \\ \vdots & \vdots & \vdots & & \vdots \\ -t^N\overline{G}^{N1} & -t^N\overline{G}^{N2} & -t^N\overline{G}^{N3} & \cdots & I \end{bmatrix}^{-1} \begin{bmatrix} t^1 \\ t^2 \\ t^3 \\ \vdots \\ t^N \end{bmatrix} . \tag{2.21}$$

If we assume that the original incident electron wave is a plane wave $\exp(i\,\vec{k}_{\rm in}\cdot\vec{r})$ and that the scattered electron wave is a plane wave $\exp(i\,\vec{k}_{\rm out}\cdot\vec{r})$, then the scattering amplitude from N atoms is given by

$$T_{LL'} = \sum_{j=1}^{N} \exp\left[i\,\left(\vec{k}_{\rm in} - \vec{k}_{\rm out}\right)\cdot\vec{r}_j\right] T^j_{LL'} \ . \tag{2.22}$$

For a single plane layer of identical atoms arranged in a periodic array, the symmetry of the layer implies that

$$T^1 = T^2 = T^3 = \cdots \equiv \tau \ . \tag{2.23}$$

The scattering due to N such planes of atoms may then be described by

$$\begin{bmatrix} \mathsf{T}^1 \\ \mathsf{T}^2 \\ \mathsf{T}^3 \\ \vdots \\ \mathsf{T}^N \end{bmatrix} = \begin{bmatrix} I & -\tau^1 G^{12} & -\tau^1 G^{13} & \cdots & -\tau^1 G^{1N} \\ -\tau^2 G^{21} & I & -\tau^2 G^{23} & \cdots & -\tau^2 G^{2N} \\ -\tau^3 G^{31} & -\tau^2 G^{32} & I & \cdots & -\tau^3 G^{3N} \\ \vdots & \vdots & \vdots & & \vdots \\ -\tau^N G^{N1} & -\tau^N G^{N2} & -\tau^N G^{N3} & \cdots & I \end{bmatrix}^{-1} \begin{bmatrix} \tau^1 \\ \tau^2 \\ \tau^3 \\ \vdots \\ \tau^N \end{bmatrix} , \tag{2.24}$$

following a development similar to that for N atoms described earlier. In this case the

Green's functions are given by

$$G_{LL'}^{ji} = -4\pi i \frac{2m_e}{\hbar^2} k \sum_{L_1} \sum_{\vec{P}}{}' i^{l_1} a(L, L', L_1)\, h_{l_1}^{(1)}\left(k\left|\vec{r}_j - \vec{r}_i + \vec{P}\right|\right) \times Y_{L_1}\left(\vec{r}_j - \vec{r}_i + \vec{P}\right) \exp\left[-i\,\vec{k}_{\rm in}\cdot\left(\vec{r}_j - \vec{r}_i + \vec{P}\right)\right] \quad , \tag{2.25}$$

where $\vec{P}$ extends over all the lattice points in any of the planes, except for $\vec{r}_j - \vec{r}_i + \vec{P} = \vec{0}$. Finally, the amplitudes of the diffracted plane waves is given by

$$M_{\rm out,in} = -\frac{16\pi^2 i m_e}{A k_{{\rm out},z} \hbar^2} \sum_{LL'} Y_L\left(\vec{k}_{\rm out}\right)\, \mathsf{T}_{LL'}\, Y_{L'}^*\left(\vec{k}_{\rm in}\right) \quad , \tag{2.26}$$

where A is the area of a two-dimensional unit cell in one of the atomic planes. The square of this amplitude gives the I-V curve intensity.

Numerous schemes have been devised to perform this calculation efficiently. For example, in the *layer doubling* method, one calculates reflection and transmission matrices for a single pair of atomic layers, then iteratively applies the method to yield matrices for 4, 8, 16, ... atomic layers. This provides an efficient means for calculating the reflection and transmission matrices for electron penetration into the bulk material, to which matrices representing the surface layers may then be applied [6].

The Renormalized Forward Scattering [6] procedure is another (somewhat more complex) such procedure, and was used to calculate the LEED I-V curves for this work.

2.2 LEED Structure Analysis

The dynamical calculations just described are too complex to allow a direct inversion, in which one could solve for the surface parameters given the I-V curve. Instead, one

calculates theoretical I-V curves for a variety of plausible surface structures, and determines which of these theoretical curves best matches the experimental curve. While one could attempt to visually determine which I-V spectra best match the experimental data, a better (and more objective) approach is to calculate a number for each of the candidate structures that indicates how closely that structure matches the experimental data, then use some method to search for the candidate structure that optimizes that fit. This section will review several methods that have been used to carry out this search for the best fit to given LEED I-V spectra.

2.2.1 Reliability Factors

We begin with a description of *reliability factors,* or *R-factors* used in LEED stuctural analysis. A reliability factor is a number which gives an indication of how closely a calculated I-V curve matches an experimental curve. By calculating an R-factor between the experimental I-V data and each of the candidate structures, one can determine the best fit to the experimental data by finding the calculated structure with the minimum R-factor.

One could, for example, calculate the root-mean-square error between the experimental data and each calculated I-V curve, and use the result as an R-factor. However, some consideration must be given to the physics of the problem, rather than searching only for the mathematical best fit between spectra. R-factors are generally designed to emphasize features of the I-V curves that are sensitive to surface structural parameters, while de-emphasizing features that are sensitive to non-structural surface properties. Typically an R-factor will emphasize such features in the I-V spectra as peak positions, relative

peak heights, peak skewness, and peak widths, although there is little agreement on the relative importance of each of these features in determining surface geometry [6]. For that reason, a variety of different R-factors have been proposed, each of which is designed to emphasize different features or to be computationally advantageous in some way.

One commonly used such R-factor is the *Pendry R-factor* [14]. It is designed to emphasize peak positions, giving equal weight to all peaks regardless of height. It also avoids calculating the second derivatives found in some other R-factors, which can lead to computational difficulties. To begin, Pendry defines the function $Y(E)$ by

$$Y(E) = \frac{L^{-1}}{L^{-2} + V_{0i}^2} \ , \tag{2.27}$$

where $L(E)$ is the logarithmic derivative of the I-V spectrum,

$$L(E) = I'/I \ , \tag{2.28}$$

I is the intensity, and V_{0i} is the electron self-energy, which is around -4 eV for most materials at electron energies above about 30 eV. The logarithmic derivative L tends to treat peaks equally, while the form of the Y-function avoids singularities when $I = 0$. Using this Y-function, the Pendry R-factor RPE is defined as

$$RPE = \frac{\sum_{\vec{g}} \int \left(Y_{\vec{g}_{\rm th}} - Y_{\vec{g}_{\rm expt}} \right)^2 dE}{\sum_{\vec{g}} \int \left(Y_{\vec{g}_{\rm th}}^2 + Y_{\vec{g}_{\rm expt}}^2 \right) dE} \ . \tag{2.29}$$

Here $Y_{\vec{g}_{\rm th}}$ is the Y-function of the theoretical (calculated) I-V curve, and $Y_{\vec{g}_{\rm expt}}$ is the Y-function of the experimental I-V curve. The sums are taken over all diffracted beams $\vec{g}$. A value of 0 for RPE indicates that the theoretical and experimental curves are identical; 1 indicates that they are uncorrelated; and 2 indicates anticorrelation.

The significance of a minimum in the Pendry R-factor may be described by the *Pendry RR-factor*,

$$RR = \frac{\sigma(R)}{\overline{R}} \quad . \tag{2.30}$$

Here $\sigma(R)$ is the standard deviation of the Pendry R-factor and $\overline{R}$ is the mean R-factor. From statistical theory,

$$\frac{\sigma(R)}{\overline{R}} = \sqrt{\frac{2}{N}} \quad , \tag{2.31}$$

where N is the number of well-separated peaks in the I-V curve. Using this result and the fact that the I-V curve peak widths are $2|V_{0i}|$ gives the RR factor as

$$RR \simeq \left(\frac{8V_{0i}}{\delta E}\right)^{1/2} \quad , \tag{2.32}$$

where δE gives the the total energy range for the I-V curve. This result allows one to estimate the 1-σ error in the R-factor due to random errors in theory and experiment.

2.2.2 Exhaustive Global Search

The simplest and most reliable (albeit slowest) search method for LEED surface structure determination is to use an *exhaustive global search*. With this method, one calculates I-V curves for a large number of possible candidate structures, calculates an R-factor for each one, then simply selects the structure with the smallest value of the R-factor. An RR-factor may also be calculated to give an indication of the significance of the minimum found.

Provided the search is performed over a sufficiently fine grid and wide range in parameter space, the exhaustive global search is guaranteed to find the structure which

globally minimizes the R-factor. One then hopes that the structure with the minimum R-factor is the one with the correct geometry.

This is not, however, necessarily the case, since I-V curves do not necessarily map one-to-one with surface structure parameters. In other words, there may be more than one surface structure that may lead to I-V curves that are indistinguishable to within calculational and experimental uncertainties. One known difficulty is that LEED I-V data is relatively insensitive to registries (layer displacements parallel to the surface), so that different registries can produce similar I-V data [6]. Another difficulty is the relative insensitivity of LEED data to half-wavelength changes in the layer spacings, since this will result in phase changes of 2π in the diffracted electron wave [6]. This has led, for example, to uncertainties in determining the interlayer spacing for the Ni(100)-c(2×2)-O system [15, 16, 17, 18].

While the exhaustive global search method is guaranteed to find the global (rather than a local) minimum in the error surface, it can also require substantial amounts of computer time; the computer time required scales exponentially with the number N of search parameters being sought. Because of this, much effort has been expended to develop faster search algorithms. Several of these are described below.

2.2.3 Steepest Descent Method

It is a well-known result from elementary calculus [19] that the directional derivative of a function of several variables $f(\vec{x})$ has its maximum in the direction of the gradient, $\nabla f(\vec{x})$. A numerical method which takes advantage of this fact, called the *steepest descent method*, [20, 21] may be used to search for a minimum in f. To implement a

steepest-descent algorithm, we begin by making an initial guess $\vec{x}^{(0)}$ for the values of the independent variables $\vec{x}$ that will minimize the function to be optimized, $g(\vec{x})$, where $\vec{x} = (x_1, x_2, \ldots, x_n)^T$. The next estimate for $\vec{x}$ will be given by

$$\vec{x}^{(1)} = \vec{x}^{(0)} - \alpha \nabla g \vec{x}^{(0)} \quad , \tag{2.33}$$

for some constant $\alpha > 0$. We wish to choose α so that $g(\vec{x}^{(1)})$ will be significantly less that the previous estimate, $g(\vec{x}^{(0)})$. The appropriate choice is the value of α for which

$$h(\alpha) = g(\vec{x}^{(0)} - \alpha \nabla g(\vec{x}^{(0)})) \tag{2.34}$$

is minimum. In order to find this α quickly without having to perform a time-consuming iterative root-finding method, one chooses three values α_1, α_2, and α_3 (hopefully near the optimum α), interpolates with a quadratic polynomial P, then uses the minimum of P to estimate the optimum α. The entire procedure is then repeated until the minimum g is found to within the desired tolerance.

The gradient descent method has an advantage over the exhaustive search method in that it scales quadratically ($\sim N^2$) with the number N of parameters being sought [22], whereas the exhaustive search method scales exponentially with N. Its primary disadvantages are that is is prone to finding local (rather than global) minima in the error surface being searched, and that convergence can be slow under some circumstances. The steepest descent method is often used for a refined search once an approximate solution has been found by some other method.

2.2.4 Simulated Annealing

Many numerical algorithms have been developed which attempt to find the global minimum error for a problem more efficiently than the exhaustive global search, while also avoiding convergence to local minima. One such algorithm is *simulated annealing*, in which a set of search parameters is treated much like a set of atoms in a crystalline metal. An error function is to be minimized, and is assigned a role analogous to energy in a physical system. One applies random perturbations (analogous to heat) to the parameters being searched and preferentially allows the parameter set to change whenever the energy function is lowered. Throughout the procedure, one slowly lowers a parameter analogous to the temperature to that the algorithm converges to a minimum of the error function.

To implement a simulated annealing algorithm, one begins be selecting a set of parameters for which an error function E (analogous to energy) is to be minimized. One also must select an initial value for the "temperature" parameter, T_0 and an initial guess for the search parameter values, for which the initial "energy" E_0 can be calculated. To begin the search, one perturbs each of the N search parameters by some random amount δ_n $(n = 1 \ldots N)$. If the energy E_j of these new parameters is less than the energy E_i of the current parameters, then the new search parameter values are accepted as the new current values. If the energy E_j of the new parameters is greater than the energy E_i of the current parameters, then the new parameters are accepted as the current parameters with probability [23]

$$\exp\left(\frac{E_i - E_j}{k_B T}\right) \quad , \tag{2.35}$$

where T is temperature parameter, and k_B is Boltzmann's constant, which serves to set E and T in the same units. This procedure is repeated a set number of times, after which the

temperature T is lowered and the entire process repeated until some convergence criterion is satisfied.

Simulated annealing has been applied to LEED structural analysis for the Ir(110)-p(2×1) surface [22]. In this case, the Pendry R-factor RPE serves the role of the energy parameter E, since the objective is to minimize the R-factor.

2.2.5 Genetic Algorithms

A relatively recent innovation in optimization problems is the use of *genetic algorithms,* which attempt to perform optimization in a way that mimics natural selection in biological organisms. One begins with a set of "chromosomes," representing the independent variable and modeled as strings of bits. One then implements a reproductive cycle (including mutations and natural selection) which produce the next generation of chromosomes. Natural selection pressures are modeled using a "fitness function," which the chromosomes will tend to maximize through the course of evolution. One can thus maximize any desired function by using it as a fitness function.

Implementation of a genetic algorithm may be most clearly illustrated with a simple example [24]. Suppose we wish to maximize the function $y = x^2$ over the range $x \in [0, 31]$, so that x may be represented as a five-bit integer. We begin by selecting several random values of x by selecting random five-bit integers (Table 2.1); this will comprise the initial population of *chromosomes.* For the purposes of illustration, the table shows a population of four chromosomes, although a practical problem would typically employ a larger population. The genetic algorithm will attempt to preferentially select those chromosomes in the population that best maximize the *fitness function* $y = x^2$.

Initial population, x			Next generation
binary	decimal	$f(x) = x^2$	counts, $f(x) \,/\, \overline{f}$
10000	16	256	3
00001	1	1	0
01001	9	81	1
00001	1	1	0

Table 2.1: Genetic algorithm results for maximizing $y = f(x)$ (initial population). The first two columns show the initial chromosome population, chosen at random. The third column gives the fitness function for each chromosome, and the last column gives the number of copies of each initial chromosome that will be used in the mating pool. The average value of $f(x)$ is $\overline{f} = 84.75$.

As shown in Table 2.1, we begin by evaluating the fitness function f for each chromosome value x in the population; larger values of f indicate chromosomes that better maximize the fitness function.

The next stage of the algorithm is to perform *reproduction* of the chromosomes in the initial population. This is done by preferentially choosing those members of the initial population with the highest values of the fitness function to make up a *mating pool* from which the next generation of chromosomes will be formed. In particular, the mating pool consists of $f(x) \,/\, \overline{f}$ copies of chromosome x, as shown in the table. From this mating pool, chromosomes are randomly arranged in pairs which will serve as the "parents" for the following generation.

The reproduction stage is followed by the *crossover* stage, which mimics genetic crossovers [25]. For each pair of chromosomes in the mating pool, a *crossover point* is chosen at random within the string of bits. Offspring are formed for the next generation

Mating pool		Crossover		New Population		New
decimal	binary	Point	Mate	Binary	Decimal	$f(x)$
16	1|0000	1	3	10000	16	256
16	100|00	3	4	10001	17	289
16	1|0000	1	1	10000	16	256
9	010|01	3	2	01000	8	64

Table 2.2: Genetic algorithm results for maximizing $y = f(x)$ (second generation). The first two columns show the chromosomes in the mating pool, where the vertical bar shows the crossover point, selected at random (column 3). The chromosomes are paired at random (column 4), and the resulting offspring, including crossovers, are shown in columns 5 and 6. The final column gives the fitness function for the new (second generation) population of chromosomes.

by copying each of the parents, while exchanging their bit patterns beyond the crossover point. In general, a crossover will occur with probability p_x; for this example, we assume a crossover probability $p_x = 1$ so that a crossover always takes place. The offspring resulting from this process for this example are shown in Table 2.2.

The final stage in a genetic algorithm is to model a *mutation*, in which bits in the chromosome string are randomly changed between 0 and 1 with probability p_m. This mutation helps keep the algorithm from converging to a local maximum in the fitness function so that it may more readily find the global maximum. For this example, p_m was chosen to be 0, so that mutations are not modeled. This set of offspring chromosomes is then used as the initial population for another iteration through the algorithm. As seen in Table 2.2, the second-generation population of chromosomes has a higher value of f than the initial population (Table 2.1), so that these chromosomes indicate values of x which

better maximize f.

A number of variations of the basic algorithm just described may be employed. One may, for example, allow multiple crossover points in the chromosomes, or employ *elitism*, in which the best one (or more) chromosomes in a population are guaranteed survival to the next generation, while the crossovers and mutations are applied to the remaining population.

Genetic algorithms have recently been used [26] for LEED surface structure determination of Ir(110)-(1×2) missing row structure, involving three independent search parameters. In this case, each chromosome models a single candidate structure in the space of surface structure parameters to be searched. Since a genetic algorithm will tend to maximize the fitness function, but we wish to *minimize* the R-factor, the fitness function was taken in this case to be $2 - RPE$, where RPE is the Pendry R-factor described above. This model employed a population size of 50, with each chromosome containing 7 bits. Elitism was used to increase the algorithm's convergence speed, in which the best five chromosomes in each population are guaranteed survival to the following generation. In this study, a genetic algorithm was used to find the approximate location of the global minimum R-factor, with a conventional steepest descent used to find the minimum more accurately.

Chapter 3

Artificial Neural Networks

"Mihi contuenti semper suasit rerum natura nihil incredibile existimare de ea."

— Pliny the Elder, *Historia naturalis, XI, 2, 6*

3.1 Introduction

THE development of the electronic digital computer has been one of the landmark achievements of twentieth century technology. It provides the physicist with a powerful tool for solving problems whose sheer size would make them difficult or impossible to solve otherwise.

A digital computer may be used to solve problems in several different ways. If the solution to a problem may be determined analytically, a computer may be programmed with instructions to carry out the numerical computations given by the solution. In many cases, however, an analytic solution to a problem cannot be found. In this case, a computer

may be used to implement a search algorithm or other iterative technique to converge upon a solution. This requires that a method for finding the solution to the problem be described to the computer in the form of computer codes, which the computer can then execute at high speeds. This is the case, for example, for the inverse LEED I-V problem described in the previous chapter.

Recent years have seen the development of innovative new approaches to the solution of such problems. In *artifical intelligence,* for example, one designs computer codes which, in some sense, are able to teach themselves how to solve certain classes of problems. One of the most interesting of these artificial intelligence algorithms is the use of *artificial neural networks,* in which a computer code is designed to mimic the operation of a biological neural network such as the human brain. This chapter will give an overview of artificial neural networks, with particular emphasis on the backpropagation networks that are of interest in solving the inverse LEED I-V problem.

3.2 Biological Neural Networks

In beginning a study of artificial neural network algorithms, it is instructive to first review the biological neural networks after which they are patterned. This section will review the molecular biology of biological neural networks as they occur, for example, in the human central, peripheral, and autonomic nervous systems. An understanding of the operation of biological neural networks will help to clarify the motivations behind several aspects of the design of artificial neural networks, which will be of interest in solving problems in LEED.

A typical biological nerve cell, or *neuron,* is shown schematically in Figs. 3.1 and

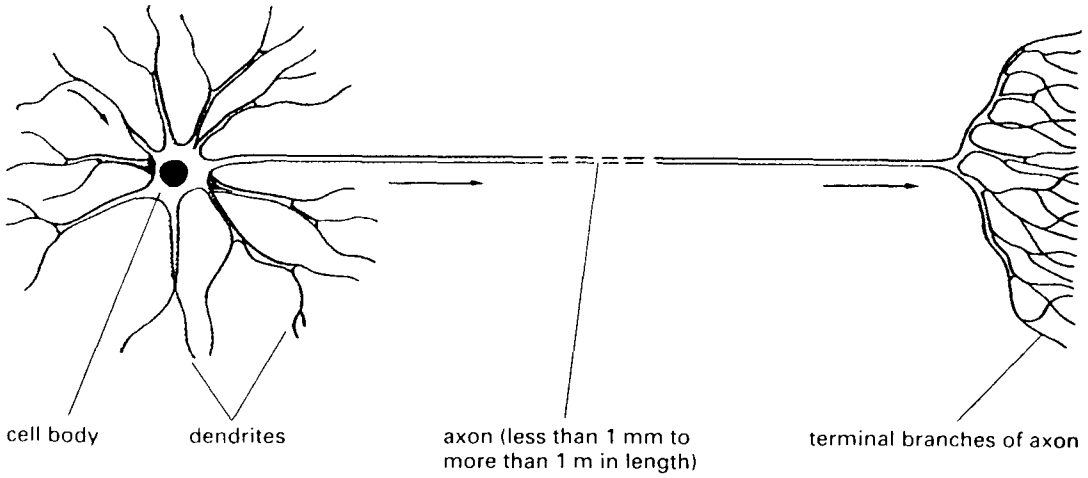

Figure 3.1: Schematic figure of a typical vertebrate neuron.
(After Alberts et al., 1994 [27].)

3.2. Its *cell body* consists of a lipid bilayer *plasma membrane* which encloses the cell nucleus containing most of the cell's genetic information. The remaining cell contents, or *cytoplasm*, consist of a jelly-like *cytosol,* in which is suspended the cell's various organelles. These organelles include the *mitochondria* which produce the adenosine 5′-triphosphate fuel used to power the cell's chemical reactions; the *endoplasmic reticulum,* where protein synthesis takes place; and the *Golgi apparatus,* which receives lipids and proteins from the endoplasmic reticulum and sends them to the parts of the cell where they are needed.

Extending out from the neuron's cell body is one long *axon,* which is the path along which an outgoing nerve impulse is sent. The axon ends in a series of *terminal branches,* each of which may be used to send nerve signals to other neurons. Also attached to the

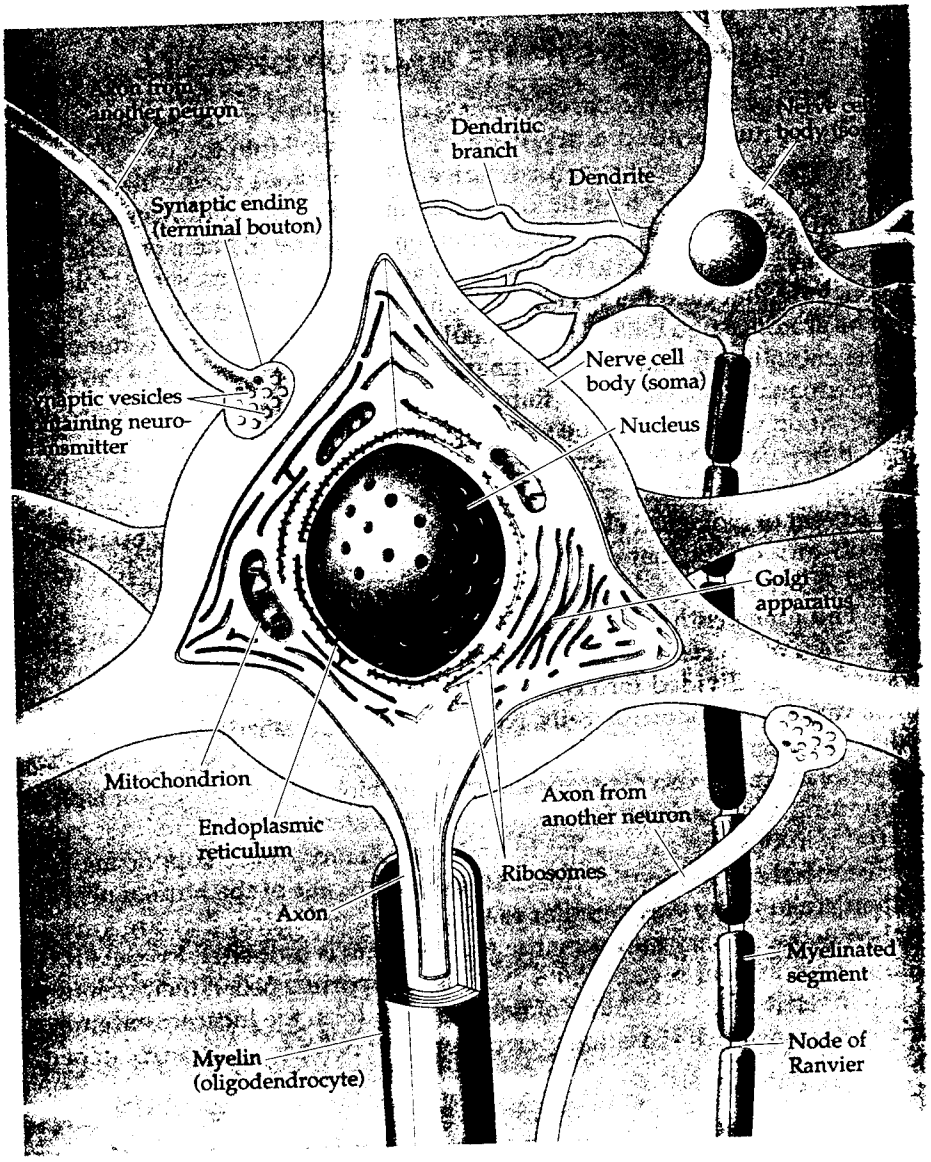

Figure 3.2: Details of nerve cell body.
(After Purves et al., 1997 [28].)

cell body are a set of shorter *dendtrites,* which act as "antennas" for receiving signals from the axons of other neurons. The dendrites essentially increase the surface area of the cell body, allowing the neuron to receive as many as 10^5 inputs from other neurons [27].

The basic mechanism by which nerve impluses are transmitted from neuron to neuron involves *gated ion channels,* which are pores in the plasma membrane through which ions (such as K^+ or Na^+) may diffuse whenever the channel's gate is open. Several types of gated ion channels exist, and are classified according to the type of stimulus which opens them. For our purposes, the most important of these are *voltage-gated ion channels,* in which an electrical potential difference across the plasma membrane causes the gate to open; and *transmitter-gated ion channels,* in which a chemical called a *neurotransmitter* acts as a sort of "key" to open the gate.

A nerve impulse (or *action potential*) travels along the axon by means of voltage-gated ion channels built into the surface of the axon. When a stimulus depolarizes the plasma membrane enough to open a voltage-gated ion channel on the axon, cations are allowed to diffuse into the axon through the channel. This causes further membrane depolarization, allowing other nearby channels to open, causing further depolarization. A separate inactivating mechanism causes the channel to quickly close, while the nearby channels, having opened slightly later, continue the process of opening their nearby channels before closing themselves. The result is a domino-like propagation of opening and closing ion channels along the length of the axon. Because of the self-sustaining nature of the propagation, this signal can travel along the axon without attenuation.

Once the action potential has reached the end of one of the axon's terminal branches (at a junction called the *synapse*), the electrical potential triggers the release of neurotransmitters, which are stored in membrane-bounded *synaptic vesicles* near the end of the

terminal branch. These neurotransmitters pass through the plasma membrane across a gap called the *synaptic cleft* which physically and electrically separates the axon from its neighboring neuron. The neurotransmitters act like "keys" which open transmitter-gated ion channels on the neighboring neuron, allowing cations to enter the cell body of that neuron. Once enough of these transmitter-gated ion channels have opened, the plasma membrane of this neighboring neuron will become sufficiently depolarized to activate an action potential along its own axon, and the whole process begins again in the neighboring neuron.

Several key features in this description of biological neurons are important to note, as they are incorporated into the design of artificial neural networks. First, each neuron may be connected to many other neurons through the terminal branches at the end of the axons. Second, an action potential in a neuron is triggered only after input stimuli from neighboring input neurons have exceeded a certain threshold. Third, once an action potential has been activated, further input from neighboring neurons has no effect. The input has been essentially saturated with inputs from neighboring neurons once the inputs have exceed the level to trigger an action potential.

3.3 Artificial Neural Networks

Artificial neural networks are algorithms that attempt to mimic, in a simplified fashion, the behavior of networks of the biological neurons described above. They are of interest to neurobiologists as a simple model of the human brain; understanding the ways in which neural networks learn to recognize patterns may provide some insight into human learning and behavior. Artificial neural networks are also of more general interest as a problem-

solving tool: since they are particularly adept at pattern recognition, they may be useful in solving certain types of problems which are intractable by traditional methods. It is this latter use of networks that will be of interest here.

Many different types of neural network algorithms have been developed since they were first conceived in the 1940s. In general, an artificial neural network consists of a set of *nodes,* each of which contains a numerical value. The nodes are connected to each other through a series of directed lines (as shown in Fig. 3.4), each of which carries an associated value called a *weight,* . Since the connecting lines are directional, each node will generally have several input connections and several output connections. A node's value will be a function of the values of the nodes which feed in to it, the weights associated with the corresponding input connections, and an *activation function* which limits a nodes value to a restricted range. The nodes are meant to be analogous to the neurons of a biological neural network; the interconnections are analogous to the biological axons and synapses; and the activation function models the saturation of a neuron by signals from input neurons.

Using an artificial neural network typically begins by setting the values of a set of *input nodes* to describe the problem to be solved. One then calculates the values of all the other nodes in the network, and finally reads out the values of a set of *output nodes* which describe the solution to the problem. The network's ability to solve the problem being asked of it is determined by the network's weights, which may be determinined by several different methods. Most often this involves what is called *supervised learning,* in which the network is shown a series of example inputs. The network calculates the output node values for these examples, and some means is used to adjust the network weights for the error between the network's results and the expected results. This process (called

training) is repeated over many iterations until the network weights have converged and the output nodes are able to produce the correct output nodes values for each of the examples on which it is being trained.

Another means of determining the network weights is *unsupervised learning,* in which the network trains itself by setting its own weights without having to be shown examples. In this case, the network will only be able to group similar inputs together and decide which group a given input is most similar to. Network weights may also be fixed; this type of network may be useful in solving constrained optimization problems, where the network weights represent the constraints in the problem

3.3.1 Examples of Artificial Neural Networks

How artificial neural networks are used in practice might best be illustrated by describing a few examples. One typical example is the design of a network to recognize letters of the alphabet [29]. Each letter may be encoded on a grid, as shown in Fig. 3.3; the value of each point in the grid (1 for "occupied" or -1 for "unoccupied") is then fed to the network at its input nodes. There may then be one output node for each possible letter, with an output value of 1 in an output node indicating the presence of the corresponding letter. Using supervised learning, such a network can be trained to recognize the letters on which it was trained—not only in their original form, but also with significant deviations from the original forms. This makes it possible for such a network to recognize human handwriting with remarkable accuracy.

Other uses for neural networks include speech recognition and production, medical diagnosis, and signal processing. Recent attempts have also been made to use neural

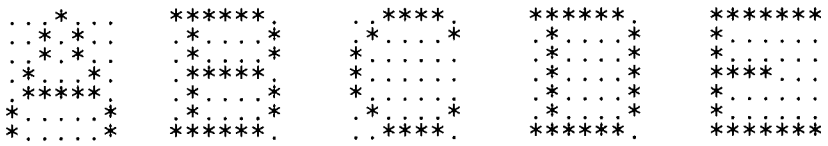

Figure 3.3: Encoding of letters for character recognition.

networks in finance, in an attempt to forecast stock and commodity prices. The success of such financial applications, however, is limited by the nature of the financial markets: if a network is ever developed that can successfully forecast stock or commodity prices, it would quickly become so widely used that it would no longer work. One must be careful, therefore, to ensure that the use of a neural network is appropriate for the problem at hand.

3.4 Backpropagation Networks

One of the most commonly used neural networks, and the type used in this work, is a *backpropagation* network. A backpropagation network consists of a set (or *layer*) of input nodes, one or more layers of *hidden* (or *medial*) nodes, and a layer of output nodes, as shown in Fig. 3.4. Data propagates through the network in one direction: from the input layer to the hidden layer to the output layer, through a series of connections that join each node in a layer to each of the nodes in the layer below it (in an arrangement known in graph theory as a *complete bipartite graph* [30]). It has been shown [29] that a single layer of hidden nodes is sufficient to approximate any continuous function of the

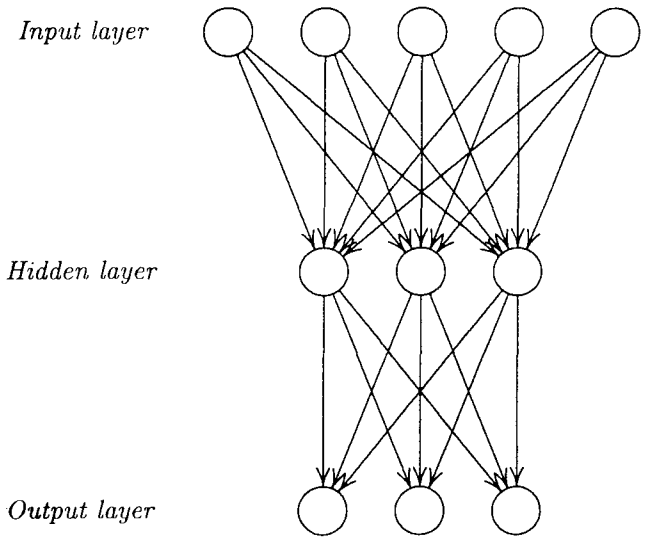

Figure 3.4: An artificial neural network.

input nodes, although in some circumstances the network may train more easily with two hidden layers. The network used for the research described in this work uses a single layer of hidden nodes.

Each connection between two nodes of a backpropagation network has an associated *weight,* which determines the fraction of a node's signal that will be received by a node in the layer below it. The set of all the network's weights determines the values of the output nodes for a given set of values in the input nodes. Specifically, suppose that there are I nodes in the input layer with values x_i, J nodes in the hidden layer with values z_j, and K nodes in the output layer with values y_k. Then the values of the hidden layer nodes in a simple network are found from the input node values from [29]

$$z_j = f\left(v_{0j} + \sum_{i=1}^{I} x_i v_{ij}\right) , \tag{3.1}$$

where v_{ij} is the weight associated with the connection between node i of the input layer and node j of the hidden layer. The weight v_{0j} is called a *bias* weight, and serves to bias the node inputs into the correct range for the activation function $f(x)$ [31]. The function $f(x)$ in Eq. (3.1) (called an *activation function* or *sigmoid function*) limits the value of its argument to a finite range, in analogy with the saturation of a biological synapse. For this work, the activation function has been chosen to be

$$f(x) = \frac{1}{1 + e^{-x}} \ , \tag{3.2}$$

to limit the weights to the range [0,1], as shown in Fig. 3.5.

Having found the hidden node values z_j, the values y_k of the output layer nodes are given by

$$y_k = f\left(w_{0k} + \sum_{j=1}^{J} z_j w_{jk}\right) \ , \tag{3.3}$$

where w_{jk} is the weight associated with the connection between node j of the hidden layer and node k of the output layer, with w_{0k} being the bias weight.

3.5 Scaling

The performance of a backpropagation network may be improved by ensuring that the values of the output nodes fall within the linear range of the activation function, This may be accomplished by *scaling* the output values. If y is a network output node value calculated by Eq. (3.3), then it may be scaled to within the linear range of the activation function by applying [32]

$$y_{\text{scaled}} = \frac{y - y_{\min}}{y_{\max} - y_{\min}}(s_{\max} - s_{\min}) + s_{\min} \ , \tag{3.4}$$

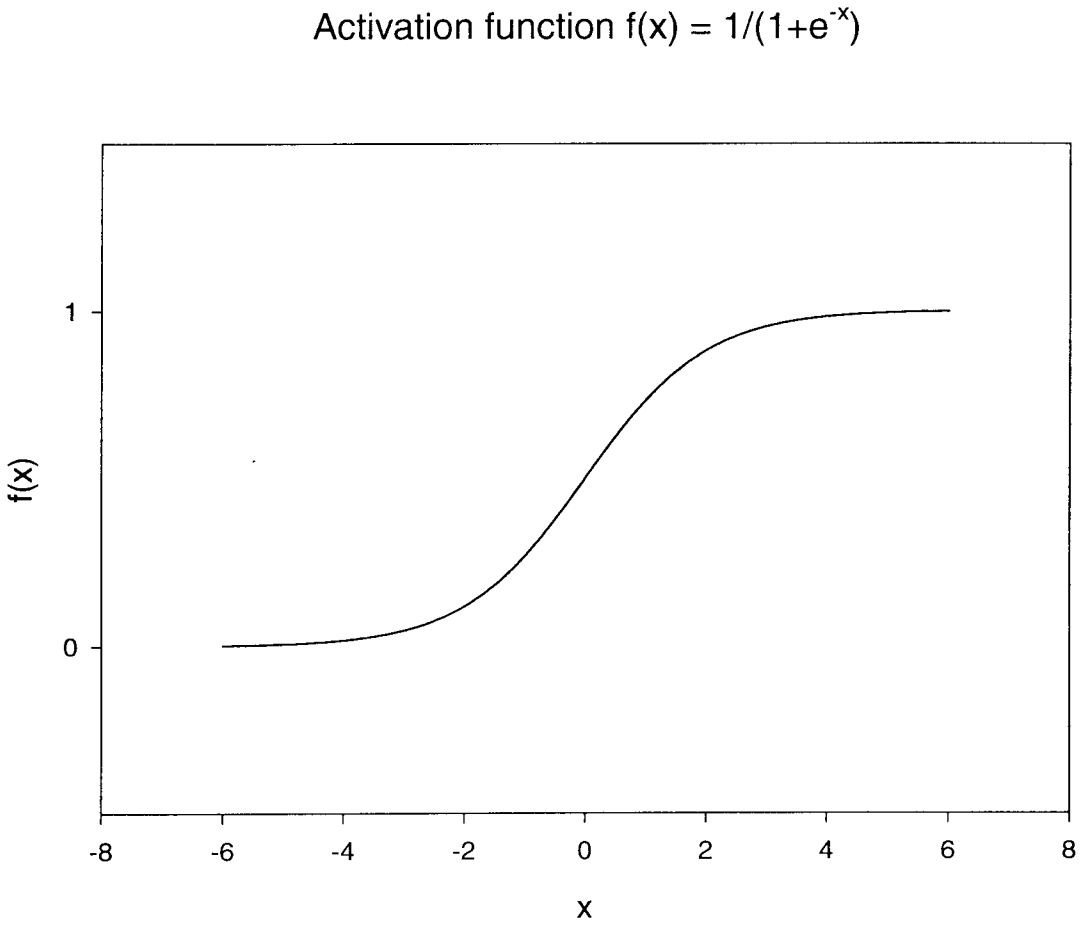

Figure 3.5: Activation function $f(x) = 1/(1 + e^{-x})$.

where $y_{\max}$ and $y_{\min}$ are the maximum and minimum expected unscaled output values and $s_{\max}$ and $s_{\min}$ are the maximum and minimum desired scaled values. Typically one chooses $s_{\max} = 0.9$ and $s_{\min} = 0.1$ for the activation function given by Eq. (3.2).

In addition to placing the network outputs in the linear range of the activation function, scaling has the additional advantage of rendering the network outputs dimensionless by cancelling any units that may be associated with the outputs. This allows different quantities, such as compositions (in percent) and interlayer spacings (in angstroms) to be on an equal footing for error backpropagation.

Similar scaling of the inputs is not required, since any such scaling would simply be absorbed by the weights in the connections between the input and hidden nodes [32].

3.6 Learning

Equations (3.1–3.3) are the basic equations for the *feedforward* phase of running the network. In this phase, the input nodes are set to the desired input values, the network node values are computed, and the output node values give the result for the given weights.

The next phase in running the network is the *backpropagation* phase, in which the network weights v_{ij} and w_{jk} are adjusted to correct for the error at the output nodes. To find the amount by which the network weights should be adjusted, we begin by writing an expression for the mean squared error in the network output [32]:

$$E_{\text{total}} = \frac{\frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{K}(y_{kn} - t_{kn})^2}{NK} \quad , \tag{3.5}$$

where N is the number of training examples available to train the network, K is the number of output nodes, y_{kn} is the k^{th} scaled network output for the n^{th} training example, and

t_{kn} is the k^{th} scaled target (i.e. "correct") output for the n^{th} training example.

3.6.1 Error Derivatives for Hidden-to-Output Weights

We begin by examining the weights w_{jk} between the hidden and output nodes. These weights may be adjusted using a gradient descent method, which will require that we find the derivative of the error with respect to the weights. Writing the squared error for one output node as E, we have

$$E = \tfrac{1}{2}(y_k - t_k)^2 \quad . \tag{3.6}$$

Using the chain rule, we differentiate E with respect to the weight w_{jk} as

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial y_k}\,\frac{\partial y_k}{\partial \eta_k}\,\frac{\partial \eta_k}{\partial w_{jk}} \quad . \tag{3.7}$$

Here η_k is the quantity

$$\eta_k = w_{0k} + \sum_{j=1}^{J} z_j w_{jk} \tag{3.8}$$

from Eq. (3.3). The first factor on the right of Eq. (3.7) is found by simply differentiating Eq. (3.6), so that

$$\frac{\partial E}{\partial y_k} = y_k - t_k \quad . \tag{3.9}$$

To compute the second factor on the right of Eq. (3.7), we note that from Eqs. (3.3) and (3.8),

$$y_k = f(\eta_k) \quad . \tag{3.10}$$

Then

$$\frac{\partial y_k}{\partial \eta_k} = f'(\eta_k) \quad . \tag{3.11}$$

Using Eq. (3.2) for f, we find

$$\begin{aligned} f'(x) &= \frac{e^{-x}}{(1+e^{-x})^2} \\ &= \frac{1}{1+e^{-x}} \; \frac{e^{-x}}{1+e^{-x}} \; . \end{aligned} \tag{3.12}$$

Now note that from Eq.(3.2) that

$$\begin{aligned} 1 - f(x) &= 1 - \frac{1}{1+e^{-x}} \\ &= \frac{e^{-x}}{1+e^{-x}} \; . \end{aligned} \tag{3.13}$$

From Eqs. (3.12) and (3.13) we see that

$$\begin{aligned} f'(x) &= \frac{1}{1+e^{-x}} \; \frac{e^{-x}}{1+e^{-x}} \\ &= f(x)\,[1 - f(x)] \; . \end{aligned} \tag{3.14}$$

Eq. (3.11) is then found using Eqs. (3.10) and (3.14) to give

$$\begin{aligned} \frac{\partial y_k}{\partial \eta_k} &= f'(\eta_k) \\ &= f(\eta_k)\,[1 - f(\eta_k)] \\ &= y_k\,(1 - y_k) \; . \end{aligned} \tag{3.15}$$

Finally, the third factor on the right of Eq. (3.7) is found by differentiating η_k with respect to the weight w_{jk}. Since

$$\eta_k = w_{0k} + \sum_{j=1}^{J} z_j w_{jk} \; ,$$

we find

$$\frac{\partial \eta_k}{\partial w_{jk}} = \begin{cases} 1 & \text{if} \quad j = 0 \; , \\ z_j & \text{if} \quad j > 0 \; . \end{cases} \tag{3.16}$$

If we define p_k to be the first two factors in the error derivative (3.7),

$$p_k \equiv \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial \eta_k} \ , \tag{3.17}$$

then Eqs. (3.9) and (3.15) give

$$p_k = (y_k - t_k)\, y_k\, (1 - y_k) \ . \tag{3.18}$$

The error derivative of Eq. (3.7) may then be written by combining Eqs. (3.16) and (3.18) to give

$$\boxed{\frac{\partial E}{\partial w_{jk}} = \begin{cases} p_k & \text{if} \quad j = 0 \ , \\ p_k\, z_j & \text{if} \quad j > 0 \ . \end{cases}} \tag{3.19}$$

Eq. (3.19) gives the change in the network output error E for a given change in the hidden-to-output weights w_{jk}.

3.6.2 Error Derivatives for Input-to-Hidden Weights

Having found the derivatives of the output node errors E with respect to the hidden-to-output nodes weights w_{jk}, we now find the derivatives with respect to the input-to-hidden weights v_{ij}. From the output node errors given by Eq. (3.6)

$$E = \tfrac{1}{2} (y_k - t_k)^2 \ ,$$

we find the derivative with respect to the weights v_{ij} using the chain rule, as before:

$$\frac{\partial E}{\partial v_{ij}} = \frac{\partial E}{\partial z_j} \frac{\partial z_j}{\partial \zeta_j} \frac{\partial \zeta_j}{\partial v_{ij}} \ . \tag{3.20}$$

where ζ_j is the quantity

$$\zeta_j = v_{0j} + \sum_{i=1}^{I} x_i \, v_{ij} \tag{3.21}$$

from Eq. (3.1). We begin with the first factor on the right of Eq. (3.20). From the analysis of the proceeding section (Eqs. (3.9) and (3.15)) we can write

$$\frac{\partial E}{\partial y_k} = y_k - t_k \ , \tag{3.22}$$

$$\frac{\partial y_k}{\partial \eta_k} = y_k \left(1 - y_k\right) \ . \tag{3.23}$$

Then the product of the first two factors in Eq. (3.20) is

$$\frac{\partial E}{\partial y_k} \ \frac{\partial y_k}{\partial \eta_k} = \left(y_k - t_k\right) y_k \left(1 - y_k\right) \equiv p_k \tag{3.24}$$

from Eq. (3.17). We then have

$$\frac{\partial E}{\partial z_j} = \sum_{k=1}^{K} p_k \, \frac{\partial \eta_k}{\partial z_j} \ . \tag{3.25}$$

Using Eq. (3.8) for η_k,

$$\eta_k = w_{0k} + \sum_{j=1}^{J} z_j w_{jk} \ , \tag{3.26}$$

we find

$$\frac{\partial \eta_k}{\partial z_j} = w_{jk} \ . \tag{3.27}$$

Then the first factor on the right of Eq. (3.20) may be written

$$\frac{\partial E}{\partial z_j} = \sum_{k=1}^{K} p_k \, w_{jk} \ . \tag{3.28}$$

To find the second factor on the right of Eq. (3.20), we note that

$$z_j = f(\zeta_j) \tag{3.29}$$

from Eqs. (3.1) and (3.21). Then, using the results from the previous section, we may write

$$\begin{aligned} \frac{\partial z_j}{\partial \zeta_j} &= f'(\zeta_j) \\ &= f(\zeta_j)\,[1 - f(\zeta_j)] \\ &= z_j(1 - z_j) \ . \end{aligned} \tag{3.30}$$

Finally, we find the third factor on the right of Eq. (3.20). Using Eq. (3.21) for ζ_j,

$$\zeta_j = v_{0j} + \sum_{i=1}^{I} x_i\, v_{ij} \ , \tag{3.31}$$

we have for the third factor on the right of Eq. (3.20)

$$\frac{\partial \zeta_j}{\partial v_{ij}} = \begin{cases} 1 & \text{if} \quad i = 0 \ , \\ x_i & \text{if} \quad i > 0 \ . \end{cases} \tag{3.32}$$

If we define

$$q_j \equiv \left(\sum_{k=1}^{K} p_k\, w_{jk} \right) z_j\ (1 - z_j) \ , \tag{3.33}$$

then the error derivative for the input-to-hidden weights (Eq. (3.20))can be written

$$\boxed{\frac{\partial E}{\partial v_{ij}} = \begin{cases} q_j & \text{if} \quad i = 0 \ , \\ q_j\, x_i & \text{if} \quad i > 0 \ . \end{cases}} \tag{3.34}$$

3.7 Learning Algorithms

Eqs. (3.19) and (3.34) give the derivatives of the network error E with respect to the hidden-to-output weights w_{jk} and the input-to-hidden weights v_{ij}, respectively. These

error derivatives may be used to determine the amounts by which the network weights should be changed, given the network output errors. Let u_m be a network weight (either v_{ij} or w_{jk}) at training epoch m, and d_m be the sum over all N training examples of the derivatives of the network output errors E with respect to the weights:

$$d_m = \sum_{n=1}^{N} \frac{\partial E}{\partial u_m} \ , \tag{3.35}$$

where the derivatives will be given by Eq.(3.19) or (3.34). At training epoch m, we wish to adjust the weight u_m by some amount c_m, so that

$$u_m = u_{m-1} + c_m \ , \tag{3.36}$$

where c_m will depend in some way on the error derivative d_m.

3.7.1 Constant Learning Rate

The simplest learning algorithm is one in which the change in the network weight is proportional to the error derivative [32]:

$$c_m = -\varepsilon \, d_m \ , \tag{3.37}$$

where ε is a constant called the *learning rate*. This algorithm is computationally fast, but may not work well with a complicated error surface such as those encountered in LEED problems. If ε is chosen to be too large, then the network weights will be adjusted in large increments—perhaps too large to find the global minimum of the error surface being searched. If ε is chosen to be too small, the network may require an excessive number of training epochs to converge.

3.7.2 Momentum

One method of improving the convergence speed of a network over that of a constant learning rate is the use of *momentum,* which will give the network the ability to adjust its weights preferentially in the direction in which they have been changing previously. A learning algorithm employing momentum will update the network weights at epoch m according to [32]

$$c_m = \mu\, c_{m-1} - (1-\mu)\,\varepsilon\, d_m \quad , \tag{3.38}$$

where μ is a momentum parameter ($0 \leq \mu < 1$). Larger values of μ will cause the weight changes to be more influenced by previous values of the weight changes; a value of $\mu = 0.9$ is typical.

Momentum is generally not needed if the network is trained by batch learning [32] (described below). Since batch learning was used for network training throughout this Dissertation, momentum was effectively disabled by setting $\mu = 0$.

3.7.3 Adaptive Learning Rates

Yet another method of improving the network's convergence speed is to use *adaptive learning rates* in which the network weights are updated in larger or smaller increments depending on whether recent updates have been in the same direction or in opposite directions. One begins by writing an exponential average f_m of the error derivative d_m [32],

$$f_{m+1} = \theta\, f_m + (1-\theta)\, d_m \quad , \tag{3.39}$$

where the parameter θ ($0 \leq \theta < 1$) determines how much weight is assigned to the past values of the error derivatives; the larger the value of θ, the more weight is assigned to past values of the error derivatives and less weight to the current value. Typically one may choose $\theta = 0.7$.

Then the change to the network weights is given by

$$c_m = -e_m d_m \ , \tag{3.40}$$

where e_m is the learning rate, which is computed from

$$e_m = \begin{cases} e_{m-1} + \kappa & \text{if} \quad d_m f_m > 0 \ , \\ e_{m-1} \times \phi & \text{if} \quad d_m f_m \leq 0 \ . \end{cases} \tag{3.41}$$

Here κ and ϕ are parameters that may be used to adjust the adaptive learning rate. If $d_m f_m > 0$, then the network weight is currently changing in the same direction as it has been in the past; in this case, it is desirable to make relatively large changes in the network weight by incrementing the learning by the parameter κ. If $d_m f_m \leq 0$, then the network updates have changed sign, and it is desirable to make smaller changes in the weight by multiplying the previous learning rate by the parameter ϕ ($0 < \phi < 1$). Note that this adaptive learning technique requires that a separate learning rate be maintained for each network weight, increasing the computer memory requirements.

Typically one may choose $\kappa = 0.1$ and $\phi = 0.5$, although it may be necessary to scale the parameter κ to the number of training examples if batch training is being used, as described shortly.

One may use both momentum and adaptive learning rates by replacing Eq. (3.40) with

$$c_m = \mu \, c_{m-1} - (1 - \mu) \, e_m \, d_m \ . \tag{3.42}$$

Since momentum was disabled in the Dissertation work in favor of batch learning, the momentum parameter μ was set to 0.

3.8 Weight Initialization

In order to begin network training, one must begin with some initial values for the network weights which can be later adjusted by the learning algorithms. The usual practice is to initialize the network weights to small random values [29, 31]. Small initial weights are chosen in order to prevent premature saturation of the network nodes; the initial weights are chosen to be random as a symmetry-breaking device, in order to keep different weights from performing the same function in the network and thus becoming redundant [33].

It has also been suggested [32] that half of the hidden-to-output node weights w_{jk} be initialized to $+1$ and the other half to -1 in order to improve convergence speed, but this technique did not seem to work well for the LEED problems being investigated here.

3.9 Batch Learning

When training the network, one may adjust the network weights after showing the network each example in the training data set. This approach, however, tends to bias the network weights; once the training is finished, the network will tend to best model the last example it was shown.

This difficulty may be avoided by training the network by *batch learning,* in which the network is shown *all* examples in the data set, one by one, without adjusting the network weights, but accumulating the total error. The weights are then adjusted according to the

total (net) errors found after all examples have been shown to the network. Batch learning was used for all network training throughout this Dissertation.

When using batch learning with an adaptive learning rate, it may be necessary to scale the learning parameter κ to the number of examples in the training set if a large number of examples is used. This was found to be the case during the course of this Dissertation, when the network training during the investigations of Chapter 6 required decreasing κ to prevent the network from diverging for large numbers of training examples.

Chapter 4

Development of Artificial Neural Networks for LEED Surface Structure Determination

"Natura materiae doctrinae est; haec fingit, illa fingitur."

— Quintilian, *Institutio oratoria, xix, 3*

As discussed earlier, one of the main computational difficulties encountered in the interpretation of experimental LEED data is the difficulty of inverting the dynamical LEED calculations; i.e. determining the arrangement of atoms near the crystal surface given the experimental I-V curves. This chapter describes the author's research in developing an artifical neural network capable of identifying a surface structure that corresponds to a given LEED I-V curve.

The general approach to using a neural network to solve the inverse LEED search problem is to use standard computer codes [11] to perform the LEED dynamical calculations that generate predicted I-V curves for a variety of candidate surface structures.

These curves are then used as training data on which to train the neural network. Once the network is trained, it is given an experimental I-V curve and asked to identify the surface structure parameters (atomic compositions and interlayer spacings).

This chapter will describe the computer codes developed by the author to implement such a neural network, as well as the initial functional and performance testing of the network.

4.1 Network Program Design

The neural network developed here for LEED surface structure determination was implemented as a computer program written in standard ANSI C and run on a Silicon Graphics computer with a UNIX operating system. C was chosen as the language of implementation because it is widely used, highly portable, and well suited for the project at hand.

The neural network program developed for this Dissertation, called *LEEDNET,* implements a backpropagation algorithm as described in Chapter 3. LEEDNET is a text-based program with a command-line interface. Commands may be entered into the program either interactively or as scripting files; the latter feature allows the program to be run as a batch job when large amounts of computing time are required. Commands are included to train, run, and test the network, format the data in the training file, adjust the network learning parameters, save and load network weights, etc. The complete set of commands is described in Appendix A (the *LEEDNET User's Guide*), which also includes general instructions for use of the program and a description of the LEEDNET input data format. A listing of the complete LEEDNET program is given in Appendices B–E.

The LEEDNET backpropagation algorithm includes adjustable adaptive learning rates

Pattern	Expected k	Network k	Error (%)
$y = \sin x$	1	1.000000	0.00
$y = \sin 2x$	2	2.000000	0.00

Table 4.1: Network recognition of $y = \sin kx$ for $k = 1, 2$.
The network was trained for 100 training epochs using 640 input nodes, 40 hidden nodes, and 1 output node, and using adaptive learning.

with an optional momentum term, each of which may be enabled or disabled as desired. The program is designed to train the network by batch learning (described in Chapter 3) to prevent biasing the network weights in favor of the last example it was shown.

4.2 Initial Testing

Once a basic version of the program had been written, it was initially tested by checking its ability to distinguish two different patterns with a single varying parameter. The two functions $y = \sin x$ and $y = \sin 2x$ for were chosen as the test patterns because of their simplicity and their similarity to the LEED I-V curves which would ultimately be used. A data set was created containing both functions sampled at 640 equally-spaced values of x for $x \in [0, 2\pi]$, and this data set was used to train the network. Once trained on both patterns, the network was shown each of the patterns individually and asked to identify the parameter k in $y = \sin kx$. The results, shown in Table 4.1, show that the network was able to identify k perfectly to seven significant digits after just 100 training epochs.

The next phase of the initial testing was to check the network's ability to recognize a

Pattern	Expected A	Network A	Error (%)
$y = \sin x$	1	1.016348	1.63
$y = 2\sin x$	2	1.984289	0.79

Table 4.2: Network recognition of $y = A\sin x$ for $A = 1, 2$.
The network was trained for 500 training epochs using 640 input nodes, 40 hidden nodes, and 1 output node, and using adaptive learning.

different parameter, the amplitude of a sinusoidal pattern. Two sinusoidal patterns were created, again with a single adjustable parameter: $y = \sin x$ and $y = 2\sin x$, using 640 equally-spaced points for $x \in [0, 2\pi]$. These patterns were used to train the network for 500 epochs, and the network was then tested by showing it the individual training patterns and asking it to identify the amplitude. The results are shown in Table 4.2, and show that the network was able to successfully identify the sinusoidal amplitudes as well. The results are not as dramatic as in the previous test, however; the network was found to be somewhat less adept at recognizing sinusoidal amplitudes than it was at identifying frequencies. Training required 500 epochs rather than 100 before the training error reached its lowest value, and the independent test results show larger errors.

The network's greater ease in recognizing sinusoidal frequencies is of benefit to the use of networks in LEED structural analysis, since it is generally regarded [11, 34] that peak positions in I-V curves are of greater importance than the magnitudes of the peaks.

Having passed these simple tests with just two training examples, the network was next tested for its ability to recognize several different values of sinusoidal frequencies. For this test, the training data consisted of the pattern $y = \sin kx$ for a single varying

Pattern	Expected k	Network k	Error (%)
$y = \sin x$	1	1.000000	0.00
$y = \sin 2x$	2	2.000000	0.00
$y = \sin 3x$	3	3.000000	0.00
$y = \sin 4x$	4	4.000000	0.00
$y = \sin 5x$	5	5.000000	0.00

Table 4.3: Network recognition of $y = \sin kx$ for $k = 1, 2, 3, 4, 5$.
The network was trained for 160 training epochs using 640 input nodes, 40 hidden nodes, and 1 output node, and using adaptive learning.

parameter $k = 1, 2, 3, 4, 5$ and 640 equally-spaced values of x for $x \in [0, 2\pi]$. After training for 160 epochs, the network was shown each of the five patterns individually and asked to identify the frequency. The results, shown in Table 4.3, show that the network was able to recognize the sinusoidal frequencies perfectly to seven significant digits after just 160 training epochs.

A similar test was performed to check the network's ability to recognize several values of the sinusoidal amplitude. The pattern $y = A \sin x$ was used to generate training data for the single varying parameter $A = 1, 2, 3$, using 640 equally-spaced values of x for $x \in [0, 2\pi]$. The results of showing the trained network each of the three individual patterns (Table 4.4) show that the network was successfully able to identify A in each case, although more trainining epochs were required than for the frequencies, and the errors were larger (although still quite small).

The final phase of this initial testing was to test the network's ability to *simultaneously* recognize both the frequencies and amplitudes of a sinusoidal pattern. Training data

Pattern	Expected A	Network A	Error (%)
$y = \sin x$	1	0.999996	0.0004
$y = 2\sin x$	2	2.000007	0.0004
$y = 3\sin x$	3	2.999994	0.0002

Table 4.4: Network recognition of $y = A\sin x$ for $A = 1, 2, 3$.
The network was trained for 2000 training epochs using 640 input nodes, 40 hidden nodes, and 1 output node, and using adaptive learning.

was generated of the form $y = A\sin kx$, with two varying parameters, $A = 1, 2, 3$ and $k = 1, 2, 3, 4, 5$, and using 640 equally-spaced values of x for $x \in [0, 2\pi]$. After the network was trained for 2000 epochs it was shown each of the individual sinusoidal patterns and asked to identify both the amplitude and frequency. The results are shown in Table 4.5, and demonstrate that the network was able to successfully identify both parameters simultaneously in each case, to within a small error.

4.2.1 Adjustment of Network Parameters

The data sets used for the initial network testing included 640 data points for each training pattern, since this was the number of data points available for the I-V data to be used later. This allowed the network parameters (such as the adaptive learning rate constants and number of nodes in the hidden layer) to be adjusted for a data set of the same size as the LEED I-V data set.

This initial testing of the network provided an opportunity to evaluate the performance of different learning algorithms and parameters. The use of an adaptive learning rate was

	A		k		Error (%)	
Pattern	Expected	Network	Expected	Network	A	k
$y = \sin x$	1	1.003167	1	1.001520	0.317	0.152
$y = \sin 2x$	1	0.999579	2	1.999990	0.042	0.001
$y = \sin 3x$	1	0.993953	3	2.999819	0.605	0.006
$y = \sin 4x$	1	0.999770	4	4.000034	0.023	0.001
$y = \sin 5x$	1	1.002643	5	4.998514	0.264	0.030
$y = 2\sin x$	2	1.998000	1	0.996934	0.100	0.307
$y = 2\sin 2x$	2	2.000197	2	1.999989	0.010	0.001
$y = 2\sin 3x$	2	2.004977	3	3.000042	0.249	0.001
$y = 2\sin 4x$	2	2.000124	4	3.999911	0.006	0.002
$y = 2\sin 5x$	2	1.998523	5	5.003198	0.074	0.064
$y = 3\sin x$	3	3.000749	1	1.001694	0.025	0.169
$y = 3\sin 2x$	3	2.999926	2	1.999971	0.002	0.001
$y = 3\sin 3x$	3	2.996878	3	2.999903	0.104	0.003
$y = 3\sin 4x$	3	2.999937	4	4.000066	0.002	0.002
$y = 3\sin 5x$	3	3.000614	5	4.998162	0.020	0.037

Table 4.5: Network recognition of $y = \sin kx$ for $A = 1, 2, 3$ and $k = 1, 2, 3, 4, 5$. The network was trained for 2000 training epochs using 640 input nodes, 40 hidden nodes, and 2 output nodes, and using adaptive learning.

found to significantly increase the convergence speed of the network. Here the learning rate e_m at epoch m in the training is taken to be

$$e_m = \begin{cases} e_{m-1} + \kappa & \text{for weights changing in the same direction,} \\ e_{m-1} \times \phi & \text{for weights changing direction.} \end{cases} \tag{4.1}$$

To demonstrate the effectiveness of an adaptive learning rate, the network was trained on problems in the previous section involving the determination of two parameters of a sinusoidal pattern (Table 4.5), both with and without an adaptive learning rate. Figure (4.1) shows the network test error as a function of the number of training epochs in each case, and clearly shows the improvement gained by the use of adaptive learning in the network. The adaptive learning case used the learning rate paramters $\kappa = 0.1$ and $\phi = 0.5$, while the non-adaptive case used the constant learning rate $\varepsilon = 0.1$.

Figure 4.2 shows a similar case in which the same problem was run with adaptive learning, but for cases both with and without a learning momentum term. The inclusion of momentum in updating the network weights, which biases weight changes in favor of continuing in the same direction, did not appear to significantly affect the network convergence and was not further used in the work for this Dissertation.

Another important network parameter to be adjusted is the number of nodes in the network's hidden layer. Neural network theory is not sufficiently developed to allow an *a priori* determination of the number of nodes in the hidden layer for a given problem [32, 31]; instead, this number must be adjusted by trial and error. Using too few nodes in the hidden layer will not provide enough weights to fit the training data, and the network will not converge. Using too many nodes in the hidden layer, on the other hand, may result in overfitting of the data and greatly increasing the training time and computer

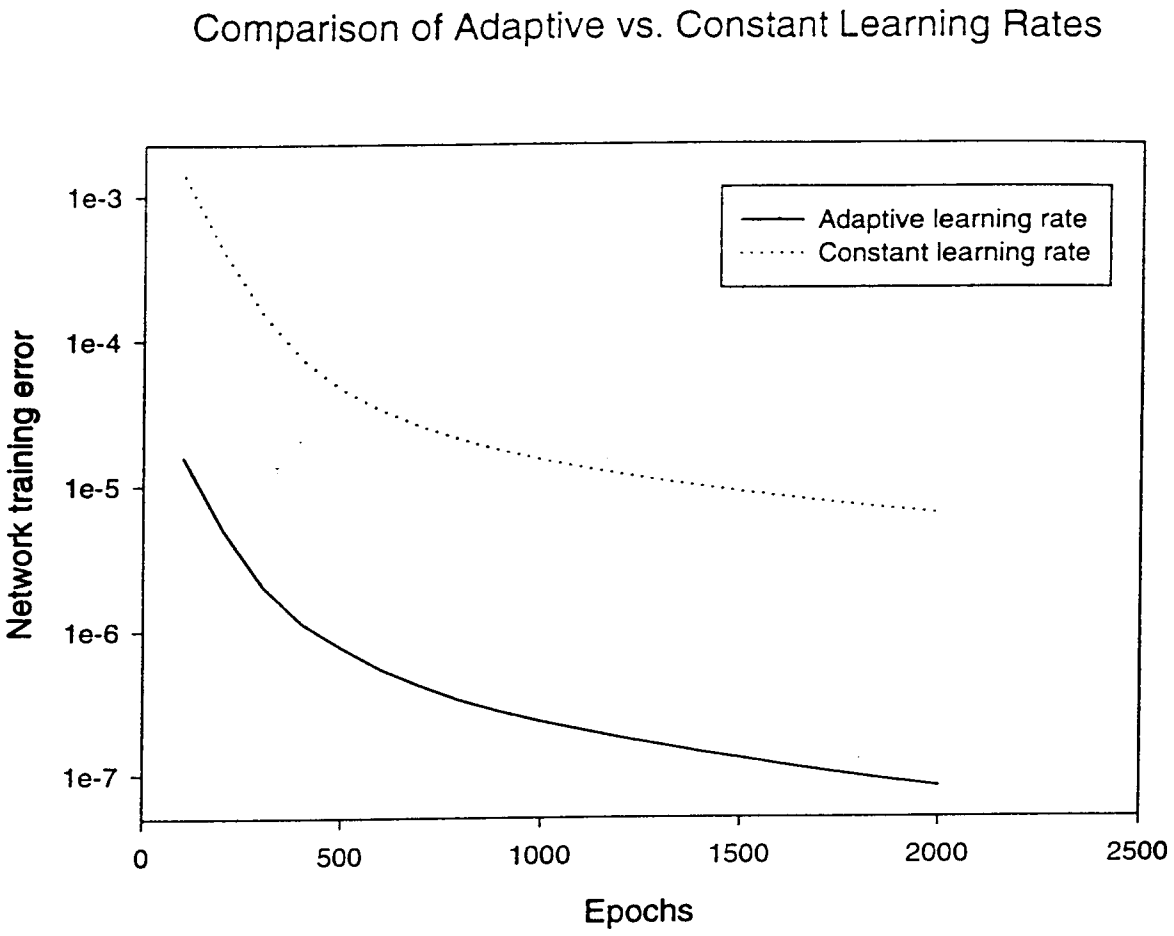

Figure 4.1: Network error vs. training epoch, with and without adaptive learning rate.

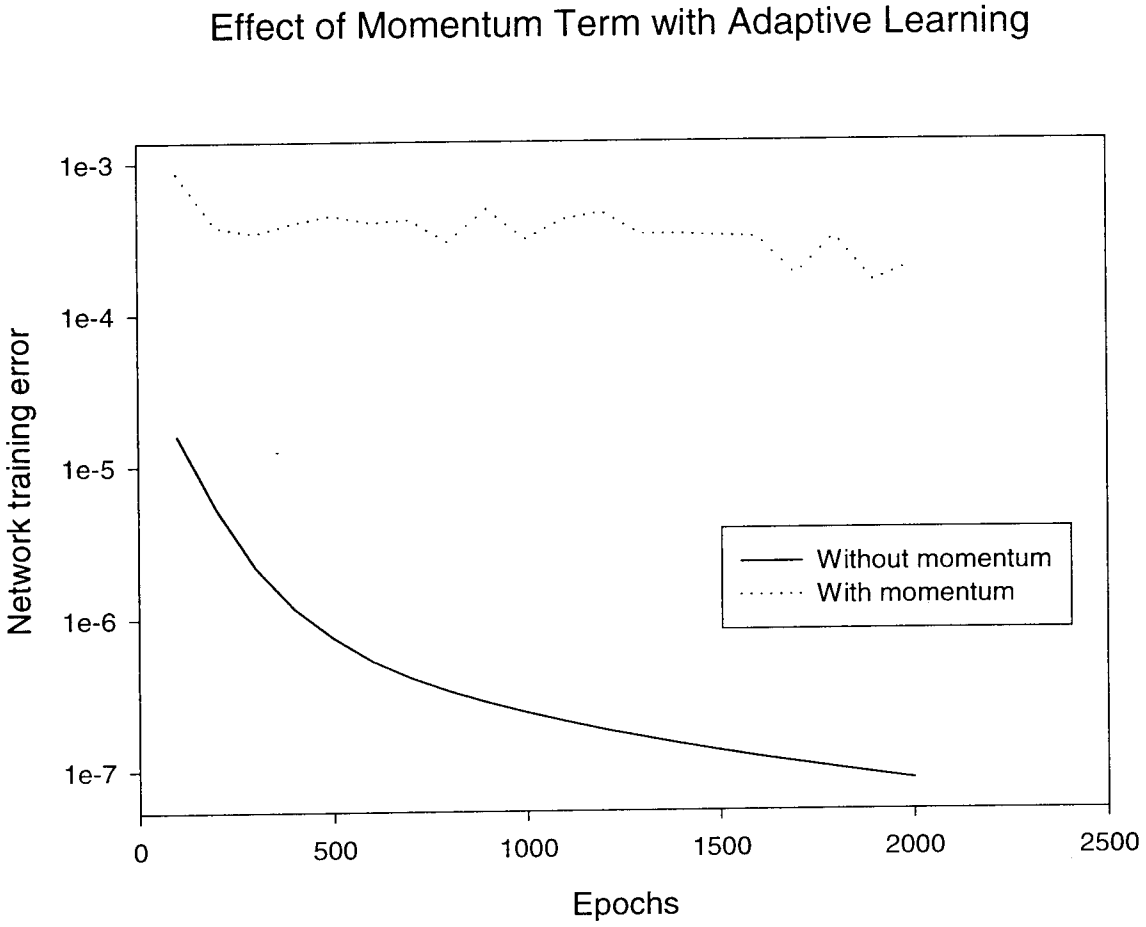

Figure 4.2: Network error vs. training epoch, with and without momentum term.

Hidden nodes	Network amplitude, A	Network frequency, k	Error (%) A	k
2	2.070583	2.083956	30.98	4.20
5	2.999886	2.000164	0.0038	0.0082
10	3.000173	2.000146	0.0058	0.0073
40	3.000051	1.999991	0.0017	0.0005
100	3.000237	1.999952	0.0079	0.0024
500	2.999259	1.999859	0.0247	0.0071

Table 4.6: Network recognition of $y = 3\sin 2x$ with different numbers of hidden nodes. The network should return $A = 3$ and $k = 2$ in each case. The network was trained for 2000 training epochs using 640 input nodes and 1 output node, and using adaptive learning. Note particularly the difficulty the network has in identifying the amplitude A when using only 2 hidden nodes.

memory requirements. By experimenting with different numbers of nodes in the hidden layer, one may develop a network which fits the training data well enough to generalize the training data without overfitting.

A simple example may help to illustrate the problems encountered when one uses too few hidden nodes in the network. In Table 4.6, the function $y = 3\sin 2x$ was used to train a network using several different sizes of hidden layers. The results show that the network had difficulty in identifying the amplitude of the pattern on which it was trained when only two hidden nodes were used. The problem did not occur when more than five hidden nodes were used.

Throughout this Dissertation, 30 or 40 nodes were generally used in the hidden layer with acceptable results. Convergence of the network was generally unaffected by small changes in the number of hidden nodes, so the exact value of this number was not found

to be critical.

4.3 Training with Calculated I-V Data

The initial testing of the network showed that it was able to correctly identify a variety of sinusoidal functions. The next step in testing the network was to train it with actual dynamically calculated LEED I-V spectra, calculated by standard computer codes [11]. The calculated training data was for the (100) surface of a $Ni_{50}Pd_{50}$ alloy, for which calculated and experimental data was available from an earlier study on surface segregation [34]. The data was calculated for normal incidence and included (00), (10), (11), and (20) beams for energies between 30 and 348 eV at intervals of 2 eV. The calculated data therefore includes 160 data points in each of its four beams, for a total of 640 data points in each structures. I-V data were calculated for 180 different structures, with the top layer compositions C_1 ranging from 0 to 50% nickel, the second layer compositions C_2 ranging from 50 to 100% nickel, and the third layer compositions C_3 ranging from 30 to 70% nickel, all at 10% intervals. For all the calculated data, the interatomic layer spacings were held constant at the bulk layer spacing of 1.87 Å.

The I-V curves were entered into the input nodes of the network by assigning one input node to each energy point, so that 160 input nodes were used for each beam. The four beams were concatenated, so that a total of 640 input nodes were used for the network. Fig. 4.3 shows schematically how the I-V intensities are entered into the network.

The first test of the network using calculated I-V curves involved training the network on six different spectra which differed only in the atomic compositions (percent nickel) in the top layer, with the second and third layer compositions and surface interlayer spacings

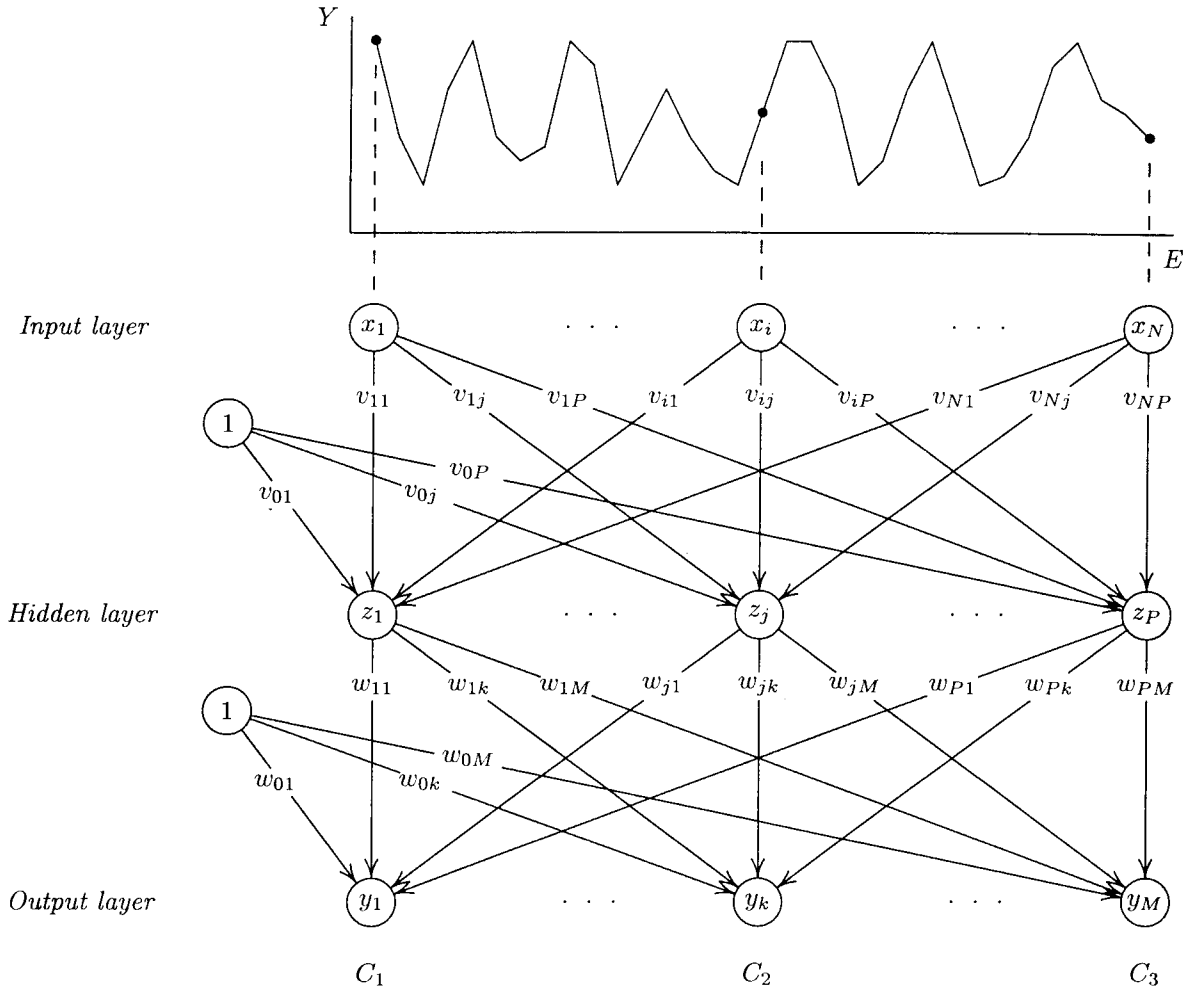

Figure 4.3: Input of I-V data to a backpropagation network.

Training data (%Ni)			Network results (%Ni)		
C_1	C_2	C_3	C_1	C_2	C_3
0	80	50	1.109107	80.008560	50.000000
10	80	50	9.097336	79.988876	49.999996
20	80	50	19.330671	79.995750	50.000000
30	80	50	30.411045	80.004921	50.000004
40	80	50	40.831749	80.005630	50.000004
50	80	50	49.409557	79.996788	50.000000

Table 4.7: Network recognition of $Ni_{50}Pd_{50}(100)$ I-V curves for six different top layer compositions.
The network was trained on all of the structures shown, and was then shown each structure individually and asked to identify the composition (%Ni) in each layer. Training was for 2000 epochs.

being held fixed. This approach was chosen because changes in the top layer compositions would result in large changes in the calculated I-V spectra, which would facilitate the network's ability to distinguish the six patterns. Once the network was trained on these six patterns, it was shown each of the patterns individually. The results are shown in Table 4.7, and show that the network was successfully able to identify the top layer compositions to within a small error.

The next stage of testing was to check the network's ability to identify surface structure parameters from I-V curves on which it had not been trained. That is, the test checked the network's ability to "interpolate" between I-V patterns on which it had been trained in order to identify a surface structure parameter.

For this test, the network was trained on the I-V curves of five surface structures, with 0, 10, 20, 40, and 50% nickel in the top atomic layer. The second layer was held at

a constant 80% nickel and the third layer at 50% nickel for all structures. The interatomic layer spacings were held at the bulk layer spacing value of 1.87 Å for all layers for each structure. The network was trained on the I-V curves for these five structures for 2000 training epochs.

After the training was complete, the network was shown a calculated I-V curve on which it had not been trained: 30% nickel in the top atomic layer. When shown this I-V curve and asked to identify the composition of the top atomic layer, the network returned a result of 30.685% nickel, in close agreement with the expected result of 30%. The full results, shown in Table 4.8, show that the network was also able to recognize the compositions of all three top atomic layers for the training data as well as for the $C_1 = 30\%$ nickel case.

A similar test was performed in which only the second layer composition was varied. The structures shown in Table 4.9 were used to train the network in which the second layer composition varied between 50 and 100% Ni. Once training was complete, each of the individual spectra was shown to the network, and the network asked to identify the compositions in each of the top three atomic layers. As shown in Table 4.9, the network was able to successfully recognize the second layer compositions to within a few tenths of a percent error.

As had been done with the top layer compositions, the network was then trained on five of these six structures, with the 30–80–50 %Ni structure having been left out. After training the network for 2000 epochs, it was shown all six structures (including the one on which it had *not* been trained) and asked to identify the compositions of all three top atomic layers. The results are shown in Table 4.10, and show that the network was successfully able to identify the compositions of all atomic layers for all structures,

	Training data (%Ni)			Network results (%Ni)		
	C_1	C_2	C_3	C_1	C_2	C_3
	0	80	50	0.980522	79.996605	50.000000
	10	80	50	9.195305	79.997948	50.000000
	20	80	50	19.588572	80.008736	50.000004
*	30	80	50	30.685244	80.013229	50.000004
	40	80	50	41.000332	80.006714	50.000000
	50	80	50	49.427132	79.991333	50.000000

Table 4.8: Network recognition of $Ni_{50}Pd_{50}(100)$ I-V curves for six different top layer compositions, having trained on five.
The network was trained on all of the structures shown *except* for the $C_1 = 30\%$ Ni case (indicated by a *). The network was then shown each structure individually and asked to identify the composition (%Ni) in each layer. Note that the network was able to successfully determine the compositions of all three layers for the case on which it was not trained. Training was for 2000 epochs.

Training data (%Ni)			Network results (%Ni)		
C_1	C_2	C_3	C_1	C_2	C_3
30	50	50	30.017166	49.524879	50.000000
30	60	50	30.007994	59.604801	50.000004
30	70	50	30.001333	70.462921	50.000004
30	80	50	29.996197	81.179176	50.000000
30	90	50	29.992945	90.609215	50.000000
30	100	50	29.989897	98.050400	49.999996

Table 4.9: Network recognition of $Ni_{50}Pd_{50}(100)$ I-V curves for six different second layer compositions.
The network was trained on all of the structures shown, and was then shown each structure individually and asked to identify the composition (%Ni) in each layer. Training was for 2000 epochs.

	Training data (%Ni)			Network results (%Ni)		
	C_1	C_2	C_3	C_1	C_2	C_3
	30	50	50	30.007107	49.687546	50.000004
	30	60	50	29.995325	59.719913	50.000011
	30	70	50	29.990582	70.631752	50.000011
*	30	80	50	29.991577	81.477409	50.000004
	30	90	50	29.998667	91.037216	49.999996
	30	100	50	30.008482	98.556068	49.999985

Table 4.10: Network recognition of $Ni_{50}Pd_{50}(100)$ I-V curves for six different second layer compositions, having trained on five.
The network was trained on all of the structures shown *except* for the $C_2 = 80\%$ Ni case (indicated by a *). The network was then shown each structure individually and asked to identify the composition (%Ni) in each layer. Note that the network was able to successfully determine the compositions of all three layers for the case on which it was not trained. Training was for 2000 epochs.

including the one that was not included in the training data.

Having demonstrated the network's ability to recognize individual surface parameters in a small number calculated I-V curves, the network was next trained on a larger set of 180 I-V curves, in which the compositions of the top three atomic layers were varied over a wide range (0–50% nickel in the top layer, 50–100% nickel in the second layer, and 30–70% nickel in the third layer, in steps of 10% in each case). The interlayer spacings for the calculated data were held constant at the bulk interlayer spacing of 1.87 Å for all layers. After the network was trained for 5000 epochs, it was shown 10 I-V curves (chosen at random) of the 180 curves on which it was trained. The results are shown in Table 4.11, and demonstrate the network's ability to recognize the compositions of the top three atomic layers when all are varied. These results also demonstrate the network's

Training data (%Ni)			Network results (%Ni)		
C_1	C_2	C_3	C_1	C_2	C_3
0	50	60	2.736262	49.857075	58.018745
0	70	70	1.986709	69.535530	70.334732
0	80	30	1.954359	80.403419	28.389162
10	70	70	9.009162	69.811073	71.389969
20	70	70	18.361916	69.959427	71.137978
20	80	60	18.194620	80.903564	61.544018
30	90	40	29.218269	90.235481	39.805630
50	90	40	50.813473	89.646935	39.306042
50	90	60	51.199688	89.840271	57.853497
50	100	60	51.065525	97.106239	58.734566

Table 4.11: Network recognition of $Ni_{50}Pd_{50}(100)$ I-V curves for 180 combinations of compositions in the top three atomic layers.
The network was trained on data with C_1 =0–50%Ni, C_2 =50–100%Ni, and C_3 =30–70%Ni, at intervals of 10%. Shown here are the network results after being shown 10 of these 180 structures, chosen at random. Interlayer spacings were held constant at the bulk value of 1.87 Å, and training was for 5000 epochs.

ability to use a larger training data set.

4.4 Tests of Data Set Reduction

4.4.1 Reduction of Number of Training Structures

Some of the early work in training the network required 100,000 training epochs and sometimes over one week of computer time. In an attempt to reduce the amount of training time required, the next phase of the network testing involved investigating various

	Expected Results (%Ni)			Network Results (%Ni)		
	C_1	C_2	C_3	C_1	C_2	C_3
	20.00	80.00	50.00	20.45	82.33	50.29
*	0.00	70.00	50.00	0.01	70.00	50.00
	0.00	50.00	30.00	-1.11	37.19	28.15
*	50.00	70.00	50.00	50.00	69.99	50.02
	50.00	100.00	70.00	45.46	92.03	65.80
	50.00	50.00	50.00	51.68	55.01	51.85
	20.00	100.00	30.00	20.23	102.86	27.89
	20.00	100.00	40.00	19.87	103.89	38.76
	20.00	70.00	60.00	19.64	69.47	60.41

Table 4.12: Network test results after training on 15 selected structures. (* indicates structures included in the training set.)

ways in which the training time might be reduced. One such strategy was to reduce the total number of structures being used to train the network, since the network had previously demonstrated the ability to "interpolate" between structures on which it had been trained. From the total set of 180 calculated structures, 15 structures covering a wide range of compositions in each layer were chosen for training the network. Once the network was trained, it was shown several of the structures from the total set of calculated data, many of which were not used to train the network. The results of this test are shown in Table 4.12. For the structures tested, the table shows that the network found the surface layer compositions with root-mean-square errors of 1.67% for the top-layer composition C_1, 5.60% for the second-layer composition C_2, and 1.85% for the third-layer composition C_3.

Another attempt at reducing the number of structures in the training data involved selecting every fifth structure out of the total set of 180 calculated structures. After the network was trained on this data set, the tests showed that the network was unable to correctly identify the compositions of the third atomic layer. The reason for this quickly became clear: by choosing every fifth structure from the total set, the training set coincidentally consisted of stuctures which all contained 30% nickel in the third atomic layer. Consequently, once the network was trained, it always returned a result of 30% nickel for the third atomic layer. This experience emphasized the importance of ensuring that the training data is carefully chosen to adquately sample the parameter space being studied. If one imagines a three-dimensional space with axes corresponding to each of the atomic layer compositions C_1, C_2, and C_3, then the total set of calculated data may be pictured as a $6 \times 6 \times 5$ lattice of points in that space. In choosing the data to use for the training set, one must be careful to properly sample this parameter space. Choosing every fifth structure in the parameter space corresponds to selecting a single plane of lattice points parallel to the C_1–C_2 plane with a constant value of 30% nickel for C_3.

Subsequent tests of data reduction involved choosing increasingly sparse samples of the total calculated data set. Training data sets for the network were created using every seventh of the 180 structures in the total data set, for a total of 26 structures. Once testing showed that this was satisfactory, another training set was created using every 17th of the 180 total structures, for a total of just 11 structures in the training set, or just 6% of the total data set. At this time the LEEDNET program was upgraded to perform a more thorough test after the network was trained: after training, the network was shown each of the 180 structures in the total training set, and LEEDNET reported statistics on the minimum, maximum, and root-mean-square errors for each atomic layer. The results are

Layer	Min. Error (%)	Max. Error (%)	RMS Error (%)
C_1	0.000	3.776	0.281
C_2	0.000	6.373	0.266
C_3	0.000	14.505	0.621

Table 4.13: Network test results after training on every 17th structure.

shown in Table 4.13.

4.4.2 Reduction of Number of Points Per Training Structure

Having tested the network's ability to identify surface compositions from a relatively small number of training structures, the next phase of network testing was to attempt to reduce the number of data points in each I-V curve. The calculated I-V curves contain data for four separate beams, with 160 data points at 2 eV intervals in each beam, for a total of 640 points for each curve. To test the network's ability to recognize I-V curves with fewer data points in each curve, the network was trained with training data at intervals of 4 eV and larger, as shown in Table 4.14. The training data for the results shown in the table are for every 17th structure of the total of 180 structures in the calculated data set, as was done for Table 4.13. The errors shown in the table have been computed from the errors found by showing the network all 180 structures in the original data set.

	C_1 Errors(%)		C_2 Errors(%)		C_3 Errors(%)	
Step(eV)	Max	RMS	Max	RMS	Max	RMS
2	3.776	1.063	6.373	2.302	14.505	4.181
4	5.529	1.727	8.735	3.101	16.673	4.451
6	4.290	1.396	6.358	2.711	16.113	4.227
8	2.792	0.987	8.859	3.348	16.847	4.827
10	6.317	1.779	13.094	3.905	13.451	4.044
12	3.917	1.216	5.266	1.923	14.878	4.448
14	3.545	0.739	4.561	2.038	16.160	4.653
16	4.636	1.386	6.669	2.678	11.573	3.166
18	7.622	2.080	13.388	3.904	19.930	4.789
20	2.558	0.770	8.539	2.421	16.133	3.892
24	4.494	1.145	6.736	3.139	18.033	5.135
30	3.126	0.832	8.318	2.774	15.617	4.560
34	7.844	1.947	5.738	1.915	14.392	4.621
40	6.120	2.092	15.048	3.642	8.907	2.623
50	4.788	1.366	7.707	1.864	25.251	6.936
60	3.911	1.168	5.342	1.681	12.492	3.295
70	4.881	0.995	10.278	2.619	14.322	3.367
80	12.869	3.298	27.639	6.365	26.633	6.265
100	9.442	3.125	7.851	2.788	18.439	5.066

Table 4.14: Network test results for reducing number of points in training sets. All results are for a training set containing every 17th structure of the total calculated data set.

4.5 Conclusions

It has been shown that a backpropagation artificial neural network can successfully identify dynamically calculated LEED I-V spectra for $Ni_{50}Pd_{50}(100)$. The network has been shown to be able to identify not only spectra on which it was trained, but can also "interpolate" between the training data and identify spectra on which it was *not* trained. In has, in fact, demonstrated its ability to identify a wide range of surface compositions when trained on only $\sim 10\%$ of the structures on which it was tested.

The network was also shown to work successfully when the I-V data was sampled at even very large intervals, demonstrating the network's ability to identify I-V spectra even with a minimum amount of training data.

Chapter 5

Application of Artificial Neural Networks to Low-Energy Electron Diffraction

"Veniet tempus quo ista quae nunc latent in lucem dies extrahat et longioris aevi diligentia."

— Seneca, *Naturales quaestiones, VII, xxv, 4*

5.1 Introduction

HAVING demonstrated the ability of an artificial neural network to recognize dynamically calculated LEED I-V spectra, the next phase of this Dissertation was to test the ability of the network to recognize structural parameters in *experimental* I-V spectra. For this next phase, LEED I-V spectra were calculated for $Ni_{50}Pd_{50}$, and these spectra were used to train the neural network. Once trained, the network was shown the experimental

I-V curves and asked to identify the compositions (percentage nickel) in the top atomic layer, in much the same way that it had been asked to determine compositions from theoretical data in the previous chapter. The results were compared to the results of a conventional exhaustive search to show that the network was able to successfully identify surface structure parameters in experimental I-V data.

5.2 Description of Experimental Sample

The experimental I-V curves used for this study were for the (100) surface of a $Ni_{50}Pd_{50}$ alloy, and were provided by Dr. Gregory Derry of Loyola College in Baltimore, Maryland. They were available from an earlier study on surface segregation [34]. The sample was a small disk, 1 cm in diameter and 1 mm in thickness, cut from a single crystal to expose the (100) plane. The sample was initially cleaned by polishing with diamond pastes; final cleaning was performed in an ultra-high vacuum ($\sim 10^{-10}$ torr) by repeated cycles of sputtering with a beam of argon ions. An annealing procedure was performed between sputtering cycles to repair the surface damage done by sputtering. Once all measurable impurities were removed from the sample (as determined by Auger electron spectroscopy), a final annealing procedure was performed and the sample was allowed to return to room temperature before performing the LEED experiment.

To generate the LEED data, low-energy electrons with energies from 50 eV to 320 eV were incident upon the cleaned $Ni_{50}Pd_{50}$ (100) surface in ultra-high vacuum. Data were collected at several incidence angles, and for several different diffracted beams. For this neural network study, only a subset of this data was used: the I-V data for the (10), (11), and (20) beams diffracted from normally incident electrons. The data for each

beam are averaged over several separate experiments and over symmetrically equivalent beams. The averaged local background intensity was subtracted from the data by the data acquisition equipment. The final data used for this work was also corrected for the instrument response function, and was normalized so that the maximum intensity for each beam is set to unity. The final data consists of this normalized intensity data for energies from 50 eV to 320 eV, in increments of 2 eV.

5.3 Theoretical Calculations

The calculated I-V spectra used to train the network were generated using standard computer codes [11]. The Renormalized Forward Scattering (RFS) method was used to perform the dynamical calculations, as described in Chapter 2. In addition, the standard computer codes were modified to include the averaged t-matrix approximation (ATA) to treat disordered alloy surfaces [34]. In this approximation, the t-matrix element that describes atomic scattering (discussed in Chapter 2) for each atom is taken to be a weighted average of the t-matrices of the elements present, weighted in proportion to the percent compositions of each element [35]:

$$t^I_{ata} = c_I t_{\mathrm{Ni}} + (1 - c_I) t_{\mathrm{Pd}} \ . \tag{5.1}$$

Here t^I_{ata} is the averaged t-matrix for atoms in layer I, whose concentrations are c_I for nickel and $1 - c_I$ for palladium. The atomic t-matrices are t_{Ni} for nickel and t_{Pd} for palladium.

$Ni_{50}Pd_{50}$ is a face-centered cubic structure in the bulk, with a lattice constant of 3.74 Å. Fig. 5.1 shows the bulk structure in the (100) plane parallel to the surface.

As shown in the figure, the nickel-palladium interatomic distance for this structure is (3.74 Å)/$\sqrt{2}$ = 2.64 Å, and the bulk interlayer spacing is $\frac{1}{2}$(3.74 Å) = 1.87 Å. The figure also shows that the bulk compositions of planes parallel to the (100) plane are 50% Ni and 50% Pd.

I-V curves were generated for four beams for this structure: (00), (10), (11), and (20). The calculations assumed an incident electron beam normal to the (100) plane, with energies ranging from 30 eV to 348 eV. Thermal effects were included for the nickel atoms only; the palladium atoms, being nearly twice as massive as the nickel atoms, have only half the thermal energy and were assumed to be stationary during the calculations. The Debye temperature of the nickel atoms was taken to be its bulk value, 440 K.

Figure 5.2 shows the experimental *I-V* curves used for this Dissertation, along with the theoretical *I-V* curves corresponding to the results of the conventional search. As the structure parameters used in the theoretical calculations are varied, the nature of the peaks also varies, often in complex ways. As shown in Fig. 5.3, varying the top-layer compositions (percent nickel) while holding the other parameters constant results in changes in the peak amplitudes and shapes. Fig. 5.4 shows that varying the interlayer spacing results in somewhat more complex changes in the peak amplitudes and shapes along with energy shifts of the peaks as the interlayer spacing Δd_{12} is varied.

5.4 Initial Results

Initial testing of the network with experimental data was aimed at identifying only the compositions of the first three atomic layers. The interlayer spacings were held constant at their bulk values of 1.87 Å. The training data thus consisted of *I-V* curves calculated

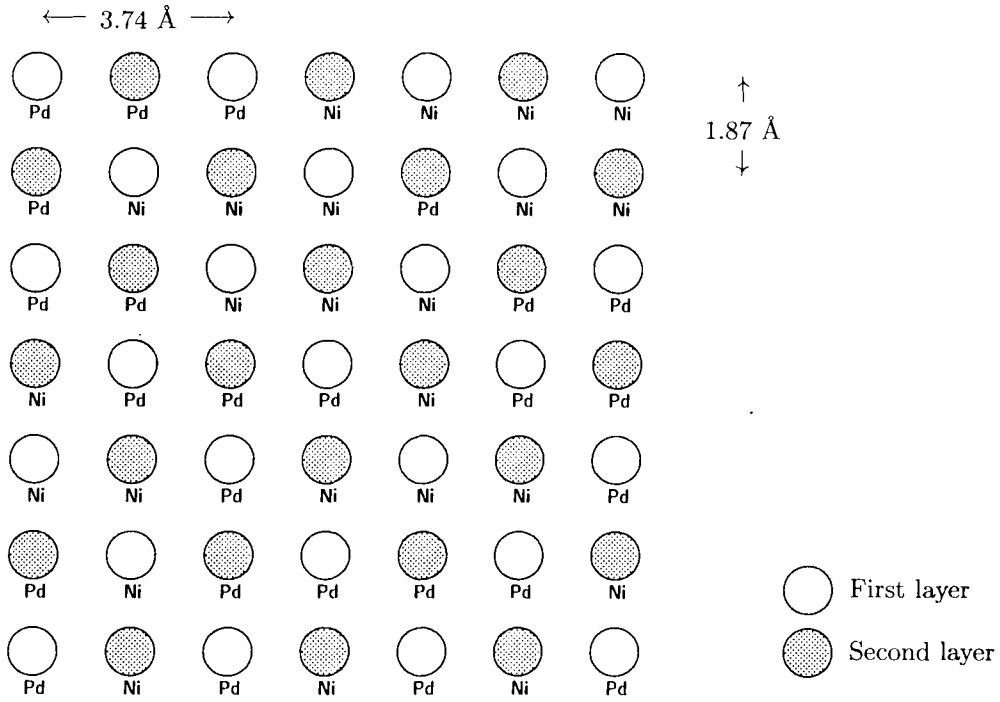

Figure 5.1: Bulk termination of $Ni_{50}Pd_{50}$ in the (100) plane.
Note that this is a randomly ordered alloy, so that nickel and palladium atoms appear randomly at each site.

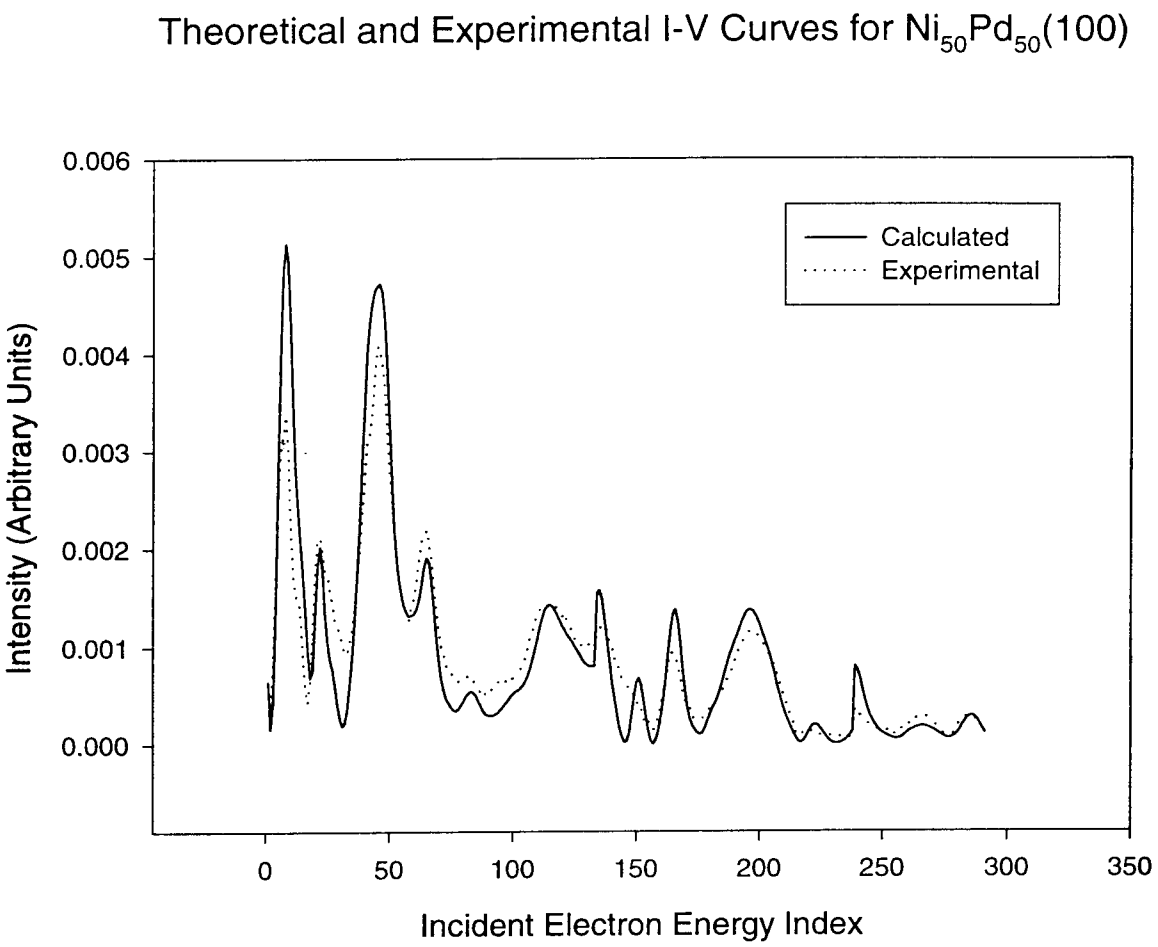

Figure 5.2: Experimental and theoretical I-V spectra for $Ni_{50}Pd_{50}(100)$. The data shown are for concatenated (10), (11), and (20) beams.

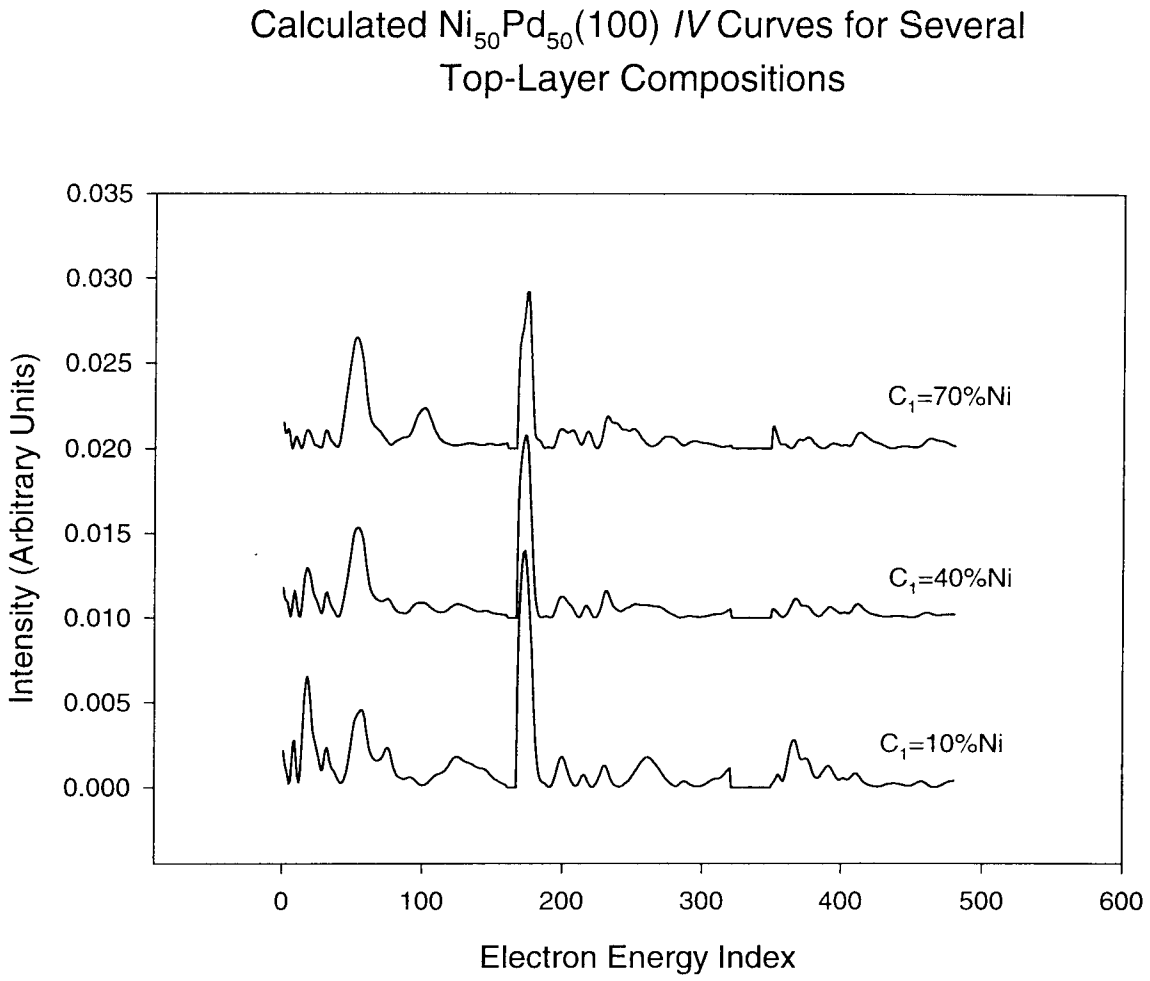

Figure 5.3: Theoretical *I-V* spectra for $Ni_{50}Pd_{50}(100)$ for several top-layer compositions. The data shown are for concatenated (10), (11), and (20) beams; the abscissa is an index indicating the electron energy, which runs from 30 eV to 348 eV for each of the three beams. Note the decrease in amplitude of the peaks around indicies 25 and 175 as the top-layer composition becomes nickel-rich, corresponding to energies of roughly 80 eV in the (10) beam and 60 eV in the (11) beam, respectively. A peak around index 60 (roughly 150 eV in the (10) beam) is seen to increase with increasing nickel content.

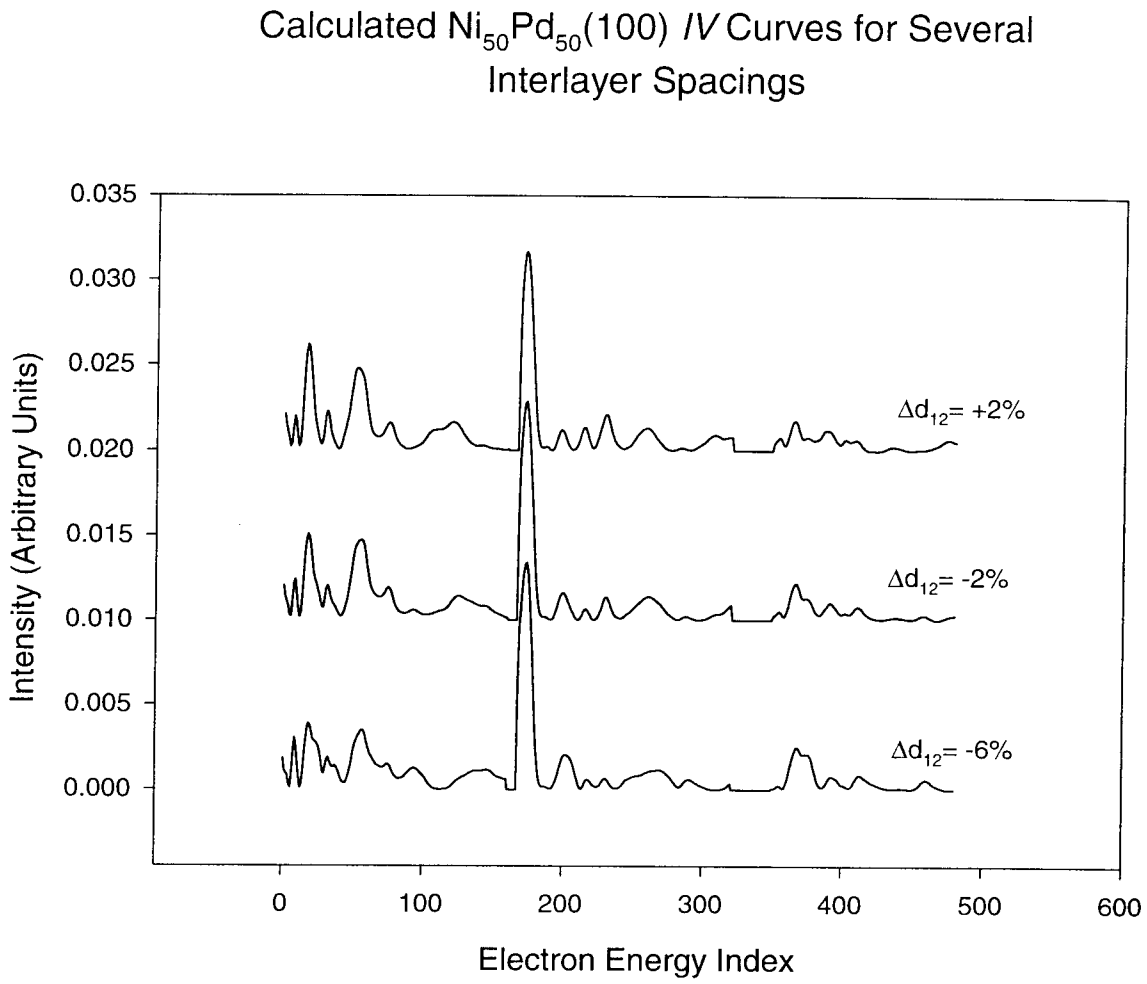

Figure 5.4: Theoretical I-V spectra for $Ni_{50}Pd_{50}(100)$ for several interlayer spacings. The data shown are for concatenated (10), (11), and (20) beams; the abscissa is an index indicating the electron energy, which runs from 30 eV to 348 eV for each of the three beams. The changes are more complex than for the case with changing top-layer compositions shown in the previous figure. Some peaks are shifted, some change in amplitude or shape, and some undergo all three changes as the interlayer spacing is changed. Those peaks that are shifted are generally shifted toward slightly lower energies as Δd_{12} is increased.

for a variety of combinations of compositions in the first three atomic layers, which model the segregation that takes place in the atomic layers in the surface. But since the interlayer spacings were held at their bulk values, relaxation of the surface layers was not modeled.

There are several differences between experimental and calculated I-V data that need to be allowed for before experimental data can be shown to a network trained on calculated data. First, the intensities for experimental data will generally not be expressed in the same units as for the calculated data. Calculated intensities are found from the squared amplitudes of the diffracted electron wave functions, while measured intensities are measured in some units that depend on the equipment being used, such as CCD counts. Since these two sets of intensity units are not easily related to each other, the usual practice is to express the intensities in arbitrary units and to normalize the diffracted beam intensities in some way. In the case of the experimental data used for this study, for example, the I-V data were normalized so that the maximum intensity in each beam is set to unity.

In order to relate the calculated intensities to the experimental intensities, it is necessary to apply some normalization scheme to both sets of data so that they are scaled to the same units. One could, for example, multiply the calculated data by a scaling factor so that the maximum intensity of each beam is set to unity; it could then be directly related to the experimental data. Because of uncertainties in the absolute values of the peak heights, however, this method was not deemed to be the best approach. Also, applying such adjustments to the calculated data would require re-training the network for each experiment, so it was considered best to apply all adjustments to the experimental data. To adjust the experimental intensities to the same scale as the calculated intensities, the experimental intensity data were normalized so that the integral under the I-V curve

between two fixed energies for both sets of data are equal:

$$\int_{E_i}^{E_f} I_{\text{experimental}}(E)\,dE = \int_{E_i}^{E_f} I_{\text{calculated}}(E)\,dE \tag{5.2}$$

This has the effect of requiring the mean intensity of both data sets to be equal. For this work, a trapezoidal rule integrator was used to find the integrals of the calculated and experimental I-V curves; the experimental intensities were then multiplied by the ratio of these integrals to set them to the same scale as the calculated intensities.

A second issue must be addressed before experimental I-V curves can be used with a neural network: since each of the network's input nodes represents the intensity at a specific incident electron energy, the experimental data must be interpolated to the same energy points as those used in the training data. For this study, the experimental data was interpolated to the energy values used in the calculated data using a cubic spline.

A third issue that must be addressed before comparing experimental and theoretical I-V spectra is that the experimental data will generally include an inner potential energy shift which is not included in the calculated data. Physically, the inner potential is due to dipole layers at the material surface, which have the effect of accelerating the incident electrons to a higher kinetic energy as they enter the surface from the vacuum. As a result, the I-V curves, being plotted against the energy of the electrons emerging from the electron gun, are shifted as a result of this inner potential [6, 11]. The magnitude of this shift is difficult to calculate since the value of the inner potential is often unknown, and the absolute energy of the calculation can be uncertain (depending of the value of the muffin tin zero). In practice, one often corrects for this effect by simply shifting the energy scale of the calculated data so that its peaks align with the experimental peaks [6], or it is treated as an additional non-structural parameter to be fit during the structure

search.

For this phase of the neural network study, the object was to study only the network's ability to recognize atomic layer compositions; the network's ability to determine the inner potential energy shift was not of immediate interest. To this end, a theoretical I-V spectrum was calculated using the best fit compositions and inner layers spacings that had been previously determined using an exhaustive global search [34]. The inner potential energy shift was then determined by shifting the experimental data in steps of 2 eV until the peaks in the two data sets were aligned. This gave an energy shift of -6 eV (i.e. the experimental data should be shifted 6 eV toward lower energy with respect to the calculated data to align the peaks in the two data sets). This value was used to eliminate the energy shift issue from the problem for this first phase of tests with experimental data.

With the inner potential energy shift determined and the interlayer spacings set at their bulk values, a set of I-V spectra was calculated in which only the layer compositions were varied: 0–50% Ni in the top layer, 50–100% Ni in the second layer, and 30–70% Ni in the third layer, all in increments of 10%. This resulted in $6 \times 6 \times 5 = 180$ separate sets of I-V spectra to be used for training the network. Once trained on these spectra, the network was shown the experimental spectrum, corrected for the difference in intensity scale and inner potential energy shift. The results are shown in Table 5.1. Comparing the network results with the error ranges for the target values shows a marginal agreement between the network results and the results found by a conventional exhaustive search.

Layer	Compositions(%Ni) Target	Network	Errors(%) Max	RMS
1	20 ± 11	38.75	2.10	0.77
2	100^{+0}_{-26}	92.95	4.15	1.45
3	34 ± 14	48.73	7.20	2.19

Table 5.1: Network test results for network shown experimental I-V data. The network was trained on calculated data assuming bulk interlayer spacings. Training data was for 0–50% Ni in the top layer, 50–100% Ni in the second layer, and 30–70% Ni in the third layer, in increments of 10%. Target compositions are from Ref. [34].

5.5 Correction for Interlayer Spacings

The results shown in Table 5.1 are only marginally in agreement with the results found by a conventional exhaustive search. One reason the results were not in better agreement with conventional results is that relaxation of the surface atoms was not modeled in the training data; instead, the interlayer spacings were all assumed to have the bulk value of 1.87 Å. A first step toward improving these results, therefore, was to generate new training data from calculations that include the correct interlayer spacings. Since this phase of the testing sought only to identify the correct layer compositions, this new set of training data was calculated using the interlayer spacings near the values that had been determined using a conventional exhaustive search: 1.8326 Å (bulk spacing minus 2%) between the first and second atomic layers, and 1.7952 Å (bulk spacing minus 4%) between the second and third atomic layers. Interlayer spacings for all deeper layers were taken to be their bulk value of 1.87 Å.

The training data generated for this test was for varying compositions in the first three

	Compositions(%Ni)	
Layer	Target	Network
1	20 ± 11	22.01
2	100^{+0}_{-26}	102.27
3	34 ± 14	42.78

Table 5.2: Network test results for network shown experimental I-V data, corrected for interlayer spacings.
The network was trained on calculated data assuming "correct" interlayer spacings. Training data was for 0–20% Ni in the top layer, 90–100% Ni in the second layer, and 25–50% Ni in the third layer, in increments of 5%. Target compositions are from Ref. [34].

atomic layers: 0–20% Ni in the top layer, 90–100% Ni in the second layer, and 25–50% Ni in the third layer, in increments of 5%, for a total of 90 training structures. The same -6 eV energy shift was applied to the experimental data as was done for the previous test, and the experimental data was normalized as before by scaling the experimental intensities so that the integrals under the experimental and calculated curves were equal. The results are shown in Table 5.2, and show much better agreement with the results obtained by an exhaustive search. This improvement may be attributed to (1) allowance being made for surface relaxation; (2) the smaller range over which the compositions were allowed to vary; and (3) the smaller increment over which the compositions in the training set were allowed to vary (5% Ni instead of 10%).

5.6 Test with Wider Training Range and Fewer Examples

The next test of the network incorporated several improvements: (1) the range of compositions was widened and increment increased to produce a more useful test; (2) the number of training structures was reduced from 90 to 11 in order to reduce the amount of time required to train the network; and (3) the training data was calculated using interlayer spacings that were closer to the values found by the conventional search.

The interlayer spacings for this test were taken to be 1.8363 Å (bulk spacing minus 1.8%) between the first and second atomic layers, and 1.7933 Å (bulk spacing minus 4.1%) between the second and third layers; these are the values that had been determined by a previous study on surface segregation [34]. I-V curves were calculated for this structure for compositions of 0–50% Ni in the first atomic layer, 50–100% Ni in the second layer, and 30–70% Ni in the third layer, all at intervals of 10%. Just 11 of these spectra were selected for training the network. After 250,000 training epochs, the network converged to the values shown in Table 5.3. Convergence of the network in this case required that the adapative learning rate be disabled and replaced with a constant learning rate.

Although these results are within the error bars of the conventional search method, the results obtained thus far were fraught with numerous difficulties. First, the network required 250,000 training epochs before it was adequately trained, requiring several days of computer time on a fast mainframe computer. This is in spite of having used only 11 different spectra to train the network.

Second, it was unclear when exactly the training should be stopped, as seen from the error plots of Figs. (5.5) and (5.6). Large oscillations in the training error (Fig. 5.5)

	Compositions(%Ni)	
Layer	Target	Network
1	20 ± 11	29.05
2	100^{+0}_{-26}	97.49
3	34 ± 14	42.47

Table 5.3: Network test results for network shown experimental I-V data, using reduced training set.
The network was trained on calculated data assuming "correct" interlayer spacings. Training data was for 11 selected structures in the range 0–50% Ni in the top layer, 50–100% Ni in the second layer, and 30–70% Ni in the third layer. Target compositions are from Ref. [34].

make it difficult to determine when the network weights have converged. Also, small differences in the choice of when to stop network training can result in large differences in the independent test error, as seen from the erratic nature of the independent test error plot (Fig. 5.6).

A third difficulty with this result is that several aspects of the problem required knowing the "answer" (determined by a conventional search) ahead of time. In particular, the experimental I-V curve was scaled to the same intensity scale as the calculated curve by requiring the integrals under the experimental and calculated curves to be equal. However, this required knowing *which* calculated curve to integrate. Furthermore, the energy shift calculation required aligning the peaks in the I-V curve for the experimental data with the peaks in the calculated curve for the known "correct" structure.

Lastly, while the results were within the error bars for the conventional calculation, it would be desirable for the network to return results that match the conventional results

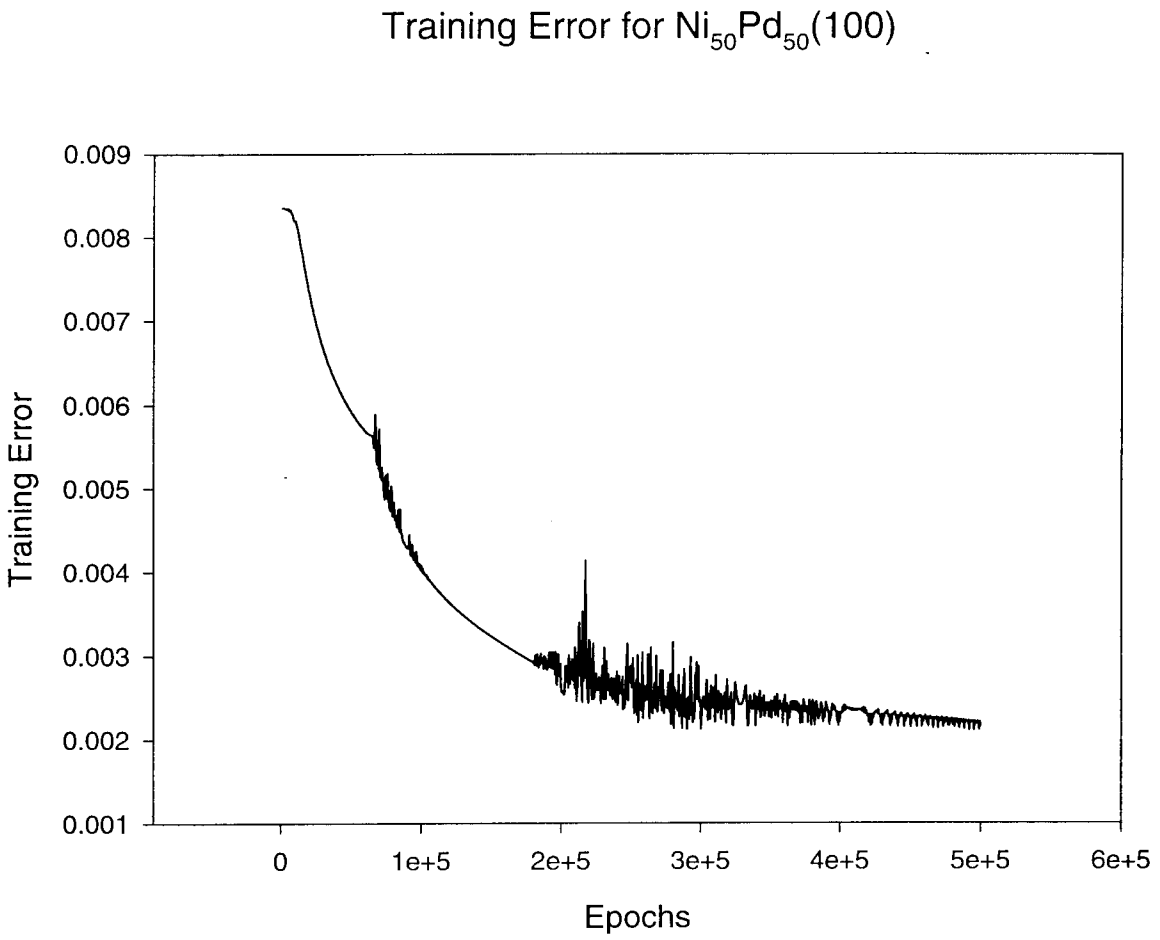

Figure 5.5: Network training error for $Ni_{50}Pd_{50}(100)$ vs. number of training epochs.

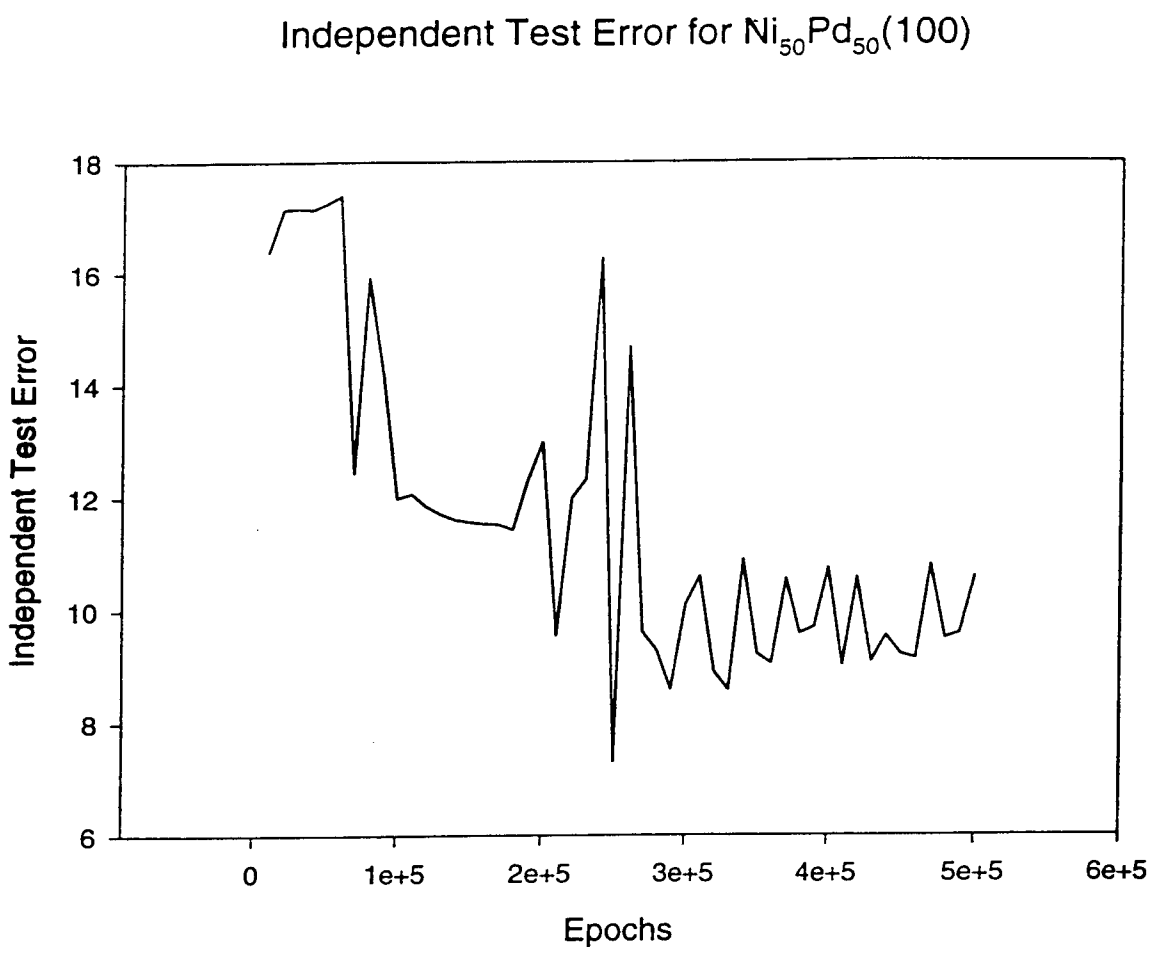

Figure 5.6: Network independent test error for $Ni_{50}Pd_{50}(100)$ vs. number of training epochs.

more closely.

5.7 The Pendry Y-function

In order to facilite the network's recognition of I-V spectra, some means was sought to improve the network's convergence, and to process the experimental data without knowing the conventional result ahead of time. To address many of these difficulties, the author decided to train the network using the *Pendry Y-function* of the dynamically calculated I-V spectra, rather than using the I-V spectra themselves. Pendry has defined the function $Y(E)$ by [14]

$$Y(E) = \frac{L^{-1}}{L^{-2} + V_{0i}^2} \ , \tag{5.3}$$

where $L(E)$ is the logarithmic derivative of the I-V spectrum,

$$L(E) = I'/I \ , \tag{5.4}$$

and V_{0i} is the electron self-energy, which is around -4 eV for most materials at electron energies above about 30 eV. Pendry designed this function to emphasize the physically important components of the I-V spectrum (such as the positions of the peaks), while tending to suppress features of that are physically less important, such as the absolute magnitudes of the intensities. The form of the function $L(E)$ given by Eq. (5.4) makes L insensitive to amplitudes for widely spaced peaks, but gives too high emphasis to zeros of $I(E)$. The form of $Y(E)$ given by Eq. (5.3) solves this problem by giving similar emphasis to zeros and peaks in $Y(E)$. This Y-function is commonly used to calculate the Pendry R-factor, a widely used measure of the correlation between calculated and experimental I-V spectra.

Using $Y(E)$ rather than $I(E)$ to train the network results in a much faster training time and more reliable results than using the I-V spectra directly, since the network may more readily recognize the physically important features of the network. Using $Y(E)$ also solves the normalization problems associated with using the I-V curves $I(E)$. Using $I(E)$ required knowing the "correct" I-V curve ahead of time in order to properly scale the intensities of the experimental data. The intensity units cancel in Eq. (5.4), and $Y(E)$ has units of eV^{-1}. Thus no normalization of the experimental data is needed when using $Y(E)$, and no foreknowledge of the results of the conventional search is required to scale the data.

Examination of the experimental data reveals the presence of many spikes in the data due to noise, which can lead to large fluctuations in the derivative I'. For that reason, both the experimental and calculated I-V data are first smoothed before the derivative is calculated, using a three-point moving average:

$$I_{\mathrm{smoothed}}(E_i) = \tfrac{1}{3}\left[I(E_{i-1}) + I(E_i) + I(E_{i+1})\right] \quad . \tag{5.5}$$

The Y-function of both the calculated and experimental data is then calculated using Eqs. (5.4) and (5.3) with this smoothed I-V data. The calculated $Y(E)$ data was then used to train the network, with greatly improved results. Figure 5.7 shows the training error vs. training epoch. Comparing with the previous results of Fig. 5.5 using I-V data shows that using $Y(E)$ for training improves the convergence speed by a factor of 500, and also results in much smoother convergence.

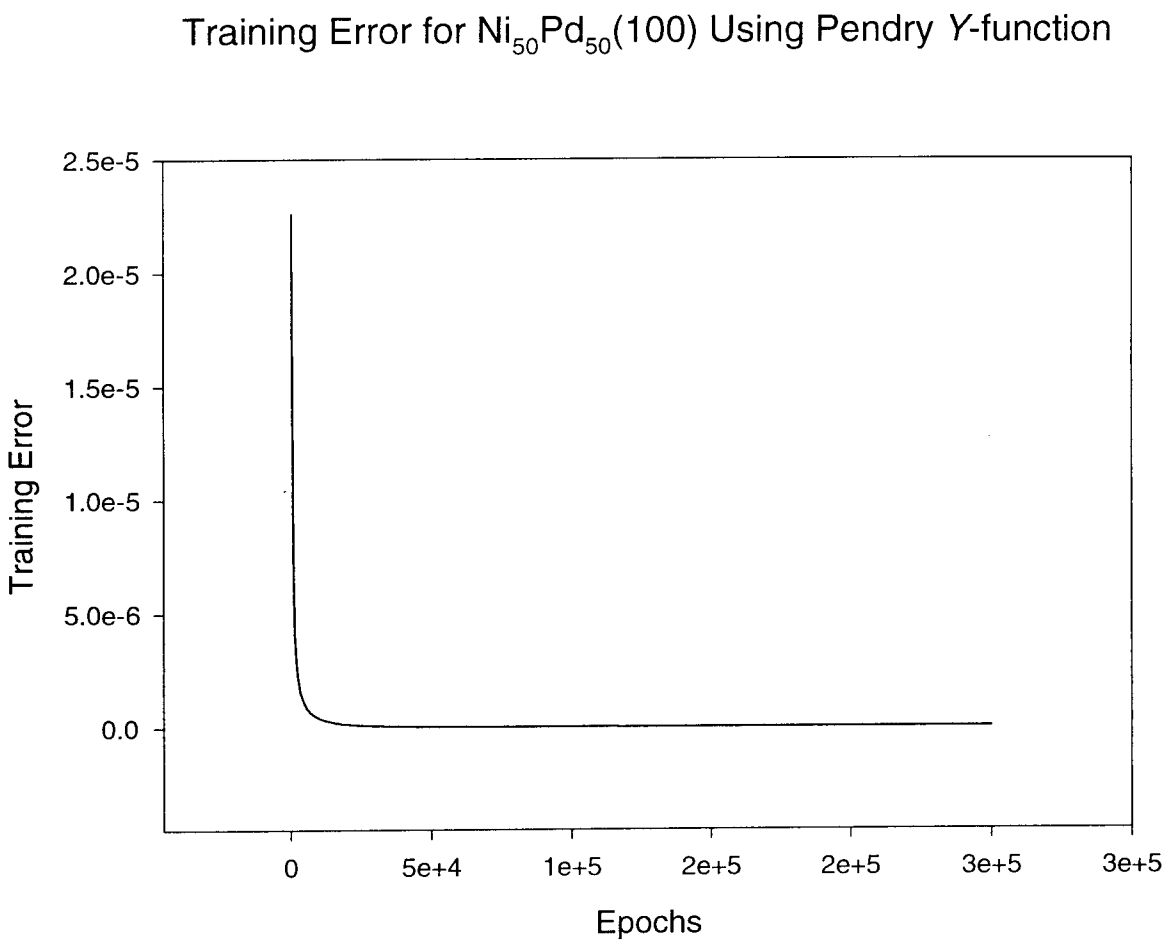

Figure 5.7: Network training error for $Ni_{50}Pd_{50}(100)$ vs. number of training epochs, using $Y(E)$.

5.8 Inner Potential Energy Shift

In order to enter the experimental data into the trained network, it is still necessary to allow for the inner potential energy shift, and to interpolate the experimental data to the same energy points as the calculated data. The inner potential energy shift was determined by shifting the experimental data with respect to the calculated data along the energy axis in steps of 0.1 eV, interpolating the experimental data at the new energy points using a cubic spline, and calculating the root-mean-square error between the Y-function of the shifted experimental data and the Y-function of the "correct" I-V spectrum found from a conventional exhaustive search. This calculation yielded a value of -4.7 eV for the energy shift (i.e. the experimental data should be shifted 4.7 eV toward lower energy with respect to the calculated data to align the peaks in the two data sets).

5.9 Testing With the Y-function

In order the test the network's ability to recognize I-V spectra using Y-functions, training data was generated consisting of the calculated I-V spectra of a wide range of possible compositions: 0–50% Ni in the top atomic layer, 50–100% Ni in the second layer, and 30–70% Ni in the third layer, at intervals of 10%, for a total of 180 different combinations of compositions. The I-V spectra for each of these surface compositions was shown to the network for 2000 training epochs before network training was stopped.

Once the network was trained, the experimental data was corrected for the inner potential energy shift and interpolated to the same energy points as those used in the calculated data to train the network. The shifted and interpolated experimental data was then stored

	Compositions(%Ni)	
Layer	Target	Network
1	20 ± 11	12.51
2	100^{+0}_{-26}	98.00
3	34 ± 14	36.43

Table 5.4: Network test results for network shown Y-function of experimental I-V data. The network was trained on calculated data assuming "correct" interlayer spacings. Training data was for 180 structures in the range 0–50% Ni in the top layer, 50–100% Ni in the second layer, and 30–70% Ni in the third layer. Target compositions are from Ref. [34].

into the input nodes of the trained network and the network was run. The resulting output, shown in Table 5.4, shows that the network was able to identify the compositions of the surface layers to within the errors stated for the conventional calculation [34], with greatly improved results over the previous attempt to use the I-V data alone.

The results shown in in Table 5.4 using the Pendry Y-function are a significant improvement over previous results using I-V spectra directly, particularly with regard to the speed and smoothness of convergence of the network. The results demonstrate the network's ability to identify the surface layer compositions when the interlayer spacings and inner potential energy shift are known. The next phase of network development was to eliminate the requirement of a foreknowledge of these parameters, so that the surface compositions and interlayer spacings could be determined without knowing any of the results of a conventional search.

Chapter 6

LEED Surface Structure Determination Using Artificial Neural Networks

καὶ γνώσεσθε τὴν ἀλήθειαν, καὶ ἡ ἀλήθεια ἐλευθερώσει ὑμᾶς.

—John 8:32

IN the previous chapter, it was shown that a backpropagation artificial neural network was able to successfully identify the compositions of the first three atomic layers of the (100) surface of a crystal of $Ni_{50}Pd_{50}$ alloy from its I-V spectrum. This required, however, network training data that was generated using the interlayer spacings that were known ahead of time. It also required knowing the correct compositions and interlayer spacings ahead of time in order to determine the inner potential energy shift.

This chapter will describe the final phase of this Dissertation: testing the ability of the neural network to identify all surface structure parameters (three layer compositions and two interlayer spacings) without a foreknowledge of the results of a conventional exhaustive search. Furthermore, the results in the previous chapter involved training the

network with layer compositions that varied over a limited range; this chapter will discuss an improvement in which the network is trained on structures whose layer compositions vary over the full range 0–100% Ni, and whose interlayer spacings span a wider range than before.

6.1 Determination of Interlayer Spacings

As a first step toward this goal, the the neural network was tested for its ability to recognize the interlayer spacings alone, with the atomic layer compositions held constant at their known values (as determined by a conventional search). Training data was generated from LEED dynamical calculations, setting the compositions of the first three atomic layers to their "known" values as determined by a conventional search [34]: 20% Ni in the top layer, 100% Ni in the second layer, and 34% Ni in the third layer. The interlayer spacings were allowed to vary over the range of -7% to $+2\%$ of the bulk spacing of 1.87 Å, in increments of 1%, for a total of 100 training structures. As was done in the previous chapter, the Pendry Y-function of the I-V curves was used to train the network, rather than the I-V curves themselves.

The network quickly converged on this calculated data after 100 training epochs. The trained network was then shown the Y-function of the experimental I-V curve, including a -4.7 eV inner potential energy shift that had been determined by comparison with the conventional results. Table 6.1 shows the network results, which are seen to be well within the error bars of the conventional search.

Layer	Conventional Results		Network Results	
	%	Å	%	Å
d_{12}	-1.8 ± 1.2	1.836 ± 0.022	-1.68	1.8386
d_{23}	-4.1 ± 1.8	1.793 ± 0.032	-3.59	1.8028

Table 6.1: Network test results for determination of interlayer spacings. The search range for both interlayer spacings was -7% to $+3\%$.

6.2 Determination of Compositions Over the Full Training Range

Having verified the network's ability to successfully determine interlayer spacings after having been trained on a relatively wide range of spacings, the next step toward improving the network was to test its ability to identify the layer compositions after training the network over the full range of possible compositions. In the previous chapter, it was shown that the network was able to correctly identify the compositions of the top three atomic layers of the $Ni_{50}Pd_{50}(100)$ surface after training the network on a moderately wide range of possible compositions. To further test the network's ability to identify surface layer compositions, a set of network training data was created using the full range of possible compositions: 0–100% Ni in each of the top three atomic layers. The results of the training after 10,000 epochs are shown in Table 6.2. The interlayer spacings in this case were held constant at the values determined by a conventional exhaustive search: -1.8% of bulk between the first and second layers, and -4.1% of bulk between the second and third layer.

Parameter	Training Range		Training Interval	Results Expected	Results Network
C_1 (%Ni)	0	100	10	20 ± 11	17.03
C_2 (%Ni)	0	100	10	100^{+0}_{-26}	91.34
C_3 (%Ni)	0	100	10	34 ± 14	20.98

Table 6.2: Network test results for determination of atomic layer compositions (full range of training data).
The experimental data was corrected for the -4.7 eV inner potential energy shift, and the training data consisted of 666 structures (every second structure of the $11^3 = 1331$ combinations of compositions). The network was trained for 10,000 epochs.

6.3 Simultaneous Determination of Compositions and Interlayer Spacings

The next phase of development of the neural network was to test its ability to simultaneously recognize *both* surface compositions and interlayer spacings. Since this involves varying three composition parameters and two interlayer spacings, this creates a much more stringent test for the network than the previous tests. To approach this problem, an initial test checked the network's ability to identify all five structure parameters for theoretically calculated data. This was followed by a test in which the calculated data was replaced by experimental data, and the network was trained over a restricted search range. The final test used experimental data with the network trained over a full search range.

Parameter	Training Range		Training Interval	Results Expected	Results Network
C_1 (%Ni)	0	100	20	20 ± 11	18.98
C_2 (%Ni)	0	100	20	100^{+0}_{-26}	100.30
C_3 (%Ni)	0	100	20	34 ± 14	34.70
Δd_{12} (%)	-7	$+3$	2	-1.8 ± 1.2	-2.05
Δd_{23} (%)	-7	$+3$	2	-4.1 ± 1.8	-4.07

Table 6.3: Network test results for simultaneous determination of compositions and interlayer spacings, with network shown calculated data.
The training data consisted of 7776 structures, and training was for 10,000 epochs.

6.3.1 Calculated Data

For this test, standard computer codes [11] were used to generate dynamically calculated I-V spectra for the (10), (11), and (20) beams diffracted from $Ni_{50}Pd_{50}$(100) for a wide range of structures: compositions of 0–100% Ni in the top three atomic layers and interlayer spacings of -7% to $+3\%$ of the bulk spacing of 1.87 Å. These spectra were then used to train the network for 10,000 training epochs. The same computer codes were used to generate I-V spectra for the structure found by a conventional search (C_1=20% Ni, C_2=100% Ni, C_3=34% Ni, $\Delta d_{12} = -1.8\%$, $\Delta d_{23} = -4.1\%$) and this data was shown to the trained network. The network successfully determined all five of the structure parameters to well within the error bars, as shown in Table 6.3.

As with other network runs described in this chapter, the network was trained on the Pendry Y-function of the I-V curves, rather than on the I-V curves themselves, because of the improvement in network performance described in Chapter 5.

In order for the network to simultaneously identify both the layer compositions and interlayer spacings, the network software had to be modified for this case so that scaling of the outputs from the network (Section 3.5) could be performed separately for each output, rather than having a single set of scaling parameters for all network outputs. This is because the "true" (unscaled) network outputs may range from 0–100 (%) for the layer compositions, while the interlayer spacings only range from a few percent around 1.87 (Å). Scaling each output separately allows for the differences in units for each output and allows the network to treat each output on an equal footing. The scaled outputs range between 0 and 1, regardless of the range or units of the unscaled outputs.

6.3.2 Experimental Data (Restricted Range)

For the next test, a set of training data was calculated in which the top three layer compositions and top two interlayer spacings were all varied over the restricted ranges shown in Table 6.4. As shown by the table, the network was able to successfully recognize all five search parameters to well within the conventional error bars in this case as well. The experimental data was shifted -4.7 eV (toward lower energies) in this case to allow for the inner potential.

6.3.3 Experimental Data (Full Range)

The final step in evaluating the network's ability to simultaneously recognize all five search parameters was to expand the range of the data used to train the network. In this case, the network was trained using layer compositions of 0–100% Ni (in steps of 20%) for all three surface layers, and the interlayer spacings Δd were searched from

Parameter	Training Range		Training Interval	Results Expected	Results Network
C_1 (%Ni)	0	50	10	20 ± 11	12.13
C_2 (%Ni)	70	100	15	100^{+0}_{-26}	95.60
C_3 (%Ni)	30	60	15	34 ± 14	35.85
Δd_{12} (%)	-3	-1	1	-1.8 ± 1.2	-1.61
Δd_{23} (%)	-5	-3	1	-4.1 ± 1.8	-3.97

Table 6.4: Network test results for simultaneous determination of compositions and interlayer spacings (restricted range of training data).
The experimental data was corrected for the -4.7 eV inner potential energy shift, and the training data consisted of 486 structures.

-7% to $+3\%$ of the bulk layer spacing in steps of 2%. Larger intervals were used for the calculated data (20% for compositions and 2% for interlayer spacings) than had been used previously in order to keep the number of calculated structures in the network training data to a reasonable number. The initial results, shown in Table 6.5, show that the network was able to identify four of the five search parameters to well within the conventional error bars. The network was not, however, initially able to correctly identify the top layer composition C_1 to within the error bars of the conventional exhaustive search in this case.

It was first thought that the network's inability to correctly identify the top layer spacing might be due to the large intervals in different values of C_1 used in the training data: the network had only been trained on examples that showed 0, 20, 40, 60, 80, and 100% Ni in the top atomic layer. Because changes in the top layer should have a larger effect on I-V data than deeper layers due to the surface sensitivity of LEED, these large intervals may result in changes in the I-V curve that are too large for the network to be

Parameter	Training Range		Training Interval	Results Expected	Results Network
C_1 (%Ni)	0	100	20	20 ± 11	-0.34
C_2 (%Ni)	0	100	20	100^{+0}_{-26}	88.25
C_3 (%Ni)	0	100	20	34 ± 14	33.84
Δd_{12} (%)	-7	$+3$	2	-1.8 ± 1.2	-1.89
Δd_{23} (%)	-7	$+3$	2	-4.1 ± 1.8	-4.44

Table 6.5: Initial network test results for simultaneous determination of compositions and interlayer spacings (full range of training data).
The experimental data was corrected for the -4.7 eV inner potential energy shift. The training data consisted of 341 calculated structures, each of which had an average composition of 40–60% Ni for the top three atomic layers.

able to successfully "interpolate" between them. With this in mind, a new set of training data was created in which the top layer composition was varied at intervals of 10% instead of 20%. It was discovered, however, that this change had no appreciable effect on the network's ability to correctly recognize the top layer composition C_1.

The difficulty with the network's ability to identify the top layer composition was ultimately traced to the network learning rate parameter ϕ, which is the factor by which changes to network weights are multiplied when the weight updates change sign from epoch to epoch (Eq. 3.41). The value of ϕ had previously been set to 0.5, which causes the changes to the network weights to decrease rapidly (and thus increase convergence speed) for a wide variety of problems [32]. In this case, however, it caused the network weights to converge too quickly, so that the weights converged to a *local* minimum in the error surface for the problem, rather than the global minimum.

Parameter	Training Range		Training Interval	Results Expected	Results Network
C_1 (%Ni)	0	100	20	20 ± 11	28.51
C_2 (%Ni)	0	100	20	100^{+0}_{-26}	83.78
C_3 (%Ni)	0	100	20	34 ± 14	40.41
Δd_{12} (%)	-7	$+3$	2	-1.8 ± 1.2	-3.93
Δd_{23} (%)	-7	$+3$	2	-4.1 ± 1.8	-1.66

Table 6.6: Improved network test results for simultaneous determination of compositions and interlayer spacings with $\phi = 0.85$ (full range of training data).
The experimental data was corrected for the -4.7 eV inner potential energy shift. The training data consisted of 7776 calculated structures, and the learning rate parameter ϕ was set to 0.85. Results are shown at 4700 training epochs.

In order to slow the network convergence to allow it to find the global error minimum, the learning rate parameter ϕ was increased to 0.85 and it was re-trained with the same I-V spectra covering the same wide range of compositions and interlayer spacings. Upon being shown the experimental spectra, the network was this time able to correctly identify the five surface structure parameters, as shown in Table 6.6.

The interlayer spacings found by the network in Table 6.6 show an interesting feature: although the results are close to the conventional search error bars, they are much closer to conventional search results for the *other* layer. In other words, the network's result for Δd_{12} is close to the conventional result for Δd_{23}, and the network's result for Δd_{23} is close to the conventional result for Δd_{12}. This is, in part, an artifact of the way in which the interlayer spacings are defined (as a percent difference from the bulk spacings). If the interlayer spacings are instead defined by the distance of each layer from the top layer,

then the network returned a distance of -3.93% bulk to the second layer and -5.59% bulk to the third layer, while the conventional search gave $-1.8 \pm 1.2\%$ bulk to the second layer and $-5.9 \pm 3.0\%$ bulk to the third layer. In other words, the network returns the same third layer distance to well within the conventional error bars, but gives a second layer distance just outside the conventional error bars.

As a further investigation of this result, theoretical I-V curves were calculated with both the conventional search results and the conventional search results with the values of d_{12} and d_{23} switched. As shown in Fig. (6.1), switching the two interlayer spacings results in very similar I-V curves. This example illustrates a difficulty common to all surface structure determination using LEED I-V spectra: since I-V data are not necessarily unambiguous, there may be some ambiguity in the structure determination.

6.4 Further Improvements

Another attempt to improve on this result involved the use of independent information in creating network training data. Since the bulk composition of the alloy is 50% nickel, one would expect that the mean composition of the first few atomic layers would also be close to 50% nickel. Accordingly, a set of training data was created which consisted entirely of structures whose mean composition in the first three atomic layers ranged from 40–60% nickel. While this reduced the size of the training data (and thus the training time) by roughly a factor of 2, it did not significantly affect the results; the network was still able to find esseentially the same results as the exhaustive search for all surface parameters.

One difficulty encountered in using a network to simultaneously determine five structural parameters is that the network would tend to diverge when given too many training

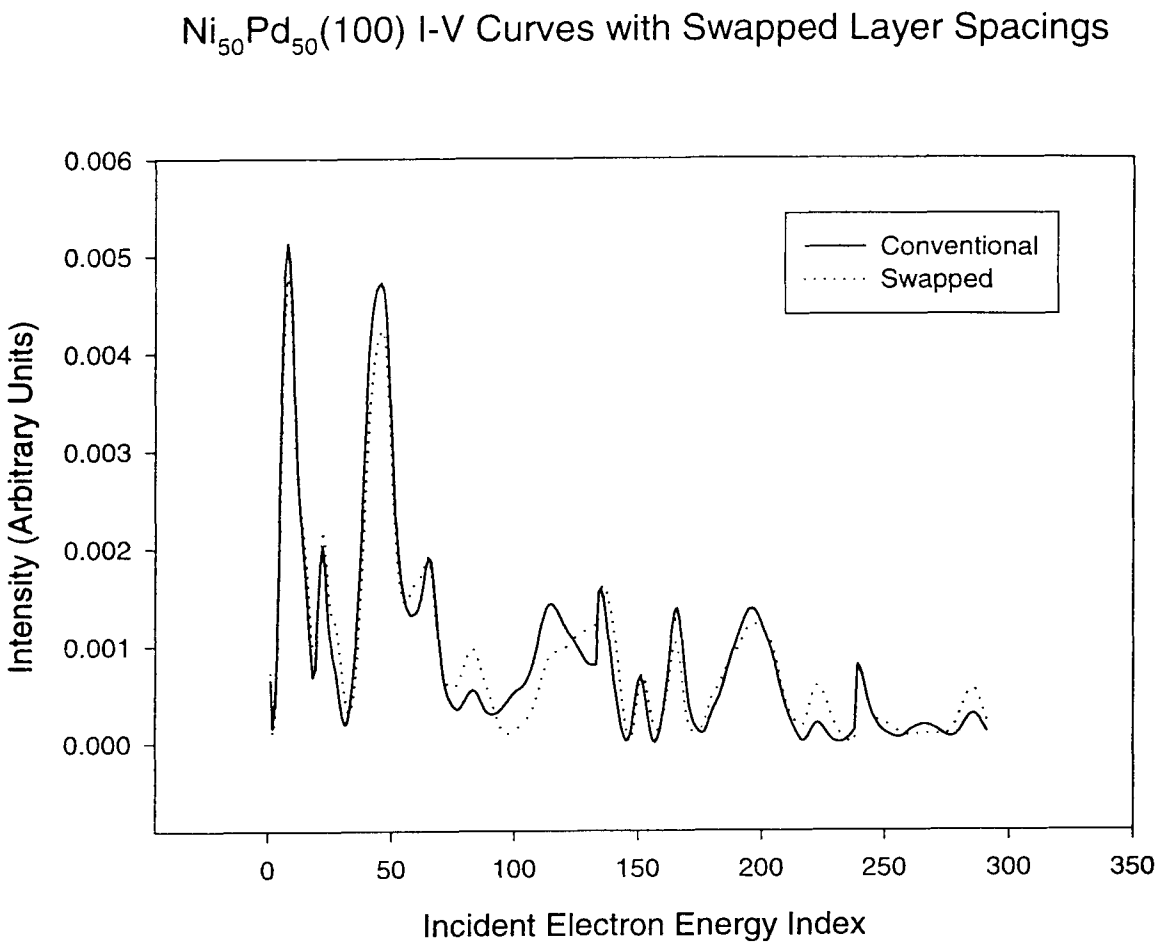

Figure 6.1: Theoretical I-V curves for $Ni_{50}Pd_{50}(100)$ (with "swapped" layer spacings). The curve labeled "conventional" is for $\Delta d_{12} = -1.8\%$, $\Delta d_{34} = -4.1\%$; the curve labeled "swapped" is for $\Delta d_{12} = -4.1\%$, $\Delta d_{34} = -1.8\%$.

examples. If, for example, one creates a set of training data consisting of five varying parameters (C_1, C_2, C_3, d_{12}, d_{23}) sampled at six points each, this creates a set of training data containing $6^5 = 7776$ different structures. It was found that training a network with data sets of this size would cause the (un-scaled) network outputs to saturate at either 0 or 1, thus giving useless scaled outputs.

Several strategies were attempted to try to remedy this situation. At first it was thought that the additional data being shown to the network might require more hidden nodes so that more weights would be available to store the additional information in the network. Many attempts were made to train the network with 7776 examples in the training data, in which the number of hidden nodes in the network was increased from 30 to as many as 250 nodes. None of these attempts succeeded in making the network converge for these large training data sets.

Another attempt at fixing this difficulty was to decrease the initial network weights. Network weights are initialized to small random values; nominally, the LEEDNET program uses random numbers $\sim 10^{-4}$. Several attempts were made to decrease these initial weights to numbers as low as $\sim 10^{-9}$, with no effect on the convergence of the network. Attempts to simultaneously increase the number of hidden nodes and lower the initial network weights also had no effect; the network continued to give saturated unscaled outputs of either 0 or 1.

One way to circumvent this convergence problem is to simply decrease the size of the training data set. Experience throughout this Dissertation shows that the network will converge well for training data sets consisting of ~ 500 spectra or fewer, but the outputs become saturated if much more training data ($\sim$ 1000 or more spectra) are used. To decrease the size of the data set, a program was written which samples a data set at fixed

intervals to produce a new data set which may then be used to train the network. One may, for example, have a set of I-V spectra in which each of five surface parameters varies over six different values, for a total of $6^5 = 7776$ spectra. If this data set is sampled so that only every 13th spectrum is used, a data set is produced consisting of 599 structures, which does converge. As noted in Chapter 4, however, one must be careful to sample the original data set at an interval which will sample the parameter space well so that each parameter in the sampled data is allowed to vary over its entire range.

This difficulty with network convergence when using large sets of training data was ultimately traced to the "batch" learning used by the network, as described in section 3.9. When training the network on a set of I-V spectra, the entire set of training data is shown to the network, one spectrum at a time. The network error is computed after each spectrum, and all errors are added together for the entire data set to find the network's net error, which is then used to adjust the network weights. Using batch learning, rather than correcting the network weights after each individual spectrum, prevents the network training from being unduly biased in favor of those examples which it was shown most recently.

The network error after each batch is updated according to the adaptive learning rate given by Eq. (3.41):

$$e_m = \begin{cases} e_{m-1} + \kappa & \text{if} \quad d_m f_m > 0 \ , \\ e_{m-1} \times \phi & \text{if} \quad d_m f_m \leq 0 \ . \end{cases}$$

The actual weight change is then proportional to the adaptive learning rate e_m. If a large number of spectra is used to train the network, then the network errors at the end of one "batch" of training will tend to be larger than for a smaller training data set. Accordingly,

the adaptive learning rate parameter κ should be scaled according to the number of spectra in the training data. This was done, and it was found that decreasing the value κ for larger numbers of spectra in the training data does, in fact, allow the network to converge properly.

However, experience with the network has generally shown that better results are achieved by training the network with a reduced set of training data, rather than using the full set of training data with a reduced learning parameter κ. For example, consider the problem of using the network to determine the top three atomic layer compositions, as discussed in the previous section. I-V spectra may be calculated for this problem for which the compositions of each of the top three atomic layers varies between 0 and 100% Ni, in steps of 10%, for a total of $11^3 = 1331$ structures. To permit the network to converge on the training data, one could either train the network with all 1331 structures using a small κ (e.g. $\kappa = 0.01$), or one could create a smaller set of training data by using only every third spectrum in the training data, thereby reducing the training data set to 444 structures. Both of these options were tried with a network training for 200 epochs, and the results are shown in Table 6.7. As shown in the table, better results are obtained by reducing the size of the training data set to less than ~ 500 spectra. As a bonus, the reduced-size training set case requires only one-third the training time of the full-size case.

6.5 Energy Shift

The inner potential energy shift (described in Chapter 2) is a rigid shift of the I-V spectrum along the energy axis due to the presence of a potential step at the metal surface. Through-

Parameter	Training Range		Training Interval	Results Full	Reduced	Conventional
C_1 (%Ni)	0	100	10	31.18	13.79	20 ± 11
C_2 (%Ni)	0	100	10	61.48	90.01	100^{+0}_{-26}
C_3 (%Ni)	0	100	10	26.80	31.27	34 ± 14

Table 6.7: Network test results for determination of atomic compositions, comparing two methods for ensuring network convergence.
The column labeled "Full" shows the results for the full training data set of 1331 spectra, with a smaller learning rate parameter $\kappa = 0.01$. The column labeled "Reduced" is for a reduced training data set of 444 spectra, with learning rate parameter $\kappa = 0.1$. The "Conventional" column gives the results of a conventional exhaustive search [34]. The atomic interlayer spacings were set at their conventionally-determined values ($d_{12} = -1.8\%$ of bulk, $d_{23} = -4.1\%$ of bulk). Network training was for 200 epochs in both cases.

out most of this study, this energy shift was known from a comparison with the known surface structure, as determined by a conventional calculation. A predicted I-V spectrum was dynamically calculated using the surface parameters found by a conventional search [34]; the Pendry Y-function of this spectrum and the experimental spectrum was then found, and the two spectra were shifted with respect to each other in steps of 0.1 eV. For each shift, the root-mean-squared error between the two spectra was computed; a value of -4.7 eV was found to be the shift which produced the minimum error, and thus the best agreement between theory and experiment.

If the inner potential energy shift of the sample being investigated is by some means known *a priori*, it may be used to correct the experimental data before showing the experimental data to the network. Even a rough approximation of inner potential energy

Energy	Structure Parameter				
Shift (eV)	$C_1(\%)$	$C_2(\%)$	$C_3(\%)$	$\Delta d_{12}(\%)$	$\Delta d_{23}(\%)$
-1	30.284	61.957	26.019	-4.78	-2.18
-2	28.693	71.246	30.698	-4.67	-2.13
-3	28.145	79.147	35.445	-4.51	-1.98
-4	28.213	83.653	38.836	-4.25	-1.80
-5	28.584	83.299	40.655	-3.82	-1.63
-6	28.726	77.730	41.110	-3.15	-1.42

Table 6.8: Network test results for a variety of inner potential energy shifts. Note the relative insensitivity of the network results to the value of the energy shift chosen.

shift is sufficient for the network to return adequate results. Table 6.8 shows the network results after being shown experimental data for $Ni_{50}Pd_{50}(100)$ shifted for energy shifts of -1 eV to -6 eV. As shown in the table, the parameters returned by the network are all similar to those returned for the optimum shift (found by comparison with the conventional search results, Table 6.6), regardless of the energy shift used.

6.6 Transferability

One potential advantage of artificial neural networks is that once trained for one crystal, it may be able to recognize the surface parameters in similar materials. To test this idea, the network was first trained on the $Ni_{50}Pd_{50}(100)$ surface described earlier. *I-V* spectra were then calculated for a similar crystal surface, $Cu_{50}Pd_{50}(100)$, by repeating the dynamical calculations for $Ni_{50}Pd_{50}(100)$ using the same structure parameters, but

Parameter	Training Range		Training Interval	Results Expected	Results Network
C_1 (%Ni)	0	100	20	20 ± 11	14.08
C_2 (%Ni)	0	100	20	100^{+0}_{-26}	103.30
C_3 (%Ni)	0	100	20	34 ± 14	39.29
Δd_{12} (%)	-7	$+3$	2	-1.8 ± 1.2	-0.87
Δd_{23} (%)	-7	$+3$	2	-4.1 ± 1.8	-4.60

Table 6.9: Network test results for simultaneous determination of compositions and interlayer spacings for $Cu_{50}Pd_{50}(100)$.
The network was trained on dynamically calculated data for $Ni_{50}Pd_{50}(100)$, then shown dynamically calculated spectra for $Cu_{50}Pd_{50}(100)$ for which C_1 = 20%Ni, C_2 = 100%Ni, C_3 = 34%Ni, $\Delta d_{12} = -1.8\%$ bulk, and $\Delta d_{23} = -4.1\%$ bulk.

with phase shifts for CuPd. This data was shown to the network; the results, shown in Table 6.9, demonstate the network's ability to successfully recognize the CuPd structure parameters in this case.

6.7 Discussion

A backpropagation neural network has been shown to be able to correctly identify two interlayer spacings (given the layer compositions) and the three top layer compositions (given the interlayer spacings) to within the error bars specified for a conventional exhaustive search. The network was also able to find all five parameters simultaneously; this required adjusting the network's learning rate parameters to accommodate the larger number of training examples and to ensure that the convergence was sufficiently slow

that the global minimum in the error surface was not missed. The network's results were shown to be relatively insensitive to the value chosen for the inner potential energy shift.

Neural networks offer some advantanges in lessening the computation time needed needed to perform a surface structure determination. The time required to perform an exhaustive search scales exponentially with N (i.e. as ε^N), where N is the number of structure parameters being sought and ε is related to the number of points in each dimension of parameter space [22]. Simulated annealing algorithms, on the other hand, scale as a polynomial in the number of degrees of freedom [36], i.e. as N^5 in this case.

The computational effort required to perform a neural network search scales exponentially with the number of search parameters as does an exhaustive search, although the number of structures in each dimension of parameters space can be smaller than for an exhaustive search. That is, the computation time required will scale as ε_{nn}^N, where $\varepsilon_{nn} < \varepsilon$.

The savings in computational time of an artificial neural network over an exhaustive search is due in large part to the coarser grid of dynamically calculated data compared to the grid needed for the exhaustive search. This ability to return results using a coarse parameter grid is related to the network's ability to "interpolate" between structures on which it has been trained. One must therefore provide the network with I-V spectra on a grid just fine enough to permit an accurate interpolation. As discussed earlier, it was found that a 20% spacing in the compositions and a 2% spacing in the grid of interlayer spacings was adequate for the network to successfully recognize the experimental I-V spectra.

A neural network search thus offers some advantages over an exhaustive search. It can be trained on a relatively coarse grid of structures in parameter space, so that fewer dynamical LEED spectra need to be calculated, resulting in a substantial savings of

computer time. Once trained, the network can be shown several experimental spectra of different samples and immediately return the structure parameters, without a need to search the parameter space again. Finally, the network's ability to recognize the structure parameters for CuPd after having been trained on NiPd spectra illustrates a potential for transferability of a trained network to other similar materials.

6.8 Concluding Remarks

6.8.1 Future Directions

Standard backpropagation neural network designs use a gradient descent method to update the network weights. Future improvements in the use artificial neural networks for LEED structural determination might be realized by finding improved methods for updating these weights. For example, one might consider developing a backpropagation network in which network weights are updated according to a simulated annealing algorithm, rather than the standard gradient descent equations.

Other future work might center on the testing the ability of a network to recognize reconstructed surfaces, or surfaces with adsorbate layers. In addition, there are many questions in neural network theory that remain unanswered (such as the *a priori* determination of the number of required hidden nodes) that would be of great practical interest in the application of networks to practical problems.

6.8.2 Summary

A backpropagation artificial neural network has been shown to identify the atomic layer compositions and interlayer spacings for the (100) surface of $Ni_{50}Pd_{50}$ in agreement with the results found by a conventional exhaustive search. The best results were found for problems involving fewer search parameters, such as finding two interlayer spacings or three compositions alone. Successfully finding five parameters simultaneously proved a bit more difficult, and required some adjustment of the network training parameters.

The surface used in this study, that of $Ni_{50}Pd_{50}$, is for a binary alloy and has five significant surface parameters that are of interest. Simpler structures, such as elemental metal crystal surfaces, would involve fewer search parameters and would therefore be good candidates for structure determination with a neural network. In such cases, use of a neural network offers several advantages. Relatively few (~ 500) I-V spectra need be calculated; since each such spectrum involves fairly complicated and time-consuming dynamical calculations, this can result in a significant time savings. The network training (using the Pendry Y-function) typically converges after just a few hundred training epochs, often involving less than one hour of computer time. Once trained, the network can immediately identify surface parameters when shown an experimental I-V spectrum. This can result in a particularly significant time savings if data from several similar samples are to be analyzed.

Appendix A

LEEDNET User's Guide

LEEDNET is an artificial neural network program that implements a backpropagation network for the analysis of low-energy electron diffraction (LEED) I-V spectra. Using standard computer codes [11], one can generate theoretical I-V spectra for a number of plausible surface structures. LEEDNET will allow this data to be formatted into its own data format, after which it can be used to train the network on these I-V spectra. Once trained, the network may be shown an experimental I-V spectrum and asked to identify the surface parameters.

LEEDNET has a command-line interface which allows it to be run either interactively or as a batch job through the execution of script files. The interface allows the network to be readily customized for the problem at hand, and it includes built-in help files.

LEEDNET is written in standard ANSI C, and should be highly portable to a wide variety of computer platforms. For this dissertation, LEEDNET was run on a Silicon Graphics computer running a UNIX operating system.

A.1 Running LEEDNET

To run LEEDNET in interactive mode under UNIX, type

```
$ leednet
  LEEDNET Version 1.04h
[1]>
```

LEEDNET begins by displaying the program version number; this may also be displayed at any time by typing the `ver` command at the command prompt. The string `[1]>` is the command prompt, and indicates that LEEDNET is waiting for a command to be input. The number between the square brackets begins at 1 when LEEDNET is started, and increments by 1 each time a command is entered.

On-line help is available for LEEDNET at any time by typing `help` at the command prompt:

```
[1]> help
```

Typing `help` without arguments prints a list of the available LEEDNET commands. One may also type `help env` at the command prompt to see a description of available environment variables (q.v.), or `help` *cmdname* for detailed help on a specific command.

To exit LEEDNET, simply type either `quit` or `exit` at the command prompt:

```
[2]> quit
$
```

LEEDNET may also be run in batch mode; this is particularly useful for network training, which can require several hours of computer time. To run LEEDNET in batch mode, one creates a scripting file with a "`.scr`" file extension which contains the LEEDNET commands to be run. One then passes the file name (sans `.scr` extension) on

the LEEDNET command line and submits LEEDNET as a batch job. If one creates a scripting file called `run001.scr`, for example, it can be run under UNIX as a batch job using

```
$ leednet run001 &
```

One should always end a scripting file with a `quit` command so that the batch job terminates the LEEDNET program.

A.2 Environment Variables

The operation of the LEEDNET program and its network parameters may be customized by the setting of *environment variables*. Subsection A.8 summarizes each of the available environment variables and their use. The values of all environment variables may be displayed at any time by typing `set` (without arguments) at the LEEDNET command line.

The value of an environment variable may be changed by using the "set" command:

set <varname>=<value>

A.3 File `LEEDNET.INI`

Whenever the LEEDNET program begins, it will look for a file called `LEEDNET.INI` in the current directory. This file is used to start LEEDNET with any desired values in the environment variables so that they need not be set manually each time LEEDNET is started.

The lines in the LEEDNET.INI file may be of three types:

(1) Any line beginning with a semicolon (;) is a comment line and is ignored.

(2) A line of the form <varname>=<value> sets environment variable <varname> to the value <value> upon starting LEEDNET.

(3) An output node scaling may be set with a line of the form scale <n> <tmin> <tmax>. See Section A.7 for a description of the scale command.

A.4 Formatting the Data

The first step in using LEEDNET is to format the dynamically calculated *I-V* data into a format usable to the program. LEEDNET data files are plain text files which contain three header lines followed by the intensity data and network parameters:

```
Number of input nodes (I)
Number of output nodes (J)
Number of structures in data file (N)
I lines of intensity data      > repeated
J lines of output parameters   > N times
```

Note than energy values are not explicitly used by the network.

To format the dynamically calculated data, begin by writing a C function to convert the dynamically calculated *I-V* data into this format. Appendix C shows an example program, FORMAT01.C. This function should then be linked into the LEEDNET main program, following the example of format01() shown in the LEEDNET listing in Appendix B. Once LEEDNET has been properly compiled and linked, it can by run and the

`format` command used to format the data. For example, suppose the original dynamically calculated I-V data is in a file called `iv.dat`, and we wish to format the data, placing the output into a new file `iv.fmt` using function `format01()`. The following LEEDNET commands would be used:

```
[1]> set original=iv.dat
[2]> set formatted=iv.fmt
[3]> set function=1
[4]> format
```

The `original` environment variable specifies the file name of the original data file; the `formatted` variable specifies the file name of the output data file; and `function` specifies which of several possible formatting functions is to be used (`1` indicates `format01()`).

If a large amount of data is involved so that a large amount of time would be needed to format the data, it may be useful to place these commands into a scripting file and to run LEEDNET as a batch program, as described earlier.

A.5 Training the Network

Once the I-V data has been properly formatted, the network can be trained using the `train` command. Since network training generally involves significant amounts of computer time, network training is usually done through a scripting file, running LEEDNET as a batch program. At a minimum, this scripting file should contain commands to set the `training` environment variable to specify the name of the file containing the training data; to set the `epochs` variable to specify the number of training epochs; and the `save` variable to save the network weights once training is completed. A typical scripting file is shown below.

```
; Network run #77
set training=/umbc/research/rous/Simpson/ivyr.055
set weightfile=/umbc/research/rous/Simpson/weights.077
set errordat=/umbc/research/rous/Simpson/error.077
set hidden=40
set epochs=10000
set delprt=100
set delanalyze=100
set errorlim=1.0e-17
set adaptive=1
set kappa=0.001
set phi=0.85
scale 1 0 100
scale 2 0 100
scale 3 0 100
scale 4 1.7391 1.9261
scale 5 1.7391 1.9261
train
save
quit
```

In this example, environment variables are set to specify the name of the file containing the (formatted) training data; the file name to hold the final network weights; and the file name to hold the network training error (at intervals of `delprt` epochs). This scripting file also specifies a network with 40 hidden nodes, that adaptive learning should be used, and gives `scale` commands to properly scale the network outputs. The `train` command near the end of the script actually performs the network training; when finished, the `save` command saves the network weights into the file specified by `weightfile`.

Two features in LEEDNET allow the user to keep track of the training of the network. The `delanalyze` environment variable specifies the interval at which to save the intermediate network weights. The network weights will be saved every `delanalyze`

epochs into a file called `i`*nnnnnnn*`.wgt`, where *nnnnnnn* is the number of epochs. This allows the user to monitor the performance of the network before the final training is completed. Also, the current training epoch and training error are saved to a a file given by the `statusfile` variable every `delprt` epochs, so that this file may by displayed at the terminal at any time to check the training status.

A.6 Using the Trained Network

Once the network has been trained, the network is used by loading the appropriate weights, specifying an input file (containing one set of I-V spectra which the network will be asked to identify), and using the `ask` command to run the network. For example,

```
[1]> set weightfile=weights.077
[2]> load
[3]> set single=expdata.dat
[4]> scale 1 0 100
[5]> scale 2 0 100
[6]> scale 3 0 100
[7]> scale 4 1.7391 1.9261
[8]> scale 5 1.7391 1.9261
[9]> ask
```

Here the `load` command loads the network weights from `weights.077`, `single` specifies the name of the data file containing the I-V spectra to be identified (e.g. experimental data), and the `ask` command runs the network and gives the network's results.

A.7 Command Reference

alloc

Allocates memory for the network. The memory allocated is based on the number of input, hidden, and output nodes currently defined (environment variables `inputs`, `hidden`, and `outputs`. The memory is dynamically allocated on the heap.

analyze

Analyze a network from its training data. The `analyze` command reads the number of inputs, outputs, and structures from the `training` file. It then loads network weights from `weightfile`. Finally, it shows each structure in `anlinfile` to the network and compares the network's outputs to the expected outputs in the training data. The statistics on the final results are sent to the file defined by the environment variable `anloutfile`.

ask

Asks a trained network to process input. Shows the file defined by environment variable `single` to the network, runs the network, and displays its outputs.

debeam

Separate dataset into individual beams. The file defined by the environment variable `formatted` contains the LEEDNET-format I(V) data, variable `beams` should be set to the number of separate beams in each spectrum, and variable `inputs` should be set to the

total number of points in each spectrum. Each I(V) spectrum is divided into `beams` equal data sets; the output files are named `beam.nnn`.

`dump`

Dump network weights. The current values of all network weights are dumped to the file whose name is given by the environment variable `dumpfile`.

`dw`

Display a single network weight. The command syntax is: `dw v|w i j` where v asks to display an input-to-hidden weight; w asks to display a hidden-to-output weight; and i,j are the weight indices.

`exit`

Quit LEEDNET. The `exit` and `quit` commands are equivalent.

`format`

Format data into LEEDNET format. I(V) data is re-formatted from its original format into the format used by LEEDNET. Several different formatting functions may be available; they are selected using the environment variable `function`. (`original` → `formatted`)

help

Help on LEEDNET commands. Type `help` for a list of available commands. Type `help cmd` for detailed help on command "cmd". Type `help env` to display help on environment variables.

load

Load network weights. Network weights are loaded from the file defined by environment variable `weightfile`. The number of input nodes, output nodes, and structures (variables `inputs`, `outputs`, and `structures`) will also be loaded.

mem

Display memory usage report. Type `mem` to display the current network memory usage. Type `mem <inputs> <hidden> <outputs>` to display the memory that would be required for a network of the specified number of input, hidden, and output nodes.

merge

Merge several data files together. The `merge` command will prompt for the name to be given to the output file. It will then ask for the names of the input files, each of which should contain one structure (created, for example, by the `separate` command). Enter a carriage return by itself after the last file name is entered. When done, manually update line 3 of the output file (total number of structures).

plot

Generate I(V) plot data. The file given by the environment variable `formatted` is broken into spectra for individual structures, and I vs. E data is saved into files named `struct.nnn`. The data in each file is in ASCII format and is suitable for plotting with a spreadsheet program.

prune

Prune points from a data set. The data file specified by the environment variable `formatted` is "pruned" by keeping only every n-th point, where n is given by the environment variable `skip`. The result is saved in the file specified by the environment variable `prune`. (`formatted` $\rightarrow$ `pruned`)

quit

Quit LEEDNET. The `exit` and `quit` commands are equivalent.

randomize

Initialize random number generator from system time.

rebeam

Combine data sets for individual structures into one file. The user is prompted for the names of the files containing the individual beam data, and the beam data is combined into a single file whose name is specified by the `beamoutfile` environment variable.

resume

Resume a training run. To resume a training run, type: set weightfile=<weight filename> set resumeepoch=<epoch number> resume

save

Save network weights. Network weights are saved to the file defined by environment variable `weightfile`. The number of input nodes, output nodes, and structures (variables `inputs`, `outputs`, and `structures`) will also be saved.

scale

Display or set network output scaling constants. Type `scale` with no arguments to display the current values of all scaling constants. Type `scale <n> <tmin> <tmax> [<smin> <smax>] [n]` to calculate and save scaling constants for output <n> for `true` values ranging between <tmin> and <tmax>. The optional minimum and maximum scaled values <smin> and <smax> default to 0.1 and 0.9, respectively. If an `n` is specified, the scaling is calculated but not stored. Type `scale <n> <value>` to perform true→scaled and scaled→true conversions of <value> for output <n>.

separate

Separate data in a data set. A single structure is isolated from the data set specified by the environment variable `formatted`. The environment variable `sepnum` should be set to the number (starting from 0) of the structure to be isolated. The `separate` command will

then place the spectrum for that structure into the file whose name is given by the `single` variable, and the remaining spectra will be placed into the file whose name is given by the `missing` variable. (`formatted` → `missing, single`)

`set`

Set/display environment variables. Type `set` with no arguments to display the current values of all environment variables. Type `set` <`varname`>=<`value`> to set an environment variable to a new value.

`status`

Show network training status. The network training status is peroidically stored in the file defined by the environment variable `statusfile`, provided the `status` variable is set to 1. The `status` command displays the contents of this file.

`train`

Train the network. The file specified by the `training` environment variable is used to train the network for "epochs" training epochs. Type `train [cont]` to continue training that has been stopped.

`ver`

Display LEEDNET version number.

A.8 Environment Variable Reference

`adaptive`: Adaptive learning rate flag (1=on, 0=off)

`anlinfile`: Analysis input file (for `analyze` command)

`anloutfile`: Analysis output file (for `analyze` command)

`beamoutfile`: Beam output file (for `rebeam` command)

`beams`: Number of beams in input (for `debeam` and `rebeam`)

`debug`: Debug mode on/off flag

`delanalyze`: Interval (epochs) to save network weights

`delprt`: Number of epochs to print msg and save error report

`dumpfile`: Network weight dump file name

`epochs`: Number of epochs to train the network

`errordat`: Error vs. epoch output file name

`errorlim`: Training error limit

`formatted`: Formatted data file name (LEEDNET format)

`function`: Selects a formatting function for `format` command

`helpfile`: Help file name

`hidden`: Number of network hidden nodes

`inputs`: Number of network input nodes

`kappa`: Kappa parameter for adaptive learning

`missing`: Missing I structure file name (for `separate` cmd)

`mu`: Momentum parameter for learning rate

`numpts`: Number of points in I(V) curve

`original`: Original data file name (for `format` command)

`outputs`: Number of network output nodes

`pause`: Training pause (epochs)

`phi`: Phi parameter for adaptive learning

`plotfile`: Plot data output file name

`pruned`: Pruned data file name (`prune` command)

`resumeepoch`: Resume epoch (for `resume` command)

`seed`: Seed for random number generator

`seedmult`: Random seed multiplier

`sepnum`: Number of structure to be isolated for `separate` cmd

`single`: 1-structure file name (`separate` and `ask` cmds)

`skip`: Number of points to skip for `prune` command

`status`: Training status on/off flag (1=on, 0=off)

`statusfile`: Training status file name

`structures`: Number of structures in data set

`theta`: Theta parameter to control averaging period

`training`: Training data file name

`weightfile`: File name under which to save or load network weights

Appendix B

Listing of Program LEEDNET.C

```
/* File  LEEDNET.C                                                          */

/**************************************************************************/
/*                                                                        */
/*                          L E E D N E T                                 */
/*                                                                        */
/*  This program implements an artificial neural network to recognize     */
/*  LEED I(V) curves.                                                     */
/*                                                                        */
/*  This source code is standard ANSI C and should be highly portable.    */
/*                                                                        */
/*  David G. Simpson                                                      */
/*  Department of Physics                                                 */
/*  University of Maryland, Baltimore County                              */
/*  Catonsville, Maryland                                                 */
/*                                                                        */
/*  July 10, 1996                                                         */
/*                                                                        */
/**************************************************************************/


/**************************************************************************/
/*                                                                        */
/*  Files used:                                                           */
/*                                                                        */
/*  Source code:                                                          */
/*     leednet.c     Main LEEDNET source code                             */
/*     format01.c    Custom function to format raw data to LEEDNET format */
/*     leednet.h     LEEDNET header file                                  */
/*                                                                        */
/*  Auxiliary file:                                                       */
/*     leednet.hlp   LEEDNET help file (for "help" command)               */
/*                                                                        */
/*  Optional initialization files:                                        */
/*     leednet.ini   Initialization file                                  */
/*                                                                        */
/**************************************************************************/


/**************************************************************************/
/*                                                                        */
/*  LEEDNET command summary                                               */
/*                                                                        */
/*  alloc       Allocate memory for the network                           */
/*  analyze     Analyze a network from its training data                  */
/*  ask         Ask a trained network to process input                    */
/*  debeam      Separate data set into individual beams                   */
/*  dump        Dump network weights                                      */
```

```
/*  dw          Display network weight                                      */
/*  exit        Quit LEEDNET                                                */
/*  format      Re-format data to LEEDNET format                            */
/*  help        Help on LEEDNET commands                                    */
/*  load        Load network weights                                        */
/*  mem         Display memory usage report                                 */
/*  merge       Merge data files together                                   */
/*  plot        Generate I(V) plot data                                     */
/*  prune       Prune points from a data set                                */
/*  quit        Quit LEEDNET                                                */
/*  randomize   Initialize random number generator from system time         */
/*  rebeam      Re-assemble data set from individual beams                   */
/*  resume      Resume a training run                                       */
/*  save        Save network weights                                        */
/*  scale       Set/display network output scaling constants                */
/*  separate    Separate data in a data set                                 */
/*  set         Set/display environment variables                           */
/*  status      Show network training status                                */
/*  train       Train a network                                             */
/*  ver         Display LEEDNET version number                              */
/*                                                                          */
/****************************************************************************/



/****************************************************************************/
/*                                                                          */
/*  Version History (Abbreviated)                                           */
/*  -----------------------------                                           */
/*  1.00a   Original version                                                */
/*  1.04a   Changed scalings to arrays to support 10 separate scalings.     */
/*  1.04b   Added range checking to above (1-9).                            */
/*  1.04c   Added "scale" command to leednet.ini.                           */
/*  1.04d   Added SEEDMULT environment variable                             */
/*  1.04e   Added format function #7                                        */
/*  1.04f   Added format function #8                                        */
/*  1.04g   Added new format function #8                                    */
/*  1.04h   Dressed up (more comments &c.) for dissertation.  Removed all   */
/*          "format" files except format01 for brevity.                     */
/*                                                                          */
/****************************************************************************/



/****************************************************************************/
/*  #includes                                                               */
/****************************************************************************/

#include <stdio.h>                      /* standard i/o                 */
#include <stdlib.h>                     /* standard library             */
#include <time.h>                       /* time functions               */
#include <math.h>                       /* math functions               */
#include <string.h>                     /* string functions             */
#include <ctype.h>                      /* character functions          */
#include "leednet.h"                    /* leednet-specific definitions  */



/****************************************************************************/
/*  #defines                                                                */
/****************************************************************************/

#define VERSION     "1.04h"             /* software version number      */
#define CMDSIZE     6                   /* num of cmd parameters + 1    */
#define ENVLINE     31                  /* size of environment line str  */
#define CMDLNSZ     80                  /* size of cmd line             */
#define INFINITY    1.0e+30;            /* effective infinity           */
#define SCALINGS    10                  /* size of scaling arrays       */



/****************************************************************************/
/*  macros                                                                  */
/*                                                                          */
/*  These function-like macros are used to access the dynamically allocated  */
/*  arrays.                                                                 */
/****************************************************************************/
```

```
#define X(i)            *(x+i)
#define V(i,j)          *(v+(i*(hidden+1))+j)
#define Z_IN(i)         *(z_in+i)
#define Z(i)            *(z+i)
#define W(i,j)          *(w+(i*(outputs+1))+j)
#define Y_IN(i)         *(y_in+i)
#define Y(i)            *(y+i)
#define YY(i)           *(yy+i)
#define T(i)            *(t+i)
#define DELTA_IN_J(i)   *(delta_in_j+i)
#define DELTA_J(i)      *(delta_j+i)
#define DELTA_K(i)      *(delta_k+i)
#define DEL_V(i,j)      *(del_v+(i*(hidden+1))+j)
#define DEL_W(i,j)      *(del_w+(i*(outputs+1))+j)
#define F_V(i,j)        *(f_v+(i*(hidden+1))+j)
#define E_V(i,j)        *(e_v+(i*(hidden+1))+j)
#define C_V(i,j)        *(c_v+(i*(hidden+1))+j)
#define F_W(i,j)        *(f_w+(i*(outputs+1))+j)
#define E_W(i,j)        *(e_w+(i*(outputs+1))+j)
#define C_W(i,j)        *(c_w+(i*(outputs+1))+j)
#define P(i)            *(p+i)
#define Q(i)            *(q+i)
#define TARGET(i)       *(target+i)
#define TRAINING_STRUCT(i,j)  *(training_struct+(i*outputs)+j)


/*************************************************************************/
/*  global variables                                                     */
/*************************************************************************/

long  i, j, k, m, ex;                       /* loop counters               */
short done = 0;                             /* 0=not done,1=done with leednet*/
long cmdnum = 0;                            /* command number (for prompt)  */
char *cp, *cq;                              /* character pointers          */
long inputs;                                /* number of input nodes       */
long hidden;                                /* number of hidden nodes      */
long outputs;                               /* number of output nodes      */
long function;                              /* data conv function number   */
long epochs;                                /* number of epochs to teach   */
long delanalyze;                            /* num of epochs to analyze    */
long delprt;                                /* num of epochs to print message*/
float mu;                                   /* momentum rate for learning  */
float kappa;                                /* amount to incr learning rate  */
float phi;                                  /* factor to multiply learn rate */
float theta;                                /* ctrl time period of averaging */
float ay[SCALINGS]={                        /* output linear scaling coeffs  */
   1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0};
float by[SCALINGS]={                        /* output const scaling coeffs   */
   0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};
float error;                                /* network error               */
double mem;                                 /* array memory used (bytes);  */
short resume_flag = 0;                      /* 1=resume a run              */
long  start_epoch;                          /* start epoch number          */
int   seed;                                 /* seed for rand num generator */
long  numpts;                               /* num of points in i(v) curve */
long  structures;                           /* num structures in data set  */
long  sepnum;                               /* structure num to extract    */
long  skip;                                 /* interval to skip pruned data  */
short status;                               /* status on/off flag          */
short adaptive;                             /* adaptive learning rate flag */
long  pause;                                /* training pause (epochs)     */
long  beams;                                /* number of beams in input    */
long  resumeepoch;                          /* resume epoch number         */
float errorlim;                             /* training error limit        */
float seedmult;                             /* random seed multiplier      */
short term_input;                           /* 1=terminal input, 0=file input*/
char line[200];                             /* input line                  */
char cmdstr[CMDSIZE][CMDLNSZ];              /* command string & params     */
char dafilename[80];                        /* delanalyze filename         */

char env_var[ENVIRONMENT][ENVLINE] = {      /* environment variables:      */
   "original",                  /*  0 */  /*   original data file name    */
   "formatted",                 /*  1 */  /*   formatted data file name   */
   "missing",                   /*  2 */  /*   missing 1 struct file name */
   "single",                    /*  3 */  /*   1 structure file name      */
   "pruned",                    /*  4 */  /*   pruned data file name      */
   "training",                  /*  5 */  /*   training data file name    */
   "errordat",                  /*  6 */  /*   error vs epoch file name   */
   "dumpfile",                  /*  7 */  /*   dump file name             */
```

```
   "helpfile",                  /*  8 */  /*   help file name              */
   "plotfile",                  /*  9 */  /*   plot data file name         */
   "inputs",                    /* 10 */  /*   number of input nodes       */
   "hidden",                    /* 11 */  /*   number of hidden nodes      */
   "outputs",                   /* 12 */  /*   number of output nodes      */
   "function",                  /* 13 */  /*   data conv function number   */
   "epochs",                    /* 14 */  /*   num of epochs to teach      */
   "delprt",                    /* 15 */  /*   num of epochs to prt message*/
   "mu",                        /* 16 */  /*   momentum rate for learning  */
   "kappa",                     /* 17 */  /*   amt to incr learning rate   */
   "phi",                       /* 18 */  /*   factor to mult learning rate*/
   "theta",                     /* 19 */  /*   ctrl time period for avg'ing*/
   "weightfile",                /* 20 */  /*   saved net weights file name */
   "seed",                      /* 21 */  /*   seed for rand num generator */
   "numpts",                    /* 22 */  /*   num of points in i(v) curve */
   "structures",                /* 23 */  /*   num structures in data set  */
   "sepnum",                    /* 24 */  /*   structure num to extract    */
   "skip",                      /* 25 */  /*   num of points to skip data  */
   "status",                    /* 26 */  /*   training status on/off flag */
   "statusfile",                /* 27 */  /*   training status filename    */
   "adaptive",                  /* 28 */  /*   adaptive learning rate flag */
   "pause",                     /* 29 */  /*   training pause (epochs)     */
   "debug",                     /* 30 */  /*   debug mode on/off           */
   "beams",                     /* 31 */  /*   number of beams in input    */
   "errorlim",                  /* 32 */  /*   training error limit        */
   "anlinfile",                 /* 33 */  /*   analysis input file         */
   "anloutfile",                /* 34 */  /*   analysis output file        */
   "beamoutfile",               /* 35 */  /*   rebeam output file          */
   "delanalyze",                /* 36 */  /*   num of epochs to analyze    */
   "resumeepoch",               /* 37 */  /*   resume epoch number         */
   "seedmult"                   /* 38 */  /*   random seed multiplier      */
};

char env_value[ENVIRONMENT][LINESIZE]  = {  /* env variable values - defaults*/
   "iv.dat",          /*  0 - original    */ /*   original data file name     */
   "iv.fmt",          /*  1 - formatted   */ /*   formatted data file name    */
   "iv.mis",          /*  2 - missing     */ /*   missing 1 struct file name  */
   "iv.sin",          /*  3 - single      */ /*   1 structure file name       */
   "iv.pru",          /*  4 - pruned      */ /*   pruned data file name       */
   "iv.fmt",          /*  5 - training    */ /*   training data file name     */
   "iv.out",          /*  6 - errordat    */ /*   error vs epoch file name    */
   "leednet.dmp",     /*  7 - dumpfile    */ /*   weight dump file name       */
   "leednet.hlp",     /*  8 - helpfile    */ /*   help file name              */
   "iv.plt",          /*  9 - plotfile    */ /*   plot data file name         */
   "100",             /* 10 - inputs      */ /*   number of input nodes       */
   "500",             /* 11 - hidden      */ /*   number of hidden nodes      */
   "3",               /* 12 - outputs     */ /*   number of output nodes      */
   "1",               /* 13 - function    */ /*   data conv function number   */
   "20",              /* 14 - epochs      */ /*   num of epochs to teach      */
   "1",               /* 15 - delprt      */ /*   num of epochs to prt message*/
   "0.0",             /* 16 - mu          */ /*   momentum rate for learning  */
   "0.1",             /* 17 - kappa       */ /*   amt to incr learning rate   */
   "0.5",             /* 18 - phi         */ /*   factor to mult learning rate*/
   "0.7",             /* 19 - theta       */ /*   ctrl time period for avg'ing*/
   "iv.wgt",          /* 20 - weightfile  */ /*   saved net weights file name */
   "1",               /* 21 - seed        */ /*   seed for rand num generator */
   "160",             /* 22 - numpts      */ /*   num of points in i(v) curve */
   "180",             /* 23 - structures  */ /*   num structures in data set  */
   "77",              /* 24 - sepnum      */ /*   structure num to extract    */
   "1",               /* 25 - skip        */ /*   num of points to skip data  */
   "1",               /* 26 - status      */ /*   training status on/off flag */
   "leednet.sta",     /* 27 - statusfile  */ /*   training status filename    */
   "1",               /* 28 - adaptive    */ /*   adaptive learning rate flag */
   "-1",              /* 29 - pause       */ /*   training pause (epochs)     */
   "off",             /* 30 - debug       */ /*   debug mode on/off           */
   "4",               /* 31 - beams       */ /*   number of beams in input    */
   "0.0",             /* 32 - errorlim    */ /*   training error limit        */
   "analyze.in",      /* 33 - anlinfile   */ /*   analysis input file         */
   "analyze.out",     /* 34 - anloutfile  */ /*   analysis output file        */
   "iv.bem",          /* 35 - beamoutfile */ /*   beam output file            */
   "-1",              /* 36 - delanalyze  */ /*   num of epochs to analyze    */
   "1",               /* 37 - resumeepoch */ /*   resume epoch                */
   "1.0e-04"          /* 38 - seedmult    */ /*   random seed multiplier      */
};

FILE *trnfp;                            /* training input file pointer   */
FILE *errfp;                            /* error output file pointer     */
FILE *statfp;                           /* status output file            */
FILE *testfp;                           /* test input file pointer       */
FILE *cmdfp;                            /* batch command input file ptr  */
```

```
FILE *dumpfp;                           /* dump output file pointer      */
FILE *anlinfp;                          /* analysis input file pointer   */
FILE *anloutfp;                         /* analysis output file pointer  */
FILE *danlfp;                           /* delanalyze file pointer       */

float *x;                               /* input neurons                 */
float *v;                               /* input-to-hidden weights       */
float *z_in;                            /* hidden neurons                */
float *z;                               /* sigmoid-limited hidden neurons*/
float *w;                               /* hidden-to-output weights      */
float *y_in;                            /* output neurons                */
float *y;                               /* sigmoid-limited output neurons*/
float *yy;                              /* unscaled network outputs      */
float *t;                               /* target values at each output  */
float *delta_in_j;
float *delta_j;
float *delta_k;
float *del_v;
float *del_w;
float *f_v;
float *e_v;
float *c_v;
float *f_w;
float *e_w;
float *c_w;
float *p;
float *q;
float *target;                          /* true outputs                  */
char  not_ready[] =
   " command not yet working\n";
struct tm *systime;                     /* system time                   */
time_t encoded_time;                    /* encoded system time           */
float rms;                              /* rms error                     */


/*****************************************************************************/
/*  function prototypes                                                      */
/*****************************************************************************/

void  alloc(void);                      /* allocate memory               */
void  analyze(void);                    /* perform error analysis        */
void  ask(void);                        /* ask trained network           */
void  display_weight(void);             /* display weight                */
void  debeam(void);                     /* sep dataset into beams        */
void  dump_weights(void);               /* dump weights                  */
extern void format01(void);             /* format function 01            */
void  help(void);                       /* display help on commands      */
void  init(void);                       /* initialize data               */
void  load_env(void);                   /* load environmental variables  */
void  load_weights(void);               /* load network weights          */
void  memory(void);                     /* display memory usage          */
void  merge(void);                      /* merge data files together     */
float net_sigmoid(float  x);            /* sigmoid function              */
void  net_update_weights(void);         /* update network weights        */
void  net_zero_dels(void);              /* zero del_v[] and del_w[]      */
void  network(int learn);               /* run the network               */
void  plot_data(void);                  /* create plot data files        */
void  prune(void);                      /* prune data                    */
float rand1(void);                      /* random float between 0-1      */
int   rand2(int n);                     /* random int between 1-n        */
float rand3(int n);                     /* random float between 1-n      */
float rand4(void);                      /* random float -1 to +1         */
void  randomiz(void);                   /* init rand num generator       */
void  rebeam(void);                     /* merge beams together          */
void  save_weights(void);               /* save network weights          */
void  scale(int verbose);               /* compute scaling               */
void  separate(void);                   /* sep structure from data       */
void  set(void);                        /* process set command           */
void  show_status(void);                /* show training status          */
void  train(void);                      /* train the network             */




/*****************************************************************************/
```

```
/*  main()                                                                    */
/****************************************************************************/

int main(int argc, char *argv[])
{

/*--------------------------------------------------------------------------*/
/*  initialize the program                                                  */
/*--------------------------------------------------------------------------*/

if (argc > 1)                           /* if an argument is on cmd line */
   {
   if (!strcmp(argv[1],"terminal"))     /* if "terminal" argument given..*/
      term_input = 1;                   /* ..then get input from terminal*/
   else
      term_input = 0;                   /* else get input from file      */
   }
else                                    /* if no cmd line argument..     */
   term_input = 1;                      /* ..then get input from terminal*/

printf("  LEEDNET Version %s\n",VERSION);  /* print software version number */
init();                                 /* initialize data               */
alloc();                                /* allocate memory               */

if (!term_input)                        /* if getting cmds from a file   */
   {
   strcpy(line, argv[1]);
   strcat(line, ".scr");
   if((cmdfp=fopen(line,"r"))==NULL)    /* open cmd input file           */
      {
      printf("  Input from terminal.\n");
      term_input = 1;
      }
   else
      {
      printf("  Input from file %s\n", line);
      term_input = 0;
      }
   }


/*--------------------------------------------------------------------------*/
/*  start of main loop                                                      */
/*--------------------------------------------------------------------------*/

do {                                    /* start of main loop            */
   if (term_input)                      /* if getting cmds from terminal */
      {
      printf("[%d]> ",++cmdnum);        /* incr cmd number & print prompt*/
      gets(line);                       /* read in input line            */
      }
   else                                 /* getting cmds from file        */
      {
      fgets(line,LINESIZE,cmdfp);       /* read in input line            */
      line[strlen(line)-1] = '\0';      /* lop off final newline char    */
      }



/*--------------------------------------------------------------------------*/
/*  parse the entered command line                                          */
/*--------------------------------------------------------------------------*/

   cp = line;                           /* point cp to start of line     */
   for (i=0; i<CMDSIZE; i++)            /* parse the input line          */
      {
      cq = cmdstr[i];                   /* point cq to start of cmd str  */

      while ((*cp==' ') || (*cp=='='))  /* skip past spaces and =        */
         cp++;

      while ((*cp)&&(*cp!=' ')&&(*cp!='=')) /* scan input until \0 or space  */
         *cq++ = *cp++;                 /* copy characters to cmd string */
      *cq = '\0';                       /* terminate cmd string when done*/
      }


/*--------------------------------------------------------------------------*/
/*  See which command was entered and process it.                           */
```

```
/*-------------------------------------------------------------------------*/

   if (!strcmp(cmdstr[0],"alloc"))          /*     ***  alloc  ***          */
      alloc();

   else if (!strcmp(cmdstr[0],"analyze"))   /*     ***  analyze  ***        */
      analyze();

   else if (!strcmp(cmdstr[0],"ask"))       /*     ***  ask  ***           */
      ask();

   else if (!strcmp(cmdstr[0],"debeam"))    /*     ***  debeam  ***        */
      debeam();

   else if (!strcmp(cmdstr[0],"dump"))      /*     ***  dump  ***          */
      dump_weights();

   else if (!strcmp(cmdstr[0],"dw"))        /*     ***  dw  ***            */
      display_weight();

   else if (!strcmp(cmdstr[0],"exit") ||    /*     ***  exit/quit  ***     */
            !strcmp(cmdstr[0],"quit"))
      done = 1;

   else if (!strcmp(cmdstr[0],"format"))    /*     ***  format  ***        */
      switch (function)
         {
         case 1:                           /* add additional cases here..  */
            format01();                    /* ..for more format functions  */
            break;
         default:
            printf("  Format function %d%s",
                   function,
                   " not defined.\n");
         }

   else if (!strcmp(cmdstr[0],"help"))      /*     ***  help  ***          */
      help();

   else if (!strcmp(cmdstr[0],"load"))      /*     ***  load  ***          */
      load_weights();

   else if (!strcmp(cmdstr[0],"mem"))       /*     ***  mem  ***           */
      memory();

   else if (!strcmp(cmdstr[0],"merge"))     /*     ***  merge  ***         */
      merge();

   else if (!strcmp(cmdstr[0],"plot"))      /*     ***  plot  ***          */
      plot_data();

   else if (!strcmp(cmdstr[0],"prune"))     /*     ***  prune  ***         */
      prune();

   else if (!strcmp(cmdstr[0],"randomize")) /*     ***  randomize  ***     */
      randomiz();

   else if (!strcmp(cmdstr[0],"rebeam"))    /*     ***  rebeam  ***        */
      rebeam();

   else if (!strcmp(cmdstr[0],"resume"))    /*     ***  resume  ***        */
      {
      resume_flag = 1;
      train();
      }

   else if (!strcmp(cmdstr[0],"save"))      /*     ***  save  ***          */
      save_weights();

   else if (!strcmp(cmdstr[0],"scale"))     /*     ***  scale  ***         */
      scale(1);

   else if (!strcmp(cmdstr[0],"separate"))  /*     ***  separate  ***      */
      separate();

   else if (!strcmp(cmdstr[0],"set"))       /*     ***  set  ***           */
      set();

   else if (!strcmp(cmdstr[0],"status"))    /*     ***  status  ***        */
      show_status();
```

```
   else if (!strcmp(cmdstr[0],"train"))     /*     ***  train  ***          */
      train();

   else if (!strcmp(cmdstr[0],"ver"))       /*     ***  ver  ***            */
      printf("  LEEDNET Version %s\n",
             VERSION);

   else if ((!strcmp(cmdstr[0],"\n")) ||    /* ignore newlines..           */
            (!strcmp(cmdstr[0],"\r")))      /* ..and carriage returns      */
      { }

   else                                     /* command not recognized      */
      printf(
      "  Command \"%s\" not recognized\n",
           cmdstr[0]);

   } while (!done);                         /* repeat until 'exit' or 'quit' */


/*-------------------------------------------------------------------------*/
/*  end of main()                                                          */
/*-------------------------------------------------------------------------*/

if (!term_input)
   fclose(cmdfp);

return 0;                                   /* return to operating system  */
}




/*************************************************************************/
/*  alloc()                                                              */
/*************************************************************************/

void alloc(void)
{
int sf;                                     /* size of float               */
static char errstr[] =                      /* error string                */
   "  Error allocating memory for \"";


sf = sizeof(float);                         /* find size of float          */

/*-------------------------------------------------------------------------*/
/*  free all memory currently used by dynamically allocated arrays.        */
/*-------------------------------------------------------------------------*/

free(target);
free(q);
free(p);
free(c_w);
free(e_w);
free(f_w);
free(c_v);
free(e_v);
free(f_v);
free(del_w);
free(del_v);
free(delta_k);
free(delta_j);
free(delta_in_j);
free(t);
free(yy);
free(y);
free(y_in);
free(w);
free(z);
free(z_in);
free(v);
free(x);


/*-------------------------------------------------------------------------*/
```

```
/*  Now allocate memory for each of the dynamic arrays on the heap.          */
/*  In each case, print an error message and return if the allocation was    */
/*  not successful.                                                          */
/*---------------------------------------------------------------------------*/

x = (float *)malloc((inputs+1)*sf);          /*   x                          */
if (!x)
   {
   printf("%s%s\"\n",errstr,"x");
   return;
   }

v = (float *)malloc((inputs+1)*(hidden+1)*sf); /*   v                        */
if (!v)
   {
   printf("%s%s\"\n",errstr,"v");
   return;
   }

z_in = (float *)malloc((hidden+1)*sf);       /*   z_in                       */
if (!z_in)
   {
   printf("%s%s\"\n",errstr,"z_in");
   return;
   }

z = (float *)malloc((hidden+1)*sf);          /*   z                          */
if (!z)
   {
   printf("%s%s\"\n",errstr,"z");
   return;
   }

w = (float *)malloc((hidden+1)*(outputs+1)*sf); /*   w                       */
if (!w)
   {
   printf("%s%s\"\n",errstr,"w");
   return;
   }

y_in = (float *)malloc((outputs+1)*sf);      /*   y_in                       */
if (!y_in)
   {
   printf("%s%s\"\n",errstr,"y_in");
   return;
   }

y = (float *)malloc((outputs+1)*sf);         /*   y                          */
if (!y)
   {
   printf("%s%s\"\n",errstr,"y");
   return;
   }

yy = (float *)malloc((outputs+1)*sf);        /*   yy                         */
if (!yy)
   {
   printf("%s%s\"\n",errstr,"yy");
   return;
   }

t = (float *)malloc((outputs+1)*sf);         /*   t                          */
if (!t)
   {
   printf("%s%s\"\n",errstr,"t");
   return;
   }

delta_in_j = (float *)malloc((hidden+1)*sf); /*   delta_in_j                 */
if (!delta_in_j)
   {
   printf("%s%s\"\n",errstr,"delta_in_j");
   return;
   }

delta_j = (float *)malloc((hidden+1)*sf);    /*   delta_j                    */
if (!delta_j)
   {
   printf("%s%s\"\n",errstr,"delta_j");
   return;
```

```
   }

delta_k = (float *)malloc((outputs+1)*sf);  /*   delta_k                    */
if (!delta_k)
   {
   printf("%s%s\"\n",errstr,"delta_k");
   return;
   }

del_v = (float *)malloc((inputs+1)*(hidden+1)*sf);   /*   del_v             */
if (!del_v)
   {
   printf("%s%s\"\n",errstr,"del_v");
   return;
   }

del_w = (float *)malloc((hidden+1)*(outputs+1)*sf);  /*   del_w             */
if (!del_w)
   {
   printf("%s%s\"\n",errstr,"del_w");
   return;
   }

f_v = (float *)malloc((inputs+1)*(hidden+1)*sf);     /*   f_v               */
if (!f_v)
   {
   printf("%s%s\"\n",errstr,"f_v");
   return;
   }

e_v = (float *)malloc((inputs+1)*(hidden+1)*sf);     /*   e_v               */
if (!e_v)
   {
   printf("%s%s\"\n",errstr,"e_v");
   return;
   }

c_v = (float *)malloc((inputs+1)*(hidden+1)*sf);     /*   c_v               */
if (!c_v)
   {
   printf("%s%s\"\n",errstr,"c_v");
   return;
   }

f_w = (float *)malloc((hidden+1)*(outputs+1)*sf);    /*   f_w               */
if (!f_w)
   {
   printf("%s%s\"\n",errstr,"f_w");
   return;
   }

e_w = (float *)malloc((hidden+1)*(outputs+1)*sf);    /*   e_w               */
if (!e_w)
   {
   printf("%s%s\"\n",errstr,"e_w");
   return;
   }

c_w = (float *)malloc((hidden+1)*(outputs+1)*sf);    /*   c_w               */
if (!c_w)
   {
   printf("%s%s\"\n",errstr,"c_w");
   return;
   }

p = (float *)malloc((outputs+1)*sf);                 /*   p                 */
if (!p)
   {
   printf("%s%s\"\n",errstr,"p");
   return;
   }

q = (float *)malloc((hidden+1)*sf);                  /*   q                 */
if (!q)
   {
   printf("%s%s\"\n",errstr,"q");
   return;
   }

target = (float *)malloc((outputs+1)*sf);            /*   target            */
```

```
if (!target)
   {
   printf("%s%s\"\n",errstr,"target");
   return;
   }

printf("  Memory successfully allocated.\n"); /* successful if we got here   */


/*-------------------------------------------------------------------------*/
/*  Now calculate how much memory we've used for the dynamically allocated   */
/*  arrays, and print the result.                                            */
/*-------------------------------------------------------------------------*/

mem  = inputs+1;                          /* arr of size "inputs"          */
mem += 5*(inputs+1)*(hidden+1);           /* arr of size "inputs * hidden" */
mem += 5*(hidden+1);                      /* arr of size "hidden"          */
mem += 5*(hidden+1)*(outputs+1);          /* arr of size "hidden * outputs"*/
mem += 7*(outputs+1);                     /* arr of size "outputs"         */
mem *= sf;                                /* convert to bytes              */
printf ("  %0.0lf bytes ",mem);           /* print memory used in bytes    */
if (mem < 1.048576e6)                     /* print memory used in Kb       */
   printf ("(%0.2f Kb) used.\n",
           mem/1024.0);
else                                      /* print memory used in Mb       */
   printf ("(%0.2f Mb) used.\n",
           mem/1.048576e6);

}




/*************************************************************************/
/*  analyze()                                                            */
/*************************************************************************/

void analyze(void)
{
int sf;                                   /* size of float                 */
int ss;
static char errstr[] =                    /* error string                  */
   "  Error allocating memory for \"";
short i, j;
float n;
short struct_num;
float delta;
float *sum2;
float *min;
float *max;
short *n_min;
short *n_max;
float *training_struct;
float rms_final;
float distance, min_distance, temp;


if((trnfp=fopen(env_value[TRAINING],"r"))==NULL) /* open input file       */
   {
   printf("  Cannot open file %s\n",
          env_value[TRAINING]);
   return;
   }

fgets (line, LINESIZE, trnfp);
sscanf (line,"%ld", &inputs);             /* read in number of input nodes */
sprintf(env_value[INPUTS],"%ld",inputs);

fgets (line, LINESIZE, trnfp);
sscanf (line,"%ld", &outputs);            /* read in number of output nodes*/
sprintf(env_value[OUTPUTS],"%ld",outputs);

fgets (line, LINESIZE, trnfp);
sscanf (line,"%ld", &structures);         /* read in number of structures  */
sprintf(env_value[STRUCTURES],"%ld",
        structures);

load_weights();                           /* load network weights          */
```

```
sf = sizeof(float);                          /* find size of float          */
ss = sizeof(short);                          /* find size of short          */

sum2 = (float *)malloc((outputs)*sf);
if (!sum2)
   {
   printf("%s%s\"\n",errstr,"sum");
   return;
   }

min = (float *)malloc((outputs)*sf);
if (!min)
   {
   printf("%s%s\"\n",errstr,"min");
   return;
   }

max = (float *)malloc((outputs)*sf);
if (!max)
   {
   printf("%s%s\"\n",errstr,"max");
   return;
   }

n_min = (short *)malloc((outputs)*ss);
if (!n_min)
   {
   printf("%s%s\"\n",errstr,"n_min");
   return;
   }

n_max = (short *)malloc((outputs)*ss);
if (!n_max)
   {
   printf("%s%s\"\n",errstr,"n_max");
   return;
   }

training_struct = (float *)malloc((structures*outputs)*sf);
if (!training_struct)
   {
   printf("%s%s\"\n",errstr,"training_struct");
   return;
   }

for (i=0; i<structures; i++)
   {
   for (j=0; j<inputs; j++)
      fgets (line, LINESIZE, trnfp);

   for (j=0; j<outputs; j++)
      {
      fgets (line, LINESIZE, trnfp);
      sscanf (line, "%e\n", &temp);
      TRAINING_STRUCT(i,j) = temp;
      }
   }

fclose(trnfp);

n = 0.0;

for (i=0; i<outputs; i++)
   {
   *(sum2+i) = 0.0;
   *(min+i) = INFINITY;
   *(max+i) = 0.0;
   *(n_min) = 0;
   *(n_max) = 0;
   }

if((anlinfp=fopen(env_value[ANLINFILE],"r"))==NULL) /* open input file       */
   {
   printf("  Cannot open file %s\n",
          env_value[ANLINFILE]);
   return;
   }

if((anloutfp=fopen(env_value[ANLOUTFILE],"w"))==NULL) /* open output file    */
   {
```

```
   printf("  Cannot open file %s\n",
          env_value[ANLOUTFILE]);
   return;
   }

struct_num = 0;

do {
   fgets (line, LINESIZE, anlinfp);
   line[strlen(line)-1] = '\0';              /* lop off final newline char    */

   if (feof(anlinfp))
      break;

   if((testfp=fopen(line,"r"))==NULL)        /* open input file               */
      {
      printf("  Cannot open file %s\n",
             line);
      return;
      }

   for (i=0; i<3; i++)                       /* read & discard header lines   */
      fgets (line, LINESIZE, testfp);


   for (i=1; i<=inputs; i++)                 /* get points of leed i(v) curve */
      {
      fgets (line, LINESIZE, testfp);
      sscanf (line,"%e\n",x+i);
      }

   for (i=1; i<=outputs; i++)                /* get target outputs            */
      {
      fgets (line, LINESIZE, testfp);
      sscanf (line,"%f\n",target+i);
      }

   fclose(testfp);

   network(0);                               /* run the network w/o teaching  */

   for (i=1; i<=outputs; i++)
      YY(i) = (Y(i)-by[i])/ay[i];            /* un-scale network output       */

   fprintf(anloutfp, "\nTest %d:\n", struct_num);

   if (error >= 1.0e-6)
      fprintf (anloutfp,"\n  Error = %f\n", error);   /* print error in f format       */
   else
      fprintf (anloutfp,"\n  Error = %e\n", error);   /* print error in e format       */

   fprintf (anloutfp,"  Target outputs:\n");          /* print expected outputs        */
   for (i=1; i<=outputs; i++)
      fprintf (anloutfp,"    %f\n",TARGET(i));

   fprintf (anloutfp,"  Network-computed outputs:\n");/* print network-computed outputs*/
   for (i=1; i<=outputs; i++)
      fprintf (anloutfp,"    %f\n",YY(i));

   for (i=0; i<outputs; i++)
      {
      delta = TARGET(i+1) - YY(i+1);
      *(sum2+i) += delta * delta;
      if (fabs(delta) < *(min+i))
         {
         *(min+i) = fabs(delta);
         *(n_min+i) = struct_num;
         }
      if (fabs(delta) > *(max+i))
         {
         *(max+i) = fabs(delta);
         *(n_max+i) = struct_num;
         }
      }

   rms = 0.0;                                /* compute rms error             */
   for (i=1; i<=outputs; i++)
      rms += (YY(i)-TARGET(i))*(YY(i)-TARGET(i));
   rms = sqrt(rms/outputs);
   fprintf (anloutfp,"  RMS error = %f\n", rms);      /* print rms error     */
```

```
    min_distance = INFINITY;

    for (i=0; i<structures; i++)
       {
       distance = 0.0;
       for (j=0; j<outputs; j++)
          {
          temp = TARGET(j+1) - TRAINING_STRUCT(i,j);
          distance += temp*temp;
          }
       distance = sqrt(distance);
       if (distance < min_distance)
          min_distance = distance;
       }

    fprintf (anloutfp,"  Distance to nearest training point = %f\n",
                      min_distance);         /* print distance */

    n++;
    struct_num++;

    } while (!feof(anlinfp));

 fprintf(anloutfp,"\n\nFinal results:\n\n");

 for (i=0; i<outputs; i++)
    {
    rms_final = sqrt(*(sum2+i)/n);
    fprintf(anloutfp,"Output %d:  Minimum error = %f  (at structure %d)\n", i, *(min+i), *(n_min+i));
    fprintf(anloutfp,"            Maximum error = %f  (at structure %d)\n", *(max+i), *(n_max+i));
    fprintf(anloutfp,"            RMS error = %f\n\n", rms_final);
    }

 free (training_struct);
 free (n_max);
 free (n_min);
 free (max);
 free (min);
 free (sum2);

 fclose (anlinfp);
 fclose (anloutfp);

 }
```

```
/*****************************************************************************/
/*  ask()                                                                    */
/*****************************************************************************/

void ask(void)
{
if((testfp=fopen(env_value[SINGLE],"r"))==NULL) /* open 'single' file        */
   {
   printf("  Cannot open file %s\n",
          env_value[SINGLE]);
   return;
   }

for (i=0; i<3; i++)                          /* read & discard header lines   */
   fgets (line, LINESIZE, testfp);


for (i=1; i<=inputs; i++)                    /* get points of leed i(v) curve */
   {
   fgets (line, LINESIZE, testfp);
   sscanf (line,"%e\n",x+i);
   }

for (i=1; i<=outputs; i++)                   /* get target outputs            */
   {
   fgets (line, LINESIZE, testfp);
   sscanf (line,"%f\n",target+i);
   }

network(0);                                  /* run the network w/o teaching  */
```

```
for (i=1; i<=outputs; i++)
   YY(i) = (Y(i)-by[i])/ay[i];              /* un-scale network output       */

if (error >= 1.0e-6)
   printf ("  Error = %f\n", error);        /* print error in f format       */
else
   printf ("  Error = %e\n", error);        /* print error in e format       */

printf ("  Target outputs:\n");              /* print expected outputs        */
for (i=1; i<=outputs; i++)
   printf ("    %f\n",TARGET(i));

printf ("  Network-computed outputs:\n");   /* print network-computed outputs*/
for (i=1; i<=outputs; i++)
   printf ("    %f\n",YY(i));

rms = 0.0;                                  /* compute rms error             */
for (i=1; i<=outputs; i++)
   rms += (YY(i)-TARGET(i))*(YY(i)-TARGET(i));
rms = sqrt(rms/outputs);
printf ("  RMS error = %f\n", rms);         /* print rms error               */

}




/*****************************************************************************/
/*  debeam()                                                                 */
/*****************************************************************************/

void debeam(void)
{
long i, j, k;                               /* loop counters                 */
char str[200];
char str2[4];
long beam_pts;
FILE *infp, **beamfpp;


/*---------------------------------------------------------------------------*/
/*  open all files                                                           */
/*---------------------------------------------------------------------------*/

beamfpp = (FILE **)malloc(beams*sizeof(FILE *));
if (beamfpp == NULL)
   {
   printf("  Error allocating beam file pointers\n");
   return;
   }

if ((infp = fopen(env_value[FORMATTED],"r"))==NULL)
   {
   printf ("  Error opening file %s.\n",
           env_value[FORMATTED]);
   return;
   }

for (i=0; i<beams; i++)
   {
   strcpy(str,"beam.");
   sprintf(str2,"%03d",i);
   strcat(str,str2);
   if ((*(beamfpp+i)=fopen(str,"w"))==NULL)
      {
      printf ("  Error opening file %s\n",
              str);
      return;
      }
   }


/*---------------------------------------------------------------------------*/
/*  copy over the first three header lines                                   */
/*---------------------------------------------------------------------------*/

fgets(str, 200, infp);                      /* number of inputs              */
inputs = atoi(str);                         /* copy to "inputs" env variable */
```

```
str[strlen(str)-1] = '\0';                  /* remove final newline          */
strcpy(env_value[INPUTS],str);              /* copy to env var string        */

beam_pts = inputs / beams;                  /* find number of points per beam */
sprintf(str,"%d\n", beam_pts);
for (i=0; i<beams; i++)
   fputs(str, *(beamfpp+i));                /* copy to "beam.xxx" files      */

fgets(str, 200, infp);                      /* number of outputs             */
for (i=0; i<beams; i++)
   fputs(str, *(beamfpp+i));                /* copy to "beam.xxx" files      */
outputs = atoi(str);                        /* copy to "outputs" env variable */
str[strlen(str)-1] = '\0';                  /* remove final newline          */
strcpy(env_value[OUTPUTS],str);             /* copy to env var string        */

fgets(str, 200, infp);                      /* number of structures          */
for (i=0; i<beams; i++)
   fputs(str, *(beamfpp+i));                /* copy to "beam.xxx" files      */
structures = atoi(str);                     /* copy to "structures" env var  */
str[strlen(str)-1] = '\0';                  /* remove final newline          */
strcpy(env_value[STRUCTURES],str);          /* copy to env var string        */


/*-------------------------------------------------------------------------*/
/*  For each structure, read in each beam and copy to the appropriate      */
/*  beam.xxx file.  Then copy the outputs to each beam.xxx file.           */
/*-------------------------------------------------------------------------*/

for (k=0; k<structures; k++)
   {
   for (i=0; i<beams; i++)
      for (j=0; j<beam_pts; j++)
         {
         fgets(str, 200, infp);
         fputs(str, *(beamfpp+i));
         }

   for (i=0; i<outputs; i++)
      {
      fgets(str, 200, infp);
      for (j=0; j<beams; j++)
         fputs(str, *(beamfpp+j));
      }
   }


/*-------------------------------------------------------------------------*/
/*  close all files                                                        */
/*-------------------------------------------------------------------------*/

fclose (infp);
for (i=0; i<beams; i++)
   fclose(*(beamfpp+i));

}




/*************************************************************************/
/*  display_weight()                                                     */
/*************************************************************************/

void display_weight(void)
{
int  i, j;                                  /* indices                   */


i = atoi(cmdstr[2]);                        /* convert 1st index         */
j = atoi(cmdstr[3]);                        /* convert 2nd index         */

if (!strcmp(cmdstr[1],"v"))                 /* v - input-to-hidden weight */
   printf("  v(%d,%d) = %f\n",
         i, j, V(i,j));

else if (!strcmp(cmdstr[1],"w"))            /* w - hidden-to-output weight */
   printf("  w(%d,%d) = %f\n",
         i, j, W(i,j));
```

```
else                                    /* oops - need to enter v or w   */
   {
   printf("  Syntax: dw v|w i j\n");     /* print error message           */
   printf("  v   for input-to-hidden weight\n");
   printf("  w   for hidden-to-output weight\n");
   printf("  i,j are weight indices\n");
   }
}




/*************************************************************************/
/*  dump_weights()                                                       */
/*************************************************************************/

void dump_weights(void)
{
int i, j;


load_weights();                         /* load network weights          */

if((dumpfp=fopen(env_value[DUMPFILE],"w"))==NULL) /* open output file    */
   {
   printf("  Cannot open file %s\n",
          env_value[DUMPFILE]);
   return;
   }

for (i=0; i<=inputs; i++)
   for (j=0; j<=hidden; j++)
      fprintf(dumpfp,"  v(%d,%d) = %f\n",
         i, j, V(i,j));

for (i=0; i<=hidden; i++)
   for (j=0; j<=outputs; j++)
      fprintf(dumpfp,"  w(%d,%d) = %f\n",
         i, j, W(i,j));

fclose(dumpfp);
}




/*************************************************************************/
/*  help()                                                               */
/*************************************************************************/

void help(void)
{
char str[16];                           /* str to hold help parameter    */
char str80[LINESIZE+1];                 /* holds line from help file     */
short found;                            /* help file section found flag  */
FILE *fp;                               /* help file pointer             */


if (!strcmp(cmdstr[1],""))              /* if no arguments given..       */
   strcpy(cmdstr[1],"general");         /* then set to show general help */

strcpy(str,"***");                      /* form help file section header */
strcat(str,cmdstr[1]);                  /* (format: ***cmdname)          */
strcat(str,"\n");                       /* add \n since fgets gives a \n */

if((fp=fopen(env_value[HELPFILE],"r"))==NULL) /* open help file          */
   {
   printf("  Help file %s not found.\n",    /* if help file not found..  */
          env_value[HELPFILE]);         /* ..print error message..       */
   return;                              /* ..and return                  */
   }

found = 0;                              /* init flag to 'not found'      */

do {                                    /* read through the help file    */
   fgets(str80,LINESIZE,fp);            /* read in one line              */
   if ((!found) && (!strcmp(str,str80)))    /* if help section found..   */
      {
```

```
      found = 1;                          /* set 'found' flag..          */
      continue;                           /* and skip to read next line  */
      }
   if (found)                             /* if help section was found..  */
      {
      if (str80[0]=='*')                  /* ..if end of section..        */
         break;                           /* ..then stop printing         */
      else                                /* else in middle of section..  */
         printf("%s",str80);              /* ..so print help info         */
      }
   } while (!feof(fp));                   /* stop at end of help file     */

fclose(fp);                               /* close help file              */
}



/*****************************************************************************/
/*  init()                                                                   */
/*****************************************************************************/

void init(void)
{
char str[LINESIZE];                       /* leednet.ini input line       */
char var[16];                             /* environment variable         */
char arg[LINESIZE];                       /* value to set env variable    */
FILE *fp;                                 /* pointer for leednet.ini      */


if ((fp=fopen("leednet.ini","r"))!=NULL)  /* if leednet.ini exists..      */
   {
   do {
      fgets(str,LINESIZE,fp);             /* read in line from leednet.ini */

      cp = str;                           /* point p to start of input str */
      cq = var;                           /* point q to start of var name  */

      while (*cp==' ')                    /* skip past leading spaces      */
         cp++;

      while ((*cp)&&(*cp!=' ')&&(*cp!='=')) /* scan input until \0 spc or =  */
         *cq++ = tolower(*cp++);          /* copy characters to string     */
      *cq = '\0';                         /* terminate string when done    */

      if (var[0]==';')                    /* if this is a comment line..   */
         continue;                        /* ..skip to read in next line   */

      while ((*cp==' ') || (*cp=='='))    /* skip over any spaces or '='   */
         cp++;

      cq = arg;                           /* point q to start of arg string*/

      while((*cp)&&(*cp!=' ')&&(*cp!='\n')) /* scan input until \0 spc or \n */
         *cq++ = tolower(*cp++);          /* copy characters to string     */
      *cq = '\0';                         /* terminate string when done    */

      for (i=0; i<ENVIRONMENT; i++)       /* check against env var list    */
         if (!strcmp(var,env_var[i]))     /* if this var found in list..   */
            {
            strcpy(env_value[i],arg);     /* ..copy its value to env_value */
            break;                        /* ..and stop search             */
            }

      if (!strcmp(var,"scale"))
         {
         strcpy(cmdstr[1], arg);

         while (*cp==' ')                 /* skip over any spaces or '='   */
            cp++;

         cq = &cmdstr[2][0];
         while((*cp)&&(*cp!=' ')&&(*cp!='\n'))
            *cq++ = tolower(*cp++);
         *cq = '\0';

         while (*cp==' ')                 /* skip over any spaces or '='   */
            cp++;
```

```
         cq = &cmdstr[3][0];
         while((*cp)&&(*cp!=' ')&&(*cp!='\n'))
            *cq++ = tolower(*cp++);
         *cq = '\0';

         scale(0);
         }

      } while (!feof(fp));                  /* repeat to end of leednet.ini  */

   fclose(fp);                              /* close leednet.ini             */
   }

load_env();                                 /* load environmental variables  */
}
```

```
/*****************************************************************************/
/*  load_env()                                                               */
/*****************************************************************************/

void load_env(void)                         /* load environmental variables  */
{
inputs     = atoi(env_value[INPUTS]);       /* conv number of input nodes    */
hidden     = atoi(env_value[HIDDEN]);       /* conv number of hidden nodes   */
outputs    = atoi(env_value[OUTPUTS]);      /* conv number of output nodes   */
function   = atoi(env_value[FUNCTION]);     /* conv data conv func number    */
epochs     = atoi(env_value[EPOCHS]);       /* conv number of epochs to teach*/
delprt     = atoi(env_value[DELPRT]);       /* conv num of epochs to prt msg */
mu         = atof(env_value[MU]);           /* conv momentum rate            */
kappa      = atof(env_value[KAPPA]);        /* conv amt to incr learning rate*/
phi        = atof(env_value[PHI]);          /* conv factor to mult learn rate*/
theta      = atof(env_value[THETA]);        /* conv factor to ctrl avg period*/
seed       = atoi(env_value[SEED]);         /* conv rand num generator seed  */
numpts     = atoi(env_value[NUMPTS]);       /* conv num pts in i(v) curve    */
structures = atoi(env_value[STRUCTURES]);   /* conv num structures in data   */
sepnum     = atoi(env_value[SEPNUM]);       /* conv structure num to extract */
skip       = atoi(env_value[SKIP]);         /* conv num of points to skip    */
status     = atoi(env_value[STATUS]);       /* conv training status flag     */
adaptive   = atoi(env_value[ADAPTIVE]);     /* conv adaptive learn rate flag */
pause      = atoi(env_value[PAUSE]);        /* conv training pause (epochs)  */
beams      = atoi(env_value[BEAMS]);        /* conv num of beams in input    */
errorlim   = atof(env_value[ERRORLIM]);     /* conv training error limit     */
delanalyze = atoi(env_value[DELANALYZE]);   /* conv num of epochs to analyze */
resumeepoch = atoi(env_value[RESUMEEPOCH]); /* conv resume epoch number      */
seedmult   = atof(env_value[SEEDMULT]);     /* conv random seed multiplier   */
}
```

```
/*****************************************************************************/
/*  load_weights()                                                           */
/*****************************************************************************/

void load_weights(void)
{
long i, j;
char wstr[LINESIZE];
FILE *wgtfp;


if((wgtfp=fopen(env_value[WEIGHTFILE],"rb"))==NULL) /* open weight file      */
   {
   printf("  Cannot open file %s\n",
          env_value[WEIGHTFILE]);
   return;
   }

fread(&inputs, sizeof(long), 1, wgtfp);     /* read num of input nodes       */
fread(&hidden, sizeof(long), 1, wgtfp);     /* read num of hidden nodes      */
fread(&outputs, sizeof(long), 1, wgtfp);    /* read num of output nodes      */

sprintf(wstr,"%d", inputs);                 /* conv num inputs to string..   */
strcpy(env_value[INPUTS],wstr);             /* ..and copy to env_value       */
```

```
sprintf(wstr,"%d", hidden);                 /* conv num hidden to string..   */
strcpy(env_value[HIDDEN],wstr);             /* ..and copy to env_value       */

sprintf(wstr,"%d", outputs);                /* conv num outputs to string..  */
strcpy(env_value[OUTPUTS],wstr);            /* ..and copy to env_value       */

alloc();                                    /* allocate memory               */

fread(v, sizeof(float),                     /* read in v weights             */
      (inputs+1)*(hidden+1), wgtfp);

fread(w, sizeof(float),                     /* read in w weights             */
      (hidden+1)*(outputs+1), wgtfp);

fclose(wgtfp);                              /* close weight file             */
}



/*****************************************************************************/
/*  memory()                                                                 */
/*****************************************************************************/

void memory(void)
{
double mem;


if (!strcmp(cmdstr[1],""))                  /* if no argument was given..    */
   {
   mem  = inputs+1;
   mem += 5*(inputs+1)*(hidden+1);
   mem += 5*(hidden+1);
   mem += 5*(hidden+1)*(outputs+1);
   mem += 7*(outputs+1);
   mem *= sizeof(float);
   printf ("  %0.0lf bytes ",mem);
   if (mem < 1.048576e6)
      printf ("(%0.2f Kb) used.\n",
              mem/1024.0);
   else
      printf ("(%0.2f Mb) used.\n",
              mem/1.048576e6);
   }

else                                        /* mem inputs hidden outputs     */
   {
   mem  = atof(cmdstr[1])+1;
   mem += 5*(atof(cmdstr[1])+1)*(atof(cmdstr[2])+1);
   mem += 5*(atof(cmdstr[2])+1);
   mem += 5*(atof(cmdstr[2])+1)*(atof(cmdstr[3])+1);
   mem += 7*(atof(cmdstr[3])+1);
   mem *= sizeof(float);
   printf ("  %0.0lf bytes ",mem);
   if (mem < 1.048576e6)
      printf ("(%0.2lf Kb) would be needed.\n",
              mem/1024.0);
   else
      printf ("(%0.2lf Mb) would be needed.\n",
              mem/1.048576e6);
   }
}



/*****************************************************************************/
/*  merge()                                                                  */
/*****************************************************************************/

void merge(void)
{
char str[200];
short ctr, i;
FILE *infp, *outfp;
```

```
/*-------------------------------------------------------------------------*/
/*  get name of output file and open it                                    */
/*-------------------------------------------------------------------------*/

printf("  Enter name of output file:  ");
fgets(str, 200, stdin);
str[strlen(str)-1] = '\0';

if ((outfp=fopen(str,"w"))==NULL)
   {
   printf("  Error opening file %s.\n", str);
   return;
   }


/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/

ctr = 0;

strcpy(str,"temp");

for (;;)
   {
   printf("  Enter name of data file:  ");
   fgets(str, 200, stdin);
   str[strlen(str)-1] = '\0';

   if (!strcmp(str,""))
      break;

   if ((infp=fopen(str, "r"))==NULL)
      {
      printf("  Error opening file %s.\n", str);
      return;
      }

   ctr++;

   if (ctr == 1)                          /* if this is the 1st data file  */
      {
      fgets(str, 200, infp);              /* ..read num of input nodes     */
      fputs(str, outfp);
      inputs = atoi(str);
      str[strlen(str)-1] = '\0';
      strcpy(env_value[INPUTS],str);

      fgets(str, 200, infp);              /* ..read num of output nodes    */
      fputs(str, outfp);
      outputs = atoi(str);
      str[strlen(str)-1] = '\0';
      strcpy(env_value[OUTPUTS],str);

      fgets(str, 200, infp);              /* ..read num of structs in file */
      structures = atoi(str);
      sprintf(str,"%d\n",structures);
      fputs(str, outfp);
      str[strlen(str)-1] = '\0';
      strcpy(env_value[STRUCTURES],str);
      }

   else                                   /* if this is not the 1st file.. */
      {
      fgets(str, 200, infp);              /* ..then skip 3 header lines    */
      fgets(str, 200, infp);
      fgets(str, 200, infp);
      }

   for (i=0; i<(inputs+outputs)*structures;
        i++)
      {
      fgets(str, 200, infp);              /* append input file..           */
      fputs(str, outfp);                  /* ..to end of output file       */
      }

   fclose(infp);                          /* close each input file         */
   }

printf("  %d files merged.\n", ctr);
```

```
fclose(outfp);                          /* close output file            */
}



/*************************************************************************/
/*  net_sigmoid()  -  sigmoid function                                   */
/*************************************************************************/

float  net_sigmoid(float  x)
{
return 1.0/(1.0+exp(-x));
}



/*************************************************************************/
/*  net_update_weights()                                                 */
/*************************************************************************/

void net_update_weights(void)
{
/*-----------------------------------------------------------------------*/
/*  adjust weights                                                       */
/*-----------------------------------------------------------------------*/

for (j=0; j<=hidden; j++)               /* adjust w weights             */
   for (k=1; k<=outputs; k++)
      {
      if (adaptive)                     /* using adaptive learning rates */
 {
         if (DEL_W(j,k)*F_W(j,k) > 0.0)
            E_W(j,k) += kappa;
         else
            E_W(j,k) *= phi;

         F_W(j,k) = (1.0-theta)*DEL_W(j,k) +
                    theta*F_W(j,k);
         C_W(j,k) = -(1.0-mu)*E_W(j,k)*DEL_W(j,k) +
                    mu*C_W(j,k);
         W(j,k) += C_W(j,k);
         }
      else                              /* using constant learning rate  */
         W(j,k) -= kappa*DEL_W(j,k);
      }

for (i=0; i<=inputs; i++)               /* adjust v weights             */
   for (j=1; j<=hidden; j++)
      {
      if (adaptive)                     /* using adaptive learning rates */
 {
         if (DEL_V(i,j)*F_V(i,j) > 0.0)
            E_V(i,j) += kappa;
         else
            E_V(i,j) *= phi;

         F_V(i,j) = (1.0-theta)*DEL_V(i,j) +
                    theta*F_V(i,j);
         C_V(i,j) = -(1.0-mu)*E_V(i,j)*DEL_V(i,j) +
                    mu*C_V(i,j);
         V(i,j) += C_V(i,j);
         }
      else                              /* using constant learning rate  */
         V(i,j) -= kappa*DEL_V(i,j);
      }
}



/*************************************************************************/
/*  net_zero_dels()                                                      */
/*************************************************************************/

void net_zero_dels(void)
```

```
{
for (i=0; i<=inputs; i++)
   for (j=0; j<=hidden; j++)
      DEL_V(i,j) = 0.0;

for (j=0; j<=hidden; j++)
   for (k=0; k<=outputs; k++)
      DEL_W(j,k) = 0.0;
}
```

```
/*************************************************************************/
/*  network()  -  run the network                                        */
/*************************************************************************/

void network(int learn)
{
/*-----------------------------------------------------------------------*/
/*  use neural network to calculate sum, and find error                  */
/*-----------------------------------------------------------------------*/

for (j=1; j<=hidden; j++)                  /* z[j] = sum of inputs*weights  */
   {
   Z_IN(j) = V(0,j);                       /* init sum to bias weight v[0j] */
   for (i=1; i<=inputs; i++)
      Z_IN(j) += X(i) * V(i,j);            /* compute sum for neuron z[j]   */
   Z(j) = net_sigmoid(Z_IN(j));            /* save sigmoid-limited z[j]     */
   }

for (k=1; k<=outputs; k++)                 /* y[k] = sum of inputs*weights  */
   {
   Y_IN(k) = W(0,k);                       /* init output sum to bias weight*/
   for (j=1; j<=hidden; j++)
      Y_IN(k) += Z(j)*W(j,k);              /* compute sum for neuron y[k]   */
   Y(k) = net_sigmoid(Y_IN(k));            /* save sigmoid-limited y[k]     */
   }

for (i=1; i<=outputs; i++)
   T(i) = TARGET(i)*ay[i]+by[i];           /* set scaled target values      */

for (k=1; k<= outputs; k++)
   error += (Y(k)-T(k))*(Y(k)-T(k));

if (!learn)                                /* if not learning..             */
   return;                                 /* ..then return                 */

for (j=0; j<=hidden; j++)                  /* zero the q[] array            */
   Q(j) = 0.0;

/*-----------------------------------------------------------------------*/
/*  back-propagate error                                                 */
/*-----------------------------------------------------------------------*/

for (k=1; k<=outputs; k++)
   {
   P(k) = (Y(k)-T(k)) * Y(k) * (1.0-Y(k));
   DEL_W(0,k) += P(k);
   for (j=1; j<=hidden; j++)
      {
      DEL_W(j,k) += P(k) * Z(j);
      Q(j) += P(k) * W(j,k);
      }
   }

for (j=1; j<=hidden; j++)
   {
   Q(j) *= Z(j) * (1.0-Z(j));
   DEL_V(0,j) += Q(j);
   for (i=1; i<=inputs; i++)
      DEL_V(i,j) += Q(j) * X(i);
   }

}
```

```
/*****************************************************************************/
/*  plot_data()                                                              */
/*****************************************************************************/

void plot_data(void)
{
long pts;
float energy, energy0;
float del_energy;
float max_intensity;
long max_struct;
long max_beam;
float max_energy;
float intensity;
char str[LINESIZE];
char str2[5];
long i, j, k;
FILE *fmtfp, *structfp;


if((fmtfp=fopen(env_value[FORMATTED],"r"))==NULL) /* open formatted file     */
   {
   printf("  Cannot open file %s\n",
          env_value[FORMATTED]);
   return;
   }

printf("  Starting energy (eV):  ");
fgets(str, LINESIZE, stdin);
sscanf(str,"%f",&energy0);

printf("  Energy step (eV):  ");
fgets(str, LINESIZE, stdin);
sscanf(str,"%f",&del_energy);

fgets(str, LINESIZE, fmtfp);               /* read in inputs               */
sscanf(str, "%d", &inputs);                /* copy to "inputs" env var     */
str[strlen(str)-1] = '\0';                 /* remove final newstr          */
strcpy(env_value[INPUTS],str);             /* copy to env var string       */

fgets(str, LINESIZE, fmtfp);               /* read in outputs              */
sscanf(str, "%d", &outputs);               /* copy to "outputs" env var    */
str[strlen(str)-1] = '\0';                 /* remove final newstr          */
strcpy(env_value[OUTPUTS],str);            /* copy to env var string       */

fgets(str, LINESIZE, fmtfp);               /* read in number of structures */
sscanf(str, "%d", &structures);            /* copy to "structures" env var */
str[strlen(str)-1] = '\0';                 /* remove final newstr          */
strcpy(env_value[STRUCTURES],str);         /* copy to env var string       */

pts = inputs / beams;                      /* find num of points per beam  */

max_intensity = 0.0;

for (i=0; i<structures; i++)
   {
   strcpy(str,"struct.");
   sprintf(str2,"%03d",i);
   strcat(str,str2);

   if ((structfp=fopen(str,"w"))==NULL)
      {
      printf ("  Error opening file %s\n",
              str);
      return;
      }

   fprintf(structfp,"%03d\n", i);          /* print structure number       */

   for (j=0; j<beams; j++)
      {
      energy = energy0;
      for (k=0; k<pts; k++)
         {
         fprintf(structfp,"%8.2f",energy); /* print energy to file         */
         fgets(str, LINESIZE, fmtfp);      /* read in next intensity       */
```

```
         fprintf(structfp,"    %s",str);    /* print intensity to file       */
         sscanf(str,"%f",&intensity);
         if (intensity > max_intensity)
            {
            max_intensity = intensity;
            max_struct = i;
            max_beam = j;
            max_energy = energy0
                        + k*del_energy;
            }
         energy += del_energy;              /* find next energy value        */
         }
      }

   for (k=0; k<outputs; k++)
      fgets(str, LINESIZE, fmtfp);          /* read in outputs & discard     */

   fclose(structfp);
   }
printf("  Max intensity = %f "
       "in structure %ld, beam %ld, "
       "at %f eV\n",
       max_intensity, max_struct,
       max_beam, max_energy);

}



/*****************************************************************************/
/*  prune()                                                                  */
/*****************************************************************************/

void prune(void)
{
/*---------------------------------------------------------------------------*/
/*  local variable declarations                                              */
/*---------------------------------------------------------------------------*/
long i, j, k;                               /* loop counters                 */
long newinputs;
char str[200];
FILE *infp, *outfp;


/*---------------------------------------------------------------------------*/
/*  open all files                                                           */
/*---------------------------------------------------------------------------*/

if ((infp=fopen(env_value[FORMATTED],"r"))==NULL)
   {
   printf ("  Error opening file %s\n",
           env_value[FORMATTED]);
   return;
   }

if ((outfp=fopen(env_value[PRUNED],"w"))==NULL)
   {
   printf ("  Error opening file %s\n",
           env_value[PRUNED]);
   return;
   }


/*---------------------------------------------------------------------------*/
/*  read the number of input and output nodes from the input file            */
/*---------------------------------------------------------------------------*/

fgets(str,200,infp);
inputs = atoi(str);

fgets(str,200,infp);
outputs = atoi(str);

rewind(infp);


/*---------------------------------------------------------------------------*/
/*  find new number of inputs                                                */
```

```
/*-------------------------------------------------------------------------*/

newinputs = 0;

for (j=0; j<inputs; j++)
   {
   if ((j % (skip+1)) == 0)
      newinputs++;
   }


/*-------------------------------------------------------------------------*/
/*  do stuff                                                               */
/*-------------------------------------------------------------------------*/

fgets(str,200,infp);
sprintf(str,"%ld\n",newinputs);
fputs(str,outfp);

fgets(str,200,infp);
fputs(str,outfp);

fgets(str,200,infp);
fputs(str,outfp);


for (i=0; i<structures; i++)
   {
   for (j=0; j<inputs; j++)
      {
      fgets(str,200,infp);
      if ((j % (skip+1)) == 0)
         fputs(str,outfp);
      }

   for (j=0; j<outputs; j++)
      {
      fgets(str,200,infp);
      fputs(str,outfp);
      }
   }


/*-------------------------------------------------------------------------*/
/*  update new number of inputs & outputs                                  */
/*-------------------------------------------------------------------------*/

inputs = newinputs;
sprintf(env_value[INPUTS],"%ld",inputs);

sprintf(env_value[OUTPUTS],"%ld",outputs);


/*-------------------------------------------------------------------------*/
/*  close all files                                                        */
/*-------------------------------------------------------------------------*/

fclose (infp);
fclose (outfp);

}




/*************************************************************************/
/*  rand1()  -  return random float  in range 0-1                        */
/*************************************************************************/

float  rand1(void)                          /* random float between 0-1      */
{
return ((float ) rand())/((float ) RAND_MAX);
}




/*************************************************************************/
```

```
/*  rand2()  -  return random int in range 1-n                              */
/*************************************************************************/

int rand2(int n)
{
n = (float ) n*rand1() + 1.0;              /* select random n, 1 to NMAX    */
return (n);
}




/*************************************************************************/
/*  rand3()  -  return random float  in range 1-n                          */
/*************************************************************************/

float  rand3(int n)
{
return (n*(((float ) rand())/((float ) RAND_MAX)));
}




/*************************************************************************/
/*  rand4()  -  return random float  in range -1 - +1                      */
/*************************************************************************/

float  rand4(void)
{
return (2.0*(((float ) rand())/((float ) RAND_MAX)) - 1.0);
}




/*************************************************************************/
/*  randomiz()                                                           */
/*************************************************************************/

void randomiz(void)
{
encoded_time = time(NULL);                 /* get encoded system time       */
systime = localtime(&encoded_time);        /* format it as local time       */
seed = (int)((10000.0/60.0)*systime->tm_sec /* create a random seed from time*/
       + (100.0/60.0)*systime->tm_min)
       * (RAND_MAX/10000.0);
sprintf(env_value[SEED],"%d",seed);        /* set env variable 'seed'       */
}




/*************************************************************************/
/*  rebeam()                                                             */
/*************************************************************************/

void rebeam(void)
{
long i, j, k;                              /* loop counters                 */
char str[200];
char str2[4];
long beam_pts;
FILE *outfp, **beamfpp;


/*-----------------------------------------------------------------------*/
/*  open all files                                                       */
/*-----------------------------------------------------------------------*/

beamfpp = (FILE **)malloc(beams*sizeof(FILE *));
if (beamfpp == NULL)
   {
   printf("  Error allocating beam file pointers\n");
   return;
   }
```

```
if ((outfp = fopen(env_value[BEAMOUTFILE],"w"))==NULL)
   {
   printf ("  Error opening file %s.\n",
           env_value[BEAMOUTFILE]);
   return;
   }

for (i=0; i<beams; i++)
   {
   printf("  Enter name of beam file %d:  ", i+1);
   fgets(str, 200, stdin);
   if ((str[0]=='\0') || (str[0]=='\n') ||
       (str[0]=='\r'))
      break;
   str[strlen(str)-1] = '\0';
   if ((*(beamfpp+i)=fopen(str,"r"))==NULL)
      {
      printf ("  Error opening file %s\n",
              str);
      return;
      }
   }

beams = i;
sprintf(env_value[BEAMS],"%ld",beams);      /* copy to env var string         */



/*-------------------------------------------------------------------------*/
/*  copy over the first three header lines                                 */
/*-------------------------------------------------------------------------*/

fgets(str, 200, *beamfpp);                  /* number of inputs from 1st beam */
beam_pts = atoi(str);                       /* find number of points per beam */
inputs = beam_pts * beams;                  /* find total number of inputs    */
sprintf(env_value[INPUTS],"%ld",inputs);    /* copy inputs to env var string  */
sprintf(str,"%d\n", inputs);                /* convert inputs to string       */
fputs(str, outfp);                          /* copy inputs to output file     */

fgets(str, 200, *beamfpp);                  /* number of outputs              */
fputs(str, outfp);                          /* copy to output file            */
outputs = atoi(str);                        /* copy to "outputs" env variable */
str[strlen(str)-1] = '\0';                  /* remove final newline           */
strcpy(env_value[OUTPUTS],str);             /* copy to env var string         */

fgets(str, 200, *beamfpp);                  /* number of structures           */
fputs(str, outfp);                          /* copy to output file            */
structures = atoi(str);                     /* copy to "structures" env var   */
str[strlen(str)-1] = '\0';                  /* remove final newline           */
strcpy(env_value[STRUCTURES],str);          /* copy to env var string         */

for (i=1; i<beams; i++)                     /* for all beams after the 1st..  */
   {
   fgets(str, 200, *(beamfpp+i));           /* ..read past the 3 header lines */
   fgets(str, 200, *(beamfpp+i));
   fgets(str, 200, *(beamfpp+i));
   }


/*-------------------------------------------------------------------------*/
/*  For each structure, read in each beam and copy to the beam output file.*/
/*-------------------------------------------------------------------------*/

for (k=0; k<structures; k++)
   {
   for (i=0; i<beams; i++)
      for (j=0; j<beam_pts; j++)
         {
         fgets(str, 200, *(beamfpp+i));
         fputs(str, outfp);
         }

   for (i=0; i<outputs; i++)
      {
      fgets(str, 200, *beamfpp);
      fputs(str, outfp);
      }
   }
```

```
/*-------------------------------------------------------------------------*/
/*  close all files                                                        */
/*-------------------------------------------------------------------------*/

fclose (outfp);
for (i=0; i<beams; i++)
   fclose(*(beamfpp+i));

}




/*************************************************************************/
/*  save_weights()                                                       */
/*************************************************************************/

void save_weights(void)
{
long i, j;
FILE *wgtfp;


if((wgtfp=fopen(env_value[WEIGHTFILE],"wb"))==NULL) /* open weight file      */
   {
   printf("  Cannot open file %s\n",
          env_value[WEIGHTFILE]);
   return;
   }

fwrite(&inputs, sizeof(long), 1, wgtfp);    /* write num of input nodes      */
fwrite(&hidden, sizeof(long), 1, wgtfp);    /* write num of hidden nodes     */
fwrite(&outputs, sizeof(long), 1, wgtfp);   /* write num of output nodes     */

fwrite(v, sizeof(float),                    /* write v weights               */
       (inputs+1)*(hidden+1), wgtfp);
fwrite(w, sizeof(float),                    /* write w weights               */
       (hidden+1)*(outputs+1), wgtfp);

fclose(wgtfp);                              /* close weight file             */
}




/*************************************************************************/
/*  scale()                                                              */
/*************************************************************************/

void scale(int verbose)
{
float ayt, byt;                             /* temp. values for ay & by      */
float tmin, tmax;
float smin, smax;
float x;
char  storeflag;
int   nn, i;


if (!strcmp(cmdstr[1],""))                  /* if no arguments were given..  */
   {
   printf("  Syntax:  scale                                "
                                  "Display scalings\n");
   printf("           scale n tmin tmax [smin smax] [n]  "
                                  "Calculate scalings\n");
   printf("           scale n value                      "
                                  "Convert\n\n");
   printf("Current scalings:\n\n");
   for (i=1; i<SCALINGS; i++)
      printf("Ay[%d] = %f\tBy[%d] = %f\n",
             i, ay[i], i, by[i]);
   return;
   }

nn = atoi(cmdstr[1]);

if ((nn<1) || (nn>=SCALINGS))
```

```
   {
   printf("Error:  Scaling index must be in"
          " the range 1-%d\n", SCALINGS-1);
   return;
   }

if (!strcmp(cmdstr[3],""))                  /* if only one argument given..  */
   {                                        /* calc. scaled/unscaled values  */
   x = atof(cmdstr[2]);                     /* convert argument to float     */
   printf("  true   -> scaled = %f\n",      /* scale if x is true value      */
          ay[nn]*x + by[nn]);
   printf("  scaled -> true   = %f\n",      /* unscale if x is scaled value  */
          (x - by[nn]) / ay[nn]);
   return;
   }

tmin = atof(cmdstr[2]);                     /* convert given tmin to float   */
tmax = atof(cmdstr[3]);                     /* convert given tmax to float   */

if (!strcmp(cmdstr[5],""))                  /* if smin & smax not given..    */
   {
   smin = 0.1;                              /* smin defaults to 0.1          */
   smax = 0.9;                              /* smax defaults to 0.9          */
   storeflag = tolower(cmdstr[4][0]);
   }

else                                        /* smin & smax are given         */
   {
   smin = atof(cmdstr[3]);                  /* convert given smin to float   */
   smax = atof(cmdstr[4]);                  /* convert given smax to float   */
   storeflag = tolower(cmdstr[6][0]);       /* get store flag                */
   }

ayt = (smax-smin)/(tmax-tmin);              /* compute ay                    */
byt = smin - ayt*tmin;                      /* compute by                    */

if (verbose)
   printf("  Ay[%d] = %f\tBy[%d] = %f\n",   /* print results                 */
          nn, ayt, nn, byt);

if (storeflag != 'n')                       /* if store flag not set to 'n'..*/
   {
   ay[nn] = ayt;                            /* store new ay                  */
   by[nn] = byt;                            /* store new by                  */
   if (verbose)
      printf("  New values stored.\n");     /* say we stored new ay & by     */
   }
}




/*****************************************************************************/
/*  separate()                                                               */
/*****************************************************************************/

void separate(void)
{
long i, j, k;                               /* loop counters                 */
char str[200];
FILE *infp, *sep1, *sep2;

/*---------------------------------------------------------------------------*/
/*  open all files                                                           */
/*---------------------------------------------------------------------------*/

if ((infp = fopen(env_value[FORMATTED],"r"))==NULL)
   {
   printf ("  Error opening formatted file \"%s\".\n",
            env_value[FORMATTED]);
   return;
   }

if ((sep1=fopen(env_value[MISSING],"w"))==NULL)
   {
   printf ("  Error opening file %s\n",
           env_value[MISSING]);
   return;
   }
```

```
if ((sep2=fopen(env_value[SINGLE],"w"))==NULL)
   {
   printf ("  Error opening file %s\n",
           env_value[SINGLE]);
   return;
   }


/*-------------------------------------------------------------------------*/
/*  copy over the first three header lines                                 */
/*-------------------------------------------------------------------------*/

fgets(str, 200, infp);                      /* number of inputs            */
fputs(str,sep1);                            /* copy to "missing" file      */
fputs(str,sep2);                            /* copy to "single" file       */
inputs = atoi(str);                         /* copy to "inputs" env variable  */
str[strlen(str)-1] = '\0';                  /* remove final newline        */
strcpy(env_value[INPUTS],str);              /* copy to env var string      */

fgets(str, 200, infp);                      /* number of outputs           */
fputs(str,sep1);                            /* copy to "missing" file      */
fputs(str,sep2);                            /* copy to "single" file       */
outputs = atoi(str);                        /* copy to "outputs" env variable */
str[strlen(str)-1] = '\0';                  /* remove final newline        */
strcpy(env_value[OUTPUTS],str);             /* copy to env var string      */

fgets(str, 200, infp);                      /* number of structures        */
structures = atoi(str) - 1;                 /* subtract one for missing struc */
sprintf(str,"%d\n",structures);             /* copy to "structures" env var   */
fputs(str,sep1);                            /* copy to "missing" file      */
fputs("1\n",sep2);                          /* copy to "single" file       */
str[strlen(str)-1] = '\0';                  /* remove final newline        */
strcpy(env_value[STRUCTURES],str);          /* copy to env var string      */


/*-------------------------------------------------------------------------*/
/*  read in ((inputs+outputs)*sepnum) lines and copy them to the "big"     */
/*  file ("missing").                                                      */
/*-------------------------------------------------------------------------*/

for (i=0; i<(inputs+outputs)*sepnum; i++)
   {
   fgets(str, 200, infp);
   fputs(str, sep1);
   }


/*-------------------------------------------------------------------------*/
/*  read in (inputs+outputs) lines and copy them to the "little"           */
/*  file ("single").                                                       */
/*-------------------------------------------------------------------------*/

for (i=0; i<(inputs+outputs); i++)
   {
   fgets(str, 200, infp);
   fputs(str, sep2);
   }


/*-------------------------------------------------------------------------*/
/*  read in (inputs-outputs)*(structures-sepnum-1) lines and copy them to the */
/*  "big" file ("missing").                                                */
/*-------------------------------------------------------------------------*/

for (i=0; i<(inputs+outputs)*(structures-sepnum-1); i++)
   {
   fgets(str, 200, infp);
   fputs(str, sep1);
   }


/*-------------------------------------------------------------------------*/
/*  close all files                                                        */
/*-------------------------------------------------------------------------*/

fclose (infp);
fclose (sep1);
fclose (sep2);
```

```
}



/*************************************************************************/
/*  set()                                                                */
/*************************************************************************/

void set(void)
{
char str[50];
short found = 0;


if (!strcmp(cmdstr[1],""))                  /* if no argument was given..    */
   {
   for (i=0; i<ENVIRONMENT; i++)            /* ..then print values of all..  */
      {                                     /* ..environment variables       */
      strcpy(str,env_var[i]);
      strcat(str,"=");
      strcat(str,env_value[i]);
      printf("  %-30s",str);

      if (i%2)                              /* if i is odd..                 */
         printf("\n");                      /* ..print a newline             */
      }

   #if (ENVIRONMENT % 2)                    /* if total num of vars is odd.. */
      printf("\n");                         /* ..print a newline             */
   #endif
   }

else
   {
   for (i=0; i<ENVIRONMENT; i++)            /* check against env var list    */
      if (!strcmp(cmdstr[1],env_var[i]))    /* if this var found in list..   */
         {
         strcpy(env_value[i],cmdstr[2]);    /* ..copy its value to env_value */
         found = 1;
         break;                             /* ..and stop search             */
         }

   if (!found)
      printf("  Variable \"%s\" not found\n",
             cmdstr[1]);

   load_env();                              /* load environmental variables  */

   }
}




/*************************************************************************/
/*  show_status()                                                        */
/*************************************************************************/

void show_status(void)
{
char statline[100];


if ((statfp=fopen(env_value                 /* open status file              */
     [STATUSFILE],"r"))==NULL)
   {
   printf ("  Cannot open file %s\n",
           env_value[STATUSFILE]);
   return;
   }

fgets(statline,100,statfp);                 /* read in one line..            */
printf(statline);                           /* ..and print it                */

fclose(statfp);                             /* close the status file         */
}
```

```
/*************************************************************************/
/*  train()                                                              */
/*************************************************************************/

void train(void)
{
char ans[20];                              /* yes/no answer to pause query  */


/*-----------------------------------------------------------------------*/
/*  set start epoch number                                               */
/*-----------------------------------------------------------------------*/

if (resume_flag == 1)
   start_epoch = resumeepoch;
else
   start_epoch = 1;


/*-----------------------------------------------------------------------*/
/*  open files & read number of input and output nodes                   */
/*-----------------------------------------------------------------------*/

if((trnfp=fopen(env_value[TRAINING], "r"))==NULL) /* open training data file */
   {
   printf("  Cannot open file %s\n",
          env_value[TRAINING]);
   return;
   }

if ((errfp=fopen(env_value[ERRORDAT], "w"))==NULL) /* open error output file */
   {
   printf("  Cannot open file %s\n",
          env_value[ERRORDAT]);
   return;
   }

fgets (line, LINESIZE, trnfp);
sscanf (line,"%ld", &inputs);              /* read in number of input nodes */
sprintf(env_value[INPUTS],"%ld",inputs);

fgets (line, LINESIZE, trnfp);
sscanf (line,"%ld", &outputs);             /* read in number of output nodes*/
sprintf(env_value[OUTPUTS],"%ld",outputs);

fgets (line, LINESIZE, trnfp);
sscanf (line,"%ld", &structures);          /* read in number of structures  */
sprintf(env_value[STRUCTURES],"%ld",
        structures);

rewind(trnfp);                             /* rewind to start of file       */

if (resume_flag == 1)                      /* if resuming a run..           */
   load_weights();                         /* ..then load old weights       */
else
   alloc();                                /* else allocate mem for network */


/*-----------------------------------------------------------------------*/
/*  initialize                                                           */
/*                                                                       */
/*  N.B.  Smith suggests the following for w weight initialization:      */
/*    if ((j%2)==0)                         if j is even..               */
/*       W(j,k) = 1.0;                      ..init w weight to +1        */
/*    else                                  if j is odd..                */
/*       W(j,k) = -1.0;                     ..init w weight to -1        */
/*  but this doesn't seem to work; dw command shows w weights all go to  */
/*  0 or 1.                                                              */
/*-----------------------------------------------------------------------*/

srand(seed);                               /* init random number generator  */

for (i=0; i<=inputs; i++)                  /* initialize v weights..        */
   for (j=0; j<=hidden; j++)
      {
      if (resume_flag != 1)
```

```
         V(i,j) = seedmult*rand4();          /* ..to random values v[i][j]    */
       E_V(i,j) = kappa;                     /* init learning rate            */
       }

for (j=0; j<=hidden; j++)                    /* initialize w weights..        */
   for (k=0; k<=outputs; k++)
      {
      if (resume_flag != 1)
         W(j,k) = seedmult*rand4();          /* ..to random values w[j][k]    */
      E_W(j,k) = kappa;                      /* init learning rate            */
      }


/*-------------------------------------------------------------------------*/
/*  teach the network to recognize leed i(v) patterns                      */
/*-------------------------------------------------------------------------*/

for (m=start_epoch; m<=epochs; m++)          /* loop to teach the network     */
   {
   fgets (line, LINESIZE, trnfp);

   fgets (line, LINESIZE, trnfp);

   fgets (line, LINESIZE, trnfp);

   error = 0.0;                              /* init the error to 0           */
   net_zero_dels();                          /* zero del_v[] and del_w[]      */

   for (ex=1; ex<=structures; ex++)
      {
      for (i=1; i<=inputs; i++)              /* get points of example         */
         {
         fgets (line, LINESIZE, trnfp);
         sscanf (line,"%e\n",x+i);
         }

      for (i=1; i<=outputs; i++)
         {
         fgets (line, LINESIZE, trnfp);
         sscanf (line, "%e\n",target+i);
         }

      network(1);                            /* run the network w/ teaching   */

      }

      net_update_weights();                  /* update weights for this epoch */

      error *= 0.5/(outputs*structures);     /* calculate error for this epoch*/

      if ((m % delprt)==0)                   /* print msg every DELPRT epochs */
         {
         printf ("  Epochs:  %ld\t", m);

         if (error >= 1.0e-6)
           printf ("Error = %f\n", error);  /* print out error in f format   */
         else
           printf ("Error = %e\n", error);  /* print out error in e format   */

         if(!strcmp(env_value[DEBUG],"on")) /* if debug mode is on..         */
            {
            printf("  Outputs:  ");
            for (i=1; i<=outputs; i++)
               printf("%e\t",Y(i));
            printf("\n");

            printf("  Un-scaled:  ");
            for (i=1; i<=outputs; i++)
               printf("%f\t",(Y(i)-by[i])/ay[i]);
            printf("\n");
            }

         if (status)
            {
            if ((statfp=fopen(env_value     /* open status file              */
               [STATUSFILE],"w"))==NULL)
               {                            /* error opening file            */
               printf("  Cannot open file %s\n",
                      env_value[STATUSFILE]);
               status = 0;
```

```
            strcpy(env_value[STATUS],"0");
            }
         else                           /* file opened ok               */
            {
            fprintf (statfp,            /* print epoch num to stat file  */
                     "  Epochs:  %ld\t",m);
            fprintf (statfp,            /* print error to stat file      */
                     "Error = %e\n",error);
            fclose(statfp);             /* close file so we can read it  */
            }
         }
      fprintf (errfp,"%ld\t%e\n",       /* send error to file            */
               m, error);
      if ((pause>0) && ((m%pause)==0))  /* if pause>0 & time to pause..  */
         {
         printf("  Continue (y/n/r/e)? "); /* ask whether to continue     */
         gets(ans);                     /* read in answer                */
         if (tolower(ans[0])=='n')      /* if we don't continue..        */
            break;                      /* ..then break out of big loop  */
         else if (tolower(ans[0])=='r') /* if we're to stop pausing..    */
            pause = -1;                 /* ..turn off the pause flag     */
         else if (tolower(ans[0])=='e')
            {
            printf("  pause=");
            fgets(ans,10,stdin);
            pause = atoi(ans);
            ans[strlen(ans)-1] = '\0';  /* remove final newline          */
            strcpy(env_value[PAUSE],ans);
            }
         }
      }

   if ((delanalyze != -1) && ((m % delanalyze)==0))
      {
      sprintf(dafilename, "i%07d.wgt", m);
      if ((danlfp=fopen(dafilename,     /* open status file              */
      "wb"))==NULL)
      {                                 /* error opening file            */
      printf("  Cannot open file %s\n",
             "dafilename");
      delanalyze = -1;
      strcpy(env_value[DELANALYZE],"-1");
      }
      else                          /* file opened ok                */
         {
         fwrite(&inputs, sizeof(long), 1, danlfp);    /* write num of input nodes      */
         fwrite(&hidden, sizeof(long), 1, danlfp);    /* write num of hidden nodes     */
         fwrite(&outputs, sizeof(long), 1, danlfp);   /* write num of output nodes     */

         fwrite(v, sizeof(float),                   /* write v weights               */
                (inputs+1)*(hidden+1), danlfp);
         fwrite(w, sizeof(float),                   /* write w weights               */
                (hidden+1)*(outputs+1), danlfp);

         fclose(danlfp);              /* close file so we can read it  */
         }
      }

   if (error < errorlim)                /* if we're below error limit..  */
      break;                            /* then stop further processing  */

   rewind(trnfp);                       /* go to start of example file   */
}

printf ("  Network is trained!\n");      /* print message                 */

if (error >= 1.0e-6)
   printf ("  Last error = %f\n", error);   /* print last error in f format  */
else
   printf ("  Last error = %e\n", error);   /* print last error in e format  */

fclose (trnfp);                         /* close training data file      */
fclose (errfp);                         /* close error data file         */
}

/******** end of file LEEDNET.C ==================*/
```

Appendix C

Listing of Program FORMAT01.C

```
/*  File  FORMAT01.C                                                        */

/*************************************************************************/
/*                                                                       */
/*                          F O R M A T 0 1                              */
/*                                                                       */
/*  This file contains the functions needed to format LEED I(V) data.    */
/*                                                                       */
/*  David G. Simpson                                                     */
/*  Department of Physics                                                */
/*  University of Maryland, Baltimore County                             */
/*  Catonsville, Maryland                                                */
/*                                                                       */
/*  This file contains one function:                                     */
/*                                                                       */
/*  format01()   Called by the LEEDNET "format" command.  This function  */
/*               re-formats the data in the file given by environmental  */
/*               variable "original", and puts the formated data into a file */
/*               whose name is given by the environmental variable       */
/*               "formatted".  (original -> formatted)                   */
/*                                                                       */
/*  Format of original data:                                             */
/*                                                                       */
/*  This data is for the Ni[50] Pd[50] (100) surface (fcc).              */
/*  (See Derry, McVey, Rous, "Surface Science", v. 326, pp. 59-66 (1995).)  */
/*                                                                       */
/*  The original data for this function (file IV.DAT) appears in 11 columns */
/*  of floating-point numbers:                                           */
/*                                                                       */
/*   Quantity      Units   Columns   Cycle order     Range      Step     Pts  */
/*                                                                       */
/*   Energy         eV      2 -  7        4        30-348 eV   2 eV     160  */
/*   d12            A       9 - 14        -        1.8700 A    const.    1  */
/*   d23            A      16 - 21        -        1.8700 A    const.    1  */
/*   d34            A      23 - 28        -        1.8700 A    const.    1  */
/*   % Ni 1         %      31 - 33        3         0-50  %     10%      6  */
/*   % Ni 2         %      36 - 38        2        50-100 %     10%      6  */
/*   % Ni 3         %      41 - 43        1        30-70  %     10%      5  */
/*   Intensity 1    -      47 - 57        -           -           -          */
/*   Intensity 2    -      61 - 71        -           -           -          */
/*   Intensity 3    -      75 - 85        -           -           -          */
/*   Intensity 4    -      89 - 99        -           -           -          */
/*                                                                       */
/*  This makes for 6 x 6 x 5 = 180 different structures, each of which has  */
/*  160 points in each of its four IV curves.  For the purposes of using the */
/*  network, the four IV curves are concatenated with each other, making one */
/*  big concatenated IV curve of 640 points for each of the 180 structures.  */
/*                                                                       */
```

```
/*                                                                          */
/*  LEEDNET format for data (ASCII text file):                              */
/*                                                                          */
/*  Number of input nodes, Ni                                  } 3-line*/
/*  Number of output nodes, Nt                                 } header*/
/*  Number of structures in file, Ns                           }       */
/*  Structure #1 1st input (beam 1 intensity at lowest energy) >       */
/*  Structure #1 2nd input (beam 1 intensity)                  > Ni    */
/*  ...                                                        > pts   */
/*  Structure #1 Ni-th input (last beam intensity at highest energy) >  */
/*  Structure #1 1st output                                    ]       */
/*  ...                                                        ] Nt    */
/*  Structure #1 last output                                   ] pts   */
/*  ...                                                                 */
/*  Structure #2 1st input                                              */
/*  ...                                                                 */
/*  Structure #2 last output                                            */
/*  ...                                                                 */
/*  Structure #Ns last output                                           */
/*                                                                      */
/*                                                                      */
/*  The file format consists of a 3-line header that defines the number of  */
/*  inputs, number of outputs, and number of structures in the file.  This  */
/*  header is followed by Ni inputs and Nt output for the 1st structure,    */
/*  Ni inputs and Nt outputs for the second structure, etc., through        */
/*  Ni inputs and Nt outputs for the Ns-th structure.  The total number of  */
/*  lines in the file should be 3 + Ns*(Ni+Nt).                             */
/*                                                                          */
/*  To write your own function to format your data into the above LEEDNET   */
/*  format, open an input "original" and output "formatted" file as shown   */
/*  here, include whatever logic is needed to format the file, and close    */
/*  the files at the end.  In this example, four "temp" files are used to   */
/*  sort the data for the individual beams before concatenating them.       */
/*                                                                          */
/****************************************************************************/

/****************************************************************************/
/*  #includes                                                               */
/****************************************************************************/

#include <stdio.h>                      /* standard i/o                     */
#include <stdlib.h>                     /* standard library                 */
#include <string.h>                     /* string functions                 */
#include "leednet.h"                    /* leednet-specific definitions     */


/****************************************************************************/
/*  external variables                                                      */
/****************************************************************************/

extern long beams;
extern long inputs;
extern long outputs;
extern long numpts;
extern long structures;
extern long sepnum;
extern long skip;
extern char env_value[ENVIRONMENT][LINESIZE];




/****************************************************************************/
/*  format01()                                                              */
/****************************************************************************/

void format01(void)
{
/*--------------------------------------------------------------------------*/
/*  local variable declarations                                             */
/*--------------------------------------------------------------------------*/
long i, j, k;                           /* loop counters                    */
double  e, d12, d23, d34, pNi1, pNi2, pNi3,
        i1, i2, i3, i4;
char str[200];
FILE *infp, *outfp, *temp1, *temp2,
     *temp3, *temp4;
```

```
/*-------------------------------------------------------------------------*/
/*  open all files                                                         */
/*-------------------------------------------------------------------------*/

if ((infp=fopen(env_value[ORIGINAL],"r"))==NULL)
   {
   printf ("  Error opening original file \"%s\".\n",
           env_value[ORIGINAL]);
   return;
   }

if ((outfp=fopen(env_value[FORMATTED],"w"))==NULL)
   {
   printf ("  Error opening formatted file \"%s\".\n",
           env_value[FORMATTED]);
   return;
   }

if ((temp1=fopen("temp1.dat","w"))==NULL)
   {
   printf ("  Error opening file TEMP1.DAT\n");
   return;
   }

if ((temp2=fopen("temp2.dat","w"))==NULL)
   {
   printf ("  Error opening file TEMP2.DAT\n");
   return;
   }

if ((temp3=fopen("temp3.dat","w"))==NULL)
   {
   printf ("  Error opening file TEMP3.DAT\n");
   return;
   }

if ((temp4=fopen("temp4.dat","w"))==NULL)
   {
   printf ("  Error opening file TEMP4.DAT\n");
   return;
   }


/*-------------------------------------------------------------------------*/
/*  start of main loop                                                     */
/*-------------------------------------------------------------------------*/
fprintf(outfp,"%d\n",inputs);
fprintf(outfp,"%d\n",outputs);
fprintf(outfp,"%d\n",structures);


for (k=0; k<structures; k++)
   {
   printf("  Structure %d\n", k);


/*-------------------------------------------------------------------------*/
/*  write one set of intensities out to temp files                         */
/*-------------------------------------------------------------------------*/
   for (j=0; j<=k; j++)
      fgets (str,200,infp);

   for (i=0; i<numpts; i++)
      {
      sscanf (str,"%lf %lf %lf %lf %lf %lf"
                  " %lf %le %le %le %le",
              &e, &d12, &d23, &d34, &pNi1,
              &pNi2, &pNi3, &i1, &i2, &i3,
              &i4);
      fprintf (temp1, "%le\n", i1);
      fprintf (temp2, "%le\n", i2);
      fprintf (temp3, "%le\n", i3);
      fprintf (temp4, "%le\n", i4);
      for (j=0; j<structures; j++)
         fgets (str, 200, infp);
      }


/*-------------------------------------------------------------------------*/
```

```
/*  done writing to temp files, so close them                                 */
/*----------------------------------------------------------------------------*/

   fclose(temp1);
   fclose(temp2);
   fclose(temp3);
   fclose(temp4);


/*----------------------------------------------------------------------------*/
/*  open temp files back up (this time for reading), and print out their      */
/*  contents into iv.out                                                      */
/*----------------------------------------------------------------------------*/

   if ((temp1=fopen("temp1.dat","r"))==NULL)
      {
      printf ("  Error opening file TEMP1.DAT\n");
      return;
      }

   for (i=0; i<numpts; i++)
      {
      fgets (str,200,temp1);
      sscanf (str,"%le", &i1);
      fprintf (outfp, "%le\n", i1);
      }


   if ((temp2=fopen("temp2.dat","r"))==NULL)
      {
      printf ("  Error opening file TEMP2.DAT\n");
      return;
      }

   for (i=0; i<numpts; i++)
      {
      fgets (str,200,temp2);
      sscanf (str,"%le", &i2);
      fprintf (outfp, "%le\n", i2);
      }


   if ((temp3=fopen("temp3.dat","r"))==NULL)
      {
      printf ("  Error opening file TEMP3.DAT\n");
      return;
      }

   for (i=0; i<numpts; i++)
      {
      fgets (str,200,temp3);
      sscanf (str,"%le", &i3);
      fprintf (outfp, "%le\n", i3);
      }


   if ((temp4=fopen("temp4.dat","r"))==NULL)
      {
      printf ("  Error opening file TEMP4.DAT\n");
      return;
      }

   for (i=0; i<numpts; i++)
      {
      fgets (str,200,temp4);
      sscanf (str,"%le", &i4);
      fprintf (outfp, "%le\n", i4);
      }


/*----------------------------------------------------------------------------*/
/*  print output parameters to iv.out                                         */
/*----------------------------------------------------------------------------*/

   fprintf (outfp,"%lf\n%lf\n%lf\n", pNi1, pNi2, pNi3);


/*----------------------------------------------------------------------------*/
/*  close temp files again                                                    */
/*----------------------------------------------------------------------------*/
```

```
   fclose(temp1);
   fclose(temp2);
   fclose(temp3);
   fclose(temp4);


/*-------------------------------------------------------------------------*/
/*  open temp files back up, this time for writing                         */
/*-------------------------------------------------------------------------*/

   if ((temp1=fopen("temp1.dat","w"))==NULL)
      {
      printf ("  Error opening file TEMP1.DAT\n");
      return;
      }

   if ((temp2=fopen("temp2.dat","w"))==NULL)
      {
      printf ("  Error opening file TEMP2.DAT\n");
      return;
      }

   if ((temp3=fopen("temp3.dat","w"))==NULL)
      {
      printf ("  Error opening file TEMP3.DAT\n");
      return;
      }

   if ((temp4=fopen("temp4.dat","w"))==NULL)
      {
      printf ("  Error opening file TEMP4.DAT\n");
      return;
      }

   rewind(infp);
   }


/*-------------------------------------------------------------------------*/
/*  close all files                                                        */
/*-------------------------------------------------------------------------*/

fclose (infp);
fclose (outfp);
fclose (temp1);
fclose (temp2);
fclose (temp3);
fclose (temp4);

}

/*======== end of file FORMAT01.C ==================*/
```

Appendix D

Listing of File LEEDNET.H

```
/*  File  LEEDNET.H                                                 */

#define LINESIZE     95                 /* size of input lines          */

#define ENVIRONMENT  39                 /* # of environment variables   */

#define ORIGINAL     0                  /*   original data file name    */
#define FORMATTED    1                  /*   formatted data file name   */
#define MISSING      2                  /*   missing I struct file name */
#define SINGLE       3                  /*   1 structure file name      */
#define PRUNED       4                  /*   pruned data file name      */
#define TRAINING     5                  /*   training data file name    */
#define ERRORDAT     6                  /*   error vs epoch file name   */
#define DUMPFILE     7                  /*   dump file name             */
#define HELPFILE     8                  /*   help file name             */
#define PLOTFILE     9                  /*   plot data file name        */
#define INPUTS       10                 /*   number of input nodes      */
#define HIDDEN       11                 /*   number of hidden nodes     */
#define OUTPUTS      12                 /*   number of output nodes     */
#define FUNCTION     13                 /*   data conv function number  */
#define EPOCHS       14                 /*   num of epochs to teach     */
#define DELPRT       15                 /*   num of epochs to prt message*/
#define MU           16                 /*   momentum rate for learning */
#define KAPPA        17                 /*   amt to incr learning rate  */
#define PHI          18                 /*   factor to mult learning rate*/
#define THETA        19                 /*   ctrl time period for avg'ing*/
#define WEIGHTFILE   20                 /*   saved net weights file name */
#define SEED         21                 /*   seed for rand num generator */
#define NUMPTS       22                 /*   num of points in i(v) curve */
#define STRUCTURES   23                 /*   num structures in data set  */
#define SEPNUM       24                 /*   structure num to extract    */
#define SKIP         25                 /*   interval to skip pruned data*/
#define STATUS       26                 /*   training status on/off flag */
#define STATUSFILE   27                 /*   status filename             */
#define ADAPTIVE     28                 /*   adaptive learning rate flag */
#define PAUSE        29                 /*   training pause (epochs)     */
#define DEBUG        30                 /*   debug mode on/off           */
#define BEAMS        31                 /*   number of beams in input    */
#define ERRORLIM     32                 /*   training error limit        */
#define ANLINFILE    33                 /*   analysis input file         */
#define ANLOUTFILE   34                 /*   analysis output file        */
#define BEAMOUTFILE  35                 /*   rebeam output file          */
#define DELANALYZE   36                 /*   num of epochs to analyze    */
#define RESUMEEPOCH  37                 /*   resume epoch number         */
#define SEEDMULT     38                 /*   random seed multiplier      */

/*======== end of file LEEDNET.H ==================*/
```

Appendix E

Listing of File LEEDNET.HLP

```
File LEEDNET.HLP
***general
  alloc      Allocate memory for the network
  analyze    Analyze a network from its training data
  ask        Ask a trained network to process input
  debeam     Separate data set into individual beams
  dump       Dump network weights
  dw         Display network weight
  env        Display help on environment variables
  exit       Quit LEEDNET
  format     Re-format data to LEEDNET format
  help       Help on LEEDNET commands
  load       Load network weights
  mem        Display memory usage report
  merge      Merge data files together
  plot       Generate I(V) plot data
  prune      Prune points from a data set
  quit       Quit LEEDNET
  randomize  Initialize random number generator from system time
  rebeam     Re-assemble data set from individual beams
  resume     Resume a training run
  save       Save network weights
  scale      Set/display network output scaling constants
  separate   Separate data in a data set
  set        Set/display environment variables
  status     Show network training status
  train      Train a network
  ver        Display LEEDNET version number
***alloc
  alloc      Allocates memory for the network.
      The memory allocated is based on the number of
      input, hidden, and output nodes currently defined
      (environment variables "inputs", "hidden", and
      "outputs".  The memory is dynamically allocated on
      the heap.
***analyze
  analyze    Analyze a network from its training data.
      "Analyze" reads the number of inputs, outputs,
      and structures from the "training" file.  It then
      loads network weights from "weightfile".  Finally,
      it shows each structure in "anlinfile" to the
      network and compares the network's outputs to the
      expected outputs in the training data.  The statistics
      on the final results are sent to the file defined by
      the environment variable "anloutfile."
***ask
  ask        Asks a trained network to process input.
      Shows the file defined by environment variable "single"
```

```
      to the network, runs the network, and displays its
      outputs.
***debeam
  debeam      Separate dataset into individual beams.
      The file defined by the environment variable "formatted"
      contains the LEEDNET-format I(V) data, variable "beams"
      should be set to the number of separate beams in each
      spectrum, and variable "inputs" should be set to the total
      number of points in each spectrum.  Each I(V) spectrum is
      divided into "beams" equal data sets; the output files are
      named "beam.nnn".
***dump
  dump        Dump network weights.
      The current values of all network weights are dumped to the
      file whose name is given by the environment variable
      "dumpfile".
***dw
  dw          Display a single network weight.
      The command syntax is:  dw  v|w  i  j
      where
          v   asks to display an input-to-hidden weight
          w   asks to display a hidden-to-output weight
          i,j are the weight indices
***env
  Environment variables are set using the "set" command:
     set <varname>=<value>
  Typing "set" with no argument displays the current values
  of all environment variables.  The available environment
  variables are:
     adaptive     Adaptive learning rate flag (1=on, 0=off)
     anlinfile    Analysis input file (for "analyze" command)
     anloutfile   Analysis output file (for "analyze" command)
     beamoutfile  Beam output file (for "rebeam" command)
     beams        Number of beams in input (for "debeam" and "rebeam")
     debug        Debug mode on/off flag
     delanalyze   Interval (epochs) to save network weights
     delprt       Number of epochs to print msg and save error report
     dumpfile     Network weight dump file name
     epochs       Number of epochs to train the network
     errordat     Error vs. epoch output file name
     errorlim     Training error limit
     formatted    Formatted data file name (LEEDNET format)
     function     Selects a formatting function for "format" command
     helpfile     Help file name
     hidden       Number of network hidden nodes
     inputs       Number of network input nodes
     kappa        Kappa parameter for adaptive learning
     missing      Missing 1 structure file name (for "separate" cmd)
     mu           Momentum parameter for learning rate
     numpts       Number of points in I(V) curve
     original     Original data file name (for "format" command)
     outputs      Number of network output nodes
     pause        Training pause (epochs)
     phi          Phi parameter for adaptive learning
     plotfile     Plot data output file name
     pruned       Pruned data file name ("prune" command)
     resumeepoch  Resume epoch (for "resume" command)
     seed         Seed for random number generator
     seedmult     Random seed multiplier
     sepnum       Number of structure to be isolated for "separate" cmd
     single       1-structure file name ("separate" and "ask" cmds)
     skip         Number of points to skip for "prune" command
     status       Training status on/off flag (1=on, 0=off)
     statusfile   Training status file name
     structures   Number of structures in data set
     theta        Theta parameter to control averaging period
     training     Training data file name
     weightfile   File name under which to save or load network weights
***exit
  exit        Quit LEEDNET.
***format
  format      Re-format data into LEEDNET format.
      I(V) data is re-formatted from its original format into the
      format used by LEEDNET.  Several different formatting
      functions may be available; they are selected using the
      environment variable "function".
        original -> formatted
***help
  help        Help on LEEDNET commands.
      Type "help"     for a list of available commands.
```

```
      Type "help cmd" for detailed help on command "cmd".
      Type "help env" to display help on environment variables.
***load
  load        Load network weights.
      Network weights are loaded from the file defined by
      environment variable "weightfile".  The number of input
      nodes, output nodes, and structures (variables "inputs",
      "outputs", and "structures") will also be loaded.
***mem
  mem         Display memory usage report.
      Type "mem" to display the current network memory usage.
      Type "mem <inputs> <hidden> <outputs>" to display the
         memory that would be required for a network of the
         specified number of input, hidden, and output nodes.
***merge
  merge       Merge several data files together.
      The "merge" command will prompt for the name to be given
      to the output file.  It will then ask for the names of
      the input files, each of which should contain one structure
      (created, for example, by the "separate" command).  Enter a
      carriage return by itself after the last file name is
      entered.  When done, manually update line 3 of the output
      file (total number of structures).
***plot
  plot        Generate I(V) plot data.
      The file given by the environment variable "formatted" is
      broken into spectra for individual structures, and I vs. E
      data is saved into files named "struct.nnn".  The data in
      each file is in ASCII format and is suitable for plotting
      with a spreadsheet program.
***prune
  prune       Prune points from a data set.
      The data file specified by the environment variable
      "formatted" is "pruned" by keeping only every n-th point,
      where "n" is given by the environment variable "skip".
      The result is saved in the file specified by the
      environment variable "prune".
        formatted -> pruned
***quit
  quit        Quit LEEDNET.
***randomize
  randomize       Initialize random number generator from system time.
***rebeam
  rebeam     Combine data sets for individual structures into one file.
      The user is prompted for the names of the files containing
      the individual beam data, and the beam data is combined into a
      single file whose name is specified by the "beamoutfile"
      environment variable.
***resume
  resume      Resume a training run.
      To resume a training run, type:
         set weightfile=<weight filename>
         set resumeepoch=<epoch number>
         resume
***save
  save        Save network weights.
      Network weights are saved to the file defined by
      environment variable "weightfile".  The number of input
      nodes, output nodes, and structures (variables "inputs",
      "outputs", and "structures") will also be saved.
***scale
  scale   Display or set network output scaling constants.
       Type "scale" with no arguments to display the current
      values of all scaling constants.
       Type "scale <n> <tmin> <tmax> [<smin> <smax>] [n]"
      to calculate and save scaling constants for output <n>
      for "true" values ranging between <tmin> and <tmax>.
      The optional minimum and maximum scaled values <smin>
      and <smax> default to 0.1 and 0.9, respectively.  If
      an "n" is specified, the scaling is calculated but not
      stored.
       Type "scale <n> <value>" to perform true->scaled and
      scaled->true conversions of <value> for output <n>.
***separate
  separate   Separate data in a data set.
      A single structure is isolated from the data set specified
      by the environment variable "formatted".  The environment
      variable "sepnum" should be set to the number (starting
      from 0) of the structure to be isolated.  The "separate"
      command will then place the spectrum for that structure
```

```
      into the file whose name is given by the "single" variable,
      and the remaining spectra will be placed into the file
      whose name is given by the "missing" variable.
        formatted -> missing, single
***set
  set         Set/display environment variables.
      Type "set" with no arguments to display the current values
      of all environment variables.
      Type "set <varname>=<value>" to set an environment variable
      to a new value.
***status
  status      Show network training status.
      The network training status is peroidically stored in the
      file defined by the environment variable "statusfile",
      provided the "status" variable is set to 1.  The "status"
      command displays the contents of this file.
***train
  train       Train the network.
      The file specified by the "training" environment variable
      is used to train the network for "epochs" training epochs.
      Type "train [cont]" to continue training that has been
      stopped.
***ver
  ver         Display LEEDNET version number.
***
```

Bibliography

[1] M. Prutton. *Introduction to Surface Physics.* Oxford University Press, 1994.

[2] A. Zangwill. *Physics at Surfaces.* Cambridge University Press, 1988.

[3] M.W. Finnis and V. Heine. *J. Phys.*, F**4:**L37, (1974).

[4] C.J. Davisson and L.H. Germer. *Phys. Rev.*, **30:**705, (1927).

[5] B.D. Cullity. *Elements of X-Ray Diffraction.* Addison-Wesley, 1978.

[6] M.A. Van Hove, W.H. Weinberg, and C.-M. Chan. *Low-Energy Electron Diffraction.* Springer-Verlag, 1986.

[7] C. Kittel. *Introduction to Solid State Physics.* Wiley, 7th edition, 1996.

[8] G. Arfken. *Mathematical Methods for Physicists.* Academic, 3rd edition, 1985.

[9] J.E. Houston and R.L. Park. *Surf. Sci.*, **21:**209, (1970).

[10] R.L. Park, J.E. Houston, and D.G. Schreiner. *Rev. Sci. Instrum.*, **42:**60, (1971).

[11] M.A. Van Hove and S.Y. Tong. *Surface Crystallography by LEED.* Springer-Verlag, 1979.

[12] P.M. Morse and H. Feshbach. *Methods of Theoretical Physics*. McGraw-Hill, 1953.

[13] J.L. Beeby. *J. Phys. C*, **1:**82, (1968).

[14] J.B. Pendry. *J. Phys. C*, **13:**937, (1980).

[15] S. Andersson, B. Kasemo, J.B. Pendry and M.A. Van Hove. *Phys. Rev. Lett.*, **31:**595, (1973).

[16] J.E. Demuth, D.W. Jepsen, and P.M. Marcus. *Phys. Rev. Lett.*, **31:**540, (1973).

[17] S.Y. Tong and K.H. Lau. *Phys. Rev. B*, **25:**7382, (1982).

[18] T.H. Upton and W.A. Goddard. *Phys. Rev. Lett.*, **46:**1635, (1981).

[19] E.W. Swokowski. *Calculus with Analytic Geometry*. Prindle, Weber & Schmidt,, 2nd edition, 1979.

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

[21] R.L. Burden and J.D. Faires. *Numerical Analysis*. PWS-KENT Publishing, 4th edition, 1989.

[22] P.J. Rous. *Surf. Sci.*, **296:**358, (1993).

[23] E. Aarts and J. Korst. *Simulated Annealing and Boltzman Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. Wiley, 1989.

[24] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[25] P.J. Russell. *Genetics*. Harper-Collins, 4th edition, 1996.

[26] R. Döll and M.A. Van Hove. *Surf. Sci.*, **355:**L393, (1996).

[27] B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, and J.D. Watson. *Molecular Biology of the Cell*. Garland Publishing, 3rd edition, 1994.

[28] D. Purves, G.J. Augustine, D. Fitzpatrick, L.C. Katz, A-S. LaMantia, and J.O. McNamara, editors. *Neuroscience*. Sinauer Associates, 1997.

[29] L. Fausett. *Fundamentals of Neural Networks*. Prentice-Hall, 1994.

[30] F. Harary. *Graph Theory*. Addison-Wesley, 1969.

[31] K. Swingler. *Applying Neural Networks: A Practical Guide*. Academic Press, 1996.

[32] M. Smith. *Neural Networks for Statistical Modeling*. Van Nostrand Reinhold, 1993.

[33] M.H. Hassoun. *Fundamentals of Artificial Neural Networks*. MIT Press, 1995.

[34] G.N. Derry, C.B. McVey, and P.J. Rous. *Surf. Sci.*, **326:**59, (1995).

[35] S. Crampin and P.J. Rous. *Surf. Sci. Lett.*, **244:**L137, (1991).

[36] P.J.M. Laarhoven and E.H.L. Aarts. *Simulated Annealing: Theory and Applications*. Reidel, (1987).

Index

action potential, 44
activation function, 46, 50
adaptive learning rates, 59
adenosine 5'-triphosphate, 42
annealing, 17
    simulated, 35
ATP, *see* adenosine 5'-triphosphate
axon, 42

backpropagation, 48
batch learning, 59, 61, 120
bias weight, 50
bipartite graph, 48

chromosomes, 36
crossover, 37
cytoplasm, 42
cytosol, 42

Davisson-Germer experiment, 6
Debye-Waller factor, 12
dendrites, 44
dynamical calculations, 19
dynamical calculations,, 11

electron gun, 12
elitism, 39
endoplasmic reticulum, 42
environment variables, 130
exhaustive global search, 32

feedforward, 52
fitness function, 36

genetic algorithms, 36
Golgi apparatus, 42
graph
    bipartite, 48

hidden nodes, 48

I-φ curves, 9
I-θ curves, 9

I-V curves, 9
inner potential, 20, 21
    energy shift, 95
input nodes, 46
instrument response function, 16, 88
intensity, 94
interatomic scattering, 20
ion channels
    gated, 44
    transmitter-gated, 44
    voltage-gated, 44

kinematic calculation, 19

layer doubling, 29
learning rate, 58
LEED, *see* low-energy electron diffraction
LEEDNET, 64
low-energy electron diffraction, 2, 6–17

mating pool, 37
medial nodes, 48
mitochondria, 42
momentum, 59
muffin-tin
    constant, 21
    potential, 20
    radius, 21
multiple scattering, 11, 19
mutation, 38

neuron, 41
neurotransmitter, 44
nodes, 46
    input, 46

Pendry
    R-factor, 31, 36, 39, 103
    RR-factor, 32
    Y-function, 31, 103, 109, 112, 122
phase shift, 24
phonon, 22
plasma membrane, 42
plasmon, 22

R-factor, *see* reliability factor
registry, 33
relaxation, 3
reliability factor, 30, 32, 36, 39, 103
renormalized forward scattering, 29, 88
reproduction, 37

RFS, *see* renormalized forward scattering

RHEED, 6

sample preparation, 17

scaling, 50, 113

scattering

    interatomic, 26

    intra-atomic, 20, 24

selvedge, 3

sigmoid function, 50

simulated annealing, 35

sputtering, 17

steepest descent, 33

suppressor grid, 14, 22

surface physics, 1

synapse, 44

synaptic cleft, 45

synaptic vesicles, 44

terminal branches, 42

thermal effects, 11

training, 47

transfer function, 16

UHV, *see* ultra-high vacuum

ultra-high vacuum, 2, 12

unsupervised learning, 47

weight, 46, 49

x rays, 6, 9, 19

Y-function, 31, 103, 109, 112, 122