12-1998

A Portable, Object-Oriented Library for Neural Network Simulation

Martin P. Franz

Recommended Citation

Franz, Martin P., "A Portable, Object-Oriented Library for Neural Network Simulation" (1998). *Master's Theses*. 4239.
https://scholarworks.wmich.edu/masters_theses/4239

A PORTABLE, OBJECT-ORIENTED LIBRARY FOR NEURAL NETWORK SIMULATION

by

Martin P. Franz

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
December 1998

ACKNOWLEDGEMENTS

My inspiration for this research was the work of people whom I have never met. Dr. Andrew Barto and Dr. Rich Sutton demonstrated by their research that neural networks (in particular those using temporal difference learning) could be put to powerful and interesting uses. The many books of Dr. Donald Knuth show that programming can be a fascinating and profitable craft. Dr. Bjarne Stroustrup deserves mention for making C++ a pleasant and powerful programming language whose many features I tried to utilize fully in this library.

I also wish to thank those who served on my committee and leant their wisdom and encouragement to me: Dr. Robert Trenary, Dr. Ajay Gupta, and Dr. John Kapenga. In particular, Dr. Trenary patiently read many drafts of this thesis with a sharp red pencil in hand. The reader may thank him as well for what is not here as for what is. I should point out, however, that any errors remaining in this work are solely my own.

Finally, I wish to thank my wife, Maria, my son, Graham, and the rest of my family, friends, and colleagues for tolerating my fascination with computing in general and neural networks in particular. They bore well the long hours I was absent while working on this project.

Martin P. Franz

A PORTABLE, OBJECT-ORIENTED LIBRARY FOR NEURAL NETWORK SIMULATION

Martin P. Franz, M.S.

Western Michigan University, 1998

A portable, object-oriented library for simulation of general Multi-layer Feedforward Neural Networks (MLFNs) is described. Unlike all-encompassing neural network simulation environments, the library was designed to allow convenient use in existing programs and in applications where training and testing data are generated using separate, often complex simulations.

The library's design goals include modularity, portability, efficiency, correctness, compactness, and type-safety. To demonstrate how these objectives are met, competing architectural choices are presented, along with the criteria used for determining the strategy actually implemented.

Sample applications using the library are presented, showing how the library's class files are used in neural network simulations. Finally, the performance of the library is evaluated to demonstrate that the neural network algorithms chosen exhibit modest run-time and storage costs.

TABLE OF CONTENTS

CHAPTER

Table of Contents—continued

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Artificial Neural Systems (ANSs) have become a popular tool for solving many computational problems. These include statistical correlation, signal processing, pattern storage and recognition, game playing, and other applications where traditional solutions are limited by the need for *a priori* understanding of the problem's constraints and structure. An ANS (also called a "neural network") offers some decided advantages over traditional computational solution techniques. These advantages include statistical robustness, a degree of built-in fault tolerance, and the ability to work in either supervised or unsupervised capacities (Masters, 1993).

Unfortunately, developing software to implement a neural network can be frustrating and time-consuming. Most useful ANS algorithms (such as backpropagation) require extensive programming and computation time to reach a solution. Properly documented test cases are difficult to find, and the software available to the researcher tends to fit the category of large, complex graphical "environments" rather than small, reusable code modules (Plonski and Joyce, 1990). These environments, while acceptable for prototyping and visualization, impose limitations on the types of problems that can be conveniently solved by the researcher and student.

This thesis describes the design and construction of a C++ library for implementing neural network programs. This library is limited to multi-layer, feedforward networks, and within this subset of neural network architectures the user may employ the object-oriented features of C++ to easily create a network to directly solve a given problem. Rather than a graphical environment, simple constructs (such as streamable classes) have been implemented to allow the user to naturally express the problem in a standard programming language. The objective was to fit the ANS model of neurons, connections, and learning rules into the structure of C++. It is hoped that this library will make it convenient and pleasant for the neural network researcher to write neural network programs.

Purpose and Scope

This thesis is not a user's guide to the library. Instead, it details the design issues involved in the construction of this library, and the way in which these issues were resolved in implementation. It is assumed that the reader knows something about neural networks, C++, and object-oriented programming, although appropriate introductory material is included here. In most cases, a design issue is first described, several competing alternative solutions are mentioned, and the solution chosen is then presented in detail. Where possible the correctness of the choice is then validated with experimental data. It is hoped that this practical approach will result in the reader's appreciation of the trade-offs involved in the library's design, and a better understanding of it's utility.

This thesis is organized into the following chapters:

Chapter II covers the basic architecture of the library. Here is where the organization of the library's class files is presented. The need for general, reusable collection classes is mentioned. Several competing storage strategies (such as using the Standard Template Library in C++, or omitting collections altogether) are described.

Chapter III covers the structure for application programs that use the library. In addition to the basics of creating and simulating a neural network, the library also provides provisions for debugging and tracing, and adjusting parameters at run time through an initialization file mechanism.

Chapter IV describes the neural network algorithms implemented using the structure described in the previous two chapters. These include backpropagation, Widrow-Hoff, and TD(λ). Introductory material on these algorithms, and analysis of their performance, is also provided.

Chapter V covers performance issues. Is the C++ implementation of these algorithms as fast and compact as a matching C implementation? If not, the utility of the library would be seriously compromised. Three test cases are shown. This chapter also compares the performance of the C++ collection classes to their C counterparts.

Chapter VI covers exception handling and type safety. One of the reasons to use C++ is that the typing mechanism in the compiler may be employed to virtually guarantee that programs that compile correctly will run correctly. To accomplish this, additional methods need to be written for each class.

Chapter VII presents three examples in detail. It is hoped that these examples will demonstrate the power and compactness of the library's design.

Chapter VIII gives some conclusions and future directions for this research.

Throughout this thesis, C++ source code has been included where appropriate. This code is set in `monospace` font. Some source statements are shown in **`boldface`** in these listings to highlight statements explained further in the text. In the text, `monospace` font has been used for items as they would appear on a computer system, such as file and function names. For clarity, however, class names are set in proportional font.

The rest of this chapter covers some history and prior implementations of neural networks.

Historical Background of Problem

The research of McCulloch and Pitts in the 1940s is generally agreed to be the founding of the field of Neural Networks (McCulloch and Pitts, 1943). They demonstrated that networks of simple neurons could compute any function computable by a Turing machine. Donald Hebb, who tied the functioning of biological neurons to a model of classical conditioning, and who presented a mathematical model by which reinforcement learning could occur, followed their work (Hebb, 1949).

In the 1950s, Frank Rosenblatt invented the Perceptron, the first practical neural network in the sense that it could be implemented on a digital computer and used for simple applications (Rosenblatt, 1958). His success, and the success of Widrow and Hoff (1960), prompted keen interest in the field. It was believed at the time that networks of neurons would, eventually, allow computers to demonstrate cognitive functions.

Unfortunately, Papert and Minsky (1969) soon demonstrated that networks like the Perceptron and the adaptive linear neurons of Widrow and Hoff were insufficient for solving some types of simple problems, and interest in the field of neural networks waned. As a result, the field lay dormant until the 1980s. Then, an increase in computing power and the development of the backpropagation training algorithm allowed newer types of networks to be developed. These breakthroughs allowed larger, multi-layer, feedforward networks to be implemented and trained, overcoming the limitations found by Minsky and Papert.

At the time of this writing, using multi-layer, feedforward neural networks (MLFNs) has become a well-understood method for solving a variety of computing problems. Because of this, there are now a number of software systems available that allow the researcher to create neural networks and simulate their function. These include SNNS, RCS, Aspirin/MIGRAINES, and others (Plonski and Joyce, 1990).

These packages may be considered "environments" in the sense that the user runs a single, large program and then interacts with it through a user interface to (a) create the network, (b) specify training and testing data, (c) simulate its function, and (d) evaluate performance. In some cases, (such as Aspirin/MIGRAINES) a separate, C-like programming language is used to control this interaction. In other cases (such as SNNS), a complete graphical user interface (GUI) has been implemented, with multiple windows, drag and drop network configuration, and graphical visualization of the network's output.

This environmental approach to neural network simulation has many advantages, but it can also be limiting. First, these tools have a substantial learning curve associated with them, even with the examples provided. Second, it is difficult to write a separate program to interact with the environment. If the researcher is constructing a control system simulation, for example, getting the system's sensory and evaluation functions to interact with the neural network part of the simulation (which is running within the environment) can be problematic. Finally, once the network has been trained, there is no easy way to encapsulate it for inclusion in a production program.

For these reasons, a library-based approach is desirable as an implementation alternative. A neural network library would allow the researcher to implement neural network functionality in a program in a more traditional, modular manner. The ideal neural network library would provide a set of reusable functions for creating the network; simulating its operation with training and testing data; and evaluating its performance. The user's program would be responsible for calling the library through a well-described, type-safe interface. In addition, it would include provisions for handling exceptions (such as running out of storage), overriding the basic learning rules, and adding new network architectures based on those already provided.

These, then, were the goals of the Monarch neural network library described in this thesis. In the next chapter, the overall architecture of the library and its associated data structures will be presented.

CHAPTER II

LIBRARY ARCHITECTURE

In this chapter, and the one that follows, we examine the architecture of the library. Here we will consider the internal construction of the library, isolating the fundamental objects and data structures needed to construct neural networks. In the next chapter, we will examine the library from an external (or application programmer's) viewpoint, to determine the amenities required to make writing neural network programs as pleasant an experience as possible.

Before going further, we shall define what we mean when we say "pleasant". Programming using a tool may be called "pleasant" when that tool allows us to write programs which are *correct* (the program performs the way we intend it to); *compact* (it provides sensible default behaviors and sufficiently powerful constructions so no extra lines of code are required), and *robust* (the tool handles exceptions properly, and creates data types that can be used naturally). These three interrelated attributes—correctness, compactness, and robustness—together comprise the idea of "pleasantness" for which we will strive as we design this library.

With these criteria in mind, let us now examine the structure of a neural network, and creation of an object-oriented design to manipulate it. Object-oriented design has been covered in many sources, including Booch (1991) and Stroustrup (1991), and will not be described in detail here. The properties of an object-oriented programming language include encapsulation (information hiding), polymorphism, dynamic binding, and inheritance. These make it possible for the programmer to create software "objects", incorporating both data and associated behavior, which accurately model the real-world problem domain. For convenience, objects are grouped into a hierarchy of *classes*, which are templates from which objects are created. In OOP terminology, the process of creating a specific object from a class is *instantiation*; the data within the object resides in one or more *instance variables*; and the action of the object is specified with a *method* activated by a *message* sent from another object in the running program (Franz, 1990).

The basic technique for object-oriented design is to (a) describe the real-world system to be modeled by the object-oriented program; (b) factor this model into various objects and actions; and (c) create software models of these objects (classes) and actions (methods) using an object-oriented language. With a sufficiently general and expressive language, such as C++, we can employ inheritance (a class's use of behavior and structure from a parent class) and polymorphism (similar classes having similar protocols for passing messages and invoking behavior) to reduce the amount of software needed.

Before factoring the design into classes and methods, we need to understand the components of the neural network model in more detail. This is covered in the next section.

The Neural Network Model

A neural network is simply a linked collection of neurons. A single neuron is a mathematical abstraction of a biological neuron. (Throughout this thesis, the notation used in Hagan, Demuth, and Beale (1995) is used.) A biological neuron, in turn, is a type of cell found in human and animal nervous systems, although the correlation between it and the abstract neuron used in a neural network is very loose.

A Single Neuron

A diagram of a single mathematical neuron is shown in Figure 1:

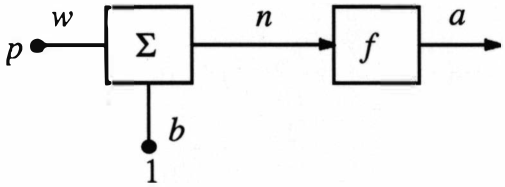

Figure 1. A Single Neuron.

In this simple model, the scalar input *p* is multiplied by a scalar weight, *w*, which is added to a scalar bias term. The bias term is typically 1. The summed output of the weight and the bias is called *n* and is fed, in turn, into an activation function *f*. This can be one of many functions, depending on the architecture of the network, although functions commonly employed are the log-sigmoid function, step function, and linear function. The output of the activation function is the scalar *a*. This is considered to be the output of the neuron.

For computational purposes, the output of the neuron is given by

$$a = f(wp + b)$$

Note that both *w* and *b* are adjustable parameters of the network. It is the network designer's job to select the function *f()*, and an appropriate learning rule, such that *w* and *b* may be computed to meet some specific goal for the network's output when presented with a given input.

Multiple Neurons

If we generalize *p* and *w* into vectors we obtain a generalized neuron. Then the equation for the neuron's output becomes:

$$a = f(\mathbf{W}\,\mathbf{p} + b)$$

From here it is a small step in generality to consider a *layer* of neurons that share the same input vector. This is illustrated in Figure 2.

In a layer of neurons, each one of *R* inputs is connected to each of every one of *S* neurons, and therefore the weights can now be considered a matrix. It is common for the number of inputs to be different than the number of neurons in the layer, so $R \neq S$. Each neuron in the layer has a separate bias, summing unit, and transfer function, although usually the transfer functions for all the neurons in a layer are the same. If different transfer functions are required for a layer, it is usually more convenient to have logically separate layers that share the same inputs and combine their outputs. The outputs from each transfer function form an output vector.

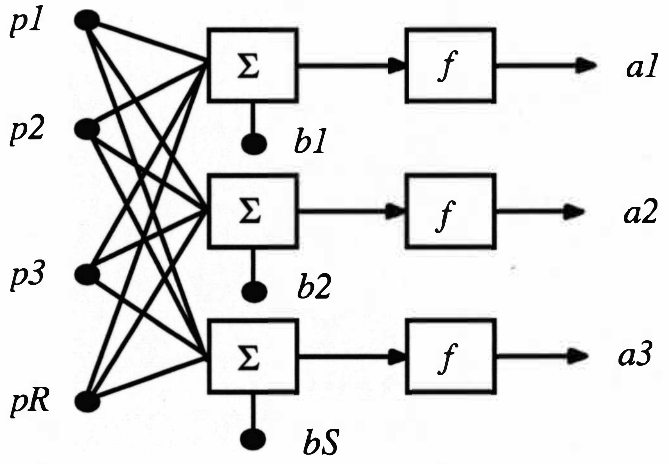

Figure 2. A Layer of Neurons.

The weights in a layer of neurons may be viewed as a matrix, as shown below:

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

It is customary to make the row indices of the elements of matrix **W** indicate the destination neuron associated with that weight and the column indices indicate the source input. Hence, $w_{2,1}$ indicates the weight to the second neuron from the first input. Note that **W** contains, in effect, the state of the layer. This means a program may save and restore **W** to effectively stop and restart training of the network from a particular point. This is a technique used in training backpropagation networks (Kutza, 1996).

Layers of neurons can be combined into multiple layers by making the output of one layer the input of another. With the appropriate learning rule, this aggregation of layers may be trained as a unit to produce a composite network of neurons. In neural network terminology, the layer whose input is the input to the network is called the *input layer*, the layer whose output is the output of the network is called the *output layer*, and all the other layers are called *hidden layers*.

Despite the generality of this model, in most applications only three layers are used, the transfer function is log-sigmoid, and all the layers are trained by a single common learning rule called backpropagation (Castleman, 1996). Backpropagation will be covered in more detail in Chapter IV.

Training and Testing a Neural Network

Using the model described above, we may now discuss how a neural network is generally implemented. This discussion assumes that some sort of feature identification exists for the system to recognize, and that data points corresponding to these features can be conveniently obtained. These are generally considered appropriate preliminary steps in any sort of pattern recognition problem, and will not be discussed further here (Castleman, 1996).

First, the designer chooses a suitable architecture. The parameters for the network include the number of layers, the number of neurons in each layer, the transfer function employed by each neuron, and the learning rule to use when training the network. Then two sets of data points are prepared: one for training, and one for testing. It is wise to make these data sets statistically independent of one another, since an improperly trained network can converge on one set of data points but nevertheless be insufficiently generalized for the full range of data points it is likely to receive as inputs (Venables and Ripley, 1994).

Next, the training data set is introduced to the network as an input/output pair, one sample at a time. The input is propagated through the network and an output computed according to the neuron rule shown above. This output is compared to the desired output in the training set for the input and an error computed. The error is used to adjust the weights in the network according to the operant learning rule, and another data point is then presented to the network. When all the data points in the training data set have been presented to the network, the process repeats until the error decreases to within some acceptable limit.

After some number of iterations through the training set (called *epochs*), the adjustment to the weights in each layer will converge. The training algorithm usually guarantees convergence provided that certain restrictions on the network's architecture and the training and testing data are observed. It also guarantees that network's output errors are within some statistical limits. At this time the network is considered "trained". A separate testing data set is then introduced and processed through the network in exactly the same manner as the training data set. Again, an error is computed. If the error from the testing data set is close to the error for the training data set, the network is considered ready for production. If not, different test data points may need to be obtained, or the architecture of the network adjusted, and the whole training and testing process repeated.

Neural Network Classes

From the preceding discussion it is clear that general library classes should include layers, which contain weights, sums, and transfer functions for behaviors, and matrix and array collections to hold the weight matrix, input vector, and output vector for a layer. We would also like to have a network class, which would hold a collection of layers, and (perhaps) some global network parameters needed by the learning rule we are using. Objects of the network class should also contain state information telling how the network is currently being used. Possible states are in training (weights are being adjusted using the learning rule) or in testing (an input vector is being propagated through the network, one layer at a time, using the vector version of the neuron equation above.) Since most learning rules require propagation (computation of output using input and weights), this would be behavior common to all types of networks and layers (Kohn, 1992).

Header Files and Source Organization

Before we go into the discussion of the library's class structure, a word about the overall organization of the library's source files is appropriate. The prototypical user of this library is assumed to be an undergraduate or graduate student who wishes to perform research or write class assignments using neural networks. Such a user typically wants an absolute minimum of implementation requirements for a project, and a shallow learning curve for any software packages they happen to use. Thus, one of the major design goals of the Monarch library is programmer convenience.

To this end, a single header file is used for all Monarch programs. It contains the declarations for the classes described in this thesis, and it calls any additional header files a Monarch program requires (such as those containing function prototypes) through the C++ `#include` file mechanism.

There is a good bit of treatment in Stroustrup (1991) about single versus multiple `#include` files for a C++ library, and more than a bit of religious feeling on the subject. Advocates of multiple `#include` files cite flexibility and reduced overhead for users who don't wish to pull in the facilities of a package they aren't using. The biggest reason to use a single `#include` file for everything, even if it grows quite large, is simplicity. Given the casual nature of the library's user, a single `#include` file has been used in Monarch.

Basic Types

Within `MONARCH.H`, it is useful to first supply a few basic types for the library. These help the application programmer to make his or her programs portable without regard to the details of a C++ implementation on a particular machine. Having these types, and using them consistently throughout the library, should ease any porting problems.

The basic types needed by a neural network program include Boolean values, integer values, and real numbers. In Monarch, these are defined as follows:

```
#ifdef __MSDOS__
typedef enum { false=0, true } boolean;
#else
typedef int boolean;
#endif

#ifndef INT_TYPE
typedef int integer;
#define INT_TYPE
#endif

#ifndef REAL_TYPE
typedef double real;
#define REAL_TYPE
#endif
```

On Unix machines supporting the newer 2.0 C++ compiler, there is already a Boolean type with `true` and `false` keywords. We can therefore make `boolean` synonymous with `int` for this case. On MS DOS machines, we don't have this convenience, so we need to create an enumerated type for Boolean values.

The Array and Matrix Classes

A good place to begin class implementation is with array and matrix classes. In OOP terminology, these are called "collection classes" or simply "collections", since they are classes that hold other objects. Other examples of collection classes include sets, lists, hash tables, and trees. In an OO language, collection classes provide data structures that can be used in an application program without the programmer needing to implement their details (Coplein, 1992).

There is a good reason to begin implementation of the library with the array and matrix collection classes—both are commonly used in neural network programs. Even "outside" the library (which we will discuss more in the next chapter) there is a compelling need for reusable arrays and matrices.

For our library, we have three strategies we can pursue. They are, in order of increasing complexity:

1. We can use the matrix and array facilities already in C++. This has the advantage of economy, but amenities such as range checking and stream I/O are not provided. Both facilities would be enormous conveniences.

2. We can implement matrix and array facilities, starting with those provided by C++ and adding the features described above. This has the advantage of providing the additional functionality mentioned above to what is a common data structure in a neural network program. The disadvantage is that we will have to write this code ourselves, and we will be adding (perhaps prohibitive) overheads to what is a fairly efficient basic implementation.

3. We can utilize the array and matrix collections provided in the Standard Template Library. This is a set of collection classes defined as part of the C++ language. This has the advantage of providing a robust set of facilities for arrays and matrices. However, the disadvantages are (a) portability to older C++ implementations that do not support templates, (b) extra overhead (neural programs are unlikely to use the other classes supported by the Standard Template Library), and (c) the library's user must learn to use the Standard Template Library to use the neural network library

For these reasons, then, following strategy (2) and writing our own array and matrix classes seems like a wise decision. As it happens, C++ makes the task relatively painless.

To design these collection classes, we use the following procedure: first, we will list all the data items that must be present "inside" an array or matrix object. In OOP terminology, these are called "instance variables." Next, we will list the functions we would like our array or matrix to perform. This is called the "external protocol" for the object. Using both of these as specifications, we can then proceed with the implementation.

First, array and matrix objects must contain the actual data stored in the collection. These items will be of type of `real`. In actual use, most neural network weights and inputs are in the range of (-0.5…0.5), but there is no reason not to use a double-length floating point number on all implementations. In any event, this can be easily changed and the library recompiled if needed.

Second, the number of elements in the array needs to be stored within the object, so range checking can be performed. For an initial implementation, we will choose a sequential storage of the objects, C-style, going from 0 to (size of the array-1).

Third, we will provide two additional indices. The first is a "from" index telling where the next element is to be obtained from the array. The second is a "to" index telling where the next element is to be stored. Both of these allow the array (or matrix) to be read sequentially as well as by a subscript. This will make streaming data to or from an array or matrix easier. Here we make the arbitrary (and simplifying) design decision that a matrix can be read or written sequentially as if it were an array stored in row-major order. This is the way C (and C++) does it.

With these variables defined, the operations we would like to perform using the array or matrix include:

1. Create the array or matrix initially, or as a copy of another array or matrix object.

2. Add elements sequentially to the array or matrix, and check the bounds to make sure we don't overflow it.

3. Retrieve elements sequentially from the array or matrix.

4. Add elements at any location, and check the location.

5. Retrieve elements (with bounds checking) using the subscript operator in C++. (The language provides convenient operator overloading facilities, so it seems a shame not to use them for this.)

6. Stream data to and from the array or matrix. This is especially useful when we want to load the object from an external file using C++ streams.

Since C++ is a compiled language, data stored in an array or matrix can be economically accessed using pointers. We use the `calloc()` library function to allocate a block of memory to hold the contents of the array or matrix, and then access an element with subscripts using exactly the same method that C++ uses for its own arrays and matrices—by computing the address of the item using the base address of the collection, the shape of the array or matrix, and the subscript(s) to be accessed. We can overload the `[]` (subscript) operator to make this process transparent to the user of an array or matrix.

Checking an array or matrix subscript against the bounds of the collection requires a bit of further explanation. Since we may not wish to have this overhead in a production program, a bounds-checking function that can be removed with `#define` seems useful. On the other hand, we would like accessing the array to be an inline method in C++.

To accommodate these two conflicting requirements, we will use a macro that conditionally expands a call to the bounds checking function inside an inline function. This is what the macros `CHECK()` (for arrays) and `CHECK2()` (for matrices) do.

With this explanation in mind, the class file structure for the array and matrix classes is shown below. First the `array` class:

```
#ifndef NDEBUG
#define _CHECK(x) check((x))
#define _CHECK2(x, y) check((x),(y))
#endif

class array
      {
protected:
      real *a;
      integer n_items;
      integer here;
      integer next;
public:
      array(void) { here = 0; next = 0; };
      array(integer n);
      array(const array &);
      ~array(void);
      void reset(void) { here = 0; next = 0; };
      int size(void) { return n_items; };
      void add(real x)
            {
            *(a+next) = x;
            next++;
            };
      void addAt(integer i,real x){_CHECK(i); *(a+i)= x;};
      real get(void)
            {
            real x = *(a+here);
            here++;
            return x;
            };
      real getAt(integer i) { _CHECK(i); return *(a+i); };
```

```
        void from(array& x, integer where=0);
        void to(array& x, integer where=0);
        array& operator = (const array&);
        real operator[] (integer i) { _CHECK(i); return *(a+i); } ;
        void check(integer);
        void randomize(real low, real high);
        void normalize(void);
        real mean(void);
        friend ostream& operator << (ostream&, array&);
        friend istream& operator >> (istream&, array&);
        };
```

These definitions implement the complete protocol described above. The array's "from" index is the integer `here`, and the array's "to" index is called `next`. The operators `[]` and `=` are overloaded to allow arrays to be subscripted normally, and for one array to be copied to another item by item. The >> and << operators are defined as `friend` functions to allow arrays to be streamed using the I/O streams package in C++. The functions `randomize()`, `normalize()`, and `mean()` perform some basic statistical calculations on the contents.

The `matrix` class is defined in a similar manner:

```
class matrix : public array
        {
        integer n_rows, n_cols;
public:
        matrix(integer i, integer j);
        matrix(const matrix&);
        ~matrix(void);
        integer rows(void) { return n_rows; };
        integer cols(void) { return n_cols; };
        void addAt(integer i, integer j, real x)
                {
                _CHECK2(i, j);
                *(a+(i*n_cols+j)) = x;
                };
        real getAt(integer i, integer j)
                {
                _CHECK2(i, j);
                return *(a+(i*n_cols+j));
                };
        real *operator[] (integer i) {return &a[i*n_cols];};
        matrix& operator = (const matrix&);
```

```
    void check(integer, integer);
    friend ostream& operator << (ostream&, matrix&);
    friend istream& operator >> (istream&, matrix&);
    };
```

The `matrix` class inherits much of its behavior from the `array` class, including the instance variable `a`, which contains the data. It retains the behavior of sequential access but adds rows and columns.

In both arrays and matrices, the principal functions are likely to be inlined by C++ since they are defined as part of the class declaration. This reduces the overhead of using these collections, since the compiler will generate no calling sequence when a method is invoked. The additional support for the array and matrix classes is contained in the `ARRAY.CPP` source file.

The collection classes above have the advantage of being concise, economical with storage (only 3 extra integers are required for arrays and 5 for matrices) and, as Chapter V will demonstrate, reasonably fast. However, they have several disadvantages:

1. Subscripts must go from 0 to 1 less than the number of items. Other ranges, (e.g. -10...+10), such as are found in Pascal, are not supported.

2. The data in these structures is contiguous in memory. Sparse arrays and matrices are not supported. They could be added using the same protocol, however.

3. The array or matrix must be allocated to a fixed size. The size cannot increase or decrease as needed. Variable length structures are not supported. Again, the same protocol could be used and the implementation changed to support this, however. This is one of the advantages of C++'s facilities for encapsulation.

None of these three disadvantages has any impact on the algorithms implemented using the library to date.

The Neural Network Classes

With the ability to create arrays and matrices, we can create neural network classes that use them. Initially, the library had only two main object types: a Layer object, which was a collection of neurons, and a Net object, which held a collection of Layers (Joyce, 1990).

This design, however, proved inadequate when implementing the algorithms in the library. This is because the matrix of weights **W** makes more sense when thought of as going between a set of neuron layers (from the output of one layer to the input of another) and not part of a set of neuron layers. Other properties of a neuron layer, such as the transfer function, make sense if they are thought of as part of the layer itself.

As a result, a synthetic entity was created, called a Link. (In OOP terminology, a synthetic entity is one created for the convenience of the programmer that does not have a physical analogue in the "real" problem.) A Link connects two Layers of neurons. The Layer contains the sum, output, and error terms for the neurons in the layer, along with the transfer function and its derivative (this becomes more important when we discuss backpropagation in Chapter IV.) The Link holds the weights and the changes in the weights for the current epoch (this quantity is used in some training methods to tell when the network has been sufficiently trained. Again, this is covered in Chapter IV.) A Link also holds pointers to the two Layers it joins.

A neural network, then, is a collection of Links, each of which joins a pair of Layers.

The declaration for a Layer is shown below:

```
class Layer : public Monarch
      {
protected:
      integer n_units;
      char *layer_name;
public:
      array *sum, *output, *error;
      Layer(char *, integer);
      ~Layer(void);
      integer numberUnits(void) { return n_units; }
      char *layerNameOf(void) { return layer_name; }
      friend ostream& operator << (ostream&, Layer &);
      friend istream& operator >> (istream&, Layer &);
      virtual real f(real x); // transfer function
      virtual real f_dot(real x); // derivative of transfer function
      };
```

The Layer class descends from the Monarch class, about which there will be more to tell below. The basic protocol includes returning the number of units, the name of the Layer, and the transfer function `f()` and its derivative `f_dot()`. The sums of each neuron, the outputs of each neuron in the layer, and the error terms (the difference between the actual and expected outputs for each neuron in the layer) are declared as pointers to `array` objects.

Here is the declaration for the Link class:

```
class Link : public Monarch
      {
protected:
      Layer *upper, *lower;
      matrix *weights, *dweights, *save_weights;
public:
      Link(Layer *from, Layer *to);
      ~Link(void);
      Layer *fromLayer(void) { return lower; }
      Layer *toLayer(void) { return upper; }
      void save(void);
```

```
        void restore(void);
        virtual void propagate(void);
        virtual void backpropagate(void) { };
        virtual void adjust(void) { };
        virtual real computeError(array&) { return 0.0; };
        virtual void newtrial(void) { } ;
        friend ostream& operator << (ostream&, Link &);
        friend istream& operator >> (istream&, Link &);
        };
```

In this class, the weights are defined as a matrix whose rows are the number of input Layer units and whose columns are the number of output Layer units, exactly as specified earlier in this chapter. There is a matching matrix called `save_weights` that may be used to hold the weights for stopped training, and another matrix called `dweights` that contains the change in each weight the last time the Link was propagated.

Like Layers, Links descend from the Monarch class.

Finally, there is the Net class. This is the collection class for the Links in the network, the controlling class for simulating its operation, and also the base class for any derived classes which actually implement the various learning algorithms. Its structure is shown below:

```
class Net : public Monarch
        {
protected:
        integer here;
        integer next;
        integer n_layers;
        char *net_name;
        phaseid phase;
        long epoch;
        real error; // total net error
        Link *links[N_LINKS];
public:
        Net(char *, integer);
        ~Net(void);
        integer numberLayers(void) { return n_layers; }
        char *netNameOf(void) { return net_name; }
        void reset(void) { here = 0; next = 0; };
        virtual void propagate(void);
        virtual void backpropagate(void);
        virtual void adjust(void);
        virtual void newtrial(void);
        virtual void computeError(array&);
        virtual void simulate(array&);
        void add(Link *l) { links[next++] = l; };
        void addAt(integer i, Link* l)
              {
              links[i] = l;
              }
        Link* get(void) { return links[here++];   };
        Link* getAt(integer i) { return links[i]; }
        void setPhase(phaseid p) { phase = p; epoch = 0; };
        real errorOf(void) { return error; }
        void save(void);
        void restore(void);
        };
```

There will be much more to say about this structure in upcoming chapters, but for now it is important to note that the array `links[]` holds pointers to all the Links in the network. Going forward through this array (from 0 to the number of Links–1) propagates through the network, from the input layer, through the hidden layers, to the output layer; going backwards (from Links-1 to 0) backpropagates from the output layer to the input layer. These iterations are what the member functions `propagate()` and `backpropagate()` do, respectively. The function `adjust()` is called if the network is in "training" phase to set the weights of each Link based on the error for each Layer. All three of these methods appear to have some correspondence in real algorithms. It is the task of the various network implementations, which descend from Net, to complete the functions mentioned above with appropriate learning rules.

The Monarch Class

The Monarch class acts as a base class for all other neural network classes in the library. The array and matrix classes do not descend from

Monarch so that they can be used in programs without dependency on the neural network structure.

The structure of the Monarch class is shown below:

```
typedef enum
      {
      monarch, net, layer, inputlayer, outputlayer, hiddenlayer,link
      } classid;

typedef enum
      {
      setup, training, testing, evaluating
      } phaseid;

class Monarch
      {
protected:
      classid id;
      char *name;
      void change(char *, classid);
public:
      Monarch(char *, classid);
      ~Monarch(void);
      classid isA(void) { return id; }
      char *nameOf(void) { return name; }
      };
```

The Monarch base class provides a name and ID for all the other classes in the class structure. The member functions `isA()` and `nameOf()` return these. These are used principally in tracing and debugging facilities, which are discussed in Chapter VI.

The InputLayer, OutputLayer, and HiddenLayer objects descend from Layer and are used for these various types of neuron layers. Their chief difference is that an InputLayer may be written to with a `to()` function, while an OutputLayer read with a `from()` function. This allows the calling program to pass data to and from the neural network during training, testing, and evaluation phases. It also allows streaming operators, discussed further in the next chapter, to be implemented.

The basic Monarch class tree is shown in Figure 3.

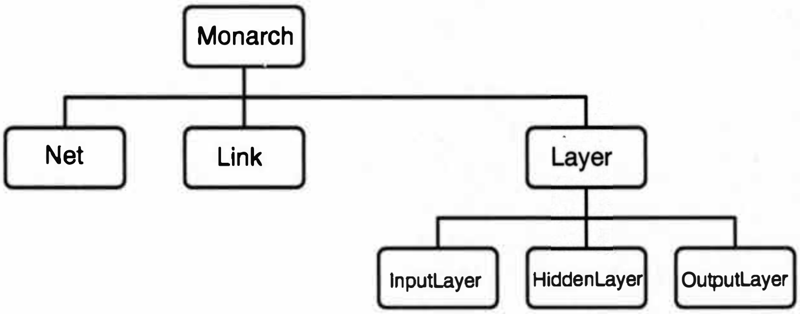

Figure 3. The Monarch Class Tree.

Note that the classes presented in this chapter are very basic. For instance, the `simulate()` method in the Net class simply follows the procedure described earlier to use the network:

```
/*
 * simulate() -- run (and possibly) train the network
 *   Runs a single cycle
 */
void Net :: simulate(array& target)
      {
      if (phase != setup)
            {
            ++epoch;
            propagate();
            computeError(target);
            if (phase == training)
                  {
                  backpropagate();
                  adjust();
                  }
            }
      }
```

The Net member functions `propagate()`, `backpropagate()`, etc. in turn simply iterate through the array of `links[]` in either forwards or backwards order. To implement a learning method using this structure, descendants for the Net and Link classes must be added for the algorithm. This is discussed in Chapter IV.

This chapter described the architecture of the Monarch library, presenting an overview of the basic classes. In the next chapter, the programmer's view of the library will be presented, and some additional classes to make programming easier will be described.

CHAPTER III

APPLICATION STRUCTURE

In this chapter we present a structure for programs that use the Monarch library. By "application structure" we mean a set of C++ classes that, when used in concert, provide a coherent framework for a Monarch neural network application. Within this framework, the programmer need only write the parts of the program needed to solve the specific problem at hand. Many details, such as creating the network and providing it with input, are already handled.

This chapter therefore covers topics relating to application structure:

1. How to use the standard header file and link with the library.

2. Basic classes, such as MonarchApp, which handle the application's initialization and termination.

3. How to use training and testing databases within an application.

4. Support for debugging and inspecting Monarch objects.

5. How network configuration information can be brought into the application when the program is run so the network can be dynamically configured.

Before we present the classes that are provided to make application programming easier, we need to take a look at how a Monarch program is compiled and linked.

Using the Library

Using the Monarch library is a two-step process. It involves (1) including the C++ header file that defines the Monarch classes in the source program, and (2) linking with the Monarch object library. A sample program and compilation, using Borland's Turbo C++ compiler, is shown below:

First the sample file:

```
C:>type sample.cpp

// SAMPLE.CPP:    Test Monarch library
//
// version:       9/16/97
// compiler:      Turbo C++ 3.x
// uses:          monarch.h, monproto.h
// module type:   .EXE

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include "monarch.h"

parameters sample_ini[1] =
      {
      { "echo", (function) _echo }
      };

class App : public MonarchApp
      {
public:
      App(void);
      ~App(void);
      };

App *theApp;

App :: App(void) : MonarchApp("SAMPLE.INI", 1, &sample_ini)
      {
      }

App :: ~App(void)
      {
      }

int main(int argc, char *argv[])
      {
      theApp = new App;
```

```
    // application goes here
    delete theApp;
    }
```

Now the compilation:

```
C:>tcc sample.cpp
C:>tlink sample,,,monarch.lib
```

Chapter II discussed the benefits and penalties of using a single header file. This topic will not be covered further here. In this example the library file `MONARCH.LIB` contains all the object modules that implement the classes in the library. All the example programs presented in this thesis were compiled the same way.

Monarch Application Classes

The program `SAMPLE.CPP` above shows the minimal code needed to use the Monarch library. The most important feature in this small example is that it creates a class called App that descends from the Monarch class MonarchApp.

The MonarchApp Class

This class is a formal class for the Monarch library. This means that specific instances of this class are never created directly. Instead, the user is expected to create a descendant tailored for his or her own use. This descendant class typically creates all the objects needed by the application program for the neural net: the Net, Link, and Layer objects described in the last chapter. It also performs any additional initialization needed by the simulation.

Here is an example, drawn from the ADALINE example program. (The ADALINE example is covered further in Chapter VII.)

```
/* ADALINE Layers and Network */

InputLayer *il;
OutputLayer *ol;
AdNet *recognize;

/*
 * App() -- create and initialize neural network
 *    Note: many values are set as part of MonarchApp initialization
 */

App :: App(int argc, char *argv[]) :
      MonarchApp(argc, argv, "CHARS.INI", N_PARMS, test_ini)
      {
      il = new InputLayer("Inputs", n_inputs);
      ol = new OutputLayer("Outputs", N_OUTPUTS);

      recognize = new AdNet("Recognize", ADALINE_LAYERS, Alpha,
                            Epsilon);
      recognize->add(new AdLink(il, ol));
          .
          .
          .
      }
```

As part of the constructor for the App, the constructor for MonarchApp is called to initialize the Monarch library functions, and then the neural network used in the program is created. The exact protocol for this is described later in this chapter.

The Parameters Class

An important class used by MonarchApp is the Parameters class. This class allows the user to create initialization files ("INI files") of parameters for the neural network, which in turn may dynamically control features such as learning constants, the number of units in a layer, etc.

To use the facility, the user must create a static structure of initialization file keywords and functions:

```
parameters sample_ini[1] =
      {
      { "echo", (function) _echo }
      };
```

The address of this structure is then passed to the MonarchApp class when its constructor is invoked:

```
App :: App(void) : MonarchApp("SAMPLE.INI", 1, &sample_ini)
      {
      }
```

The initialization file may either be a file proper, or a sequence of command-line arguments.

A number of alternative methods for performing this function were considered. These were:

1. Strictly using UNIX command-line arguments, passed through `argc` and `argv[]`. These are portable to all C and C++ implementations, but they are limiting: if all the arguments required are too long to fit on a command line, then no support exists for retrieving them.

2. Using an operating system-dependent facility for .INI file processing. Microsoft's Windows 3.1, Windows 95, and Windows NT have such a facility, available through the Application Program Interface (API) for these environments. It handles all the required functions conveniently. The problem, of course, is that this mechanism is limited to these environments.

3. Writing a basic facility that builds on the UNIX command line processing but augments it with files.

4. Writing a more complex facility that replaces the UNIX command line processing using the UNIX parsing tools Lex and Yacc. This is the approach adopted by environments such as Aspirin/MIGRAINES. It allows ultimate flexibility, but it is much more general than is needed for this project—most of the parameter specifications required by Monarch obey the rule "parameter = value".

In this application, option (3) was chosen. Implementing it in a hybrid object-oriented language like C++ is problematic, since an important feature of this facility is that the Monarch library's parameter file handler calls code the application programmer writes within his or her program. In a "pure" object-oriented language such as Smalltalk, code can be passed between objects as blocks (Goldberg, 1983), but this ability is not included in C++. (There are alternative idioms for this, discussed in Coplien (1992).) The closest we can come to a proper block without adding too much apparatus to the library is with C-style pointers to functions.

This mechanism is what the following types establish:

```
typedef void (* function) (char * t);
                        // keyword function table entry

struct parameters
      {
      char *what;       // control token
      function where;   // function to call
      };
```

These declarations create a type of `function` that is a pointer to a function that returns `void`, and a structure called `parameters` that pairs a string (called a "control token") with a `function` to process it. The class Parameters, shown below, takes an array of parameters, a file, and a UNIX-style argument list. We can consider both the initialization file and the command line argument list as sources of parameters. Using the parameter structure, the Parameters class calls the functions in the structure whenever a keyword is encountered in either parameter source.

All the programmer writes are relatively simple functions to handle argument processing, such as:

```
void _alpha(char *s)   { Alpha = atof(s); }
void _epsilon(char *s) { Epsilon = atof(s); }

#define N_PARMS 4

parameters test_ini[N_PARMS] =
      {
      { "echo",    (function) _echo },    // in library
      { "*",       (function) _comment }, // in library
      { "alpha",   (function) _alpha },
      { "epsilon", (function) _epsilon }
      };
```

Here `_alpha()` and `_epsilon()` simply process their arguments with the C++ standard library function `atof()` and set the parameters in the simulation accordingly. In an .INI file, a user could then include:

```
alpha 0.05
epsilon 0.95
```

to set these values, while on the program's command line, the construction

```
      eta=0.05 epsilon=0.95
```

would be used instead, since C++ runtime initialization breaks up parameter strings using blanks. (In this example, `eta=0.05` and `epsilon=0.95` are treated as separate arguments.)

All the processing for parameter handling is embedded in the class Parameters. This synthetic class handles this chore for the entire library. Its declaration is shown below:

```
class Parameters
      {
        FILE *inifp;
        integer num_keywds;
        parameters *key_table;
protected:
        void getline ( char *s );
        integer search (char * key);
public:
        Parameters(char *fn, integer n, parameters *tbl);
        ~Parameters(void);
        boolean arglist(integer argc, char *argv[]);
        boolean obtain(void);
        };
```

The details of this class need not concern us, except to mention that the user never calls the Parameters class directly—the standard constructor for MonarchApp handles this processing when the program is initiated.

Creating the Neural Network

Within the App class, the library functions to actually build the network are called. The exact protocol is described in this section. Here there are several design options to consider:

1. C++ calls to constructors for Nets, Links, and Layers can be used. This is the most straightforward and obvious approach.

2. A "design language" can be implemented where the schema for the network is read from an ASCII file. This is the approach used in packages such as Aspirin/MIGRAINES. The file can be edited outside of the simulation program, and different network configurations interchanged.

3. A graphical design tool can be constructed which outputs the needed objects for the network directly. This is the most elaborate implementation, but operation would be easiest for the non-programming user. This is the approach used by environments such as SNNS (Zell, 1995).

In this application, approach (1) was adopted. Using C++ and the Unix language tools such as Lex and Yacc, approach (2) or even (3) could be implemented at a later time.

For approach (1), there are two sub-cases to be considered:

a. The network can be constructed at compile-time, using statically initialized data structures. This makes it easy to design tools to create the network, but it is difficult for programmers who don't have (or like) the tools to use the package. Preprocessor macros can make this process more palatable, however.

b. The network can be constructed at run-time, using C++ function calls. This makes it easy to use the library without any supporting tools, although adding a language or visual tool later means that an interpreter is required to translate a schema into C++ calls.

Approach (b) was the one adopted. The linked data structures within the Net object make adding additional tools later a straightforward process.

Sample code to create a network is shown below:

```
/*
 * App() -- create and initialize neural network
 *   Note: many values are set as part of MonarchApp initialization
 */

App :: App(int argc, char *argv[]) :
      MonarchApp(argc, argv, "BPN.INI", N_PARMS, test_ini)
      {
      il = new InputLayer("Inputs", n_inputs);
      ol = new OutputLayer("Outputs", N_OUTPUTS);
      hl = new HiddenLayer("Hidden", n_hidden);

      sunspots = new BpNet("Sunspots", N_LAYERS, Gain, Eta, Alpha);
      sunspots->add(new BpLink(il, hl));
      sunspots->add(new BpLink(hl, ol));
      }
```

The network is created from the bottom up. First the Layers are created, then the Net object, and then two Links (one for the input to hidden connection, one for the hidden to output connection). Each Layer's constructor takes as an argument the number of units for the layer. This is obtained from the initialization file when the App's ancestor, MonarchApp, is instantiated.

When the Net object is instantiated, learning parameters set during the program's startup (either from default constants or the initialization file) are passed to the constructor. These parameters vary depending on the type of network and learning method. Here, standard backpropagation parameters gain, ε (eta), and α (alpha) are shown. These are described in detail in Chapter IV.

Using the relatively simple initialization file facility and run time function calls, a variety of network configurations can be easily created in an application program.

Destroying the Neural Network

When the program has ended, the destructor for the App class typically deletes the neural network Net object.

```
/*
 * ~App() -- delete network when program finishes
 *    Layers, Links, etc. are deleted automatically
 */
App :: ~App(void)
      {
      delete sunspots; // delete Net
      }
```

Since a Monarch neural network is a linked object, all the associated Links and Layers, and their associated arrays and matrices, are also destroyed from "within" the Net object. Several different neural networks can be independently created and destroyed in a single simulation.

Streams in the Monarch Library

In a neural network simulation program, there are a several uses for type-safe input and output facilities:

1. Files of training and testing data need to be handled. (These are sometimes referred to as databases in neural network literature, although the traditional view of a database as a hierarchical or relational organization of data items is not applicable here. Instead, a neural network database is a set of arrays or matrices of paired input and output data items.)

2. Visualization tools, such as MATLAB and GNUPLOT, are often used to graph data output from the simulation, or to prepare input for a simulation. These tools usually require ASCII files.

3. Debugging an application sometimes requires "dumping" the contents of an array or matrix for examination.

All three uses fall into three broad areas of input and output as defined in C++ (Stroustrup, 1991):

1. *Input* can be thought of as the conversion of a sequence of characters (usually read from a file) into an instance of a specific type.

2. *Output* is the conversion of an instance of a specific type into a sequence of characters.

3. *Formatting* is changing the layout of the sequence of characters before they are input to a type, or after the type is output.

In C++ vocabulary, these sequences of characters from which types are input or output are called *streams*. By default, a C++ program has access to three streams when it is run: `cin`, which is connected to standard input; `cout`, which is connected to standard output, and `cerr`, which is connected to the error output. By connecting to streams through standard C++ library calls additional files may be used.

Operator Overloading

C++ uses the << and >> operators to get data from and put data to a stream. Defining stream behavior for a class therefore requires overloading these two operators. An example demonstrates the use of the overloaded "put to" << operator:

```
ostream& operator << (ostream& s, array &a)
        {
        for (integer i = 0; i < a.n_items; i++)
                {
                s << *(a.a+i) << "\n";
                }
        return s;
        };
```

The function `operator` << is defined as a friend of the array class in `MONARCH.H`. This allows the function access to the normally private instance variables of an array object. For an array, we simply want to output the individual elements of the array (note that we also use << for these) to whatever stream we happen to be using. The << operator is considered dyadic, with the stream to output to and the object to be output as the two operands.

For a network object such as a Link or a Layer, specifying the behavior of the stream operators is problematic. What do we want them to accomplish? What does "putting to" a Link or "getting from" a Layer mean?

For Layers, we arbitrarily decide that the stream operators work with the `output` array in the object. This is what we are interested in most of the time. The code to handle "put to" is shown below:

```
istream& operator >> (istream& s, Layer &l)
      {
      s >> *l.output;
      return s;
      }
```

This allows us to set the InputLayer of a neural network very conveniently. The code for "get from" is similar:

```
ostream& operator << (ostream& s, Layer &l)
      {
      s << l.output;
      return s;
      };
```

Using this, we can get the contents of an OutputLayer easily.

For Links, we define the same functions to work with the weight matrix. This is useful for debugging and tracing the output. The functions are shown below:

```
ostream& operator << (ostream& s, Link &l)
      {
      s << l.weights;
      return s;
      };

istream& operator >> (istream& s, Link &l)
      {
      s >> *l.weights;
      return s;
      }
```

In both the Link and Layer examples shown above, we call the `operator <<` and `operator >>` functions defined for arrays and matrices. The member functions for Links and Layers simply call them for the appropriate instance variable.

Database Classes

In addition to making neural network classes streamable, Monarch provides direct support for database classes. In Monarch, a database is usually associated with a file, and it contains an array or matrix of ASCII numeric data. The two sub-types of databases are streams and blocks. Like arrays and matrices, streams and blocks are available in programs without requiring the rest of the neural network library class tree. They may be used wherever ASCII files of numeric data need to be handled.

In designing the database classes, the goal was to provide a class structure where (a) basic ASCII file operations could be done easily by casual users and (b) a more knowledgeable Monarch user would have a suitable hierarchy to use for their own, more specific "databases". These databases may not associate with files at all, but could, for example, interface to the move generator in a board game.

The protocol for accessing a database relies on the polymorphism available in C++: the programmer opens, closes, gets, and puts data to a database exactly like using a file. The definition of the `database` class in `MONARCH.H` is shown below.

```
typedef enum {closed=0, opened=1} db_state;

class database
    {
protected:
      real x;
      integer items_read, items_written;
      db_state s;
public:
      database(void) { s = closed; items_read = 0;
                       items_written = 0; };
      ~database(void) { } ;
      void dump(void);
      virtual void reset(void) { } ;
      virtual boolean isError(void) { return false; } ;
      virtual boolean isEof(void) { return false; } ;
      virtual int errorOf(void) { return 0; } ;
      db_state stateOf(void) { return s; };
      virtual real get(void) { items_read++; return x; } ;
      real last(void) { return x; };
      virtual void put(real y ) { x = y; ++items_written; } ;
      integer read(void) { return items_read; };
      integer written(void) { return items_written; };
      };
```

This formal class defines the basic behavior for a database object. It provides the ability to get an item from the database and query its various attributes, such as whether the file associated is it is at end of file or not.

The supported descendants of the database class are `stream` and `blocks` databases. These map to the C++ Standard Library's stream and block I/O file handling, respectively. The `stream` class is shown below:

```
class stream : public database
      {
protected:
      FILE *f;
      char *fname, *fmt, *mode;
public:
      stream(char *, char *);
      ~stream(void);
      stream(const stream&);
      stream& operator = (const stream&);
      char *fileNameOf(void) { return fname; };
      char *formatOf(void) { return fmt; };
      void format(char *s) { delete fmt; fmt = strdup(s); };
      boolean isError(void);
      boolean isEof(void);
      int errorOf(void);
      FILE *fileOf(void) { return f; };
      real get(void);
      void put(real);
      void reset(void); // flush and seek to start
      long locate(long);
      friend int operator == (const stream& a, const stream& b);
      friend int operator != (const stream& a, const stream& b);
      };
```

Note that this defines most of the actual behavior, while `database` acts as a template. The `blocks` class is similar:

```
class blocks : public database
    {
protected:
    int handle;
    char *fname;
public:
    blocks(char *, int);
    ~blocks(void);
    blocks(const blocks&);
    blocks& operator = (const blocks&);
    char *fileNameOf(void) { return fname; };
    boolean isError(void);
    boolean isEof(void);
    int errorOf(void);
    int fileOf(void) { return handle; };
    real get(void);
    void put(real);
    void reset(void); // seek to start
    void close(void);
    void open(int mode);
    long locate(long);
    friend int operator == (const blocks& a, const blocks& b);
    friend int operator != (const blocks& a, const blocks& b);
    };
```

There is a substantial amount of code in these classes to make them type-safe; this is covered in more detail in Chapter VI.

Additional Functions

While stream operators and databases are convenient, there are applications where we want to simply transfer data to or from the neural network using arrays. This is the case when, for example, we dynamically generate data in our program without reading a database file.

To handle this situation, we may use the buffered I/O features of the C++ stream library, but it is just as easy to define two additional functions. One belongs to the InputLayer class and is called `to()`, while the other belongs to the OutputLayer class and is called `from()`. Both copy the contents of an array to or from the `output` array of the Layer.

The `from()` function is shown below:

```
/*
 * from(a) -- set contents of input layer from 'a'
 */
void InputLayer :: from(array& a)
      {
      assert(a.size() < output->size());
      for (integer i = 0; i < a.size(); i++)
            {
            output->addAt(i+1, a[i]);
            }
      }
```

The `to()` function is the inverse:

```
/*
 * to(a) -- get contents of output layer into 'a'
 */
void OutputLayer :: to(array& a)
      {
      assert(a.size() < output->size());
      for (integer i = 0; i < a.size(); i++)
            {
            a.addAt(i, ((*output)[i+1]));
            }
      }
```

Here is an example of how these are used. The sunspot simulation (see Chapter VII) uses these functions to copy the input and target arrays to and from the neural network during a simulation epoch:

```
/*
 * Simulate(in, out, target) -- run a single simulation cycle
 */
void Simulate(array& in, array& out, array& target)
      {
      il->from(in);
      sunspots->simulate(target);
      ol->to(out);
      }
```

First, the InputLayer is loaded from the `in` array. Next, the `simulate()` method is called for the network. Finally, the `out` array is loaded from the OutputLayer of the network. This completes a single simulation cycle.

Debugging Support

Many C++ compilers, such as Borland's, provide interactive debugging tools such as class browsers and object inspectors for C++ programs. However, there are times when it is convenient to display the contents of an object in an application-specific manner during a program's execution. All Monarch classes support a `dump()` method for this. An example is shown below:

```
/*
 * dump() -- dump array's instance variables
 */
void array :: dump(void)
    {
    cout << " array" ;
    cout << " n_items=" << n_items ;
    cout << " here=" << here ;
    cout << " next=" << next ;
    cout << " &a=" << a << "\n";
    }
```

The preprocessor macro `DUMP` invokes the dump method for a single object:

```
#define DUMP(x) cout << #x; x.dump()
```

Since all objects descending from the Monarch class tree have a type and identifier, this is included in the output.

The Complete Framework

In this chapter we have seen all the elements of a Monarch application. This includes: (a) the MonarchApp class; (b) the App class that descends from it; (c) how configuration parameters for the simulation may be read from an initialization file; (d) how the neural network is created when the App object is instantiated, and destroyed at termination; and (e) how C++ streams may be used to send data to and from the neural network for simulation, and how databases may be used within simulations.

These pieces together comprise the Monarch application framework. Once the network has been constructed, it's up to the programmer to write the parts of the simulation that obtain or generate the testing and training databases, pass them to the network, and evaluate the outputs that come back. The details of this will be shown later in Chapter VII, when three specific examples are presented.

In the next chapter, we will consider the neural algorithms that were implemented in the library.

CHAPTER IV

DETAILS OF ALGORITHMS

In the last two chapters, we considered the internal and external architecture of the library. We introduced classes for Nets, Links, Layers, and MonarchApps. This exposition was necessary to describe a framework in which neural network simulation programs could be conveniently written. We have yet to implement any neural network algorithms, however.

That state of affairs will change with this chapter. Here we present three neural network algorithms, in increasing order of complexity. We will show how the code for these algorithms may be "plugged into" the library structure developed in the previous chapters. The three algorithms are the ADALINE network of Widrow and Hoff (1960), standard backpropagation, and the TD(λ) algorithm of Barto and Sutton (1990). The derivation of these algorithms is necessarily brief—there are many other sources which cover this material in more detail, including the important analysis of convergence once the algorithm and its performance index have been derived (Hagan Demuth, and Beale, 1995). Here enough detail has been included to give the reader a feel for the algorithm and how the implementation in C++ reflects it.

Adaptive Linear Neurons (ADALINE)

As mentioned in the Introduction, the earliest types of neural networks were the Perceptrons of Rosenblatt (1958) and the adaptive linear neurons (ADALINE) of Widrow and Hoff (1960). A single layer of neurons, each of which used a relatively simple transfer function, characterized these network architectures. In the case of the Perceptron, the transfer function was the hard limit. In the case of ADALINE, it is a pure linear function.

Unlike the Perceptron, which uses an arithmetic learning rule, the ADALINE uses the Least Mean Squares (LMS) learning rule. LMS gives better performance than the Perceptron learning rule, which simply moves the decision boundary in the direction of the error vector when a misclassification occurs. Perceptron weights are susceptible to input noise; weights computed by LMS are generally not. More importantly, the ADALINE network may be viewed as a simplified form of backpropagation network, so its implementation can be generalized.

Following the notation described in Chapter II, an ADALINE network is shown in Figure 4.

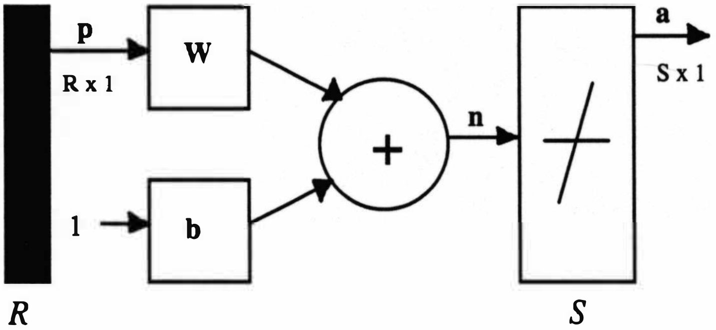

Figure 4. A Sample ADALINE Network.

The output of this network is given by:

$$\mathbf{a} = \mathrm{purelin}(\mathbf{Wp} + \mathbf{b}) = \mathbf{Wp} + \mathbf{b}$$

Like the Perceptron, this equation creates a decision boundary at the point where $n = 0$ or $\mathbf{Wp+b} = 0$. This implies that an individual ADALINE may be used to classify objects into two categories, but only if the features are linearly separable.

With this basic description, let's turn our attention to the LMS algorithm for training an ADALINE network. To do this we will summarize the exposition in Hagan, Demuth, and Beale (1995). This is a supervised algorithm, meaning that the network must be provided with examples of proper behavior. The network is presented with pairs of inputs and associated targets in the sequence:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\} \ldots \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

The weights and bias may be lumped into a single vector:

$$\mathbf{x} = \begin{bmatrix} {}_1\mathbf{w} \\ b \end{bmatrix}$$

Similarly, we include the bias input "1" and create the composite input vector:

$$\mathbf{z} = \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix}$$

Now the network output can be written in vector form as

$$a = \mathbf{x}^T \mathbf{z}$$

Using this notation, the mean square error function $F(\mathbf{x})$ is based on the expected value of the error over all the input-target pairs. If we use the notation $E[\ ]$ to denote expected value, the mean square error function may then be written:

$$F(\mathbf{x}) = E[e^2] = E[(t-a)^2] = E[(t-\mathbf{x}^T\mathbf{z})^2]$$

This can be expanded and re-written as

$$F(\mathbf{x}) = c - 2\mathbf{x}^T\mathbf{h} + \mathbf{x}^T\mathbf{R}\mathbf{x}$$

Where $c=E[t^2]$, $\mathbf{h}=E[t\mathbf{z}]$, and $\mathbf{R}=E[\mathbf{z}\mathbf{z}^T]$. The vector **h** is sometimes called the cross-correlation between the input and the target, and **R** is called the correlation matrix.

To compute the LMS error, we need to minimize this $F(\mathbf{x})$ function. This is a quadratic function, fitting the general form:

$$F(x) = c + \mathbf{d}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}$$

where

$$\mathbf{d} = -2\mathbf{h} \text{ and } \mathbf{A} = 2\mathbf{R}$$

Because this is a quadratic function, its characteristics (and therefore the existence of a minimum error point) are determined by Hessian matrix **A**. If the eigenvalues of the Hessian matrix are all positive, then the function will have one unique global minimum. Since the Hessian matrix is twice the correlation matrix **R**, and all correlation matrices may be demonstrated to be either positive definite or positive semidefinite, then the performance index will have either one unique global minimum, a weak minimum, or no minimum. All of this can be determined by the vector **d**=-2**h**.

The minimum of the performance index is determined by the gradient of $F(\mathbf{x})$. This is given by:

$$\nabla F(\mathbf{x}) = \nabla\left(c + \mathbf{d}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}\right) = \mathbf{d} + \mathbf{A}\mathbf{x} = -2\mathbf{h} + 2\mathbf{R}\mathbf{x}$$

The stationary point can be found by setting the gradient equal to zero:

$$-2\mathbf{h} + 2\mathbf{R}\mathbf{x} = 0$$

If the correlation matrix is positive definite there will be a unique stationary point, which will be a strong minimum:

$$\mathbf{x}^* = \mathbf{R}^{-1}\mathbf{h}$$

Note that the existence of a solution depends only on the input correlation matrix, **R**. If we could calculate **h** and **R** directly, we would have the minimum point. However, this is not always practical. The important insight of Widrow and Hoff is that the minimum may be estimated from the square of the original error function, and the gradient of F(x) need not be computed exactly. It is much easier to compute an error vector *e* at each iteration of the algorithm, and rely on the fact that

$$\hat{F}(\mathbf{x}) = (t(k) - a(k))^2 = e^2(k)$$

In this approximation, the expectation of the squared error is replaced by the squared error at each iteration. Then the gradient estimate becomes:

$$\hat{\nabla} F(\mathbf{x}) = \nabla e^2(k)$$

The first *R* elements of this are derivatives with respect to the network weights, while the $(R+1)^{st}$ element is the derivative with respect to the bias. This can be written:

$$[\nabla e^2(k)]_j = \frac{\partial e^2(k)}{\partial w_{1,j}} = 2e(k)\frac{\partial e(k)}{\partial w_{1,j}} \text{ for } j = 1,2,\ldots R$$

and

$$[\nabla e^2(k)]_{R+1} = \frac{\partial e^2(k)}{\partial b} = 2e(k)\frac{\partial e(k)}{\partial b}$$

Using the notation defined previously, this simplifies to

$$\hat{\nabla}F(\mathbf{x}) = \nabla e^2(k) = -2e(k)\mathbf{z}(k)$$

In our matrix notation from Chapter II, then, the LMS algorithm looks like this:

$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha\mathbf{e}(k)\mathbf{p}^t(k)$$

and

$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha\mathbf{e}(k)$$

The parameter α (alpha) is called the *learning rate* and may be calculated from the eigenvalues of **R**. It controls the size of the gradient descent interval at each iteration of the algorithm. Rather than compute this, it is customary to select a small, conservative number to use in the simulation.

We have now developed enough theory to talk about the algorithm as it is implemented in C++. First, let's look at the structure of an ADALINE Link object, since it contains most of the data structures needed by the algorithm. It descends from the standard Link in the class tree. It is shown below:

```
/* ADALINE version of Link */

class AdNet;

class AdLink : public Link
      {
protected:
      AdNet *net;
      array *activation;
public:
      AdLink(Layer *from, Layer *to);
```

```
    ~AdLink(void);
    AdNet *netOf(void) { return net; };
    void netIs(AdNet *n) { net = n; };
    virtual void propagate(void);
    virtual void adjust(void);
    virtual real computeError(array&);
    };
```

In addition to the standard `output` and `weight` arrays contained in the Link, there is also an `activation` array. This will be discussed below.

The basic structure of a simulation algorithm was shown in Chapter II. It is handled by the `simulate()` function in the Net class, presented again here:

```
/*
 * simulate() -- run (and possibly) train the network
 *   Runs a single cycle
 */
void Net :: simulate(array& target)
    {
    if (phase != setup)
        {
        ++epoch;
        propagate();
        computeError(target);
        if (phase == training)
            {
            backpropagate();
            adjust();
            }
        }
    }
```

When this method is executed, each method in it calls its counterpart for each of the Links in the network.

The first function to examine for an ADALINE Link is `propagate()`. Its construction is straightforward. It simply computes the output of each layer as shown in the diagram above and passes it on to the next one. In the case of an ADALINE network, there is only one layer.

```
/*
 * propagate() -- pass signals forward thru AdLink
 *   Output of upper layer is set from output of lower
 *     layer and transfer function.
 */
void AdLink :: propagate(void)
      {
      real sum;

      for (integer i = 1; i <= upper->numberUnits(); i++)
            {
            sum = 0.0;
            for (integer j = 0; j <= lower->numberUnits(); j++)
                  {
                  sum += ((*weights)[i][j])*(*(lower->output))[j] ;
                  }
            activation->addAt(i, sum);//needed for error computation
            upper->output->addAt(i, lower->f(sum));
            }
      }
```

Note that the array `activation` holds the output of the summer (*n*) from the units before it goes into the `purelin()` function. This is needed when computing the error term.

The next function to examine is `computeError()`. Using the array of target values passed to the Net, it computes the error values in the layer:

```
/*
 * computeError() -- compute error for a link
 */
real AdLink :: computeError(array& target)
      {
      real error, err, act;

      error = 0.0;
      for (integer i = 1; i <= upper->numberUnits(); i++)
            {
            act = (*activation)[i];
            err = target[i-1] - act;
            upper->error->addAt(i, err);
            error += 0.5 * sqr(err);
            }
      return error;
      }
```

For ADALINE networks, the error value is simply the target minus the output of the summer.

The final method to cover here is `adjust()`. This computes the new weights and bias terms for the network. It is shown below:

```
/*
 * adjust() -- adjust weights within a BpLink
 */
void AdLink :: adjust(void)
      {
      real err, out, dw;

      for (integer i = 1; i <= upper->numberUnits(); i++)
            {
            for (integer j = 0; j <= lower->numberUnits(); j++)
                  {
                  out = (*(lower->output))[j];
                  err = (*(upper->error))[i];
                  dw  = dweights->getAt(i, j);
                  weights->addAt(i,j,
                        weights->getAt(i,j)+ALPHA*err*out);
                  dweights->addAt(i, j, ALPHA*err*out);
                  }
            }
      }
```

This is a straightforward implementation of the matrix version of the equations shown earlier. The constant `ALPHA` is used in the ADALINE Net code as the learning rate (α) shown earlier.

In Chapter VI, we will complete this example with a variation on the character recognition problem used by Widrow and Hoff. For now, we will continue the presentation of the algorithmic part of the library by turning to backpropagation.

Backpropagation

As mentioned earlier, both ADALINE networks and Perceptrons were limited. Because there was no way to train multiple layers of neurons, inputs presented to the network had to be linearly separable to allow proper classification. This severely restricted the kinds of problems that could be solved with a neural network, although ADALINE found a home in many signal-processing applications.

The problem with training a multiple layer neural network lies in adjusting the weights of the hidden and input layers. The LMS algorithm used for ADALINE can be used for adjusting the output weights, but how can the error be propagated backwards through the layers which connect to the output layer and used to correctly adjust these weights as well? This is the problem that backpropagation solves.

Paul Werbos (1974) proposed the first algorithm to train multilayer networks of neurons. It was then independently rediscovered and widely publicized in the mid-1980's by Rumelhart, Hinton, and Williams (1986), among others.

To develop the backpropagation algorithm, let us first look at a typical multi-layer network, shown in Figure 5.

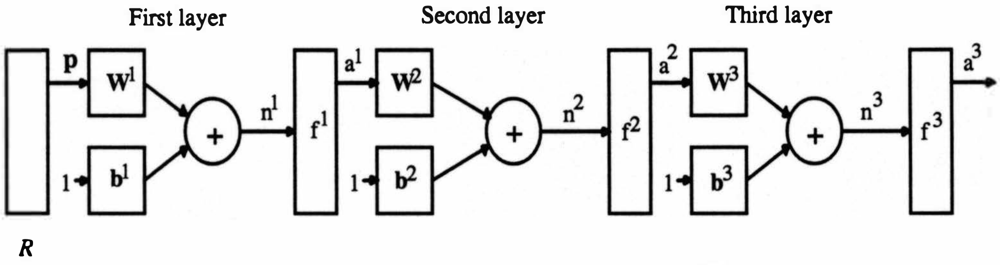

Figure 5. A Three-Layer Neural Network.

Using our notation, the output $\mathbf{a}^3$ is given by:

$$\mathbf{a}^3 = f^3(\mathbf{W}^3 f^2(\mathbf{W}^2 f^1(\mathbf{W}^1 p + \mathbf{b}^1) + \mathbf{b}^2) + \mathbf{b}^3)$$

and in general the output for a given layer is given by

$$\mathbf{a}^{m+1} = f^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}) \text{ for } m = 0,2,\ldots M-1$$

with the neurons in the first layer receiving the external inputs, and the neurons in the last layer being the network outputs.

The performance index for a multilayer network is a generalization of the LMS algorithm used for ADALINE. Again, the network is provided with a set of examples of proper behavior. And again, the mean square error is used to compute the correct weights and biases. For the approximate steepest descent algorithm (using the squared error) the weights and biases in a particular layer are given by:

$$w_{i,j}^m(k+1) = w_{i,j}^m(k) - \alpha \frac{\partial \hat{F}}{\partial w_{i,j}^m}$$

$$b_i^m(k+1) = b_i^m(k) - \alpha \frac{\partial \hat{F}}{\partial b_i^m}$$

Where α is the learning rate.

In an ADALINE network, it was easy to compute the partial derivative using just the weights and biases in the current layer. In a multilayer network, the chain rule must be applied. In the case of the partial derivatives above, this is:

$$\frac{\partial \hat{F}}{\partial w_{i,j}^{m}} = \frac{\partial \hat{F}}{\partial n_{i}^{m}} \times \frac{\partial n_{i}^{m}}{\partial w_{i,j}^{m}}$$

$$\frac{\partial \hat{F}}{\partial b_{i}^{m}} = \frac{\partial \hat{F}}{\partial n_{i}^{m}} \times \frac{\partial n_{i}^{m}}{\partial b_{i}^{m}}$$

Note that the second term in these equations can be computed easily since it is the net input to the layer:

$$n_{i}^{m} = \sum_{j=1}^{S^{m-1}} w_{i,j}^{m} a_{j}^{m-1} + b_{i}^{m}$$

Therefore

$$\frac{\partial n_{i}^{m}}{\partial w_{i,j}^{m}} = a_{j}^{m-1}, \frac{\partial n_{i}^{m}}{\partial b_{i}^{m}} = 1$$

If we now define *sensitivity* (*s*) as follows:

$$s_{i}^{m} = \frac{\partial \hat{F}}{\partial n_{i}^{m}}$$

We can think of this as the sensitivity of *F* to changes in the i^{th} element of the net input at layer *n*. At this point, our approximate steepest descent algorithm becomes:

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha s^m (\mathbf{a}^{m-1})^T$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha \mathbf{s}^m$$

We now have equations for weights and biases that involve only quantities known at each step. The remaining problem becomes computing the sensitivities. This process involves computing the sensitivity of the *m*th layer in the network and backpropagating it to the (*m*-1)th layer. This is where the term "backpropagation" comes from. The recurrence relation for this is:

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \mathbf{s}^{m+1}$$

This equation expresses the sensitivity of the *m*th layer in terms of the derivative of the transfer function for that layer, and the weights and sensitivities of the (*m*+1)th layer.

We now have an algorithm for training a multilayer neural network so long as the transfer function **F**(x) is differentiable. In summary, the algorithm is:

1. The input is propagated forward through the network, with the output of each layer used as the input of the successive layer. The output of the last layer is the output of the entire network.

2. The sensitivities are computed. The sensitivity of the output layer is computed directly from the error. The sensitivity of the other layers are computed using the sensitivities of the successive layer, using the recurrence

relation above. The sensitivities are propagated backwards through the network.

3. The weights and biases for each layer are updated using the sensitivities and the LMS error and approximate steepest descent.

We can easily fit this algorithm into our existing library structure. We first define a BpLink class:

```
/* backpropagation version of Link */

class BpNet; // forward reference

class BpLink : public Link
      {
protected:
      BpNet *net;
public:
      BpLink(Layer *from, Layer *to) : Link(from, to) { };
      BpNet *netOf(void) { return net; };
      void netIs(BpNet *n) { net = n; };
      virtual void backpropagate(void);
      virtual void adjust(void);
      virtual real computeError(array&);
      };
```

The `propagate()` method is straightforward. In fact, it is common to both backpropagation and TD(λ) algorithms, so it is placed in the standard Link class rather than BpLink:

```
/*
 * propagate() -- pass signals forward thru Link
 *   Output of upper layer is set from output of lower
 *     layer and transfer function.
 */
void Link :: propagate(void)
      {
      real sum;


      for (integer i = 1; i <= upper->numberUnits(); i++)
            {
            sum = 0.0;
            for (integer j = 0; j <= lower->numberUnits(); j++)
                  {
                  sum += ((*weights)[i][j])*(*(lower->output))[j] ;
```

```
                }
            upper->output->addAt(i, lower->f(sum));
            }
        }
```

Next, we need to provide a `computeError()` method:

```
/*
 * computeError() -- compute error for a link
 */
real BpLink :: computeError(array& target)
        {
        real error, err, out;

        error = 0.0;
        for (integer i = 1; i <= upper->numberUnits(); i++)
                {
                out = (*(upper->output))[i];
                err = target[i-1] - out;
                upper->error->addAt(i, GAIN * upper->f_dot(out) * err);
                error += 0.5 * sqr(err);
                }
        return error;
        }
```

The `error` array actually contains the sensitivity for the upper layer in it. It is up to the programmer to correctly specify the `f_dot()` function with the derivative of the layer's transfer function before the network is constructed. The constant `GAIN` is provided as a way to attenuate the size of the error if needed.

With outputs and errors computed, we now backpropagate the sensitivity values to each layer:

```
/*
 * backpropagate() -- pass errors backwards thru BpLink
 */
void BpLink :: backpropagate(void)
        {
        real err, out;

        for (integer i = 1; i <= lower->numberUnits(); i++)
                {
                out = (*(lower->output))[i];
                err = 0.0;
                for (integer j = 1; j <= upper->numberUnits(); j++)
                        {
                        err += ((*weights)[j][i]) * (*(upper->error))[j];
```

```
                  }
            lower->error->addAt(i, GAIN * upper->f_dot(out) * err);
            }
      }
```

This implements the recurrence equation shown earlier: the error array from the upper layer is passed back to the lower layer, multiplied by the derivative. This computes the sensitivity for the lower layer.

Using everything computed thus far we finally adjust the weights in each layer:

```
/*
 * adjust() -- adjust weights within a BpLink
 */
void BpLink :: adjust(void)
      {
      real err, out, dw;

      for (integer i = 1; i <= upper->numberUnits(); i++)
            {
            for (integer j = 0; j <= lower->numberUnits(); j++)
                  {
                  out = (*(lower->output))[j];
                  err = (*(upper->error))[i];
                  dw  = dweights->getAt(i, j);
                  weights->addAt(i, j,
                        weights->getAt(i,j)+ETA*err*out+ALPHA*dw);
                  dweights->addAt(i, j, ETA*err*out);
                  }
            }
      }
```

The constant ETA is used when computing the weight adjustment as a "momentum" term. The designer chooses it in much the same way as α. It is multiplied by the magnitudes of the error and output values. This causes the algorithm to descend the error surface faster when the error is larger. This improves the convergence of the algorithm in most cases.

There is a detailed example showing the use of backpropagation to predict sunspot numbers in Chapter VI, and an analysis of the algorithm's convergence performance in Chapter V.

Temporal Difference Learning and TD(λ)

One may think of the problem of adjusting weights in a neural network in terms of credit assignment: how does each individual weight in the matrix **W** affect the operation of the network with each given input? Both ADALINE and backpropagation are algorithms providing what might be termed structural credit assignment. That is, the effect of each weight is determined by presenting all the inputs in succession until a previously agreed-upon error value is reached. The difference between the expected and actual outputs is used to drive the adjustment in the weights.

But in many types of problems there is also a temporal component to the problem of credit assignment. Take, for example, the problem of predicting the weather. If we wish to predict Friday's weather on Monday with a neural network, we might want to have the various inputs (such as temperature, wind speed and direction, and barometric pressure) weighted differently than when we predict Friday's weather on Thursday. Presumably, we will have more accurate data on Thursday than Monday. At this time an appropriate reinforcement (or penalty) is sent to the network and the weights adjusted accordingly. This is type of learning is called reinforcement learning, to distinguish it from supervised learning.

As it turns out, there are many interesting problems that fall into this category, and no good way to solve them with a neural network using the backpropagation algorithm. Barto and Sutton (1990), motivated by an interest in these types of problems, turned their attention to so-called "temporal difference learning" and the TD(λ) algorithm.

The theory behind TD(λ) is that weights in the network are updated at each prediction using the difference in expected outputs at each time step. The weight update rule is:

$$w_{ij}^{t+1} = w_{ij}^{t} + \alpha \sum_{k \in O} (P_k^{t+1} - P_k^{t}) e_{ijk}^{t}$$

This rule states that the difference between the successive predictions

$$(P_k^{t+1} - P_k^{t})$$

(called the *temporal difference*) is used to update individual weights according to an "eligibility" term *e* computed for each weight in **W**. In the example above, the notation e_{ijk}^{t} means "the k^{th} eligibility at time *t* of the weight from unit *i* to unit *j*". This is computed using the following:

$$e_{ijk}^{t} = \sum_{n=1}^{t} \lambda^{t-n} \left(\frac{\partial P_k^{n}}{\partial w_{ij}^{n}} \right)$$

This is another gradient approximation based on the LMS error. Once again, computation of this term is simplified by using a recursive, backpropagation-like process.

First, the eligibility is simplified to:

$$e_{ijk}^{t+1} = \lambda e_{ijl}^{t} + \delta_{kj}^{t+1} y_i^{t+1}$$

where

$$s_j^t = \sum_{i \in FI_j} w_{ij}^t y_i^t$$

(This is the weighted sum of the inputs to unit *j* at time *t*) and

$$\delta_{kj}^{t+1} = \frac{\partial P_k^{t+1}}{s_i^t}$$

We must now have an algorithm to compute δ. This is defined as:

$$\delta_{ki}^t = \frac{\partial P_k^t}{\partial s_i^t} = \begin{cases} y_i^t(1 - y_i^t) \text{ if } k = i \\ 0 \text{ if } k \in O \text{ and } k \neq i \\ \sum_{j \in FO_i} \delta_{kj}^t w_{ij}^t y_i^t (1 - y_i^t) \text{ otherwise} \end{cases}$$

Note that we are only using either the outputs of the network (y), or the deltas for the successive layer (δ). Since both of these quantities are known, we can now write code to adjust the weights in a temporal-difference neural network (Sutton, 1987).

Once again, we may use the structure we have developed for ADALINE and backpropagation. First, we create a TdLink, which holds the additional arrays needed for the eligibility (*e*) and δ:

```
class TdLink : public BpLink
      {
protected:
      matrix *eligibility;
      array *delta;
      void newtrial(void);
      TdNet *net;
public:
      TdLink(Layer *from, Layer *to);
      ~TdLink(void);
      void netIs(TdNet *n) { net = n; };
      virtual void backpropagate(void);
      virtual void adjust(void);
      virtual real computeError(array&);
      };
```

The function `propagate()` is straightforward and in fact we can use the one we have for Link. The next function we need to write is computeError():

```
/*
 * computeError() -- compute error for a TD(lambda) link
 */
real TdLink :: computeError(array& target)
      {
      real error, err, out;

      error = 0.0;
      for (integer i = 1; i <= upper->numberUnits(); i++)
            {
            out = (*(upper->output))[i];
            err = target[i-1] - out;
            upper->error->addAt(i, err);
            error += 0.5 * sqr(err);
            }
      return error;
      }
```

Note that this is very similar to the functions for ADALINE and backpropagation. This makes sense since the overall error for the TD(λ) network is LMS. The array `target` contains the reinforcement at the appropriate point in the simulation.

The function where TD(λ) is more complex than either of the two prior algorithms is in `backpropagate()`. Here we need to perform several tasks:

1. We must backpropagate the error at the current level to the one feeding it. The error must be in a "self-contained" form since Links are generally independent objects from one another.

2. We must compute δ for the current Link.

3. We must compute the eligibility for the current Link.

All this is performed by the function below:

```
/*
 * backpropagate() -- pass errors and delta backwards thru TdLink
 */
void TdLink :: backpropagate(void)
      {
      integer j;
      TdLink *fo;
      real err, delt, out, outdelt, elg, wt;

      // backpropagate error to next lower layer
      for (integer i = 0; i <= lower->numberUnits(); i++)
            {
            err = 0.0;
            for (integer j = 1; j <= upper->numberUnits(); j++)
                  {
                  err += (*(upper->error))[j];
                  }
            lower->error->addAt(i, err);
            }

      // compute delta for this Link
      if (upper->isA() == outputlayer)
            {
            for (i = 1; i <= upper->numberUnits(); i++)
                  {
                  out = (*(upper->output))[i];
                  delta->addAt(i, (out * (1.0 - out)));
                  }
            }
      else
            {
            for (i = 0; i <= upper->numberUnits(); i++)
                  {
                  fo = net->FO(upper);
                  for (outdelt=0.0, j=1;
                        j<=fo->upper->numberUnits(); j++)
                        {
                        outdelt += fo->delta->getAt(j);
                        }
                  out = (*(upper->output))[i];
                  wt = weights->getAt(i, 0);
                  delt = outdelt*wt*(i ? (out * (1.0 - out)) : 1);
                  delta->addAt(i, delt);
                  }
            }
// compute eligibilities for this Link
if (upper->isA() == outputlayer)
      {
      for (i = 0; i <= lower->numberUnits(); i++)
            {
            out = (*(lower->output))[i];
            for (delt=0.0,j=1;j <= upper->numberUnits(); j++)
                  {
                  delt += delta->getAt(j);
                  }
            elg = eligibility->getAt(i, 0);
            elg *= LAMBDA;
            elg += delt * out;
            eligibility->addAt(i, 1, elg);
            }
      }
else
      {
      for (i = 0; i <= lower->numberUnits(); i++)
            {
            for (integer j=1; j <= upper->numberUnits(); j++)
                  {
                  out = (*(lower->output))[i];
                  delt = delta->getAt(j);
                  elg = eligibility->getAt(i, j);
                  elg *= LAMBDA;
                  elg += (i ? delt*out : delt);
                  eligibility->addAt(i, j, elg);
                  }
            }
      }
}
```

The observant reader will have noticed several things about this code when comparing it to the other algorithms presented in this chapter (other than the embarrassing length, that is.) Namely:

1. It checks to see if we are processing an OutputLayer or not, and performs slightly different processing for eligibility and δ if we are. This is in keeping with the equations for δ and *e* shown earlier. This task requires additional predicates for a Link to return information about the Link's type.

2. There was no provision in the original structure to return the "fan out" of a layer. This has also been added. It is performed by the following function:

```
/*
 * FO() -- return Link with fanout of current Link
 *   Passed Layer which must be "lower" of Link
 */
TdLink *TdNet :: FO(Layer *l)
      {
      for (integer i = n_layers-2; i >= 0; i--)
            {
            if (links[i]->fromLayer() == l)
                  return (TdLink *)links[i];
            }
      return NIL_LINK;
      }
```

This returns a pointer to the Link which is the fanout of the Layer passed to it. In short, we needed to stretch the library structure a bit to accommodate the TD(λ) algorithm, but not much.

Now we can adjust the weights. This code is more straightforward:

```
/*
 * adjust() -- adjust weights within a TdLink
 *   Learning rule is weight + rate * error * eligibility
 */
void TdLink :: adjust(void)
      {
      real err, elg, w, dw;

      for (integer i = 1; i <= upper->numberUnits(); i++)
            {
            for (integer j = 1; j <= lower->numberUnits(); j++)
                  {
                  err = (*(upper->error))[i];
                  elg = eligibility->getAt(j, i);
                  w = weights->getAt(i, j);
                  dw = (ALPHA * err * elg);
                  w += dw ;
                  weights->addAt(i, j, w);
                  }
            }
      }
```

There is only one more detail to clean up. This is in the handling of the eligibility trace. The eligibilities are cumulative, so we require a way to clear the trace between trials. In the original design for the library, we had no provision for "between trial" housekeeping. We need to add it now. This is performed by the `newtrial()` method.

```
/*
 * newtrial() -- handle between-trial housekeeping
 *   For TD(lambda), this means clearing eligibilities
 */
void TdLink :: newtrial(void)
      {
      for (integer i = 0; i <= lower->numberUnits(); i++)
            {
            for (integer j = 0; j <= upper->numberUnits(); j++)
                  {
                  eligibility->addAt(i, j, 0.0);
                  }
            }
      }
```

The TD(λ) version of the Net class calls this function at the start of every `simulate()` cycle so we can reset the eligibility trace.

In Chapter VI there is an extended example of using TD(λ) to compute Markov random walk probabilities. This will demonstrate how the algorithm and the library are used in a practical application.

We now have completed the development of the three main algorithms in the library. We hope it is apparent that the library's structure may accommodate other types of ANS algorithms as well.

In the next chapter, we will examine the performance of these algorithms, especially comparing this C++ implementation with a strictly C implementation. Note that the performance of the library at run time is important, but secondary to development time for the project.

CHAPTER V

PERFORMANCE ISSUES

In this chapter we examine the library's runtime performance. Specifically, we are interested in answers to the following questions:

1. Are the data structure classes we have created efficient?

2. How do learning rules perform as the size of the neural network increases?

3. Is the C++ implementation of Monarch efficient when compared to a strictly C implementation?

For the answers to the first and third questions, we will rely on timing data to provide an empirical answer. For the answer to the second question, we will use more detailed analysis of the algorithms presented in Chapter IV.

Collection Class Performance

As the source code for the algorithms presented in Chapter IV showed, Monarch makes heavy use of array and matrix collection classes. Therefore, we can obtain insight into performance of the library by looking at the performance of these classes compared to their "standard" C++ counterparts.

There are two fundamental operations performed on an array or matrix in C and C++:

1. *Declaration* introduces the variable's name into the scope of the program, and allocates storage for its contents.

2. *Referencing* uses the variable's name and (optionally) a subscript to compute an address to a part of the variable's store. From this point, the access may entail a fetch or a store of data.

Using the native array facility in C++, both these operations require minimal overhead. Since the language was designed to run "on top of C", these operations both deal with storage using pointers, which the compiler translates appropriately. The drawback to this design is that there is no bounds checking on arrays, and no provision for arrays of varying size. Both of these are actually considered "features" in native C/C++: since storage is allocated linearly when the program is compiled, the programmer can utilize tricks to use less or more of an array or matrix as long as care is taken.

In Chapter II, we introduced the array and matrix classes. At the time, we stated the desire to provide for automatic bounds checking, but only when the program was being debugged. We used a preprocessor macro to determine if the extra bounds checking code should be included in the program.

In order to test the array and matrix classes, we will adopt the following methodology. First, we will write and compile the famous Sieve of Eratosthenes. This has long been used as a "standard" benchmark in computing product comparisons (Gilbreath, 1981), sometimes with questionable validity (the principal objection to the Sieve is that a clever compiler can use optimizations that are unrealistic for a "real" program to produce very good benchmark results). In this case, as a test of subscripting, it should be a reasonable test of performance.

The theory behind the Sieve is simple. The program finds and/or counts prime numbers. An array is created which holds 1...*n* Boolean flags. All the flags are initially set "true". The array is scanned in ascending order and as factorable numbers are found, the matching array elements are set "false." When the end of the array is reached, the elements still "true" are prime numbers.

The program shown below is two test cases in one. If the preprocessor macro `CONTAINERS` is defined, the Monarch collection classes are used. If not, an array of real data (type `real`) is defined instead. The only difference in the two versions is the way in which the array of flags is defined and referenced. Even the rest of the Monarch application framework is omitted.

The source file `SIEVE.CPP` is shown below.

```
/* sieve
 * Eratosthenes Sieve Prime Number Routine
 * BYTE, Aug. 1983, pg. 112
 * ver. 1.0   01/08/84
 */

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

const size = 1024;
const iterations = 10000;

#ifdef COLLECTIONS
array flags(size + 1);
#else
real flags[size+1];
#endif

int main(int argc, char *argv[])
      {
#ifdef COLLECTIONS
```

```
      puts("Using collection classes");
#else
      puts("Using `standard' arrays");
#endif
      printf("sizeof(flags) = %d bytes\n", sizeof(flags));

      clock_t begin = clock();
      integer count = 0;
      printf("%d iterations\n", iterations);
      for(integer iter = 0; iter < iterations; iter++)
            {
            count = 0;
            for(integer i = 0; i <= size; i++)
                  {
#ifdef COLLECTIONS
                  flags.addAt(i, true);
#else
                  flags[i] = true;
#endif
                  }
            for(i = 0; i <= size; i++)
                  {
                  if(flags[i])
                        {
                        integer prime = i + i + 3;
                        for(integer k=i=prime;
                              k <= size; k = k + prime)
                              {
#ifdef COLLECTIONS
                              flags.addAt(k, false);
#else
                              flags[k] = false;
#endif
                              }
                        count++;
                        }
                  }
            }
            printf("\n%d primes. ", count);
            clock_t elapsed = clock() - begin;
            float elapsed_sec = (float)elapsed/CLOCKS_PER_SEC;
            printf("It took %10.6f seconds\n", elapsed_sec);
            return 0;
      }
```

This is the (in)famous *Byte* magazine benchmark program, modified for C++ by removing explicit initialization of variables (in the interest of brevity) and adding timing code. Constants are used rather than `#define` for the size of the array and the number of test iterations.

Timing is accomplished by the `clock()` function, included in most IBM PC compatible C compilers' libraries. This function returns a `long` integer containing the "ticks" that have elapsed since the program was started. These ticks are based on the computer's timer interrupt, which for historical reasons occurs 18.2 times a second. Thus, the resolution of the timer is approximately 55 milliseconds. For higher precision timing, additional software is required. For our purposes, this resolution is sufficient.

The `SIEVE.CPP` program was run on a 133 MHz Micron® Pentium system in MS DOS under Windows 95. The programs were compiled under Borland Turbo C++ 3.0 using the large memory model.

Table 1 shows the results of running the program in several configurations.

Table 1

Collection Class Benchmark Results

	Using subscripts	Using collection classes
Bounds checking off	2.47 sec.	6.53 sec.
Bounds checking on	N/A	12.03 sec.
Register variables off	3.73 sec.	7.03 sec.
Executable size	32,474 bytes	43,464 bytes

As the table shows, using collection classes is slower than using just native C-style arrays. One hypothesis for this result is that the `addAt()` and `getAt()` (subscript reference) methods require C++ message processing, unlike their native C counterparts, which index array data through a CPU register. On this benchmark, an optimizing compiler can easily put some key variables in registers and obtain a sizable speed improvement, as the times without register variables (this is a compiler switch on Turbo C++) show. This is one reason the Sieve is not considered a good benchmark for general application performance (Gilbreath, 1983).

A second hypothesis for this result is that the arrays in the collection classes are dynamically allocated, and therefore accessed through an extra pointer. (This pointer contains the address of the array base.) This extra pointer reference increases the amount of time needed to access the data in the array.

Nevertheless, the speed penalty here is not prohibitive, and the convenience of having collection classes is probably worth the extra runtime required. This is also true of executable memory size, which increases a bit when the Monarch library routines that implement collection classes are added to the program.

Performance and Network Size

We now turn to the issue of runtime performance versus network size. Here it makes sense to analyze the algorithms comprising the `simulate()` function based on the number of neurons in the network. For example, let us consider the ADALINE `adjust()` function, shown below:

```
/*
 * adjust() -- adjust weights within an AdLink
 */
void AdLink :: adjust(void)
      {
```

```
real err, out, dw;

for (integer i = 1; i <= upper->numberUnits(); i++)
      {
      for (integer j = 0; j <= lower->numberUnits(); j++)
            {
            out = (*(lower->output))[j];
            err = (*(upper->error))[i];
            dw  = dweights->getAt(i, j);
            weights->addAt(i,j,
                  weights->getAt(i,j)+ALPHA*err*out);
            dweights->addAt(i, j, ALPHA*err*out);
            }
      }
}
```

The run time of this algorithm is directly proportional to the size of the two layers connected by the Link. If *m* is the number of neurons in the upper layer, and *n* the number of neurons in the lower, then the runtime is bounded by *n* × *m*. Since the ADALINE network consists of a single Link connecting two layers (one of which is simply a set of inputs), this is the total runtime for the `adjust()` function.

Runtimes for the remaining functions called by `simulate()` can be estimated the same way.

If there are *R* layers in the network (see Chapter II for more on this notation), then the runtime will be (*R*-1) × *n* × *m*. This is because there are *R*-1 links in the network.

The runtime behavior of the algorithms supported by the library is summarized in Table 2. The results can be summarized by stating that theoretically, the algorithm run times are directly proportional to the number of neurons in the network. Therefore, we may compare implementations as long as the size of the networks being compared is the same.

Table 2

Summary of Algorithm Runtime Behavior

Algorithm Step	ADALINE	Standard backpropagation	TD(λ)
`Propagate()`	$n \times m$	$(R\text{-}1) \times n \times m$	$(R\text{-}1) \times n \times m$
`ComputeError()`	M	$(R\text{-}1) \times n \times m$	$(R\text{-}1) \times m$
`Backpropagate()`		$(R\text{-}1) \times n \times m$	$(R\text{-}1) \times 3 \times n \times m$
`Adjust()`	$n \times m$	$(R\text{-}1) \times n \times m$	$(R\text{-}1) \times n \times m$

Performance in Applications

We have covered enough material on performance in the previous two sections to conclude that the neural algorithms should have a linear performance increase as the number of neurons increases, and the performance penalty for a C++ implementation should be modest. We will now verify these two conclusions with a further comparison, benchmarking C versions of ADALINE, standard backpropagation, and TD(λ) against C++ versions.

First, let's look at ADALINE. The details of this example will be presented in detail in Chapter VII. For now, it suffices to remark that the program trains an ADALINE network to recognize digits encoded as ASCII characters in a grid.

Since the initial random weights of the network affect it's performance as the minimum error is reached, we must measure each individual simulation cycle and then average these elapsed times over the number of epochs run to get an idea of how fast the library performs. This is done with measurement code added to the simulation cycle as follows:

```
/*
 * Simulate(in, out, target) -- run a single simulation epoch
 */
void Simulate(array& in, array& out, array& target)
      {
      il->from(in);
      clock_t starting = clock();
      recognize->simulate(target);
      clock_t ending = clock();
      total_elapsed += (ending-starting);
      ol->to(out);
      }
```

Note that the code above will average times for both training and testing simulation epochs together. This is done for convenience; the two could be easily separated. The average is probably more indicative of how the network is used in a production setting, however. The global variable `total_elapsed` holds the clock ticks tallied over the entire simulation.

When the application terminates, statistics can be printed in the `~App` destructor, since it contains any application-specific termination code:

```
/*
 * ~App() -- delete network when program finishes
 *   Layers, Links, etc. are deleted automatically
 */
App :: ~App(void)
      {
      fclose(f); // close the output file when done
      delete recognize; // delete Net

      cout << "Total clocks=" << total_clocks << "\n";
      cout << "Total epochs=" << total_epochs << "\n";
      float total_time = (float)total_clocks / CLOCKS_PER_SEC;
      cout << "Total time=" << total_time << "\n";
      cout << "Average time=" << total_time/total_epochs << "\n";
      }
```

After the Net has been deleted and the output file closed, the code shown above prints the statistics for the simulation run, including the total number of clock ticks, the total time, and the average clock ticks.

This measurement code is added to the C versions of the simulations in a similar way. All the simulations were run on the same 133 MHz Micron® Pentium computer used for the Sieve benchmarks mentioned earlier.

The results of the simulations are shown in Table 3. The benchmark results support some interesting conclusions.

First, in every case the "native" ANSI C version of the simulation was faster, by a factor of roughly 3 times. This was (again, roughly) the same factor by which the C++ collection classes underperformed their C counterparts in the Sieve benchmark. It suggests that most of the performance overhead in the simulation is due to just this one implementation difference.

Second, the C++ ADALINE and backprop average simulation cycles are similar. We might expect this since, as the derivations in Chapter IV showed, the two are very similar algorithms. This gives us a good feeling about the soundness of the implementation.

Third, the TD(λ) algorithm appears faster than the backprop or ADALINE algorithms. While it is difficult to make exact comparisons in this type of benchmarking environment (see below), it is faster on average. This echoes the findings of Barto and Sutton, who found that TD(λ) had a smaller incremental computation load than backprop. It demonstrates that TD(λ) is not to be shunned because of the (perceived) complexity demonstrated in its algorithmic derivation. When it comes to performance, it appears to work as well or better than backpropagation.

Table 3

Summary of C/C++ Comparison Benchmarks

Program	Total Clocks	Total Epochs	Time	Average seconds/cycle
ADALINE C++	1772 ticks	30930	97.36 sec.	0.00315 sec./cycle
ADALINE C	340 ticks	28510	19.01 sec.	0.000667 sec./cycle
Backprop. C++	4154 ticks	67740	228 sec.	0.000337 sec./cycle
Backprop C	23174 ticks	1197430	1273.3 sec.	0.001063 sec./cycle
TD(λ) C++	1748 ticks	344438	96.04 sec.	0.000276 sec./cycle
TD(λ) C	386 ticks	229896	21.21 sec.	0.0000923 sec./cycle

Performance Caveats

There are some factors to bear in mind when considering this performance data, especially when making comparisons between the different types of algorithms. (The exact details of the examples will be presented in Chapter VII.)

1. The architectures of the networks are dissimilar. We are using the examples presented in Chapter VII to run these benchmarks. One should bear in mind that the ADALINE network has only two layers of neurons, one of which is used for inputs. They both have (approximately) the same

number of neurons, however, so the observation earlier about their average cycle times is accurate.

2. The TD(λ) network is smaller than either the ADALINE or backprop networks in the examples. As the section on network size showed, this directly affects performance. It also uses reinforcement learning rather than supervised learning, so in a sense we are comparing apples to oranges when we speak of one outperforming the other. The important point here is that the performance of TD(λ) was reasonable in both the C and C++ versions, and the model should not be rejected from consideration because of its (apparent) complexity.

Backpropagation Performance

The backpropagation example reveals that the algorithm's convergence time is highly variable. During testing, the network took between 67740 epochs and 2045130 epochs to reach an acceptable error level on the training data. The runtime varied in proportion to this. The number of epochs required depends on the random weights when the network is initialized. If the error "surface" described by the gradient is complex, with many hills and valleys, the adjustments to the network during each simulation cycle can overshoot a minimum point and require correction on the following iterations. This lengthens the time required for the network to reach an appropriate minimum.

In the library, a modified "momentum" term is used to control the algorithm's descent down the network's error gradient. This is the function of `ETA` in the library's backpropagation code shown in Chapter IV:

```
/*
 * adjust() -- adjust weights within a BpLink
 */
void BpLink :: adjust(void)
      {
      real err, out, dw;

      for (integer i = 1; i <= upper->numberUnits(); i++)
            {
            for (integer j = 0; j <= lower->numberUnits(); j++)
                  {
                  out = (*(lower->output))[j];
                  err = (*(upper->error))[i];
                  dw  = dweights->getAt(i, j);
                  weights->addAt(i, j,
                        weights->getAt(i, j)+ETA*err*out+ALPHA*dw);
                  dweights->addAt(i, j, ETA * err * out);
                  }
            }
      }
```

Even with this additional constant to control the size of the "jump" at each iteration, it still requires a variable number of iterations during training.

Several more complex methods exist for improving backpropagation's performance. These include "batching" (saving all the weight updates for each training sample and then averaging them together), the computing conjugate gradient, and the Levenberg-Marquardt algorithm. These are documented in other sources (Hagan, Demuth, and Beale, 1995), and have not been implemented in the library yet. They are discussed further in Chapter VIII.

Conclusions

The benchmarks run in this chapter demonstrate that, while the C++ implementation imposes a significant overhead on the library, this can be traced to the collection classes used in all the algorithms. One would expect that if the library were re-written to use static arrays, runtime would improve considerably.

This said, the advantages of having these collection classes can outweigh the additional performance penalties they impose. For instance, since the collections are allocated dynamically, the configuration of the network may be changed at runtime. This allows a Monarch network to be tuned using the following heuristic: initially, a large number of neurons can be used, and the network trained until a given error reached. This network will fit the training set very well, but not be generalized to all testing data sets. Subsequent training cycles can then be run with neurons successively removed from this network until an appropriate tradeoff between generality and error performance is reached.

Another advantage of the C++ collection implementation is exception handling. C++ provides a much richer set of tools for handling exceptions generated within objects, such as subscripting errors, domain errors in transfer functions, etc. This topic will be covered in detail in the next chapter.

CHAPTER VI

TYPE SAFETY AND EXCEPTION HANDLING

We have covered a lot of material in the last five chapters, so a brief summary is in order. We now have a class hierarchy for the library; a set of classes providing initialization and configuration services for neural network applications; collection classes for representing arrays and matrices comprising neuron layers; and implementations for three common types of neural networks. We have also tested and measured the library's performance compared to a strictly C version and found the overheads imposed by our design to be reasonable.

In this chapter we will complete construction of the library by focusing on two areas normally overlooked in neural network simulations:

1. The additional behavior needed by the classes to make them more object-oriented. This includes providing copy constructors and relational operators.

2. Exception handling, so error conditions likely to occur in normal use (such as array subscripts out of range or mathematical function overflow) are handled gracefully.

In C++ parlance, the first topic is sometimes called "type safety". That is, we want our assortment of collections, layers, and links to behave identically to other C++ "concrete" types such as floats, ints, and doubles.

Type Safety

Providing type safety for our neural network classes means adding additional methods to the class declarations. These cover situations where objects in the class are used as if they were normal "concrete" types like integers, real numbers, and characters.

These additional methods include: (a) a copy constructor invoked automatically by C++ when a new object is created from another object through assignment; (b) an assignment operator used when one object is assigned to another of the same class; (c) an equality operator, so two similar objects may be compared to one another and tested for equality; and (d) an inequality operator.

These four methods are generally considered to comprise a minimal set of type safe behaviors for a class. It is considered good C++ programming practice to provide at least these methods. This is not a requirement of the language, but it seems necessary for completeness, to consider these cases and account for them in the behavior of the objects created (Oualline, 1995).

In addition to these four basic methods, there are other methods we wish to provide. For example, we want to have arrays or matrices usable in arithmetic expressions, so we will need to overload the arithmetic operators for them.

The additional type safety methods needed in the library are summarized in Table 4. In the case of Net objects, we do not want the user to be able to freely create one network from another.

Table 4

Summary of Type Safety Functions

Class Name	Copy constructor?	Assignment Operator?	Equality and inequality?	Additional Requirements
`Net`		Yes	Yes	
`Link`	Yes	Yes	Yes	
`Layer`	Yes	Yes	Yes	
`Database`	Yes			
`Array`	Yes	Yes	Yes	Arithmetic operators
`Matrix`	Yes	Yes	Yes	Arithmetic operators

This is an arbitrary restriction, designed to keep the programmer out of trouble by ensuring that the construction protocol is followed. The user can assign one Net to another, however, knowing that the Net should be constructed when doing so. We will also make it possible to copy and assign Links and Layers, and this should allow a great deal of freedom to dynamically configure a network prior to the simulation's start.

With this explanation in mind, let's begin by examining the type-safety code for collections. The following allows arrays to be copied and assigned:

```
/*
 * array(const array&) -- copy constructor
 *   This is a new, complete copy of the array
*/
array :: array(const array &x)
      {
      a = (real *)calloc(x.size(), sizeof(real));
      n_items = x.n_items;
      here = x.here;
      next = x.next;
      for (integer i = 0; i < x.size(); i++)
            {
            *(a+i) = x.getAt(i);
            }
      }

/*
 * array operator = -- handle assignment
 *   Get rid of old array and copy new one to it
*/
array& array :: operator = (const array &x)
      {
      if (this != &x)
            {
            free(a);
            a = (real *)calloc(x.size(), sizeof(real));
            n_items = x.n_items;
            here = x.here;
            next = x.next;
            for (integer i = 0; i < x.size(); i++)
                  {
                  *(a+i) = x.getAt(i);
                  }
            }
      return *this;
      }
```

These two functions appear nearly identical, but the second version (`operator =`) must discard any allocations in the object passed to it, since a new object is being created which replaces the old one via the assignment. The first function is the copy constructor proper and C++ guarantees that a completely new object is being created.

For the equality comparison, we arbitrarily deem two arrays to be "equal" if all of their elements are equal and their sizes are the same. This has a benefit for matrices, which will be discussed shortly. Here is the code:

```
/*
 * array a == b
 */
int operator == (const array& a, const array& b)
    {
    if (a.size() != b.size())
        return (0==1);
    for (integer i = 0; i < a.size(); i++)
        {
        if (a.getAt(i) != b.getAt(i))
            return (0==1);
        }
    return (1==1);
    }
```

Once we have the equality operator written, inequality is trivial:

```
/*
 * array a != b
 */
int operator != (const array& a, const array& b)
    {
    return (!(a==b));
    }
```

Matrices may inherit these methods directly if we make our definition of "equality" generous enough to consider two matrices equal if their sizes are equal, and not necessarily the exact number of rows and columns (their shape). Thus, a 2 × 5 matrix may be equal to a 5 × 2 matrix if their elements are the same in the same order. This is an extension of other methods such as `mean()` and `randomize()` which treat matrices as arrays in row-major order.

Along with these basic methods, we wish to add support for arithmetic operators. While not used directly by any of the learning algorithms, they nonetheless might prove useful to anyone using the library. The additional arithmetic operators for arrays are:

```
array& operator += (const real&);
array& operator -= (const real&);
array& operator *= (const real&);
array& operator /= (const real&);
array& operator += (const array&);
array& operator -= (const array&);
array& operator *= (const array&);
array& operator /= (const array&);
friend array& operator - (array&);
friend array& operator + (array&, const real&);
friend array& operator - (array&, const real&);
friend array& operator * (array&, const real&);
```

```
      friend array& operator / (array&, const real&);
      friend array& operator + (const real&, array&);
      friend array& operator - (const real&, array&);
      friend array& operator * (const real&, array&);
      friend array& operator / (const real&, array&);
      friend array& operator + (array&, const array&);
      friend array& operator - (array&, const array&);
      friend array& operator * (array&, const array&);
      friend array& operator / (array&, const array&);
```

This is a lot of methods. The reason is that we need to handle the case where two arrays are being operated on together, plus the cases where a real and array is being used, plus the cases where an array and a real is being used. C++ does not have any idea of commutativity. Thankfully, most of the methods are brief (only a sample in each category above is shown here):

```
/*
 * a += r
 */
array& array :: operator += (const real& r)
    {
    for (integer i = 0; i < size(); i++)
        {
        addAt(i, getAt(i)+r);
        }
    return (*this);
    }


/*
 * a += a
 */
array& array :: operator += (const array& b)
    {
    for (integer i = 0; i < size(); i++)
        {
        addAt(i, getAt(i)+b.getAt(i));
        }
    return (*this);
    }


/*
 * unary -
 */
array& operator - (array& a)
    {
    for (integer i = 0; i < a.size(); i++)
        {
```

```
        a.addAt(i, -a.getAt(i));
        }
    return a;
    }

/*
 * a + r
 */
array& operator + (array& a, const real& r)
    {
    for (integer i = 0; i < a.size(); i++)
        {
        a.addAt(i, a.getAt(i)+r);
        }
    return a;
    }


/*
 * r + a
 */
array& operator + (const real& r, array& a)
    {
    for (integer i = 0; i < a.size(); i++)
        {
        a.addAt(i, r+a.getAt(i));
        }
    return a;
    }


/* Arithmetic operations on conforming arrays. */

/*
 * a + b
 */
array& operator + (array& a, const array& b)
    {
    for (integer i = 0; i < a.size(); i++)
        {
        a.addAt(i, a.getAt(i)/b.getAt(i));
        }
    return a;
    }
```

Again, because of the way that matrices are defined, these operations are also valid with matrices of conforming size.

Note that there is no explicit length checking in these operations. Instead, this is handled by the `getAt()` method being invoked within the `operator` method. This means all the operators work with the shorter of the two operands, and generate an exception (see the next section) for the longer. This shortcut allows the canny user a bit of C-like flexibility when writing programs, but it will still catch any subscripting errors leading to memory overwriting.

The remaining type-safety code in the library follows the pattern established for arrays. Some points worth mentioning are:

1. Layers are considered equal if their outputs are considered equal. This in turn means they must have the same number of neurons and the same values.

2. Links are considered equal if both of their constituent Layers are equal.

3. Nets are considered equal if all constituent Links are equal. Note that two Nets may have different training parameters or even different learning algorithms, but if they have the same outputs at each layer, we consider them equal anyway!

4. Databases may be copied and assigned, but the underlying files, handles, and descriptors are not duplicated in any way. This avoids unpleasantness with operating systems such as MS DOS. Two databases are considered equal if they are either streams or blocks and their filenames are identical. It is up to the programmer to define these type-safety methods for any database class descendants needed in his or her own program.

This completes the type-safety behavior required by the library. It is interesting to note that this additional code adds roughly 20% to the overall size of the library.

Exception Handling

The library has several circumstances where exception handling is in order. In this section, we examine how the library exploits C++ exception handling.

C++ has a graceful exception handling mechanism using the keywords `throw`, `catch`, and `try`. A brief review of this mechanism is appropriate here. Within each class definition, exception classes are defined, which may be signaled (invoked) during execution using the `throw` keyword. In the application, the programmer attempts execution of a block of code using `try`. If an exception occurs while this code is executed, a corresponding `catch` block is executed, with a mechanism for selecting among one of many exceptions thrown.

There are several instances in the library where exceptions may occur. Three of the most important are:

1. When a subscript is out of range, and the application is still being debugged. (Earlier it was demonstrated that range checking has prohibitive performance overhead in a "production" program.)

2. When a transfer function is out of range. In particular, the exponential sigmoid functions behave badly when their arguments exceed ±500.

3. When transferring data to or from a Layer and the number of items transferred exceeds the number of neurons in the Layer.

In these cases, we normally wish to halt execution of the simulation, especially while the program is being debugged. However, there may be times when the programmer would like to handle the exception in a more graceful manner. For this "fail most times" strategy, C++'s exception mechanism seems a good solution, better than using an `assert()` macro.

The only restriction on using the C++ exception mechanism is that it is not implemented in all versions of C++. In particular, Borland's Turbo C++ does not have it in versions 3.xx and earlier. Since this is a popular compiler, a workaround is needed. The following preprocessor definitions allow the standard C++ exception mechanism to be "commented out" and a call to a function called `exception()` inserted in place instead:

```
// Exception handling

#define EXCEPTIONS

#ifdef EXCEPTIONS
#define TRY try
#define CATCH(x) catch(x)
#define THROW(x) throw x
#define EXCEPT(x) class x { };
#else
void exception(char *, char *, int); // defined in EXCEPT.CPP

#define TRY
#define CATCH(x) if (1==0)
#define THROW(x) exception(#x,__FILE__,__LINE__)
#define EXCEPT(x)
#endif
```

If the preprocessor variable `EXCEPTIONS` is defined, then the "normal" C++ keywords for exception handling are enabled. If not, then any `catch` blocks in the program will be skipped (by using the empty conditional `if (1==0)`) and any `throw` functions will instead call a function called `exception()`with the offending exception, source file name, and line number. This function in turn calls the Standard Library's `abort()` handler, ending the program:

```
#ifndef EXCEPTIONS
void exception(char *s, char *f, int n)
    {
    printf("exception: %s file: %s line: %d\n", s, f, n);
    abort();
    }
#endif
```

This workaround is a good solution while the program is being debugged, and not too objectionable in a production program, since it causes termination of the program with a listing of the line number and filename where the exception occurred.

As an example of how exceptions are used, let us look at the code for Layers. The following function handles transfers between arrays and Layers:

```
/*
 * from(a) -- set contents of input layer from 'a'
 */
void InputLayer :: from(array& a)
      {
      if (a.size() >= output->size())
          THROW(Layer :: Size());
      for (integer i = 0; i < a.size(); i++)
            {
            output->addAt(i+1, a[i]);
            }
      }
```

This code checks to make sure the number of neurons in the Layer is greater than or equal to the number of elements in the array a. If not, `THROW(Layer :: Size())` invokes the exception handler for Layer. The declaration of the Layer class has the following in it now:

```
class Layer : public Monarch
      {
protected:
      integer n_units;
      char *layer_name;
public:
      array *sum, *output, *error;
      Layer(char *, integer);
      Layer(const Layer&);
~Layer(void);
void dump(void);
EXCEPT(Domain);
EXCEPT(Size);
integer numberUnits(void) { return n_units; }
char *layerNameOf(void) { return layer_name; }
Layer& operator = (const Layer&);
friend ostream& operator << (ostream&, Layer &);
friend istream& operator >> (istream&, Layer &);
friend database& operator << (database&, Layer &);
friend database& operator >> (database&, Layer &);
friend int operator == (const Layer&, const Layer&);
friend int operator != (const Layer&, const Layer&);
virtual real f(real x); // transfer function
virtual real f_dot(real x); // derivative of transfer function
};
```

The two boldfaced lines specify the exceptions for the Layer object. The Domain exception is signaled whenever a transfer function is passed an argument bigger than the constant `Overflow`:

```
const Overflow = 500; // for assertions

/*
 * logsig() -- sigmoid log transfer function
 */
real logsig(real n)
      {
      if (abs(n) >= Overflow)
          THROW(Layer :: Domain());
      return 1/(1 + exp(-n));
      }
```

Returning to the Size exception, in a program, we would try the code using `from()` like this:

```
/*
 * Train(epochs) -- train net for many epochs
 */
void Train(integer epochs)
      {
      array in(N_INPUTS), out(N_OUTPUTS), target(N_OUTPUTS);

      sunspots->setPhase(training);
      for (integer n = 0; n < epochs*TRAIN_YEARS; n++)
            {
            integer year = RandomEqual(TRAIN_LWB, TRAIN_UPB);
            TRY
                {
                in.from(Sunspots, year-N_INPUTS);
                target.from(Sunspots, year);
                Simulate(in, out, target);
                }
            CATCH (Layer::Size)
                {
                puts("Layer size error");
                DUMP(in);
                DUMP(target);
                // additional error recovery as required
                }
            }
    }
```

Should a Size exception occur, the code within the `CATCH` block is executed. In this case, the various offending objects are dumped. Note that if `EXCEPTIONS` is not defined, this block is skipped and the `exception()` function called instead.

With the addition of type-safety and exception handling to the library, we now have a complete protocol for creating and simulating neural networks. In the next chapter, we will examine several complete simulations, which will put all the software developed to use.

CHAPTER VII

EXAMPLES

This chapter presents several examples, demonstrating how Monarch is used, and how the C++ object-oriented model makes it possible to write neural network simulations that are concise and correct.

Before examining the examples in detail, it should be mentioned that there is little in the way of benchmark neural network simulations available either in literature or as public-domain software. A software engineer wishing to construct his or her own simulation faces the task of first implementing a learning algorithm and testing it with known data in a known network (to ensure the algorithm functions correctly), then collecting their application's data and writing their own simulation. It is hoped that Monarch will serve as a platform to avoid this repetition of work.

The examples chosen are the following: (a) the character recognition application of Widrow and Hoff, which uses an ADALINE network; (b) sunspot prediction using a backpropagation network; and (c) a Markov model example using TD(λ).

These examples are presented in order of increasing algorithmic complexity, following the order in Chapter IV. Two of the examples are based on the unpublished package "Neural Networks at your Fingertips" (Kutza, 1996), since this provided a set of C programs to use as a baseline for comparison. (For more on benchmark data, please refer to Chapter V.)

Character Recognition With ADALINE

In this example, an ADALINE network is trained to recognize characters in a 5×7 grid. This is a variation on the original work by Widrow and Hoff in 1960. In the original work, there were three characters (T, G, and F) each sited one of two ways on a 4×4 grid. The targets were –60, 0, and 60, which worked well with a meter that Widrow and Hoff used for their output display.

In this simulation, there are ten digits sited on a 5×7 grid. Samples are shown in Figure 6. These are the representations of the digits "0" and "1".

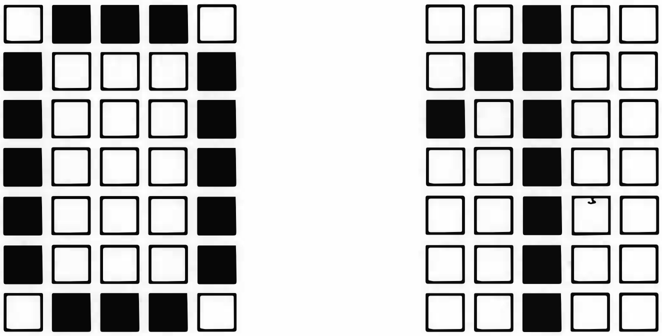

Figure 6. Sample Character Grids.

The goal of the simulation is to train a neural network to recognize all 10 digits. When presented with all 35 inputs corresponding to the 5×7 grid, the output of the neuron corresponding to the value of the digit should be 1. The architecture of this network is a single layer with ten neurons, each of which has 35 inputs, as shown in Figure 7. The drawing is simplified for clarity. There is actually a weight going from each cell of the input grid to each neuron, 350 in total.

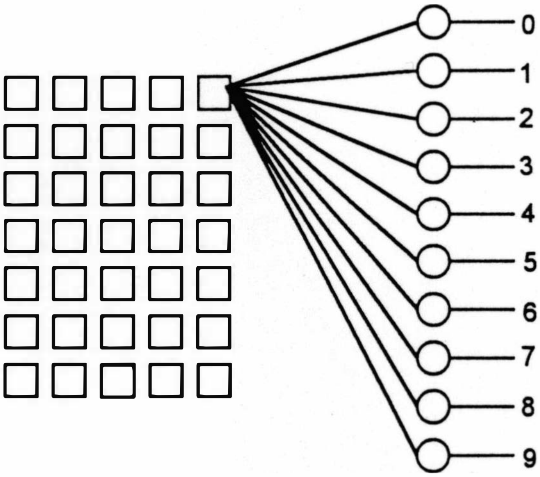

Figure 7. ADALINE Network Schematic.

Our approach to writing this simulation is the following: we will take the simulation framework presented in Chapter III and supply the appropriate `App`, `~App`, and `main()` functions to perform the simulation.

The basic architecture of the network will be a single ADALINE Link connecting 35 input "neurons" with 10 output neurons. This is reflected in the following constants and definitions:

```
#define X 5 // number of pixels across
#define Y 7 // number of pixels down

#define NUM_DATA 10 // number of characters to recognize
#define ADALINE_LAYERS 2
#define N_INPUTS (X*Y)
#define N_OUTPUTS (NUM_DATA)
#define N (X*Y)
#define M N_OUTPUTS
```

We then define the following objects:

```
/* Objects required for simulation */

array inputs(N_INPUTS), outputs(N_OUTPUTS), targets(N_OUTPUTS);

/* Transfer functions for ADALINE Layers */

real Layer :: f(real x) { return hardlims(x); }
real Layer :: f_dot(real x) { return 0; } // not used

/* ADALINE Layers and Network */

InputLayer *il;
OutputLayer *ol;
AdNet *recognize;
```

These create the input, output, and target arrays; associate the `hardlims()` transfer function with the Layers to be used in the simulation; and provide pointers to the various components to be configured in the App constructor.

The App constructor builds the neural network at runtime when the program is initialized:

```
/*
 * App() -- create and initialize neural network
 *   Note: many values are set as part of MonarchApp initialization
 */

App :: App(int argc, char *argv[]) :
      MonarchApp(argc, argv, "CHARS.INI", N_PARMS, test_ini)
    {
    il = new InputLayer("Inputs", n_inputs);
    ol = new OutputLayer("Outputs", N_OUTPUTS);

       recognize = new AdNet("Recognize", ADALINE_LAYERS, Alpha,
                             Epsilon);
    recognize->add(new AdLink(il, ol));

    f = fopen("ADALINE.TXT", "w"); //

    // This is here so you can enter the characters to recognize
    // in ASCII, then have them converted to input values.
    for (integer n = 0; n < NUM_DATA; n++)
        for (integer i = 0; i < Y; i++)
            for (integer j = 0; j < X; j++)
                  Input[n][i*X+j] =
                           (Pattern[n][i][j] == 'O')? HI : LO;
    }
```

The lines following `fopen()` require additional explanation. The input to the neural network is a value of –1 or 1 corresponding to whether the cell in the grid is "off" or "on", respectively. The training set consists of a set of 10 grids with the appropriate cells initialized to these values. Since typing these grids would be unpleasant, an ASCII character matrix is provided in the program instead. The grids can then be entered with a text editor:

```
Char Pattern[NUM_DATA][Y][X] = { { " OOO ",
                                   "O   O",
                                   "O   O",
                                   "O   O",
                                   "O   O",
                                   "O   O",
                                   " OOO "  },
```

Some way is needed, however, to transfer/convert the ASCII characters into input values. That is what the remaining code in `App()` does.

Deleting the network when the program is finished is trivial:

```
/*
 * ~App() -- delete network when program finishes
 *   Layers, Links, etc. are deleted automatically
 */
App :: ~App(void)
    {
    fclose(f); // close the output file when done
    delete recognize; // delete Net
    }
```

As we saw in Chapters II and VI, deletion of Links and Layers is handled automatically.

The rest of the simulation is handled in `main()`. This function is shown below:

```
integer main(integer argc, char *argv[])
      {
      theApp = new App(argc, argv);

      integer stop;
do
      {
      // evaluate network
      real error = 0.0;
      stop = 1;
      recognize->setPhase(testing);
      for (integer n = 0; n < NUM_DATA; n++)
            {
            SetInput(Input[n], false);
            SetTarget(Output[n]);
            Simulate(inputs, outputs, targets);
            error = max(error, recognize->errorOf());
            stop = stop &&
              (recognize->errorOf() < recognize->epsilonOf());
            }
      error = max(error, recognize->epsilonOf());
      printf("Training %0.0f%% completed ...\n",
            (recognize->epsilonOf()/error) * 100.0);

      // train network
      if (!stop)
            {
            recognize->setPhase(training);
            for (integer m = 0; m < 10*NUM_DATA; m++)
                  {
                  n = RandomEqual(0, NUM_DATA-1);
                  SetInput(Input[n], false);
                  SetTarget(Output[n]);
                  Simulate(inputs, outputs, targets);
                  }
            }
      } while (!stop);

// test network
for (integer n = 0; n < NUM_DATA; n++)
      {
      SetInput(Input[n], true);
      SetTarget(Output[n]);
      Simulate(inputs, outputs, targets);
      GetOutput(Output[n], true);
      }

delete theApp;
return 0;
}
```

The following steps are performed in `main()` as part of the complete simulation:

1. The network is trained using character data. The data is presented by the `SetInput()` function. The appropriate target is set with the

`SetTarget()` function. Both functions are shown below. During the training phase, the data is presented in random order.

2. After each epoch is completed, the maximum error in the network is compared to the value of ε set when the simulation was started. This is obtained with the `epsilonOf()` message. If the error for the test set is less than ε, training is complete.

3. When the error has reached an acceptable level, the final test set is run through the network and the output written to a file. A sample of the file's output is shown below:

```
 ooo
o   o
o   o
o   o
o   o
o   o
 ooo  -> 0

  o
 oo
o o
  o
  o
  o
  o   -> 1
```

The functions to set the input and target values are straightforward:

```
/*
 * SetInput() -- copy input to array for simulation
 */
void SetInput(real *Input, boolean writeit)
      {
      for (integer i = 0; i < inputs.size(); i++)
            inputs.addAt(i, Input[i]);
      if (writeit == true)
            WriteInput(Input);
      }

/*
 * SetTarget() -- set target array from Outputs
 */
```

```
void SetTarget(real *Output)
      {
      for (integer i = 0; i < outputs.size(); i++)
            targets.addAt(i, Output[i]);
      }
```

The `SetInput()` function copies the input data to the network's inputs from the `Input[][]` array, which was initialized from the ASCII data by `App()` when the program was started. The `SetTarget()` function does likewise for the `Output` array.

Once these arrays have been set up, running the actual simulation cycle is trivial:

```
/*
 * Simulate(in, out, target) -- run a single simulation epoch
 */
void Simulate(array& in, array& out, array& target)
      {
      il->from(in);
      recognize->simulate(target);
      ol->to(out);
      }
```

Depending on which phase the network is currently set to (training or testing), an error will be computed and learning (weight adjustment) will occur. When this simulation is run, the error value converges very rapidly to a nominal value, as shown in Figure 8. Streaming the error value to an ASCII database and then using MATLAB produced the graph.

Sunspot Prediction With Backpropagation

In this example, astronomical sunspot data is used to predict the distribution of sunspots over time. Sunspots are dark areas on the Sun's surface. They tend to follow a 21-year cycle (of keen interest to amateur radio operators) but the underlying causes of this cycle are poorly understood. Here, annual sunspot data from the years 1700-1960 is used to predict the distribution of sunspots from the years 1960 to 1978. The first data set constitutes the training set, and the second the testing set. Both sets will be kept strictly separate by the program so that the network's predictive ability can be accurately measured.

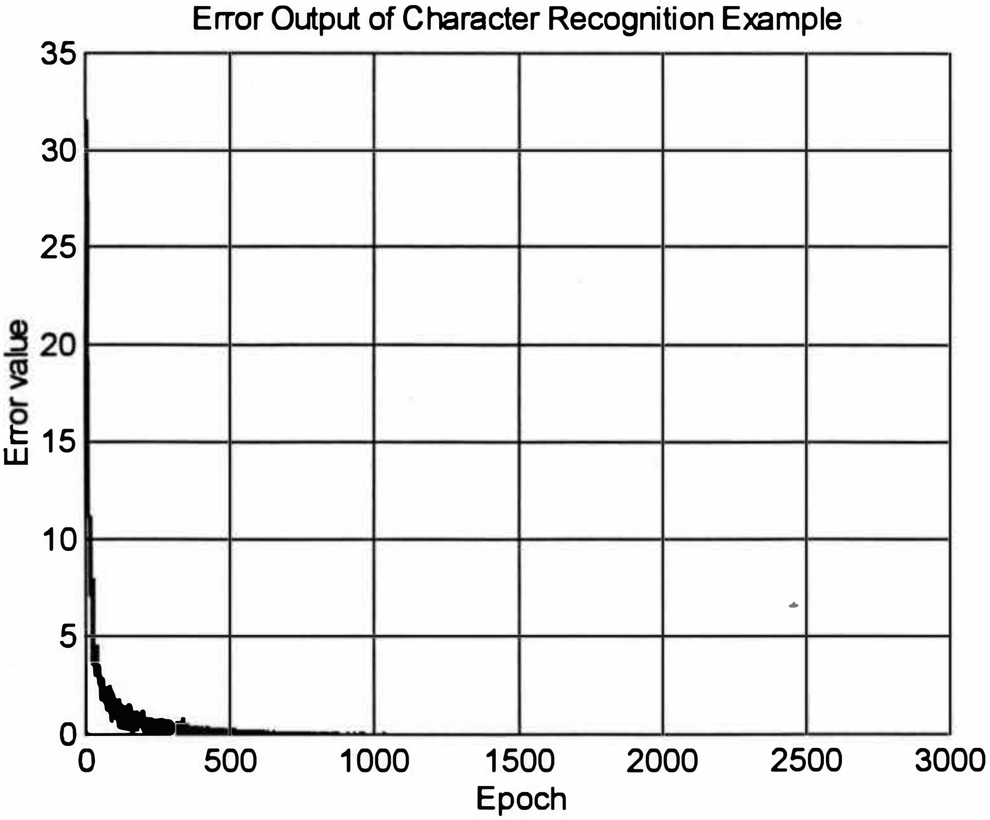

Figure 8. ADALINE Error Convergence.

For this simulation, a three-layer backpropagation neural network is used. The input layer receives a 30-year period of observed sunspot counts, normalized so these counts are between 0 and 1. (This normalization is critical.)

Once again we follow the template for a Monarch program. The App function is shown below:

```
/* Layers and Network */

InputLayer *il;
OutputLayer *ol;
HiddenLayer *hl;
BpNet *sunspots;

/*
 * App() -- create and initialize neural network
 *    Note: many values are set as part of MonarchApp initialization
 */

App :: App(int argc, char *argv[]) :
      MonarchApp(argc, argv, "BPN.INI", N_PARMS, test_ini)
      {
      il = new InputLayer("Inputs", n_inputs);
      ol = new OutputLayer("Outputs", N_OUTPUTS);
      hl = new HiddenLayer("Hidden", n_hidden);

      sunspots = new BpNet("Sunspots", N_LAYERS, Gain, Eta, Alpha);
      sunspots->add(new BpLink(il, hl));
      sunspots->add(new BpLink(hl, ol));
      }
```

The number of input neurons, hidden neurons, and the constants for the gain, η, and α are set from the initialization file. They can be easily changed for different runs of the simulation.

Once again the actual simulation is handled by the main() function:

```
integer main(integer argc, char *argv[])
      {
      theApp = new App(argc, argv);
      ifstream from("sunspots.dat");
      if (!from)
            {
            cout << "Can't open input file\n";
            exit(1);
            }
      from >> Sunspots;
      Sunspots.normalize();
      Mean = Sunspots.mean();
      Sunspots_ = Sunspots;
      ComputeInitialError();

      boolean stop = false;
      real MinTestError = MAX_REAL;
      do {
            Train(10);
            Test();
            if (TestError < MinTestError)
                  {
                  printf(" - saving weights...");
                  MinTestError = TestError;
                  sunspots->save();
                  }
            else
                  {
                  if (TestError > 1.2 * MinTestError)
                        {
                        printf(" - stop and restore weights");
                        stop = true;
                        sunspots->restore();
                        }
                  }
            } while (!stop);

      Test();
      Evaluate();

      delete theApp;
      return 0;
      }
```

The functions `Test()`, `Train()`, and `Evaluate()` are self-explanatory. They are shown below:

```
/*
 * Train(epochs) -- train net for many epochs
 */
void Train(integer epochs)
      {
      array in(N_INPUTS), out(N_OUTPUTS), target(N_OUTPUTS);

      sunspots->setPhase(training);
      for (integer n = 0; n < epochs*TRAIN_YEARS; n++)
            {
            integer year = RandomEqual(TRAIN_LWB, TRAIN_UPB);
            in.from(Sunspots, year-N_INPUTS);
            target.from(Sunspots, year);
            Simulate(in, out, target);
            }
      }

/*
 * Test() -- test net after each epoch
 */
void Test(void)
```

```
      {
      array in(N_INPUTS), out(N_OUTPUTS), target(N_OUTPUTS);

      sunspots->setPhase(testing);
      TrainError = 0.0;
      for (integer year = TRAIN_LWB; year <= TRAIN_UPB; year++)
            {
            in.from(Sunspots, year-N_INPUTS);
            target.from(Sunspots, year);
            Simulate(in, out, target);
            TrainError += sunspots->errorOf();
            }
      TestError = 0.0;
      for (year = TEST_LWB; year <= TEST_UPB; year++)
            {
            in.from(Sunspots, year-N_INPUTS);
            target.from(Sunspots, year);
            Simulate(in, out, target);
            TestError += sunspots->errorOf();
            }
      printf("\nNMSE is %0.3f on training set and %0.3f on test
      set",
            TrainError/TrainErrorPredictingMean,
            TestError/TestErrorPredictingMean);
      }

/*
 * Evaluate() -- compute network's performance
 */
void Evaluate(void)
      {
      array in(N_INPUTS), target(N_OUTPUTS);
      array out(N_OUTPUTS), out_(N_OUTPUTS);

      sunspots->setPhase(evaluating);
      printf("\n\n\n");
      printf("Year Sunspots Open-Loop Prediction Closed-Loop
      Prediction\n");
      for (integer year = EVAL_LWB; year <= EVAL_UPB; year++)
            {
            // open-loop prediction
            in.from(Sunspots, year-N_INPUTS);
            target.from(Sunspots, year);
            Simulate(in, out, target);

            // closed-loop prediction
            in.from(Sunspots_, year-N_INPUTS);
            target.from(Sunspots_, year);
            Simulate(in, out_, target);
            Sunspots_.addAt(year, out_[0]); // note assignment

            // print results
            printf("%4d %0.3f %0.3f %0.3f\n",
                FIRST_YEAR+year,
                Sunspots[year],
                out[0],
                out_[0]);
        }
}
```

This simulation outputs a normalized prediction for sunspots between the years 1960 and 1978. This is shown below:

```
Year    Sunspots    Open-Loop Prediction    Closed-Loop Prediction
1960      0.587                  0.357                  0.346
1961      0.282                  0.224                  0.230
1962      0.196                  0.103                  0.130
1963      0.146                  0.096                  0.138
1964      0.053                  0.268                  0.293
1965      0.079                  0.245                  0.282
1966      0.246                  0.328                  0.358
1967      0.491                  0.549                  0.533
1968      0.554                  0.588                  0.549
1969      0.552                  0.489                  0.474
1970      0.546                  0.404                  0.391
1971      0.348                  0.390                  0.373
1972      0.360                  0.265                  0.265
1973      0.199                  0.193                  0.223
1974      0.180                  0.108                  0.150
1975      0.081                  0.088                  0.118
1976      0.066                  0.084                  0.105
1977      0.143                  0.135                  0.170
1978      0.484                  0.317                  0.323
```

When this data is streamed to a database and plotted using MATLAB, the graph shown in Figure 9 results. This is the distribution of sunspots both from actual data and as predicted by the simulation. The smooth line joins the actual sunspot counts, while the open and closed-loop predictions are shown by the "+" and "o", respectively.

The graph demonstrates that the simulation is reasonably good at predicting sunspot activity using nothing more than the count of sunspot data.

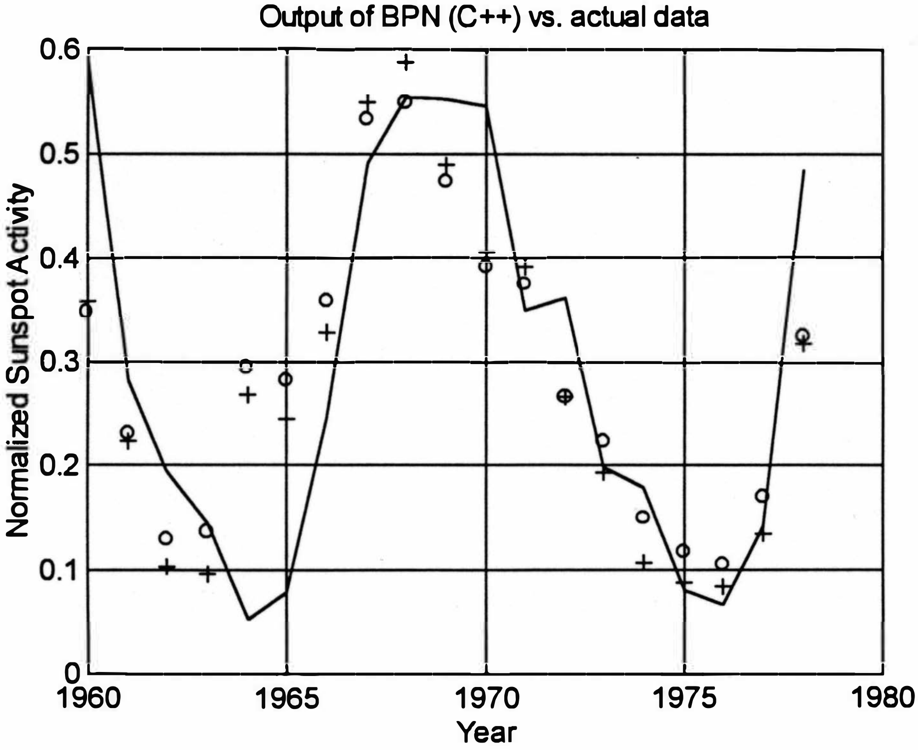

Figure 9. Graph of Actual and Predicted Sunspot Counts.

Random Walk and TD(λ)

There is no standard example in the literature for this learning algorithm. Therefore, this program replicates the random walk example in Sutton (1988). In it, there are nine states, labelled A-I. The probability of moving from one state to either adjacent state at any time is 0.50. This is shown in Figure 10. In the simulation, a random walker begins at state E. Successive states are generated until either state A or I is reached. When state A is reached, a reward signal z=0 is generated, while at state I, reward z=1.

The walk ends when either terminal state is reached. The neural network is trained to predict the reward signal for each of the intermediate states. Since rewards are only generated when two of the eight states are reached, efficient training using conventional backpropagation methods would be difficult.

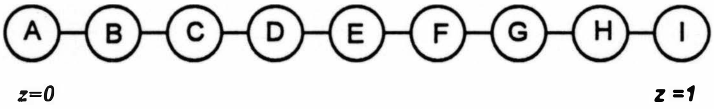

Figure 10. "Random Walk" Markov Model.

To simulate this problem, a three-layer network is used, with seven input neurons, three hidden neurons, and one output neuron. The inputs reflect the presence of the walker in any of states B-H, while the output is the probability of reaching the I state at each of these states.

To create the network with Monarch, an App constructor function is written:

```
App *theApp;

/*
 * App() -- create and initialize neural network
 *   Note: many values are set as part of MonarchApp initialization
 */

App :: App(int argc, char *argv[]) :
      MonarchApp(argc, argv, "MARKOV.INI", N_PARMS, test_ini)
      {
      il = new InputLayer("Inputs", n_inputs);
      ol = new OutputLayer("Outputs", N_OUTPUTS);
      hl = new HiddenLayer("Hidden", n_hidden);

      markov = new TdNet("Markov",N_LAYERS,Gain,Eta,Alpha,Lambda);
      markov->add(new TdLink(il, hl));
      markov->add(new TdLink(hl, ol));
      }
```

This creates the three Layers, then makes the Net, then links the Input and Hidden Layers together, followed by the Hidden and Output Layers.

Running the simulation consists of: (a) generating the states of the model, (b) passing them to the network, (c) obtaining a prediction of the probability, and (d) training the network with an updated prediction, or testing the network's predictive ability.

To simulate the network, the following function is used:

```
/*
 * Simulate(in, out, target) -- run single simulation epoch
 */
void Simulate(array& in, array& out, array& target)
      {
      il->from(in);
      markov->simulate(target);
      ol->to(out);
      }
```

This sets the input layer from the array `in`, runs a single simulation epoch, and returns the array `out` from the output layer. Since Monarch automatically handles training and testing modes internally, performing the TD(λ) learning algorithm on each training epoch, these functions are very similar:

```
/*
 * Test(t) -- test network
 *   Passed t, which row of state vector to use
 */
real Test(integer t)
      {
      array in(7), out(1), target(1);

      markov->setPhase(testing);
      CopyState(t, in);
      target.addAt(0, 0.0); // doesn't matter here
      Simulate(in, out, target);
      return out[0];
      }

/*
```

```
 * Train(p) -- train network
 *   Update weights
 *   Passed state t and prediction p for current state
 */
real Train(integer t, real p)
      {
      array in(7), out(1), target(1);

      markov->setPhase(training);
      CopyState(t, in);
      target.addAt(0, p);
      Simulate(in, out, target);
      return out[0];
      }
```

Only association of a predicted probability *p* to the input separates the two functions, although they are kept separate in the interest of clarity.

Note well that generating moves in the random walk occurs "outside" of the Monarch library. This is where having a library to implement the algorithm is preferable to a complete simulation "environment" where simulating this part of the problem would be difficult. The function `CopyState()` transfers state information from the part of the program where the walker's move is generated to the part of the program that interfaces with the library. This is shown below:

```
/*
 * CopyState(where) -- copy state to an `array' structure
 */
void CopyState(integer to, array& where)
      {
      for (integer i = 0; i < 7; i++)
            {
            where.addAt(i, state[to][i] ? 1.0 : 0.0);
            }
      }
```

When run, this program displays a crude ASCII indicator of the walker's state. This is shown below:

```
Trial 14
      0               1
      A B C D E F G H I
                *       Prediction: 0.396191
```

```
              *         Prediction: 0.394591
                *       Prediction: 0.396187
              *         Prediction: 0.394587
                *       Prediction: 0.396182
                  *     Prediction: 0.389263
                *       Prediction: 0.396177
                  *     Prediction: 0.389259
                *       Prediction: 0.396172
                  *     Prediction: 0.389254
                    *   Prediction: 0.389252
                      * Reinforcement = 1.0
```

Unlike the other simulations, a fixed number of trials are run. At completion, the walker's probabilities are close to the expected values, indicating that the network has "learned" a Markov model of how state transitions occur:

```
Trial 9999
      0               1
      A B C D E F G H I
                  *     Prediction: 0.717366
                *       Prediction: 0.570204
                  *     Prediction: 0.717663
                *       Prediction: 0.571133
              *         Prediction: 0.453579
            *           Prediction: 0.310332
              *         Prediction: 0.453189
                *       Prediction: 0.570005
              *         Prediction: 0.453108
            *           Prediction: 0.310447
          *             Prediction: 0.205433
            *           Prediction: 0.309549
          *             Prediction: 0.20515
            *           Prediction: 0.308638
              *         Prediction: 0.450955
            *           Prediction: 0.308975
              *         Prediction: 0.45057
                *       Prediction: 0.567068
                  *     Prediction: 0.715164
                    *   Prediction: 0.857647
                  *     Prediction: 0.716282
                    *   Prediction: 0.858192
                      * Reinforcement = 1.0
```

In this chapter, three examples were presented. These demonstrate that Monarch's implementation of the algorithms in Chapter IV is correct, and the library may be successfully used to create simulations of ADALINE, backpropagation, and TD(λ) neural networks.

CHAPTER VIII

CONCLUSIONS

With presentation and discussion of the examples in Chapter VII, the original aim of this thesis is complete. Nonetheless, other issues have been raised during the course of this research, and in this chapter we discuss them and suggest directions for further work on the library.

Clearly, the library as it stands now is incomplete as a production tool. It works well for simple, multi-layer feedforward networks, but it has limitations for applications that are more complex and demanding. While it demonstrates that C++ is an effective vehicle for constructing neural network simulations, it may be profitably extended with the following additions: (a) additional learning algorithms, (b) "front end" processors for configuring and tuning the network's performance, and (c) a "back end" tool for delivering a trained and tested network to an embedded system.

Let us address each of these issues in turn.

Additional Learning Algorithms

It is generally agreed that standard backpropagation has drawbacks in practical use. The biggest of these is its slow convergence. A thorough treatment of this subject may be found in Masters (1995). A number of methods for improving convergence have been developed (Hagan, Demuth, and Beale 1995). These include:

1. “Batching” weight updates and performing them at the completion of an epoch, using the mean of the update.

2. A “momentum” term to descend smoother parts of the error gradient more rapidly.

3. Varying the learning rate based on the magnitude of the change in error at each iteration.

4. Applying numerical optimization techniques such as computing the conjugate gradient and the Levenberg-Marquardt algorithm.

Currently the backprop algorithm in the library uses method (2) since it is straightforward to implement. For the types of programs a typical Monarch user wishes to write, this is sufficient. But a future direction might be implementing either of the numerical optimization algorithms as a descendant of the BpLink and BpLayer classes.

Three-layer backpropagation is far and away the most popular method for training neural networks in production settings, but there are other learning algorithms that could be included into the library for the student and experimenter. These include associative, competitive, and adaptive resonance networks. All of these could be implemented as descendants of the fundamental Net, Link, and Layer classes.

Front-End Processing

The design of the library makes it possible to create specialized “front-end” processing tools to increase its utility. One such tool might be a program to read a network’s schema as a file and translate it into the C++ statements to initialize and simulate it. This approach has already been used in the Aspirin/MIGRAINES package (Leighton, 1992), where Aspirin is the language to specify a network's architecture. Here is a sample Aspirin "program" for a character-recognition problem similar to the one shown in Chapter VII:

```
/* The abcd.aspirin file in this directory is
   far more complex than necessary for the
   problem. It is meant to illustrate capabilities
   of the Aspirin language. This network will
   solve the problem much more effectively.
   Even smaller networks than this are possible
   using tessellation.
   -Russ
*/

DefineBlackBox ABCD
{

  OutputLayer-> OUTPUT
  UpdateInterval-> 1
  InputSize-> [16 x 16]


  Components->
  {
      PdpNode OUTPUT [4]
        {
          InputsFrom-> H
        }
      PdpNode H [2 x 2]
        {
          InputsFrom-> $INPUTS
        }

   }
  }
```

The Aspirin syntax resembles C or C++ so programmers familiar with one of these languages can feel comfortable with the package. This file will be processed into a C file, then linked with a library/simulator for execution. There is no reason the same strategy cannot be employed with Monarch. The UNIX tools Lex and Yacc would be appropriate for this task.

Back-End Processing

Once a neural network has been trained and tested, it is usually be employed in a control or pattern recognition system. There is currently no facility for conveniently embedding a network into such a system. Instead, some portion of the simulation must be re-coded for the target program. This is typically the part of the network that propagates the signals from the input neurons to the outputs. (Of course, the weights have been established by training and may be treated as constants.)

This propagation algorithm may have performance overheads that are unacceptable in a process control or real-time system. This is because library facilities that make simulation more convenient (such as floating point numbers and collection classes) may not be appropriate for an embedded system.

Another useful tool, therefore, would take a trained network and allow convenient emplacement into an embedded software system. It should support scaled or fixed-point arithmetic and integer weights (Kahn and Wilson, 1996) as opposed to floating point arithmetic and use contiguous, statically allocated arrays instead of collection classes. It may even more efficient to output the network's weights and propagation algorithms directly into assembly language instead of C. Such a tool would build on the support already in Monarch for stopped training and saving weights.

Concluding Remarks

Designing a library in C++ is both a pleasant and challenging task. It is pleasant because the language provides features to tackle annoyances such as store allocation and exception handling in a consistent and elegant way. It is challenging because the designer now must confront these issues rather than push them onto the application programmer.

In particular, adding appropriate type safe behavior to the library is an important (and often overlooked) part of the overall implementation cycle. The designer must confront issues such as "what does addition of two arrays mean?" or "what happens if one layer is assigned to another?" and in doing do clarify one's thinking about both the specific objects one is trying to manipulate and the overall computing problem one is trying to solve. This I found particularly challenging.

It is hoped Monarch demonstrates the original intent of the thesis, that an object-oriented design and C++ implementation of neural network algorithms makes writing such programs convenient and pleasant. More importantly, it is hoped that this work (and the Monarch library) will serve as a useful tool for other programmers to go forward and develop their own simulations. If students and experimenters find it useful in this way, then it will have, in my opinion, been successful.

BIBLIOGRAPHY

Booch, Grady. 1991. *Object-Oriented Design with applications*. Redwood City, California: Benjamin/Cummings.

Castleman, Kenneth R. 1996. *Digital Image Processing*. Englewood Cliffs, New Jersey: Prentice-Hall.

Coplein, James O. 1992. *Advanced C++ Programming Styles and Idioms*. Reading, Massachusetts: Addison-Wesley.

Franz, Marty. 1990. *Object-Oriented Programming Featuring Actor*. Glenview, Illinois: Scott, Foresman & Company.

Gilbreath, Jim. 1981. "A High-Level Language Benchmark," *Byte*, Septmber 1981, 180.

________. 1983. "Eratosthenes Revisited: Once More Through The Sieve", *Byte*, January 1983, 283.

Goldberg, Adele and David Robson. 1983. *Smalltalk-80: The Language and its Implementation*. Reading, Massachusetts: Addison-Wesley.

Hagan, Martin T., Demuth, Howard B., and Mark Beale. 1995. *Neural Network Design*. Boston: PWS Publishing Company.

Hebb, D. O. 1949. *The Organization of Behavior*. New York: Wiley.

Isbell, Charles L. 1992. Explorations of the Practical Issues of Learning Prediction-Control Tasks Using Temporal Difference Learning Methods, Master's Thesis, Massachusetts Institute of Technology.

Joyce, C. W. 1990. Network data structures and representation in the simulation of neural networks, Master's thesis, the University of Tennesee.

Kahn, Altaf H. and Roland Wilson. 1996. Integer-Weight Approximation of Continuous-Weight Multilayer Feedforward Nets. Technical report, University of Warwick, Coventry, England.

Kohn, Phil. 1992. *Connectionist Layered Object-Oriented Network Simulator (CLONES) User's Manual.* Berkeley, California: International Computer Science Institute TR-91-073.

Kutza, Karsten. 1996. Neural Networks at your fingertips. Unpublished public-domain software package available on the Internet at `http://www.geocities.com/CapeCanaveral/1624/`.

Leighton, Russell R. 1992. *The Aspirin/MIGRAINES Neural Network Software User's Manual,* Technical Report MP-91W00050, the MITRE Corporation

Masters, Timothy. 1993. *Practical Neural Network Recipes in* C++. San Diego: Academic Press.

________. 1995. *Advanced Algorithms for Neural Networks: a C++ Sourcebook.* New York: John Wiley & Sons.

McCulloch, W. and Pitts, W. 1943. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics* 5:115-133.

Minsky, Marvin and Seymour Papert. 1969. *Perceptrons*. Cambridge, Massachusetts: MIT Press.

Oualline, Steve. 1995. *Practical* C++ *Programming*. Sebastapol, California: O'Reilly & Associates, Inc.

Plonski, M. and C. Joyce. 1990. RCS, GENESIS, and SFINX: Three "public domain" simulators for neural networks. *Neural Network Review*. 4(3/4)

Rosenblatt, Frank. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review* 65:386-408.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. 1986. Learning internal representation by error propagation. *In Parallel Distributed Processing: Explorations in the Microstructure of Cognition,* Vol. 1. Cambridge, Massachusetts: MIT Press.

Stroustrup, Bjarne. 1991. *The C++ Programming Language, Second Edition.* Reading, Massachusetts: Addison-Wesley.

Sutton, Richard S. 1987. Implementation details of the TD(λ) procedure for the case of vector predictions and backpropagation. GTE Laboratories Technical Note TN87-509.1.

________. 1988. Learning to predict by the methods of temporal differences. *Machine Learning*, 3, 9—44.

Sutton, Richard S. and Andrew G. Barto. 1990. Time-derivative models of Pavlovian reinforcement. *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, M. Gabriel and J. Moore, Eds. Cambridge: MIT Press.

Tsitsiklis, John N. and Benjamin Van Roy. An Analysis of Temporal-Difference Learning with Function Approximation. Cambridge, Massachusetts: Laboratory for Information and Decision Systems

Venables, W.N. and Riply, B.D. 1994. *Modern Applied Statistics with S-Plus.* New York: Springer-Verlag.

Werbos, Paul. 1974. "Beyond regression: New tools for prediction and analysis in the behavioral sciences", Ph.D. Diss., Harvard University.

Widrow, B. and M. E. Hoff. 1960. Adaptive switching circuits. In *1960 IRE WESCON Convention Record* by the IRE Part 4, 96-104. New York: IRE Part 4.

Zell, Andreas. 1995. *Stuttgart Neural Network Simulator User Manual Version 4.01*, Technical Report 6/95, University of Stuttgart. Available electronically at www.informatik.uni-stuttgart.de/`ipvr/bv/projekte/snns`.