THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

UTILIZATION OF STATE INFORMATION VIA RECURRENT CONNECTIONS IN EXPERT NETWORKS

By

CATHERINE LEBLANC

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Summer Semester, 1997

The members of the Committee approve the dissertation of Catherine LeBlanc defended on July 10, 1997.

S. I. Hruska
Professor Directing Dissertation

T. Baker
Committee Member

C. J. Kacmar
Committee Member

R. C. Lacher
Committee Member

T. Logan
Outside Committee Member

Approved:
R.C. Lacher, Chair, Department of Computer Science

To Kyle Scott and Alysha Aubrey
the next generation

ACKNOWLEDGEMENTS

Many graduate students and faculty members have helped to make my time at FSU interesting and meaningful. Without Emily Ratliff, Renee Renner and Tarina Whitfield, I surely would have lost my sanity during the prelim exam. Thanks also to Kristi Adair, Alan Levis, Sophia Roberts, Pam Strozier, and Bob Timpany for their advice and support at the various stages of my graduate career. Dr. Ted Baker, Dr. Chuck Kacmar, Dr. Chris Lacher, and Dr. Tim Logan, while serving on my dissertation committee, have shared their time and vast knowledge with me. Thanks to Dr. Mike Blaber for teaching two outstanding classes on proteins.

I could not have asked for a better major professor than Dr. Susan Hruska, from whom I have learned how to be a good teacher, a good researcher and a good person.

I would like to thank my family, Melvin and Kitty LeBlanc, Mark and Jenn LeBlanc and June LeBlanc, for their support, love, and patience. During the last, hard year of work on this dissertation, Dr. Cynthia Waddell has acted as sounding board, mentor, cheerleader and good friend. Finally, I would like to thank Dr. Evelyn Stiller. A list of all she has given me would run thousands of pages.

Contents

List of Tables x

List of Figures xi

ABSTRACT xiii

1 INTRODUCTION 1

1.1 **Motivation** 1

1.2 **Feedforward Neural Networks and Background Knowledge** 4

1.3 **Recurrent Neural Networks and Background Knowledge** 6

1.4 **Protein Secondary Structure Prediction** 7

1.5 **Dissertation Outline** 8

2 ANNOTATED EXAMPLE 10

2.1 **Description of Protein Secondary Structure Prediction Example** 10

2.2 **The Chou-Fasman Algorithm** 12

2.2.1 **The Chou-Fasman Rules** 14

2.2.2 **Difficulties with the Chou-Fasman Algorithm** . . . 15

2.3 **Implementation of the Chou-Fasman Algorithm** 20

2.3.1 **Rules for Finding Helices** 20

2.3.2 **Rules for Finding Strands** 21

2.3.3 **Rules for Prediction of Coil (The Default)** 22

3 EXPERT NETWORKS **23**

3.1 **Background** 23

3.2 **Design of System for Building Expert Networks** 26

3.2.1 **CBaseObj** 26

3.2.2 **CNode** 28

3.2.3 **COpNode** 30

3.2.4 **CBias** 31

3.2.5 **CSynapse** 32

3.2.6 **CNetwork** 33

3.2.7 **CPattern** 37

3.2.8 **CPatFile** 37

3.2.9 **CUserNode** 37

3.3 **Summary** 38

4 RECURRENT EXPERT NETWORK METHODOLOGY **41**

4.1 **Recurrence and State Information** 41

4.2 **Previous Work on Recurrent Networks** 43

4.2.1 **Elman Networks** 43

4.2.2 **Jordan Networks** 45

4.3 **Key Ideas and Heuristics** 47

4.3.1 **Types of Recurrence** 47

4.3.2 Recurrent Architectures 48
4.3.3 Types of Error 51
4.3.4 Learning Algorithms 54
4.3.5 Time In Network Activation and Learning Phases 56
4.3.6 Averaging of Weight Changes 58
4.3.7 Determination of Prediction Accuracy 59
4.3.8 Intermediate Concepts in the Network Architecture 63

5 EXPERIMENTS AND RESULTS 70

5.1 Experimental Methods 70
5.2 Helix Finding Expert Sub-Network 75
5.3 Strand Finding Expert Sub-Network 77
5.4 Synthesis of the Two Sub-Networks 78
5.4.1 Simple Combine 80
5.4.2 User Nodes, Combine Nodes 83
5.4.3 User Nodes, No Combine Nodes 85
5.4.4 No User Nodes, No Combine Nodes 86
5.5 Adding State Nodes 89
5.5.1 Experiment 1: State Nodes and Equal Add Node Thresholds 89
5.5.2 Experiment 2: Increasing the Add Node Thresholds 91
5.5.3 Experiment 3: Requiring More Evidence for Coil Prediction 94
5.5.4 Experiment 4: Requiring More Evidence for State Node Prediction of Helices 96

5.5.5 **Experiment 5: Requiring More Evidence for Helix Finding Subnetwork Prediction of Helices** 96

5.5.6 **Experiment 6: Requiring Less Evidence for Helix Finding Subnetwork Prediction of Helices** 100

5.6 **Results on FKBP-type Proteins** 105

6 DISCUSSION AND CONCLUSIONS **113**

6.1 **Summary** 113

6.2 **Future Work for Improved Prediction Accuracy** 117

6.3 **Future Work in Computer Science** 119

6.4 **Conclusions** 120

APPENDICES **123**

A CLASS DEFINITIONS **123**

A.1 **CBaseObj** 123

A.2 **CListNode** 123

A.3 **CList** 124

A.4 **CSynapse** 124

A.5 **CHardSynapse** 126

A.6 **CBias** 126

A.7 **CNode** 127

A.8 **COpNode** 129

A.9 **CAddNode** 130

A.10 **CCombineNode** 130

A.11 **CAndNode** 131

A.12 **CNotNode** 132

A.13 **COrNode** 132

A.14 **CPattern** 133

A.15 **CPatFile** 134

A.16 **CNetwork** 135

A.17 **CUserNode** 136

B FILE FORMAT FOR NETWORK PERSISTENCE **138**

B.1 **CNode Objects** 138

B.1.1 **Regular Nodes** 138

B.1.2 **Operation Nodes** 138

B.1.3 **User Nodes** 139

B.2 **CSynapse Objects** 140

REFERENCES **141**

BIOGRAPHICAL SKETCH **147**

List of Tables

2.1 Twenty Amino Acids 11

2.2 Conformational Parameters for Helical and Strand Residues . . 13

2.3 Chou-Fasman Conformational Parameters 18

List of Figures

3.1 Transforming a Rule Base into an Expert Network 25
3.2 Expert Network Class Relationship Diagram 27
3.3 Algorithm for Topological Sort of Expert Network Hidden Nodes 34
3.4 Illustration of the Topological Sort Algorithm 35
3.5 Hierarchical Symbolic/Sub-symbolic Architecture 39
4.1 Elman Network 44
4.2 Jordan Network 46
4.3 Recurrent Architectures for Protein Secondary Structure Prediction 48
4.4 Learning Algorithm for Fully Recurrent Networks 57
4.5 Original Helix Finding Expert Network 64
4.6 Helix Finding Expert Network with No Intermediate Concepts . 66
4.7 Helix Finding Expert Network 67
5.1 Preprocessing of Amino Acid Sequence into Input Patterns . . . 73
5.2 Separate Helix and Strand Finding Expert Network Results . . 76
5.3 Strand Finding Expert Network 79
5.4 Overall Scheme for Prediction Expert Network 80
5.5 Protein Folding Expert Network Results 81
5.6 Simple Combine Algorithm 82
5.7 Network Using User Nodes and Combine Nodes 83
5.8 Combination Nodes Inputs and Outputs 84

5.9 Network Using User Nodes and No Combine Nodes 86
5.10 Network Using No User Nodes and No Combine Nodes 87
5.11 Prediction Network Incorporating State Information 90
5.12 Results of Incorporating State Information 92
5.13 Third Prediction Network Incorporating State Information . . . 95
5.14 Fourth Prediction Network Incorporating State Information . . 97
5.15 Fifth Prediction Network Incorporating State Information . . . 98
5.16 Sixth Prediction Network Incorporating State Information . . . 101
5.17 Output Values for Individual Amino Acids 102
5.18 Results on FKBP-type Proteins 107
5.19 Predictions on FKBP Region of Interest 108

ABSTRACT

Hybrid systems which combine rule-based knowledge about a problem domain with data-driven knowledge discovery methods have been implemented in a number of ways, including *expert networks*. An expert network is a computational network whose topology is derived from an underlying knowledge base. Each of the nodes in the network has a unique functionality according to the role the node plays in encoding the knowledge base. Such systems allow human expert knowledge to be explicitly encoded in the system and then refined by automated examination of data. The topologies and learning algorithms developed for expert networks prior to the work described in this dissertation have been strictly feedforward.

Some classification tasks, however, are difficult to complete using a strictly feedforward architecture. In particular, the solution to many problems requires that *state* information be maintained. State information is the context in which the problem is currently being solved. The context of a problem solution will change as the solution proceeds and certain sets of rules need only be considered in certain contexts. In order to maintain such state information, I have added recurrent connections to the topologies and modified the learning algorithms allowed for expert networks.

A particular problem domain for which state information is important is the problem of predicting the secondary structure of a protein from its primary

sequence. Algorithms developed for prediction of protein secondary structure do not take into account information concerning interactions between amino acids which occur far apart along the protein primary sequence although many studies have shown that such interactions play a significant role in the formation of actual protein structure. Information concerning these long-range interactions constitutes the state information necessary to adequately address this prediction problem.

I have encoded the rules from the Chou-Fasman protein secondary structure prediction algorithm into an expert network and augmented this expert network with knowledge about longer-range amino acid interactions. Results of protein secondary structure prediction experiments show that the addition of state knowledge to existing algorithms encoded as expert networks aids in the development of better prediction tools.

CHAPTER 1
INTRODUCTION

1.1 Motivation

An expert network is an intelligent hybrid system technique, combining data-driven learning methods with explicit encoding of expert knowledge [1]. The earliest definitions of these networks included a criterion of acyclicity, requiring that the architectures have only feedforward architectures [2]. Some classification tasks, however, are difficult to complete using a strictly feedforward architecture. In particular, the solution to many problems requires that *state* information be maintained. State information is the context in which the problem is currently being solved. The context can change as the solution proceeds. Certain sets of rules need only be considered in certain contexts. For example, if a person is faced with the problem of reading email, the state must include information about whether the computer is on or off. The rules to turn on the computer need only be considered if the state tells us that the computer is off. The primary contribution of the work described in this dissertation is the addition of recurrent connections in expert networks so that these knowledge based networks can have access to state information during the course of a problem solution.

Research has shown that computational techniques such as neural networks often provide classification abilities that are more accurate than methods which rely on explicit knowledge acquisition alone [3]. On the other hand, because no explanation for a particular classification can be given when a computational technique has been used, human experts tend to be skeptical of such systems [4]. As a result, many researchers have developed tools, called *hybrid systems*, which combine the pattern recognition capabilities and parallel processing of neural systems while retaining the explicit domain knowledge encoded in expert systems [4]. Such hybrid systems illustrate an understanding that neural networks and expert systems are simply two ways of representing and manipulating knowledge and that making use of these alternative representations in a cooperative rather than competitive manner is likely to be useful in finding solutions to real-world problems [5].

Because the widely known *knowledge acquisition bottleneck* [6] makes explicit knowledge acquisition tools (such as expert systems) expensive to create, Kuncicky, Hruska and Lacher [1] have developed *expert networks*, which eliminate the need for the expert to associate a certainty factor with each rule in the expert knowledge base. Instead, the expert system rules, acquired from the human expert, are translated into the topology of an acyclic computational network, called an expert network. The individual nodes in an expert network are not all identical, as they would be in a typical neural network, but instead have functionalities that match the part of the knowledge base which they encode. For example, a node which encodes an AND between two pieces of knowledge in the knowledge base might take the minimum of its inputs and if that minimum is above a certain threshold, outputs that minimum [7]. The certainty factors on the original rules correspond to the trainable weights between the nodes. Example data are presented to the network and the weights, that is, the certainty factors, are learned via a backpropagation-like algorithm [8]. The topologies and learning algorithms developed for expert networks prior to the work described in this dissertation have been strictly feedforward.

As stated earlier however, many problems exist that are difficult to solve using feedforward architectures because of the need for the context or state of the problem solution to be maintained. Another, more complex example of a problem in which state information is likely to be important is the prediction of weather. Although many of the factors concerning weather are well-understood, the interactions among those factors is not well-understood by human beings. However, it is well-understood that yesterday's weather somehow affects today's weather which in turn affects tomorrow's weather. The lining up of such effects means that yesterday's weather also affects tomorrow's weather. When predicting the weather for tomorrow, today's weather and yesterday's weather and the weather of the day before in addition to the conditions found in the atmosphere today represent the context in which the problem must be solved. It is not clear how many days one must go back to capture all relevant interactions for prediction of tomorrow's weather. Capturing these interactions would be important in the expert network solution to accurate weather prediction.

Such state information is difficult to manage using feedforward architectures. In fact, in standard artificial neural networks, such state information is handled by the addition of recurrent connections in the topology of the network [9]. The recurrent connections allow the context information to be input to the network at succeeding steps. Therefore, the work described in this dissertation extends the notion of expert networks so that they are able to maintain state information via recurrent connections while at the same time encoding previously discovered expert knowledge.

The work is fielded in the area of protein secondary structure prediction, although it is not a goal of the work to solve this prediction problem. Rather, this dissertation describes an investigation of methods for, as well as the effect of, adding state information to an existing prediction algorithm.

1.2 Feedforward Neural Networks and Background Knowledge

Feedforward neural networks using backpropagation learning [10] have been used successfully in a number of applications. For example, Sejnowski and Rosenberg [11] created a neural network that learned to pronounce English text, Pomerleau [12] created a neural network that learned to keep a car on a road, and Qian and Sejnowski [13] created a neural network that learned to predict protein secondary structure with some accuracy. Such neural networks, however, are typically unable to take advantage of existing problem-specific symbolic knowledge. In other words, such networks may learn rules from the data presented to them but are unable to directly encode previously discovered knowledge about the problem domain. In fact, many of the developers of artificial neural network tools that solve particular problems have stated that the fact that neural networks discover their own rules for associating input patterns with output patterns is an advantage of the tools [13]. However, researchers understand that to solve many of the tough problems that remain in artificial intelligence and to create more powerful and effective systems, approaches which combine several techniques, such as the data-driven refinement of existing expert knowledge found in expert networks, are necessary [14].

To allow feedforward neural networks to use previously discovered knowledge, several researchers have created systems which combine neural networks with expert systems. These hybrid systems use the strengths of multiple techniques to minimize the weaknesses of the individual techniques. For example, Kuncicky, Hruska and Lacher [1] developed expert networks, which are networks that derive their topology from the problem-specific knowledge base and use a variation of backpropagation [8] to learn the weights for the network.

One of the strengths of neural networks is the use of data in the development of an automated tool to solve a problem with little input from a human domain expert. A weakness of neural networks is that they then exist as a "black box." That is, determining how the network has come to its conclusions is not trivial. Expert systems, on the other hand, have an explanation facility which explains the conclusions of the system. Extracting knowledge from human experts in a form readily translated into an expert system is often difficult, however. By combining the two techniques into an expert network, some of the knowledge extraction problem of expert systems is ameliorated by the data-driven methods of neural networks while some of the black box criticism facing neural techniques can be answered by the expert system underlying the expert network. The weaknesses of the individual techniques are strengthened by the other techniques utilized in the hybrid system.

Expert networks, as developed by Kuncicky, Hruska and Lacher [1], are required to be *acyclic*, which disallows the use of recurrent connections in the architecture of the network. As described above, the solutions to some problems are best-suited to a recurrent network architecture. In order to allow expert network technology to be applied to a wider array of problems, this dissertation describes work that removes the acyclic condition from the definition of expert networks.

1.3 Recurrent Neural Networks and Background Knowledge

A number of researchers have proposed networks that maintain state information via recurrent connections in the network architecture but do not allow the encoding of previously discovered domain knowledge [15] [16] [17]. Several of these recurrent neural networks are described in later chapters of this dissertation. The idea of maintaining state information across training patterns occurs primarily in work addressing natural language problems [15] [18]. These network architectures provide a means for preserving past activations of the network for use in processing of later inputs to the network.

Maclin and Shavlik [19] describe a technique which encodes previously discovered expert knowledge as well as maintains state information via recurrent connections. Their technique, known as Finite State Knowledge-Based Artificial Neural Networks (FSKBANN), translates the rule-like knowledge in an automaton into a neural network and then refines the automaton using the backpropagation of error learning algorithm [10]. The states in the automaton correspond to the output layer nodes in the neural network while the transition arcs among the states in the automaton correspond to hidden layer nodes in the neural network. The input nodes in the neural network correspond to the input to the automaton. Once the architecture of the neural network is derived from the background knowledge, the network is made to be fully connected via the addition of synapses with low initial weights. These additional synapses allow the neural network to discover additional knowledge that was not encoded in the original automaton. On the other hand, the additional synapses disallow easy access to the set of rules used by the network during its processing. FSKBANN allows state information to be maintained during the course of the problem solution via a small, selective set of fixed recurrent connections from the output layer of the fully-connected neural network to a set of context units in the network [19]. Because the resulting network has a fully connected architecture, the rules of the original algorithm cannot be accessed after training.

In contrast to FSKBANN, the knowledge-based expert networks described in this dissertation are not given full connectivity and therefore, the original algorithm rules are accessible after training.

1.4 Protein Secondary Structure Prediction

The recurrent expert networks of this dissertation are applied to the specific domain area of the prediction of protein secondary structure from the primary sequence of the protein. Protein secondary structure prediction is the major outstanding problem in structural molecular biology [20]. Researchers have used many techniques to try to accurately predict secondary structure from the primary sequence of the protein. Some of these techniques are based on statistical analysis of proteins of known structure [21] [22]. Other techniques are based on knowledge of the stereochemistry of the amino acids in the primary sequence [23]. Still other researchers have used various artificial intelligence techniques to try to obtain accurate prediction results [24] [25] [26].

One of the artificial intelligence techniques commonly applied to protein secondary structure prediction is artificial neural networks. For example, Maclin and Shavlik [19] [27], Kneller, *et al.* [28], Qian and Sejnowski [13], and Holley and Karplus [29] have all attempted to encode protein sequences so that the neural network techniques employed can discover the function that maps the primary sequence to the secondary sequence. These neural networks perform about as well as other methods of secondary structure prediction. Most of these methods take only short range amino acid interactions into account although longer-range interactions are known to significantly influence the final structure of the protein [30] [31] [32]. These longer-range amino acid interactions represent the state of the problem in which the solution must proceed. The work in this dissertation explores methodologies for capturing some of this state information. The results of using the state information to augment an existing prediction algorithm as encoded in an expert network are also discussed.

1.5 Dissertation Outline

In Chapter 2 of this dissertation, the problem domain in which the recurrent expert network methods are tested and the Chou-Fasman algorithm for secondary structure prediction are described. In Chapter 3, the general tool of expert networks is described, along with a discussion of the particular, object-oriented implementation of expert networks used in the work in this dissertation. Some of the key ideas and heuristics needed to understand the recurrent expert network methodology are then discussed in Chapter 4. Next, in Chapter 5, the experimental methods employed for the implementation of recurrent expert networks for protein secondary structure prediction are discussed, followed by descriptions and analyses of the variety of experiments for implementing the Chou-Fasman algorithm, refining it via training, and augmenting it with state information. Finally, Chapter 6 contains conclusions about the work described in the dissertation along with suggestions for future work in the protein secondary structure prediction realm as well as in computer science.

CHAPTER 2
ANNOTATED EXAMPLE

2.1 Description of Protein Secondary Structure Prediction Example

Many of the functions of life are the result of complex chemical interactions carried out by biomolecules such as proteins. These proteins play a wide variety of roles in any living being. Despite their wide array of functions, all proteins are made from the same basic components: amino acids. The twenty most commonly occurring amino acids are listed in Table 2.1 [20].

In a protein, the amino acids are strung together, bead-like, in a sequence that is unique to that protein. The sequence of amino acids that makes up the protein is called the *primary sequence* of the protein. The role that the protein plays in the living cell is determined by its *tertiary structure*, that is, the linear sequence of amino acids folded into a three-dimensional entity. The folding of a protein is called its *conformation.* The *secondary structure* of a protein refers to arrangements of relatively short amino acid subsequences into intermediate-level conformations that appear over and over in many folded proteins. Secondary structure conformations include α-helices, which are spirals of amino acids in three dimensions, and β-strands, which bind to each other to form sheets of amino acids [33].

To determine the function of a protein, its tertiary structure must be found. In 1961, Anfinsen and his colleagues demonstrated that the conformation (and

Table 2.1: Twenty Amino Acids

Amino Acid Name	Three Letter Abbreviation	One Letter Abbreviation
Alanine	Ala	A
Arginine	Arg	R
Asparagine	Asn	N
Aspartate	Asp	D
Cysteine	Cys	C
Glutamine	Gln	Q
Glutamate	Glu	E
Glycine	Gly	G
Histidine	His	H
Isoleucine	Ile	I
Leucine	Leu	L
Lysine	Lys	K
Methionine	Met	M
Phenylalanine	Phe	F
Proline	Pro	P
Serine	Ser	S
Threonine	Thr	T
Tryptophan	Trp	W
Tyrosine	Tyr	Y
Valine	Val	V

therefore, the function) of a protein is determined by its primary sequence [34]. Many researchers have approached the problem of predicting the tertiary structure of a protein as a problem of first predicting the secondary structure of the protein and then assembling the secondary structure elements into a tertiary structure [35]. Thus, many have viewed the prediction of the secondary structure of a protein as a necessary step toward prediction of its tertiary structure.

The problem domain in which the recurrent expert network research of this dissertation has been fielded is that of the prediction of the secondary structure of a protein based solely on its primary sequence.

2.2 The Chou-Fasman Algorithm

The Chou-Fasman algorithm [36] is a widely used algorithm for protein secondary structure prediction. Its prediction accuracy, however, is estimated to be between 50% [20] and 58% [37].

In the Chou-Fasman approach to protein secondary structure prediction, two conformational parameters, a helix propensity value, P_α, and a strand propensity value, P_β, are calculated for each amino acid from the relative frequency of the amino acid in helices and strands [38]. From the helix propensity value, each amino acid is classified as a strong helix former (H_α), a helix former (h_α), a weak helix former (I_α), an indifferent helix former (i_α), a weak helix breaker (b_α) or a strong helix breaker (B_α). Similarly, from the strand propensity value, each amino acid is classified as a strong strand former (H_β), a strand former (h_β), a weak strand former (I_β), an indifferent strand former (i_β), a weak strand breaker (b_β) or a strong strand breaker (B_β). Table 2.2 shows the Chou-Fasman parameters calculated from 29 proteins with known tertiary structure [38].

Table 2.2: Conformational Parameters for Helical and Strand Residues

P_α			P_α			P_β			P_β		
Glu	1.51	H_α	*Arg*	0.98	i_α	*Val*	1.70	H_β	*Arg*	0.93	i_β
Met	1.45	H_α	*Thr*	0.83	i_α	*Ile*	1.60	H_β	*Asn*	0.89	i_β
Ala	1.42	H_α	*Ser*	0.77	i_α	*Tyr*	1.47	H_β	*His*	0.87	i_β
Leu	1.21	H_α	*Cys*	0.70	i_α	*Phe*	1.38	h_β	*Ala*	0.83	i_β
Lys	1.16	h_α				*Trp*	1.37	h_β			
Phe	1.13	h_α				*Leu*	1.30	h_β	*Ser*	0.75	b_β
Gln	1.11	h_α	*Tyr*	0.69	b_α	*Cys*	1.19	h_β	*Gly*	0.75	b_β
Trp	1.08	h_α	*Asn*	0.67	b_α	*Thr*	1.19	h_β	*Lys*	0.74	b_β
Ile	1.08	h_α				*Gln*	1.10	h_β			
Val	1.06	h_α				*Met*	1.05	h_β	*Pro*	0.55	B_β
Asp	1.01	I_α	*Pro*	0.57	B_α				*Asp*	0.54	B_β
His	1.00	I_α	*Gly*	0.57	B_α				*Glu*	0.37	B_β

2.2.1 The Chou-Fasman Rules

The Chou-Fasman algorithm uses the conformational parameters shown in Table 2.2 in a simple set of rules to predict the secondary structure designation of a region in a primary sequence. The rules are described below [38]:

1. Four helical residues (H_α or h_α) out of six will initiate a helix, with I_α residues counting as half of a helical residue. The helical segment will be extended in both directions along the protein sequence until sets of five residues whose average helix conformational parameter, $\langle P_\alpha \rangle$, is less than 1.00 are reached, where $\langle P_\alpha \rangle$ is the average helix propensity value for the region. Proline cannot occur in the inner helix or at the C-terminal end of the helix but can occur within the last three residues at the N-terminal end. The inner helix contains all residues in the helix except the three at each end. Any segment that is at least six residues long with $\langle P_\alpha \rangle > 1.03$ and $\langle P_\alpha \rangle > \langle P_\beta \rangle$ is predicted as helical. $\langle P_\beta \rangle$ is the average strand propensity value for the region.

2. A cluster of three strand formers (H_β or h_β) out of five residues along the protein sequence will initiate a strand. The strand is extended in both directions until sets of five residues whose average strand conformational parameter, $\langle P_\beta \rangle$ is less than 1.00. Any segment with $\langle P_\beta \rangle > 1.05$ and $\langle P_\beta \rangle > \langle P_\alpha \rangle$ is predicted as strand.

3. Any segment containing overlapping helix and strand regions is helical if $\langle P_\alpha \rangle > \langle P_\beta \rangle$ and strand if $\langle P_\beta \rangle > \langle P_\alpha \rangle$.

4. Any segment not predicted as helix or strand will be predicted as coil.

2.2.2 Difficulties with the Chou-Fasman Algorithm

One of the main criticisms of the Chou-Fasman algorithm is that it is extremely difficult to computerize. In fact, it has been pointed out that the original predictions for 19 proteins used by Chou and Fasman [36] can never be reproduced by a computer [37] because Chou and Fasman appear to have used information about known structure in making their structure predictions [37].

A second difficulty is that although the Chou-Fasman algorithm is an algorithm in the sense that it is a complete, easy-to-follow set of instructions [39], the algorithm is recursive in both directions along the protein primary sequence. What this means is that to predict the secondary structure at amino acid i, one must first predict the structure at amino acid $i-1$ while moving in one direction along the primary sequence and at amino acid $i+1$ while moving in the other direction. Prediction of the structure for amino acid $i-1$ requires prediction of the structure for amino acid $i-2$ while moving in one direction and amino acid i while moving in the other direction. In other words, one must have all prediction information available before starting the prediction. Humans are able to move back and forth along the protein sequence and deal fairly easily with incomplete information which changes over time. Humans can make several prediction passes at the protein, revising each successive prediction based on whatever new information is available. When attempting to predict the structure of a protein, however, one does not know in advance how long the protein might be, with some proteins being short (tens of amino acids) and others being very long (thousands of amino acids). A computer is not easily able to maintain an unknown amount of information while the problem solution proceeds.

For these reasons, the algorithm is relatively easy for a human to use but very difficult to implement on a computer.

Several researchers have made modifications to the Chou-Fasman rules to allow their use by a computer [37] [40] [41]. The modifications that I use to computerize the algorithm in part follow those of Maclin and Shavlik [19]. The set of rules developed by Maclin and Shavlik are shown in Section 2.3. The conformational parameters used with these rules differ from those in the original Chou-Fasman algorithm in that the helix and strand former values have been normalized by dividing the conformation value of the given former by the conformation value of the weakest former. For example, since Glutamic acid has a helix conformational parameter of 1.51 [38] and the weakest helix former, Valine, has a helix conformational parameter of 1.06 [38], the helix former value for Glutamic acid is 1.42. Breaker values have been created in a similar manner except that the value used to calculate the breaker value is the multiplicative inverse of the conformation value. Thus, to determine the breaker value for a given amino acid, the multiplicative inverse of the original Chou-Fasman value for the amino acid is divided by the multiplicative inverse of the weakest breaker amino acid value [19]. For example, since Glycine has a helix conformational value of .57 and the weakest helix breaker, Tyrosine, has a helix conformational value of .69, the helix breaker value for Glycine is $\frac{1}{.57}$ divided by $\frac{1}{.69} = 1.21$.

The published Chou-Fasman propensity values [38] are not used in conjunction with the Maclin and Shavlik rules for two reasons. First, smaller helix propensity values are desired to decrease the number of times that three strong helix-formers add up to more than 4 (and similarly for strands). Secondly, in the original Chou-Fasman parameters, the breaker values were between 0 and 1 with stronger breakers having values closer to 0. In the Maclin and Shavlik rules, it is desirable that stronger breakers have larger breaker values.

Table 2.3 shows the conformational parameters that I calculated with the method of Maclin and Shavlik and which are used in my work. These numbers are not identical to Maclin and Shavlik's because these values are based on those found in Table 2.2 [38] while theirs are based on an older set of propensity values [36].

Many factors make the prediction of secondary structure in proteins a difficult task. First, the notion of *secondary structure* is imprecise since researchers, given the same data, may arrive at different conclusions about the secondary structure of a protein. The assignment of secondary structure to a protein requires some amount of judgement on the part of the human researcher and thus one or more residues may be added or deleted at the ends of helices and strands [32]. Second, even if secondary structure is assigned in an automated manner (which is not how the structure for most proteins in the Protein Data Bank was assigned), for example, based on the detection of periodic repeats of hydrogen bonded structure [42], the sharp cutoffs in hydrogen energy that occur at the ends of the secondary structural elements mean that small variations in three dimensional coordinates can lead to variations in the numbers of residues assigned to the helix or strand structure [32]. Because of these many difficulties, 100% prediction accuracy is unlikely to be achieved.

Long-range Amino Acid Interactions. Many authors (including Chou and Fasman) have pointed out that the Chou-Fasman algorithm takes only *local*

Table 2.3: Chou-Fasman Conformational Parameters

Amino Acid	Helix Former Value	Helix Breaker Value	Strand Former Value	Strand Breaker Value
Ala	1.34	0.00	0.00	0.00
Arg	0.00	0.00	0.00	0.00
Asn	0.00	1.03	0.00	0.00
Asp	0.50	0.00	0.00	1.39
Cys	0.00	0.00	1.13	0.00
Gln	1.05	0.00	1.05	0.00
Glu	1.42	0.00	0.00	2.02
Gly	0.00	1.21	0.00	1.00
His	0.50	0.00	0.00	0.00
Ile	1.02	0.00	1.52	0.00
Leu	1.14	0.00	1.24	0.00
Lys	1.09	0.00	0.00	1.01
Met	1.37	0.00	1.00	0.00
Phe	1.07	0.00	1.31	0.00
Pro	0.00	1.21	0.00	1.36
Ser	0.00	0.00	0.00	1.00
Thr	0.00	0.00	1.13	0.00
Trp	1.02	0.00	1.31	0.00
Tyr	0.00	1.00	1.40	0.00
Val	1.00	0.00	1.62	0.00

amino acid interactions into account. Local interactions are those interactions which occur among amino acids that are found close together in the primary sequence of the protein. The role of long-range amino acid interactions in the final conformation of the protein is known to be significant, especially for large proteins [31]. The formation of secondary structure in proteins can be thought of as $C = C_{local} + C_{global}$, where C is the complete secondary structure conformation of the protein, C_{local} is the secondary structure that is determined by local amino acid interactions, C_{global} is the secondary structure that is determined by long-range amino acid interactions and $+$ is a rough, first order approximation of the relationship between C_{local} and C_{global} [32]. For example, Logan and his colleagues have discovered a region in the FK506 Binding Protein, FKBP, which is an α-helix in the unfolded conformation, is predicted α-helix by Chou-Fasman's algorithm, but exists as a β-strand in the folded conformation due to long-range amino acid interactions [43]. Several other authors have found regions in other proteins which exhibit similar characteristics [44] [45] [46] [47]. Such long-range interactions cause difficulties for most secondary structure prediction algorithms because most algorithms are able to take only shorter-range, local amino acid interactions into account when making predictions. In fact, some authors have suggested that the long-range interactions impose an upper limit on the predictive accuracy possible for algorithms which take only local interactions into account [30] [32].

Since most studies compare predicted secondary structures with secondary structures observed in the folded conformation of a protein, the Chou-Fasman algorithm is penalized for an incorrect prediction in the region of FKBP discovered by Logan [43]. In the system described in this dissertation, the Chou-Fasman algorithm is augmented with knowledge about longer-range interactions between amino acids. The recurrent connections in the expert network encoding the Chou-Fasman rules maintain the knowledge about longer-range interactions. This knowledge represents the *state* of the problem solution as the solution proceeds. The state of the problem solution is necessary for a more accurate solution to the problem at hand.

2.3 Implementation of the Chou-Fasman Algorithm

The Chou-Fasman rules [19] as used in my system are shown in this section. These rules are based on the Chou-Fasman parameter values shown in Table 2.3 [38] and on the work by Maclin and Shavlik [19]. In the rules, **@0** means "at the current amino acid." For example, **if P@0** means "if the amino acid for which a prediction is currently being made is proline." Likewise, **@1** means "at the amino acid next to the current amino acid." For example, **if G@1** means "if the amino acid after the one for which a prediction is being made is glycine."

2.3.1 Rules for Finding Helices

1. **Initiating a Helix.**

   (a) $InitHelix \leftarrow \left(\sum_{position=0}^{5} HelixFormer(AminoAcid@position)\right) > 4$ and $\left(\sum_{position=0}^{5} HelixBreaker(AminoAcid@position)\right) < 2$

2. **Terminating a Helix.**

   (a) *HelixBreak@0* $\leftarrow$ N@0 or Y@0 or P@0 or G@0

(b) *HelixBreak@1* $\leftarrow$ N@1 or Y@1 or P@1 or G@1

(c) *HelixIndiff@1* $\leftarrow$ R@1 or T@1 or S@1 or C@1

(d) *BreakHelix* $\leftarrow$ *HelixBreak@0* and *HelixBreak@1*

(e) *BreakHelix* $\leftarrow$ *HelixBreak@0* and *HelixIndiff@1*

3. **Continuing a Helix.**

(a) *ContHelix* $\leftarrow \neg$ P@0 and $\neg$ *BreakHelix*

4. **Prediction of Helix by Initiation or Propagation From Last State.**

(a) $helix_i \leftarrow$ *InitHelix*

(b) $helix_i \leftarrow helix_{i-1}$ and *ContHelix*

2.3.2 Rules for Finding Strands

1. **Initiating a Strand.**

(a) *InitStrand* $\leftarrow \left(\sum_{position=0}^{4} StrandFormer(AminoAcid@position)\right) > 3$ and $\left(\sum_{position=0}^{4} StrandBreaker(AminoAcid@position)\right) < 2$

2. **Terminating a Strand.**

(a) *StrandBreak@0* $\leftarrow$ P@0 or D@0 or E@0 or S@0 or G@0 or K@0

(b) *StrandBreak@1* $\leftarrow$ P@1 or D@1 or E@1 or S@1 or G@1 or K@1

(c) *StrandIndiff@1* $\leftarrow$ R@1 or N@1 or H@1 or A@1

(d) *BreakStrand* $\leftarrow$ *StrandBreak@0* and *StrandBreak@1*

(e) *BreakStrand* $\leftarrow$ *StrandBreak@0* and *StrandIndiff@1*

3. **Continuing a Strand.**

   (a) *ContStrand* $\leftarrow \neg$ P@0 and $\neg$ E@0 and $\neg$ *BreakStrand*

4. **Prediction of Strand by Initiation or Propagation From Last State.**

   (a) $strand_i \leftarrow$ *InitStrand*

   (b) $strand_i \leftarrow strand_{i-1}$ and *ContStrand*

2.3.3 Rules for Prediction of Coil (The Default)

1. $coil_i \leftarrow helix_{i-1}$ and *BreakHelix*

2. $coil_i \leftarrow strand_{i-1}$ and *BreakStrand*

3. $coil_i \leftarrow coil_{i-1}$ and $\neg$ *InitHelix* and $\neg$ *InitStrand*

CHAPTER 3
EXPERT NETWORKS

3.1 Background

Expert networks have been described in detail by various researchers at Florida State University [2] [5] [48] [49]. Early exploration of expert networks showed how to create the acyclic topology of an expert network using the rules from a rule-based expert system [2] [5] and how to use a backpropagation-like learning algorithm to extract knowledge from data [48]. Later work removed the need for an existing rule base as the basis of the expert network topology [49] [50] [51]. The work described in this dissertation removes the acyclic constraint from the expert network topologies in an attempt to allow expert networks to maintain information about the state of the solution as that solution proceeds.

Although expert networks take inspiration for their learning algorithms from neural network learning algorithms, they differ from neural networks in a number of ways:

1. In an expert network, each neuron has a specific meaning, that is, represents a *concept* in the knowledge base. In a neural network, on the other hand, no neuron is endowed with specific meaning. Meaning is instead distributed among the connection strengths between the neurons.

2. Since each neuron in an expert network has a specific meaning, the functionality of each neuron is unique. For example, a neuron encoding an

AND relationship should have different functionality than a neuron encoding an OR relationship. In a neural network, however, the neurons are indistinguishable in their functionalities. Again, meaning arises from the particular pattern of connection strengths between the neurons.

3. In an expert network, sparse connectivity between neurons encodes specific relationships among the concepts. Therefore, the knowledge represented in the network is not distributed throughout the entire network. In contrast, neural networks are fully connected and any knowledge encoded in the network is distributed throughout the entire network.

Kuncicky [5] describes the process of transforming a rule-based expert system into an expert network so that exemplar data may be used to refine the *certainty factors* associated with each rule. A rule has the form IF *a* THEN *b (cf)*, where *a* and *b* are *assertions*, *a* is the *antecedent* of the rule, *b* is the *consequent* of the rule, and *cf* is the *certainty factor* of the rule. The certainty factor represents our a level of confidence in the rule.

The transformation from rules to the architecture of the expert network occurs in two steps [2]. In the first step, each assertion in the rule is represented by a node in the expert network. Since a rule is simply a relationship between two or more assertions, the implication of the rule is represented by the weighted connections between the nodes representing the assertions of the rule. The second step of the transformation concerns the assertions that contain logical operations, such as AND, OR, or NOT. In this step, the logical operation is separated from the elements that the operation connects. Figure 3.1 shows the transformation process.

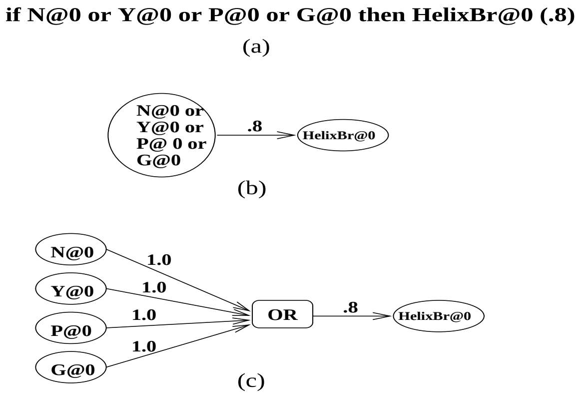

Figure 3.1: Transforming a Rule Base into an Expert Network

In the rule shown in Figure 3.1(a), there are two assertions. The first is `OR(N@0,Y@0,P@0,G@0)` and the second is `HelixBr@0`. The certainty factor on this rule is 0.8. This rule is transformed into a piece of an expert network, shown in Figure 3.1(b), by creating a node to represent `OR(N@0,Y@0,P@0,G@0)` and a node to represent `HelixBr@0`. These two nodes are connected with a synapse whose weight is the same as the certainty factor on the original rule, that is, 0.8. Figure 3.1(c) shows the second step of the transformation process. In the assertion, `OR(N@0,Y@0,P@0,G@0)`, several elements are bound together in an OR operation. In this step, an *operation* (OP) node is created to represent the logical operation, in this case, the OR. Then a node for each element in the OR operation is created. Finally, each element node is connected to the OP node with a synapse having a weight of 1.0. The synapses going into an OP node are *hard* synapses, which means that their values are not updated during training. The functionality of the various types of OP nodes is described in Section 3.2.3.

Recently, researchers at Florida State University have shown that one need not start with a rule base to build an expert network [50]. Instead, one only needs expert knowledge to build an expert network. The key point is that the architecture of the network is determined by the expert knowledge base that is to be encoded in the network.

Each synapse in an expert network has an input value, u, a weight value, w, and an output value, x, and the relationship among these data members is: $x = u \times w$. Each node in the expert network has a combining function, g, an internal state, y, an activation function, f, and an output value, z, such that $y = g(\mathbf{x})$, where $\mathbf{x} = (x_1, \ldots, x_n)$ with n being the number of input synapses for this node and $z = f(y)$. The input, u, for each output synapse of the current node takes the value of the node output, z. That is, $z_i = u_{ji}$, where z_i is the output of node i and u_{ji} is the input to the synapse connecting node i to node j.

3.2 Design of System for Building Expert Networks

The software developed for the design and implementation of the expert networks described in this dissertation is written in C++ and Tcl/TK [52]. I call my software the Expert Network Development Environment (ENDE). The neural objects and the design of the class hierarchy are discussed below. The class hierarchy is shown in Figure 3.2 and the class definitions are shown in Appendix A.

3.2.1 CBaseObj

Every class in the class hierarchy is a child of the `CBaseObj` class. This class contains a data member called *object id* and thus, the *object id* data member is

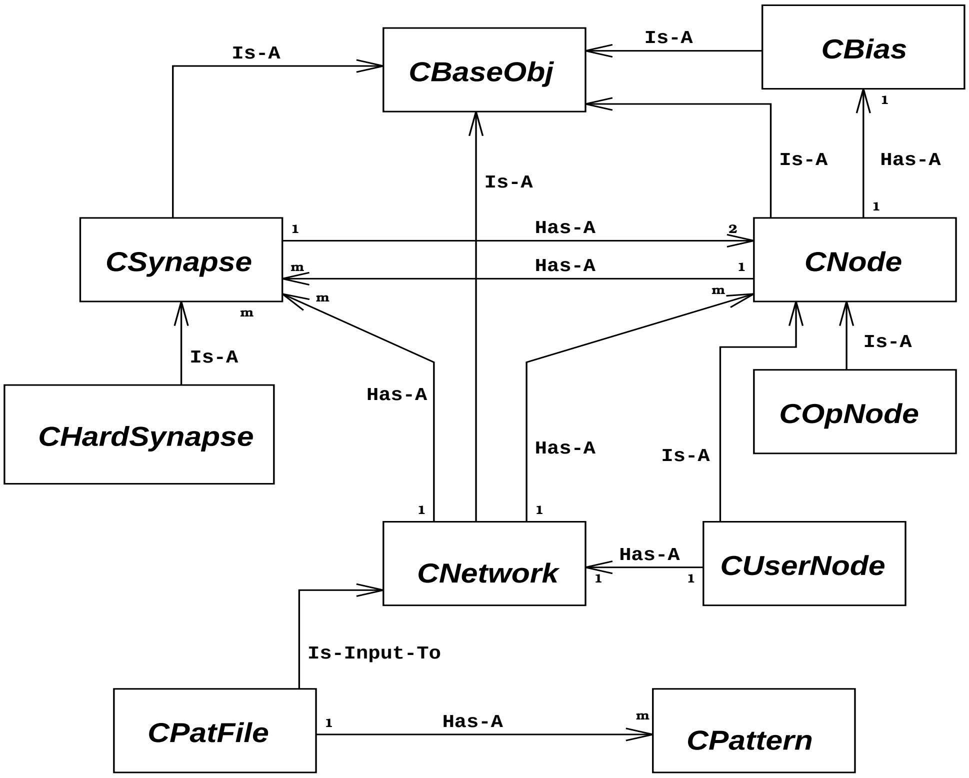

Figure 3.2: Expert Network Class Relationship Diagram

inherited by every instantiation of the child classes. The software assigns each object a unique object id which allows the network objects to be *persistent*, that is, to be saved between executions of the network. In object-oriented research, the issue of persistence, and, thus, of *object identity* has received much attention (*e.g.*, [53]). The technique for implementing persistence in my system is very simple. The first object to be instantiated receives an object id of 1 and each subsequent object is assigned the next number in sequence. The objects can then be stored in a file along with their ids and the network can be recreated at a later time. Appendix B shows the file format for a persistent network.

3.2.2 CNode

The `CNode` class is the building block for an instantiation of an expert network. This class contains all the information for an individual node, or *neuron*, in the network. Each `CNode` object contains information about whether that particular neuron is an input, output, or hidden layer neuron. The layer structure is not strict since, for example, an output layer node may input its value to other nodes in the network. The layer structure here is used only for convenience to distinguish the ways in which the various layers interact with the external world. The input layer nodes receive input from a file while output layer nodes have an ideal output value associated with them. The hidden layer nodes have no overall structure imposed upon them and have no direct interaction with the external world.

If the neuron is a hidden layer neuron, it may be either an operation (**OP**) node or a regular (**REG**) node [1]. A subclass of `CNode`, called the `COpNode` class, contains special properties that are associated with **OP** nodes and is described in Subsection 3.2.3.

In addition, a `CNode` object contains information about input and output connections, internal state, output value, current error, combining function and firing function. The combining function for `CNode` objects is

$$y = g(\mathbf{x}) = \sum_{i=1}^{N} x_i + \beta, \tag{3.1}$$

where $\mathbf{x} = (x_1, \ldots, x_N)$, N is the number of input synapses and β is the bias, which is discussed later in this chapter, for the `CNode` object. For ordinary neural networks, the firing function for `CNode` objects is typically

$$z = f(y) = tanh(y). \tag{3.2}$$

Although ENDE allows the above firing function to be used, the expert networks described in later chapters use a linear firing function. That is,

$$z = f(y) = y. \tag{3.3}$$

While the *tanh* firing function forces the outputs to be between -1 and 1, a linear firing function does not restrict the outputs of the `CNode`. Not limiting the output values of the `CNode` objects is important for properly encoding the Chou-Fasman algorithm because several different sets of consecutive amino acids may result in similar output values for a particular `CNode` object and the small differences in the output values appear to be important for the network to be able to distinguish between the sets. Squashing the output of the node via the firing function obscures such small differences.

3.2.3 COpNode

The `COpNode` class is a subclass of the `CNode` class and contains information about the operation (**OP**) nodes. The possible types of **OP** nodes are `CAndNode`, `COrNode`, `CNotNode`, `CAddNode` and `CCombineNode`. Each of these **OP** node types is implemented as a separate subclass of `COpNode` and has unique functionality.

CAndNode. An object of type `CAndNode` has a combining function of the form

$$y = g(\mathbf{x}) = MIN_{i=1}^{N}(x_i), \tag{3.4}$$

where $\mathbf{x} = (x_1, \ldots, x_N)$ and N is the number of input synapses for the `CAndNode` object. The firing function for `CAndNode` objects is Equation 3.3.

COrNode. An object of type `COrNode` has a combining function

$$y = g(\mathbf{x}) = MAX_{i=1}^{N}(x_i), \tag{3.5}$$

where $\mathbf{x} = (x_1, \ldots, x_N)$ and N is the number of input synapses for the `COrNode` object. The firing function for `COrNode` objects is given by Equation 3.3.

CNotNode. The combining function of a `CNotNode` object is given by Equation 3.1 and the firing function for objects of type `CNotNode` is

$$z = f(y) = 1 - y. \tag{3.6}$$

CCombineNode. The combining function of a `CCombineNode` object is

$$y = g(\mathbf{x}) = 1 - \prod_{i=1}^{N}(1 - x_i) \tag{3.7}$$

where $\mathbf{x} = (x_1, \ldots, x_N)$ and N is the number of input synapses for the `CCombineNode` object. The firing function of `CCombineNode` objects is given by Equation 3.3.

CAddNode. Objects of type `CAddNode` have two data members not seen in other `COpNode` objects. The first data member is the *comparison type*, CT, which states whether the comparison to be performed is to be a less than (lt) comparison or a greater than (gt) comparison. The second data member is a *threshold value*, θ, which is used in the firing function as the value which the internal state must be less than or greater than. The combining function of a `CAddNode` object is given by Equation 3.1. The firing function of a `CAddNode` object is

$$z = f(y) = \begin{cases} 1 & (\text{if CT} = \text{`lt'} \text{ and } y < \theta) \text{ or} \\ & (\text{if CT} = \text{`gt'} \text{ and } y > \theta) \\ 0 & \text{otherwise,} \end{cases} \tag{3.8}$$

where CT is the comparison type and θ is the threshold value.

3.2.4 CBias

In addition to its incoming synapses, each neuron in a neural network typically also receives input from a *bias* node. The bias or *threshold* for a neuron represents the value above which a neuron must activate to actually fire. The bias is represented in my system by a class called `CBias`. Many neural network software packages create, for each neuron in the network, an additional input node to represent the bias.

Unlike other nodes, the node representing the bias has its input value clamped to -1. Other nodes receive a different input value for each pattern presented to the network. The inputs into a node, including the bias, are typically treated all the same in other neural network software packages. In the design of the software described here, I chose to separate the `CBias` class from other classes because of the unique clamping behavior in its implementation.

3.2.5 CSynapse

For many of the same reasons described for the `CBias` class, I designed my software so that the `CSynapse` class is a separate class. In most object-oriented implementations of neural networks, the synapse is part of some other class, perhaps a layer class [54] or a neuron class [54]. In a neural network, all synapses have the same behavior, that is, multiply the output of the neuron at one end of the synapse by the weight on the synapse and input this multiplied value into the neuron at the other end of the synapse. In an expert network, on the other hand, individual synapses may have unique behavior, depending on the role that synapse plays in the overall network. For example, the synapses that send input into **OP** nodes are *hard* synapses, which have weights that do not change. The basic `CSynapse` objects implement the *soft* synapses which behave as synapses in ordinary neural networks do. During the activation of the network, a `CSynapse` object connecting node i with node j transmits a value from node i to node j according to

$$x_j = u_i * w_{ji}, \tag{3.9}$$

where x_j is the input to node j, u_i is the output of node i, and w_{ji} is the weight value of the `CSynapse` object connecting node i to node j.

When ΔW averaging [49], a technique described in Chapter 4, is turned off, the weight value on a `CSynapse` object is updated according to

$$w_{ji}^{new} = w_{ji}^{old} + \sum_{k=1}^{P} \Delta w_{ji}^{k}, \tag{3.10}$$

where w_{ji}^{new} is the new weight value on the `CSynapse` object, w_{ji}^{old} is the old weight value on the `CSynapse` object, P is the number of training patterns, and Δw_{ji}^{k} is the weight change for the `CSynapse` object for pattern k.

When ΔW averaging is turned on, the weight changes are accumulated by example type. That is,

$$w_{ji}^{new} = w_{ji}^{old} + \Delta W_{ji}^{positive} + \Delta W_{ji}^{negative}, \tag{3.11}$$

where w_{ji}^{new} is the new weight value on the `CSynapse` object, w_{ji}^{old} is the old weight value on the `CSynapse` object,

$$\Delta W_{ji}^{positive} = \frac{\sum_{k \in PositiveExamples} \Delta w_{ji}^{k}}{NumberOfPositiveExamples}, \tag{3.12}$$

and

$$\Delta W_{ji}^{negative} = \frac{\sum_{k \in NegativeExamples} \Delta w_{ji}^{k}}{NumberOfNegativeExamples}. \tag{3.13}$$

The learning algorithm for updating these weights is discussed in more detail in Chapter 4.

CHardSynapse. The `CHardSynapse` class is a child class of `CSynapse`. The functionality of the `CHardSynapse` class differs from that of the `CSynapse` class in that the weight of the `CHardSynapse` object is never updated. In other words, the weight on a `CHardSynapse` object cannot be trained. These weights typically are set to 1.0. A `CHardSynapse` object connecting node i to node j transmits values from node i to node j according to Equation 3.9.

3.2.6 CNetwork

The `CNetwork` class instantiates the expert network. A `CNetwork` object contains data members of type `CNode` and `CSynapse`, as well as member functions

```
Let L be an empty list of nodes
Let M be an adjacency matrix representing the network
while there are more rows in M
   find the first row i with all zeroes
   if number of rows > 0 but none have all zeroes, network
       has cycles in middle layer and an error occurs
   otherwise
      add node represented by row i as first item in L
      delete row i from M
      delete column i from M
end while
return L
```

Figure 3.3: Algorithm for Topological Sort of Expert Network Hidden Nodes

for activation and training of the network. Node objects are placed in either the input layer, which takes input values from a pattern file, the output layer, which computes error based on ideal values taken from the pattern file, or the hidden layer, which contains all nodes that are not in the input or output layer. Although no true layer structure exists, the layer metaphor is used strictly for convenience in distinguishing various node behaviors.

To activate the nodes in a `CNetwork` object in an efficient manner, the nodes in the hidden layer of the network are maintained in topologically sorted order [55]. The nodes are then activated in this order which ensures that at the time of its activation, an individual node has received input from all of its input nodes. To maintain the sorted order, each time a synapse is added to the network, the hidden nodes of the network are resorted. The algorithm for sorting the hidden nodes is shown in Figure 3.3. List L contains the nodes in topologically sorted order.

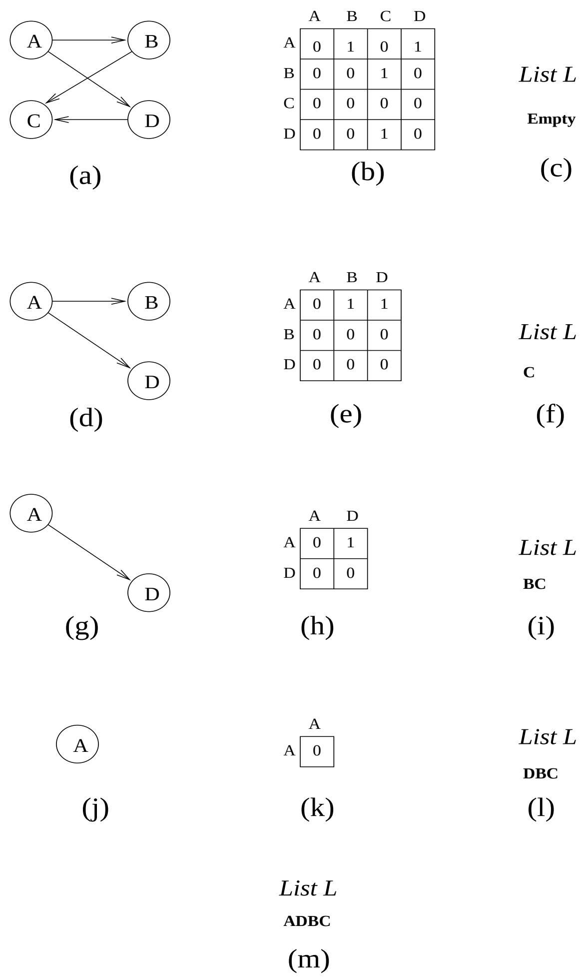

Figure 3.4: Illustration of the Topological Sort Algorithm

Figure 3.4(a) shows a network for which a topological sort is desired. Figure 3.4(b) shows the adjacency matrix for this network, with a 1 in row i, column j representing an edge from node i to node j. A zero means that no such edge exists in the network. Figure 3.4(c) shows that the topologically sorted list of nodes, L, starts with nothing in it.

To sort the nodes in this network, look for the first row that contains all zeroes and find that row 3, representing node C, contains all zeroes so remove node C from the network, resulting in the network shown in Figure 3.4(d) and remove row 3 and column 3 from the adjacency matrix, resulting in Figure 3.4(e). Put node C at the beginning of list L, which now looks like Figure 3.4(f). Repeat the process on the adjacency matrix shown in Figure 3.4(e) and find row 2 representing node B. Remove B from the network resulting in Figure 3.4(g) and remove row 2 and column 2 from the adjacency matrix resulting in Figure 3.4(h). Put B at the beginning of list L, which now looks like Figure 3.4(i). Repeat the process on Figure 3.4(g) and find row 2 representing node D. Remove D from the network, resulting in Figure 3.4(j), and row 2 and column 2 from the adjacency matrix, resulting in Figure 3.4(k). Add D to the list L which now looks like Figure 3.4(l). Repeat the process to find node A. Remove the representations of node A from both the network and the adjacency matrix and find that there are no more nodes in the network. Add A to the beginning of list L, which now looks like Figure 3.4(m) and contains all the nodes of the network in topologically sorted order. The nodes of this network can be activated in this order to ensure that at the time of activation, each node has all its needed inputs. For example, one would not want to activate node C before either node

D or node B since node C takes the output of the other two nodes as input.

Updating the weights during learning also proceeds according to the topological order of the hidden layer. The algorithm for updating the weights on the `CSynapse` objects is discussed in Chapter 4.

3.2.7 CPattern

The `CPattern` class contains the input and output elements for an individual pattern to be input to the `CNetwork` class. The input elements are presented to the input layer nodes in the `CNetwork` class while the output elements represent the ideal output for the pattern for determination of error at the output layer nodes.

3.2.8 CPatFile

The `CPatFile` class contains a number of `CPattern` objects so that all the patterns can be passed to the `CNetwork` object at the same time. The `CNetwork` object then processes the `CPattern` objects in the order that they were created.

3.2.9 CUserNode

The `CUserNode` class is a subclass of `CNode`. This class allows the user to build the expert network in stages, training a piece of the network, fixing those weights, connecting the subnetwork to a larger piece of the network, training the larger piece of the network, and so on. Objects of type `CUserNode` contain a data member of type `CNetwork` and a data member of type `CPatFile`. The `CPatFile` object contains the input pattern file for the `CNetwork` object. The `CNetwork` object represents a subnetwork of the larger expert network. This subnetwork has fixed weights so no further training can occur. When the larger network is activated, the subnetwork contained in the `CUserNode` object receives input from the `CPatFile` data member, the subnetwork `CNetwork` object is activated, and an output value for the `CUserNode` object results. This output value is transmitted along the output synapses of the `CUserNode` object.

3.3 Summary

In this chapter, the basic expert network methodology has been described. In addition, the design of a set of expert network classes has been discussed. Although early work on expert networks contains some theoretical discussion of object-oriented implementations of these networks [2] [5], the actual implementations of early expert networks are not object-oriented [48] [50]. The work described in this chapter represents one manner in which expert networks can be implemented using the object-oriented programming paradigm.

The design of the expert network class hierarchy described here contains several features not usually found in the definitions of classes designed for use with ordinary artificial neural networks. In particular, the bias input and the synapse have both been encapsulated into individual classes for the implementation of expert networks. This feature arises from the fact that expert networks function in a manner that is different from typical neural networks. For example, in a typical neural network, every synapse has the same functionality. In an expert network, there are at least two types of functionality for the synapses. Therefore, because each synapse has potentially unique behavior, the creation of a separate synapse class is desirable.

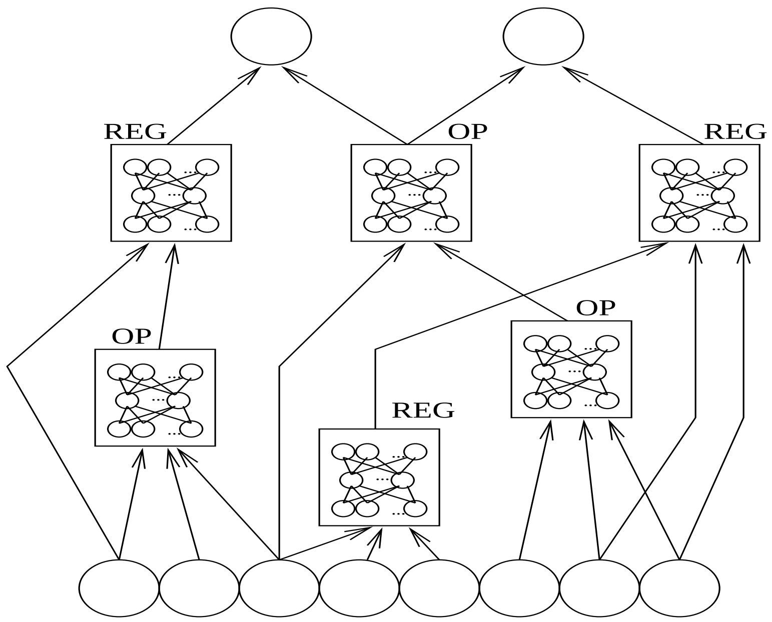

Figure 3.5: Hierarchical Symbolic/Sub-symbolic Architecture

An additional unique feature of the class design described in this chapter is the creation of a user node class. The user node allows the user to build expert networks in stages. The user can thus train a subnetwork, fix the weights on the subnetwork, encapsulate the subnetwork into a user node and finally, place the user node into a larger network. Such a process allows the user to express an expert network as a neural network with a *hierarchical symbolic/sub-symbolic architectural organization* [7]. Figure 3.5 is a schematic view of such a hierarchical implementation. The entire network in Figure 3.5 is an expert network because each of the nodes has *symbolic* meaning outside of the network. Lacher and Nguyen [7] have shown that each of the meaningful neurons in the expert network can be implemented as an artificial neural network, which performs *sub-symbolic* processing because the neurons have meaning only when taken as a group. In their work, each of the neural networks comprising the expert network has been trained to perform the input/output function needed in that place in the expert network. The user node class allows the development of expert networks as a set of sparsely connected user nodes, each of which contains a fully-connected artificial neural network trained to perform its required function.

CHAPTER 4
RECURRENT EXPERT NETWORK METHODOLOGY

4.1 Recurrence and State Information

Feedforward artificial neural networks have been applied to a number of problems [11] [12] [13]. However, researchers have long recognized that certain types of problems cannot be addressed adequately with feedforward architectures [9]. In particular, problems in which elements of the input patterns interact with each other over variable distances are difficult, if not impossible, to solve using feedforward architectures.

For example, in Dutch, the infinitive form of any verb must be placed at the end of the sentence. Therefore, if one wants to say "I want to go there," in Dutch, one would actually say "I want there to go." If the task is to understand the meaning of the Dutch sentence, one must connect the verb "want" with what one wants to do, which is "to go." The distance between the two related verbs is variable in any Dutch sentence. If a neural network-like tool is developed for understanding Dutch sentences, it would likely have to be a recurrent network of some sort in order to keep information in the system long enough for it to be related to the later information. In fact, much of the research done on recurrent neural networks has been focused on the problem domain of natural language understanding (*e.g.* [18]).

The protein secondary structure prediction problem has many instances of such longer-range interactions although most of the work done on the problem has focused on the local interactions [32]. The representation of proteins used in neural network approaches to the problem does not allow knowledge of longer-range interactions among amino acids in the protein to be accounted for, despite the fact that these longer-range interactions are known to be significant in the formation of protein secondary structure [31]. In many of the studies in which neural networks are applied to the protein secondary structure prediction problem [13] [28] [29], a single protein is represented by several patterns, such that the patterns act as a "sliding window" across the length of the protein. The first pattern represents the first amino acid in the protein, the second pattern represents the second amino acid in the protein, and so on. The window allows some of the local context of each amino acid to be used by the feedforward neural network for prediction [13]. In this representation, a limit is imposed on the length of time that an individual amino acid can influence the prediction of other amino acids in the protein. That is, once an amino acid leaves the sliding window, it can no longer influence subsequent predictions. In addition, it is known that the domain algorithm developed by Chou and Fasman secondary structure prediction does not take longer-range interactions into account [38].

The experiments described in this dissertation show how to augment the Chou-Fasman algorithm with information concerning longer-range interactions by adding recurrent connections to the expert network encoding the Chou-Fasman rules. The recurrent connections feed information about previous activations of the network into the current activation of the network. The previous activations of the network constitute the state or context in which the current problem is being solved. Therefore, these connections allow state information, which concerns the predictions made by the network for amino acids occurring early in the primary sequence, to be used by the network during the course of making predictions for amino acids occurring later in the primary sequence.

The addition of recurrent connections to expert networks allows expert network technology to be applied to any problem domain which requires the maintenance of state information during the problem solution. This chapter describes some of the key ideas necessary to understand the recurrent expert network methodology.

4.2 Previous Work on Recurrent Networks

Researchers have attempted to generalize neural network architectures by removing the restriction that allows only feedforward connections. Many of these generalization attempts, however, impose the restriction that the recurrent connections among neurons in the networks must be symmetric [56] [57]. Other researchers have developed models which have other restrictions on the types of connections allowed but which do not require the connections to be symmetric. Two of these models are of particular interest to the work presented in this dissertation and will be addressed below.

4.2.1 Elman Networks

Figure 4.1 is a schematic view of the types of networks described by Elman [16] [18]. In these networks, a set of input nodes, called the *input vector*, receives input from the input pattern, just as in a typical feedforward neural network.

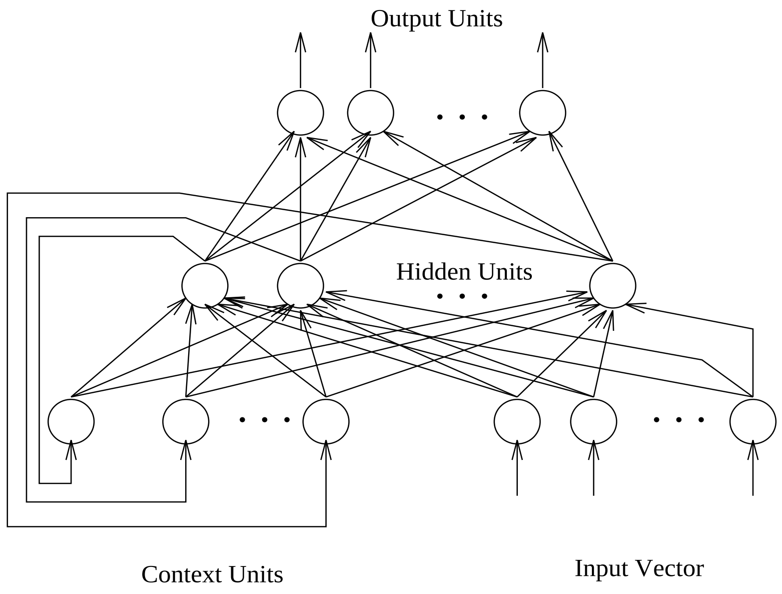

Figure 4.1: Elman Network

In addition, another set of input nodes, called *context units*, are added to the network architecture to capture information about the context in which the problem solution is proceeding. The context units can also be considered to be hidden units in that they do not interact with the outside world in any manner. However, the context units are considered input units during learning because error is not propagated back through their input connections [9]. Rather than receiving input from an external input pattern, the context units receive their input from the output of the hidden layer of nodes. That is, context units are created to receive copies of the activation of the hidden units on a one-for-one basis and thus, remember exactly what the previous internal state of the network was. The synapses between the hidden layer units and their respective context units have fixed weights of 1.0. Because the weights on these recurrent connections are fixed, standard backpropagation of error can be used to learn the weight values on the feedforward synapses.

Elman has shown that networks which capture context information can be used to acquire some of the basic principles needed for language-understanding [18]. In particular, his simulations make it clear that simple recurrent networks of the type described above can extract information from sound sequences concerning the boundaries of linguistic units, such as "phoneme," "morpheme," "word," that must be learned when learning a language. In addition, these simple recurrent networks can learn principles about constraints on word order that allow the creation of simple sentences. In these networks, the internal representations of the input vectors are maintained as the context for solving the problem at hand. The internal representations can be thought of as the feature vector that the network has extracted from the current input vector combined with the previously extracted feature vectors. Note that because the hidden units receive input from both the context units and the input units, there is no isolated representation of a particular input vector. Instead the internal representation is of the current input vector combined with all previous input vectors as processed through the network. In these internal representations, Elman has discovered very complex structures which suggest that the context-sensitivity inherent in the networks imparts greater ability to abstract and generalize [16].

4.2.2 Jordan Networks

Figure 4.2 is a schematic view of neural network architectures proposed by Jordan [17]. In these networks, the recurrent connections are used to associate a static pattern, called a *plan*, with a serially ordered output pattern, that is,

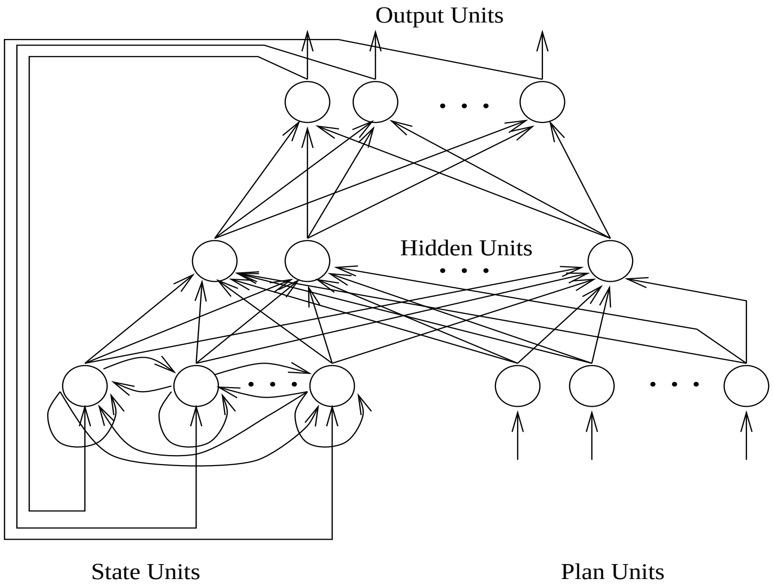

Figure 4.2: Jordan Network

a sequence of *actions*. Each plan vector is associated uniquely with an output pattern and is presented to the *plan units* of the network. The recurrent connections feed values from the output units of the network to a set of *state units*, allowing the hidden units of the network to see their own previous output, so that subsequent network behavior can be influenced by previous network responses. In addition, the state units have recurrent connections among themselves to allow the output of the state units to be influenced by all previous states in the input sequence. This influence represents the context for the next output vector in the sequence. The weight on a synapse between an output unit and a corresponding state unit is fixed at 1.0, allowing the use of standard backpropagation of error to be used for learning the other feedforward weights in the network [17].

4.3 Key Ideas and Heuristics

4.3.1 Types of Recurrence

Recurrent neural networks can be categorized into *partially recurrent* networks, which are recurrent in their activation only, and *fully recurrent* networks, which are recurrent in both their activation and their learning [9]. In other words, in partially recurrent networks, the learning algorithm updates only the feedforward connections while in fully recurrent networks, all synapses are updated during the course of learning. Fully recurrent networks allow arbitrary connections among nodes and implement learning algorithms which allow learning of the weights on all connections in the network. The learning algorithms for fully recurrent networks are often based on backpropagation of error with an additional *relaxation step* [58] [59] [60]. The algorithms, therefore, require long training periods. In addition, convergence in fully recurrent networks is difficult to guarantee. Fully recurrent networks are, therefore, often impractical for use in real applications. The fundamentals of these learning algorithms are described in Subsection 4.3.5.

Partially recurrent networks, on the other hand, have a small, carefully chosen set of recurrent connections whose weights are fixed and cannot be learned. Because these weights are fixed, standard backpropagation of error [10] can be used to learn the weights on the other, feedforward connections. The Elman networks and Jordan networks described above are both examples of partially recurrent neural networks. The recurrent expert networks implemented for solving the protein secondary structure prediction problem are also examples of partially recurrent networks.

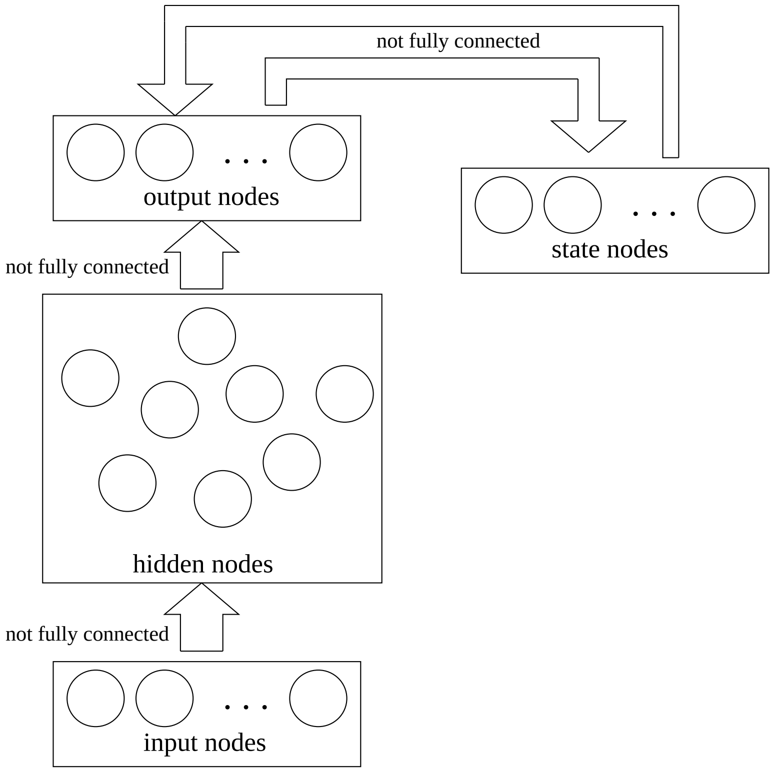

Figure 4.3: Recurrent Architectures for Protein Secondary Structure Prediction

4.3.2 Recurrent Architectures

Whether the feedback connections in recurrent architectures come from the hidden units, as in Elman networks, or from the output units, as in Jordan networks, the behavior of the recurrent networks is probably similar. Individual problems, however, might be better suited to one architecture or the other [61]. Although the software developed for this dissertation work supports Elman networks, Jordan networks and various other variations of partially recurrent architectures, the application of protein secondary structure prediction suggests the use of certain partially recurrent architectures rather than others.

Figure 4.3 is a schematic view of the recurrent architectures implemented to augment the Chou-Fasman prediction algorithm with information about long-range amino acid interactions. In these architectures a set of *input nodes* receives external input from an input pattern file. These input nodes are connected, consistent with the knowledge found in the underlying knowledge base, to a subset of the *hidden nodes*. The hidden nodes have a variety of connections among them, again in a manner which encodes the knowledge from the knowledge base. A subset of the hidden nodes are then connected to the *output nodes*, which receive external information concerning the ideal outputs for each of the nodes for each input pattern. For each output node, a *state node* node and a synapse between the output node and the state node are created. Each of the state nodes then contains an output synapse to another state node which in turn contains an output synapse to another state node. This sequence of state nodes attempts to maintain the state information over the course of many subsequent activations of the network. The number of state nodes in this sequence is determined via experimentation.

The outputs of all state nodes corresponding to a particular output node are combined, typically via a `CAddNode` object, and the combined value is sent into the corresponding output node. These connections allow each output node to receive information concerning the previous prediction values for that state node. Chapter 5 shows several specific implementations of these networks and the results of activating them with protein sequence data.

Note that the use of state information in the above networks is different than that seen in Elman networks. In the networks for prediction of protein secondary structure, each of the hidden nodes represents a particular concept, such as *InitHelix*, and the connections between the nodes represents a relationship between the concepts. In other words, the topology of the hidden nodes in these networks is determined by the background knowledge found in the underlying knowledge base. Because the topology of the hidden nodes is complex, unlike in the Elman networks where the topology of the hidden nodes is simply one layer with full connectivity, it would be difficult to feed the state nodes from the hidden nodes. In Elman networks, the context units, which serve the same role as the state nodes in the protein secondary structure prediction networks, hold an exact copy of the previous activation of the internal state of the network. To accomplish this task in the recurrent expert networks, the entire hidden node topology would have to be copied. In such a network, it is not clear which of the state nodes would receive input from the output nodes because each of the state nodes, as a copy of an expert network hidden node, now has meaning by itself.

The notion of recurrence seen in Jordan networks is also not perfectly suited to implementation in an expert network. In Jordan networks, the output nodes send their output to a set of state units which then feed into the hidden nodes so that the hidden nodes combine the previous predictions with the current input to determine the current prediction. As stated above, the hidden nodes in the protein secondary structure prediction networks each have specific meaning with regard to the underlying knowledge base and, therefore, it is not clear which of them should receive input about all the previous predictions made by the network.

Because of the differences between the functionalities of the hidden nodes in neural networks like the Jordan and Elman networks and the hidden nodes in expert networks as described in this dissertation, I have chosen to implement the recurrent connections in the manner seen in Figure 4.3. In these networks, the state nodes are allowed to see the previous predictions made by the network and then feed that prediction information back to the output nodes. The output nodes then make their current predictions based on the current input combined with previous prediction information.

4.3.3 Types of Error

In standard backpropagation neural networks, the notion of output error refers to the difference between the actual output of an output node and the ideal output for that same output node. The backpropagation algorithm attempts to minimize the difference between the actual and ideal outputs by adjusting the weighted connections between nodes such that the next time a particular input pattern is presented to the network, the actual output of the network is closer to the ideal output for that pattern. This error minimization algorithm is a *gradient descent* method, because the path the network follows in error space is the path of most rapid descent toward some local error minimum [9].

Actual error, as defined above, relies on a notion of *local* error. If error is local, for a particular node, it depends on only the ideal output for that node and the actual output for that node. That is, local error depends only on information that is local to the node itself. For many applications, minimization of this type of error yields the locally optimum solution to the problem.

Other types of problems require a more *global* notion of error. In particular, problems which deal with some sort of prediction, as opposed to some type of true pattern association, are likely to require this more global notion of error. In many prediction problems, the ideal output pattern contains only one high value (that is, a 1) and all the rest of the ideal values are the low value (-1 or 0, depending on the allowed range of values from the firing function for the node). Typically, in this type of problem, the output nodes represent the set of possible classes into which the input pattern can be placed and the ideal output pattern contains a high value in the position corresponding to the output node representing the correct class for the particular input pattern. In other words, the input pattern can belong to only one of the possible classes.

In such prediction problems, it does not matter whether the actual output value reaches the ideal value. Rather, it is more important that the node corresponding to the correct class has the highest relative actual output value. For example, assume the network in question has three output nodes, that is, three possible classes, and that for a particular input pattern, the ideal output pattern is (1, 0, 0). If the actual output pattern from the network is (.7, .2, .3), the actual error, as described above, is $((1-0.7)^2)+((0-0.2)^2)+((0-0.3)^2) = 0.22$. If, on the other hand, the assumption is that the network predicts that the input pattern belongs to the class corresponding to the output node having the highest output value, the network has a global, or *prediction*, error of 0 because the output node corresponding to the correct class has the highest actual output. The fact that the highest actual output value is not quite as high as the highest ideal output value or the low actual output values are not quite as low as the lowest ideal output values does not prevent the network from making the correct prediction for the input pattern.

The prediction error is considered to be global because the error for a node cannot be determined by looking at features of the individual node alone, without respect to other nodes in the network. For example, if, for a particular input pattern, the ideal output pattern is (1, 0, 0) and the actual output pattern is (.7, .2, .3), none of the output nodes are considered to be in error. If, on the other hand, for the same input pattern, the actual output pattern is (.7, .9, .3), both the first and second output nodes are considered to be in error because the first node, which corresponds to the correct class for the input pattern, no longer has the highest actual output value. The input pattern is instead predicted to belong to the class corresponding to the second output node. Notice that in both situations, the first node has an actual output value of 0.7 but since error cannot be determined locally with only information about the node in question, in one case, the node is not in error while in the other, the node is in error. Actual error and prediction error are related but, as can be seen in this example, are not identical.

The CAEN project [50] is an example of a situation where the network prediction is determined by the output node with the highest value. In that project, an expert network was developed to aid in the diagnosis of causes of machine faults. The inputs to the network are a set of symptoms and the outputs are a set of possible causes. The network prediction of the cause for the machine fault corresponds to the output cause node with the highest output value. For diagnosis of machine faults, the use of actual error makes less sense than the use of prediction error. The prediction of protein secondary structure is another problem in which it does not really matter how close the actual output of the output nodes are to the ideal values. Instead, it is important that the node with the highest output value corresponds to the position in the output pattern that contains the high value.

4.3.4 Learning Algorithms

The learning algorithm used in a network for solving a particular domain problem requires an investigation of the type of error that makes sense for that domain. General purpose learning algorithms typically minimize actual error because actual error is the same regardless of problem domain. For the partially recurrent expert networks described in Chapter 5, the learning algorithm attempts to minimize actual error but for evaluation of the predictive capability of the networks, prediction error is used. The choice to use two different types of error for the two different tasks is based on the use of object-oriented technology for the implementation of the Expert Network Development Environment (ENDE). The object-oriented programming paradigm is well-suited to the implementation of the *artificial neural objects* used in neural network technology [5].

In the object-oriented paradigm, there are generally three types of relationships that exist among classes, *Is-A*, *Has-A* and *As-A* [62]. Figure 3.2 shows the relationships, along with cardinalities, of the classes in the software described in this dissertation. If two classes have an *Is-A* relationship, the child class is a more specific example of the parent class. For example, the `CHardSynapse` class is a more specific example of the `CSynapse` class and it inherits the properties of the `CSynpase` class, much like a child inherits many properties of his or her parents. The relationship between the `CNetwork` class and the `CNode` class is an example of a *Has-A* relationship between two classes. The *Has-A* relationship is a containment relationship in which one class contains an instantiation of another class. The third relationship between two classes, *As-A*, is less common than the other two and there are no examples of this relationship in the software described here. In an *As-A* relationship, one class is implemented as an example of another class, allowing the first class to access the member functions of the second class but disallowing users of the first class access to the member functions of the second class. The various relationships among classes illustrate the principle of *encapsulation* [62], in which the data and behaviors of an object are hidden from the user of the object. The user must instead interact with the object via an agreed upon public protocol, without knowledge of how the object accomplishes its behaviors.

The principle of encapsulation suggests that processing should be localized. For example, to determine the error at a particular node, the user should ask the node for its error value and have no indication of how the error value is determined. The global notion of error described above suggests that prediction error cannot be localized, that is, determined by querying the node itself concerning its error. An object of some class type, perhaps the `CNetwork` class, which can be considered to be the user of the `CNode` class, must look inside the node, violating the principle of encapsulation, to determine the error value at the node. Although the global determination of error can indeed be implemented, it violates the object-oriented principle of encapsulation. For this reason, I have chosen a learning algorithm which does not minimize prediction error but instead minimizes actual error.

The determination of the predictions made by the network is accomplished outside the network. That is, the network does not output a set of predictions but rather outputs prediction values. The determination of which of these output values is the highest and, therefore, the actual secondary structure prediction made by the network for the current amino acid must be completed separately from the network processing. By maintaining a separation between the output values of the network and the actual secondary structure prediction associated with those values, I have ensured that the expert network development environment software is not tied to the particular problem domain of predicting protein secondary structure.

4.3.5 Time In Network Activation and Learning Phases

In the activation phase of a feedforward network, the output is an explicit function of the input. In other words, the input is propagated through the network and produces the output immediately. In contrast, during activation, fully recurrent networks always need some kind of relaxation to reach an attractor or stable state [9]. In addition, fully recurrent networks require a relaxation step during the training phase, which adds considerable time to already long training times. The partially recurrent networks described in this dissertation represent a cross between feedforward networks, which do not require a relaxation step, and fully recurrent networks, which are able to maintain state information. Therefore, partially recurrent networks are able to capture some sense of state

```
         ___ for each iteration
        |         for each pattern
        |         ___ for each relaxation time step
        |        |        for each node in the network
        |        |            activate
        | time 1-|            accumulate weight changes
time 2-|         |        end for
        |        |___ end for
        |         end for
        |         for each synapse
        |             update with accumulated weight changes
        |         end for
        |___ end for
```

Figure 4.4: Learning Algorithm for Fully Recurrent Networks

during the problem solution without prohibitively long training times.

When a learning algorithm has a relaxation step, there are several notions of time in the algorithm. The weights on the synapses change slowly with respect to the activation of the entire network. The relaxation step activates the network a certain number of times for each pattern during a single iteration. Figure 4.4 shows the basic loops in a learning algorithm for a fully recurrent network. The loop labeled *time 1* represents the faster, relaxation loop for activation of the network and the loop labeled *time 2* represents the slower loop for learning of the weights on the synapses in the network. In a partially recurrent network, the loop labeled *time 1* does not exist and therefore, the training of a partially recurrent network occurs more quickly than the training of a fully recurrent network.

4.3.6 Averaging of Weight Changes

The number of negative examples for any given expert network in this dissertation work is far greater than the number of positive examples. For example, when training an expert network to recognize helical regions in a protein, the number of non-helical amino acids is far greater than the number of helical amino acids. If the weight changes which are applied at the end of an iteration are simply summations of the weight change values calculated for each of the input patterns, the negative examples in the training dataset wield far more influence than the positive examples. The result of this heavier influence is that the weights quickly move to the point at which the network correctly predicts every negative example and incorrectly predicts every positive example.

To prevent this tendency, I implemented an option in my software that allows the use of a version of ΔW *averaging* [49]. ΔW averaging is a technique in which the weight change values are averaged by example type for each example in each iteration. When ΔW averaging is not used, the weight change values are accumulated for each input pattern using the standard batch mode implementation of the update rule. The new weight on the synapse between node i and node j is the old weight on the synapse between these two nodes plus the sum of the weight changes calculated for this synapse over all the training patterns for the network. In other words,

$$w_{ji}^{new} = w_{ji}^{old} + \sum_{k=1}^{P} \Delta w_{ji}^{k}, \tag{4.1}$$

where P is the number of training patterns and Δw_{ji}^{k} is the weight change calculated for the synapse between node i and node j for pattern k. When, on the other hand, ΔW averaging is on, the new weight on the synapse between node i and node j is the old weight on the synapse plus the average of the weight changes calculated for this synapse over all the positive training examples for the network plus the average of the weight changes calculated for this synapse over all the negative training examples for the network. In other words,

$$w_{ji}^{new} = w_{ji}^{old} + \Delta W_{ji}^{Positive} + \Delta W_{ji}^{Negative}, \tag{4.2}$$

where

$$\Delta W_{ji}^{Positive} = \frac{\sum_{k \in PositiveExamples} \Delta w_{ji}^{k}}{NumberOfPositiveExamples}, \tag{4.3}$$

$$\Delta W_{ji}^{Negative} = \frac{\sum_{k \in NegativeExamples} \Delta w_{ji}^{k}}{NumberOfNegativeExamples}, \tag{4.4}$$

Positive Examples is the set of all positive training examples and *Negative Examples* is the set of all negative training examples.

4.3.7 Determination of Prediction Accuracy

The most common method for determining the accuracy for a network designed for prediction is to count the number of input patterns for which the correct output is predicted and then divide by the number of input patterns. For example, if there are five input patterns and the correct class (corresponding to the output node with the highest output value) is predicted for three of them, the prediction accuracy for the network is $\frac{3}{5} = 60\%$.

For many tasks, however, there are many more training examples available for one of the classes than for the other classes. In this situation, the network could learn to simply predict the most commonly occurring class no matter what the input pattern is and the prediction accuracy is relatively high. Such a network is virtually useless, however. For example, if there are five input patterns and two potential output classes with four of the patterns coming from class 1 and only one pattern coming from class 2, the network could simply predict every input to fall into class 1. In the case of this training set, the network achieves $\frac{4}{5} = 80\%$ prediction accuracy but the high accuracy is an artifact of the fact that one of the classes occurs only rarely. In such a situation, perhaps a network with a lower overall prediction accuracy but which predicts the rare cases with high accuracy should be rewarded more than a network with a higher overall prediction accuracy that never predicts the rare cases.

In prediction of the secondary structure of proteins, there are three possible output classes, *helix*, *strand*, and *coil* [33]. Every amino acid falls into one of these three categories. In nature, however, many more amino acids participate in coil structures than in either helices or strands. In the benchmark training dataset created by Qian and Sejnowski [13], there are 4601 helical amino acids, 3636 strand amino acids and 9868 coil amino acids. In the test dataset created by Qian and Sejnowski, there are 849 helical amino acids, 746 strand amino acids, and 1923 coil amino acids. Of the 21,623 amino acids in these two datasets, 55% participate in coil structures while only 25% are helical and 20% are strands. By predicting all amino acids to be coil, a prediction algorithm can easily achieve 55% prediction accuracy. Such an algorithm is not useful, however, for actual prediction of protein secondary structure.

As a result of the high number of coil amino acids in proteins, other measurement tools for prediction accuracy have been developed. Of these tools, Matthews' correlation coefficient is the most commonly used [63]. In this method for determining prediction accuracy, the correlation between the observed structure for the protein and the predicted structure for the protein is calculated. The correlation coefficient for each secondary structure type is measured separately and algorithms with good prediction accuracy perform equally well on all three structure types. The correlation coefficient is:

$$C = \frac{(p/N) - (\bar{P} \times \bar{S})}{(\bar{P} \times \bar{S} \times (1 - \bar{S}) \times (1 - \bar{P}))^{\frac{1}{2}}}, \tag{4.5}$$

where

$$\bar{P} = \frac{(p + v)}{N}, \tag{4.6}$$

$$\bar{S} = \frac{(p + u)}{N}, \tag{4.7}$$

N is the total number of residues in the protein, p is the number of true positives for the secondary structure in question, u is the number of false negatives, v is the number of false positives and

$$N = p + q + u + v. \tag{4.8}$$

$C = 1$ means a perfect correlation between the observed secondary structure and the predicted secondary structure, $C = -1$ means no correlation whatsoever has been found and $C = 0$ means the correlation is what one would expect from a completely random prediction [63]. Typical correlation coefficients for protein secondary structure prediction algorithms are in the 0.2 to 0.4 range. Note that this formulation of the standard statistical correlation coefficient requires an assumption that the predicted structure for residue n, P_n, and the observed structure for residue n, S_n, are both step functions with two possible values, 1 and 0. For example, if $P_n = 1$, then residue n is predicted to be helical while if $P_n = 0$, then residue n is predicted to be non-helical. Similarly, $S_n = 1$ if residue n is observed to be helical and $S_n = 0$ if residue n is observed to be non-helical.

Many factors make perfect prediction on an individual amino acid basis a very difficult, if not impossible, goal. The most difficult factor to overcome is the fact that different researchers arrive at different secondary structure assignments for the same amino acid in the same protein, using either visual inspection or different feature extraction algorithms [32]. If, in the training set, the manner in which amino acids are assigned their ideal, or observed, secondary structure is not consistent, a network may have difficulty learning a single functional relationship between the primary sequence and the secondary structure.

Because of the problems with the idea of single residue prediction accuracy, Rost and his colleagues [32] have suggested that the goals of protein secondary structure prediction should be redefined. They have developed a new measure of *segment overlap* that is somewhat insensitive to small variations in secondary structure assignments. Segment overlap is defined to be

$$Sov = \frac{1}{N} \times \sum_{s} \frac{minov(s_1, s_2) + \delta}{maxov(s_1, s_2)} \times len(s_1), \tag{4.9}$$

where N is the total number of residues in the protein, the sum is taken over all segment pairs $s = (s_1, s_2)$, where s_1 and s_2 are two segments that have in common at least one residue position in the same secondary structure, the subscript 1 usually refers to the sequence of observed structure for the protein, the subscript 2 usually refers to the the sequence of predicted structure for the protein, *minov* is the number of residues for which both the observed and predicted sequence has the same secondary structure, δ is the accepted variation between the two sequences, *maxov* is the number of residues for which either of the two sequences has a particular secondary structural element and $len(s_1)$ is the length of the sequence labelled with subscript 1 (usually the observed structural sequence). Sov counts the fractional extent to which two segments overlap, with some allowance for non-matching residues at the ends [32].

In the work described in this dissertation, Matthews' correlation coefficient is used when measuring prediction accuracy. This measurement tool was chosen over the others because it precludes the extreme situation of prediction of only coil producing the most accurate results while at the same time giving results that can easily be compared to others in the secondary structure prediction literature. In addition, this measurement tool can easily be adapted to apply to other prediction problems where a choice between classes must be made using the output values. The segment overlap measurement, on the other hand, is application specific and is, therefore, difficult to apply to other problem domains.

4.3.8 Intermediate Concepts in the Network Architecture

The original rules for the Chou-Fasman algorithm [38], shown in Section 2.2.1, are difficult to use in a completely automated manner. As a result, Maclin and Shavlik [19] modified the original rules somewhat and developed the rules shown in Section 2.3. In the work described here, the helix finding and strand finding subnetwork architectures were created by transforming the Maclin and Shavlik rules into a network via the process described in Kuncicky, *et al.* [1]. The helix finding network created by this transformation is shown in Figure 4.5.

Note that there are several places in the helix finding network where one operation (`OP`) node feeds into another `OP` node. The synapses feeding into `OP`

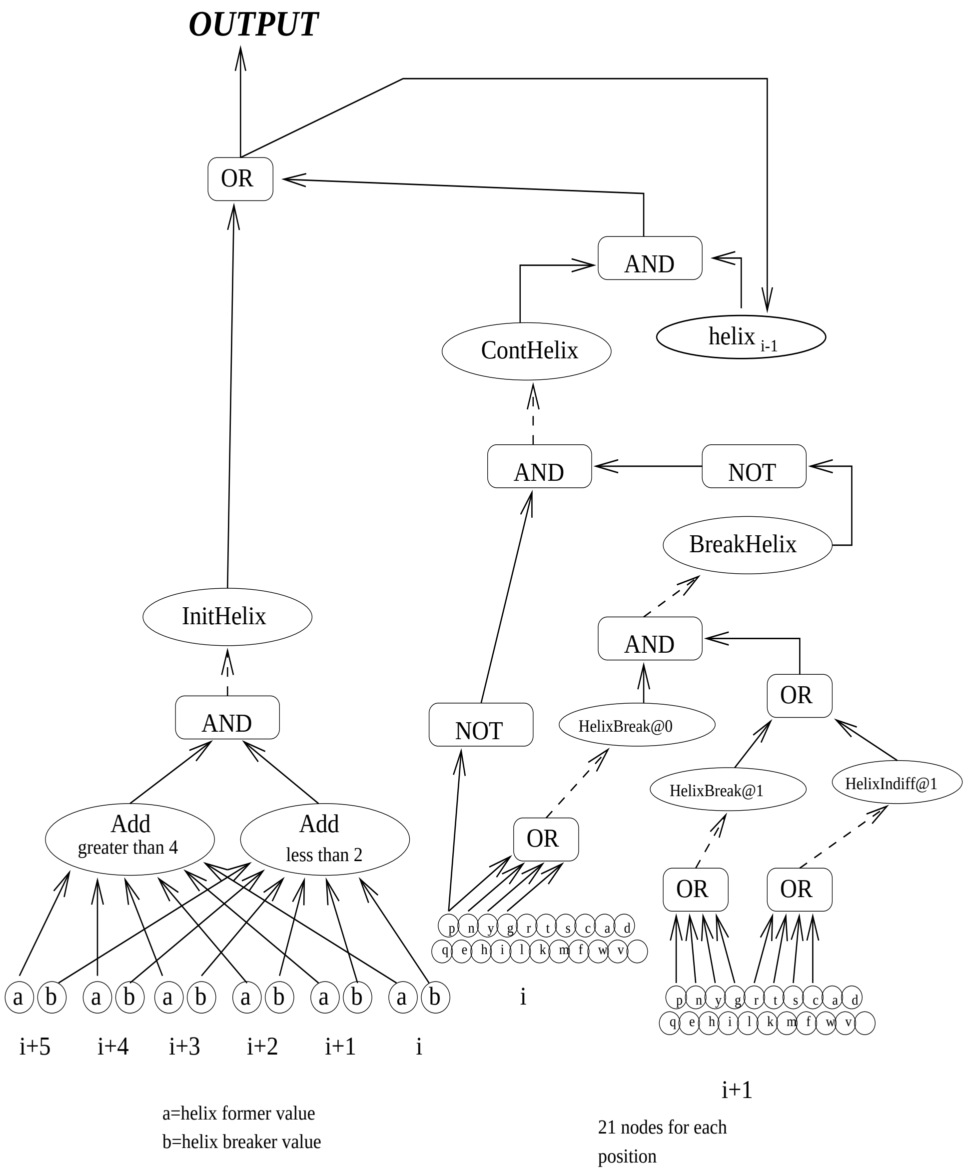

Figure 4.5: Original Helix Finding Expert Network

nodes are hard synapses, represented by solid lines in Figure 4.5. The weights on the hard synapses are fixed and non-trainable. For example, in Figure 4.5, the input node representing the amino acid *Proline*, labeled p, feeds into an `OP` node of type `CNotNode`. The `CNotNode` object then feeds into another `OP` node of type `CAndNode`. Finally, the `CAndNode` object feeds into a regular (`REG`) node representing an intermediate concept called *ContHelix*. Because `CSynapse` objects leading into `OP` nodes are always of type `CHardSynapse` on which no learning can occur, in this path in the network, little learning can occur. The intermediate concept *ContHelix* does not appear in the original rules developed by Chou and Fasman although it does appear in the rules created by Maclin and Shavlik. In previous work on expert networks, the intermediate concepts created by the human expert were considered to be necessary for the proper processing of the expert network [5] but it is clear that in the Chou-Fasman rules developed by Maclin and Shavlik, the intermediate concepts are somewhat arbitrary.

Figure 4.6 shows an implementation of the rules developed by Maclin and Shavlik in which all intermediate concepts have been removed. This network represents what the Maclin and Shavlik rules would look like if all intermediate concepts were removed and the individual rules were instead combined into one long rule. In such a network, no refinement of the background knowledge can occur because none of the synapses in this network have trainable weights. Instead, all of the synapses are of type `CHardSynapse`. Such a network has no ability to refine the background knowledge using data and thus, is not a very useful example of an expert network.

One should, therefore, be extremely wary of removing all intermediate con-

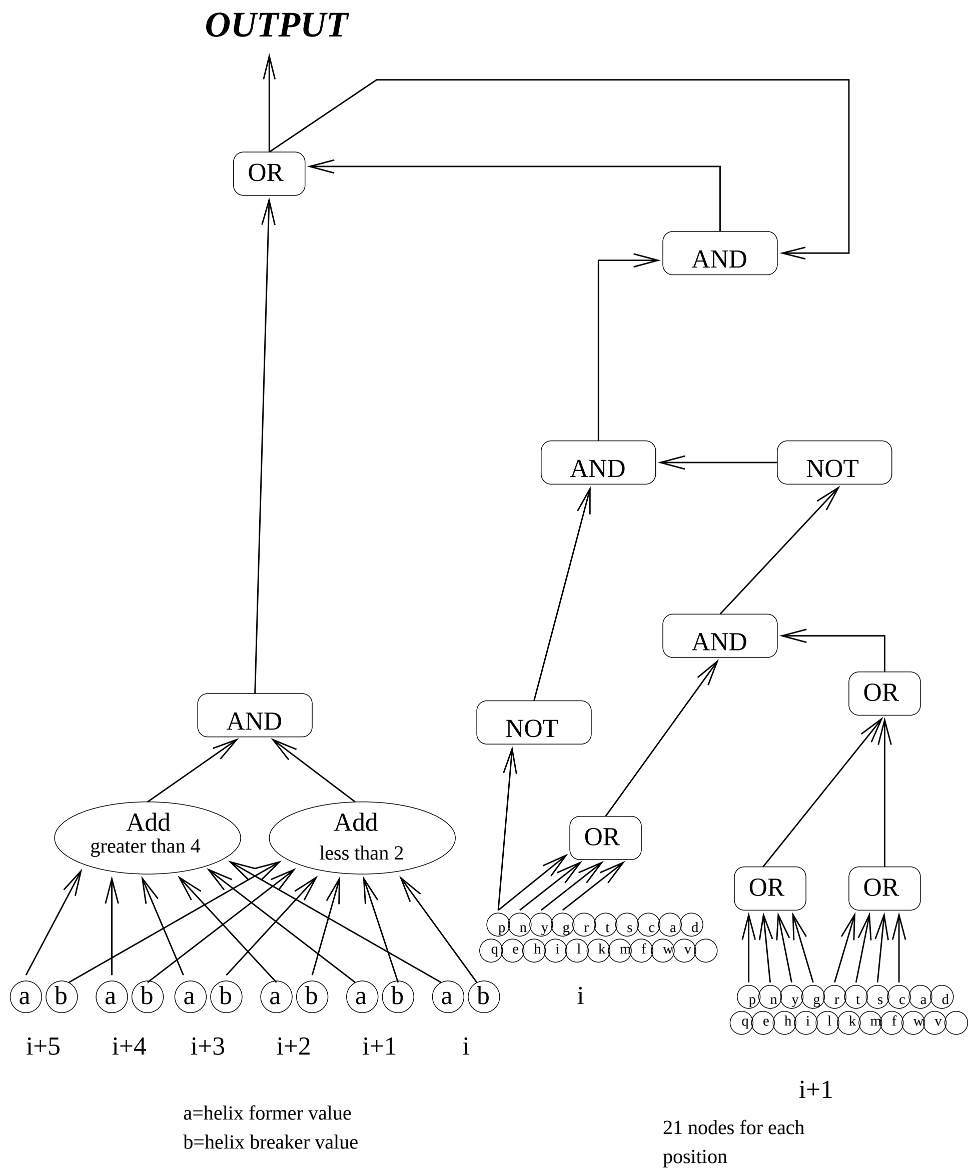

Figure 4.6: Helix Finding Expert Network with No Intermediate Concepts

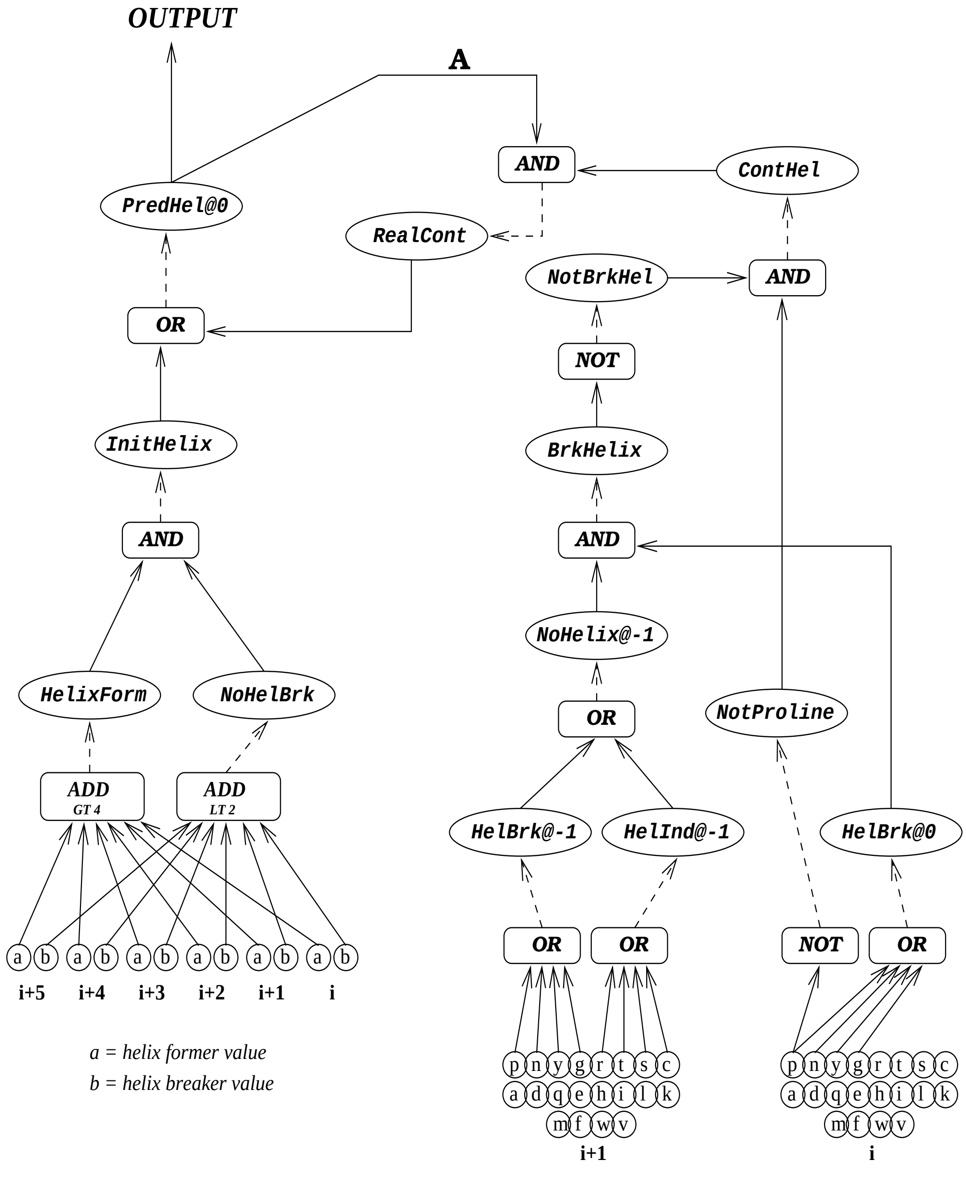

Figure 4.7: Helix Finding Expert Network

cepts in the knowledge base before transforming the knowledge into the expert network topology. On the other hand, adding intermediate concepts between every pair of `OP` nodes results in a network with the maximum number of possible trainable synapses. In other words, disallowing situations where one `OP` node feeds directly into another `OP` node results in an expert network in which the maximal amount of learning from the data can occur. Figure 4.7 shows the network that results from transforming the Maclin and Shavlik rules according to the algorithm given by Kuncicky [5], followed by an additional step of creating intermediate concept nodes, all of which are `REG` nodes, between any two `OP` nodes that feed directly into one another. In other words, to create the network shown in Figure 4.7, first use the two steps described in [5] to create the network shown in Figure 4.5. Next, determine pairs of `OP` nodes that are connected directly to each other. Between each of these pairs, add a `REG` node representing an intermediate concept such that the first `OP` node feeds into the `REG` node with a soft synapse with an initial weight of 1.0 and the `REG` node feeds into the second `OP` node with a hard synapse with an initial weight of 1.0. The previous hard synapse between the two `OP` nodes is removed.

The issue of adding intermediate concepts to the knowledge base did not come up in the original research on expert networks because the topology of early expert networks came strictly from actual expert system rule bases. These rule bases tended to have lots of intermediate concepts because the rules containing intermediate concepts are simpler and thus, easier for the human expert to understand. In this early research, much care was taken to duplicate the inferencing process that occurred in the expert system on which the expert network was based [5]. In the work described in this dissertation, the rules developed by Maclin and Shavlik were not used in an expert system environment and thus, there is no expert system inferencing to which the work described here can be compared. Therefore, there is no concern with duplication of the manner in which an expert system comes to its conclusions. Adding intermediate concepts then only gives the expert network more places in which the background knowledge can be refined. These intermediate concepts may also provide the domain expert with new, potentially important concepts with which to reason about the problem domain.

CHAPTER 5
EXPERIMENTS AND RESULTS

5.1 Experimental Methods

The benchmark protein datasets created by Qian and Sejnowski [13] were used in all the studies reported here. These proteins are split into a training dataset of 111 proteins and a test dataset of 17 proteins. I chose a random subset of the training data as my training dataset. My training dataset contains 1965 helical amino acids, 1135 strand amino acids and 2989 coil amino acids taken from 35 proteins. The test dataset contains 849 helical amino acids, 746 strand amino acids and 1923 coil amino acids taken from 17 proteins. All training proceeded using the ΔW averaging technique [49] and a learning rate of .0001. The number of iterations varied from experiment to experiment and is reported in the description of each experiment. Each protein is scanned only in the forward direction. Recall that to completely automate the Chou-Fasman algorithm, information from scanning the protein in both directions is required for prediction (see Subsection 2.2.2). Such an implementation, however, cannot be network based. I have, thus, chosen to ignore potentially helpful information from scanning in the opposite direction.

To show how well an automated version of the Chou-Fasman algorithm performs without benefit of refinement using data, the results from Nishikawa [37] are shown at the bottom of Figure 5.5. Although Nishikawa tested his implementation of the Chou-Fasman algorithm on a different set of proteins than used in this dissertation, one can expect that if Nishikawa had tested his implementation on the same benchmark protein dataset used in my work, the results would not be significantly different than those reported in his paper since both datasets, the one used by Nishikawa and the benchmark set, are randomly selected sets of globular proteins. I have chosen the Nishikawa implementation of the Chou-Fasman algorithm as a baseline rather than the Maclin and Shavlik implementation [19] because although Maclin and Shavlik use the Chou-Fasman rules as the starting point for the architecture of their networks, they then fully connect the resulting set of network nodes. By doing so, the underlying algorithm is obscured so that after training, the algorithm is no longer accessible. The Maclin and Shavlik method is successful, achieving an average correlation coefficient of .35. These results do not reflect the accuracy of the Chou-Fasman algorithm itself but rather reflect the accuracy that can be brought about using the Maclin and Shavlik method.

Although the above discussion involves comparisons of various implementations of the Chou-Fasman algorithm, such comparisons are not the point of the work described in this dissertation. The goal of this work is to show how the knowledge encoded in an expert network can be augmented with state information and that such augmentation leads to a more accurate solution to the problem. Therefore, I did not worry about optimizing the expert network by improving the knowledge base used to determined the topology of the original expert network. Instead, my focus is to determine the effect of adding state knowledge to the network in question, given that it is created from a particular knowledge base, despite the fact that the knowledge base is not optimized.

Section 6.2 discusses a variety of methods for improving the knowledge base which underlies the expert network. These methods constitute future work in improving protein secondary structure prediction.

The transformation from amino acid sequence to a set of training patterns is accomplished via a C program, which creates a set of patterns for each protein and inserts a pattern of all zeroes between the proteins. Each pattern contains information about the Chou-Fasman conformational parameters of the local region of the amino acid sequence along with information about the actual amino acids found in that region.

Figure 5.1 shows the transformation from amino acid sequence to the set of input patterns for the helical finding sub-network. Note that a protein of length five amino acids results in a twelve input patterns because the patterns simulate a sliding window along the amino acid sequence. Pattern 1 in Figure 5.1 is a pattern containing all zeroes. In Pattern 2, the first amino acid, *Alanine*, appears in position $i+5$ of the conformational parameters part of the pattern, with the helix former value of *Alanine*, 1.34. In Pattern 3, *Alanine* moves to position $i+4$ and the second amino acid, *Glycine*, appears in position $i+5$. The process continues until *Alanine* is in position $i+1$ which is Pattern 6 in the Figure. At this time, *Alanine* also appears in the Amino Acids part of the pattern, with a one appearing in position $i+1$. In Pattern 7, *Alanine* has finally moved into position i and the output pattern for this input contains the information about whether this amino acid participates in a helix or not. When the end of the protein is reached, that is, the last amino acid of the protein has appeared at postion i of the input pattern, another pattern of all zeroes is

Moving in this direction --->

Protein Sequence Ala Gly Val Val Lys

Patterns	**Conformational Parameters**												**Amino Acids**	
	i		**i+1**		**i+2**		**i+3**		**i+4**		**i+5**		**i**	**i+1**
Pattern 1	0	0	0	0	0	0	0	0	0	0	0	0	00000000000000000000	00000000000000000000
Pattern 2	0	0	0	0	0	0	0	0	0	0	1.34	0	00000000000000000000	00000000000000000000
Pattern 3	0	0	0	0	0	0	0	0	1.34	0	0	1.21	00000000000000000000	00000000000000000000
Pattern 4	0	0	0	0	0	0	1.34	0	0	1.21	1.00	0	00000000000000000000	00000000000000000000
Pattern 5	0	0	0	0	1.34	0	0	1.21	1.00	0	1.00	0	00000000000000000000	00000000000000000000
Pattern 6	0	0	1.34	0	0	1.21	1.00	0	1.00	0	1.09	0	00000000000000000000	00000000100000000000
Pattern 7	1.34	0	0	1.21	1.00	0	1.00	0	1.09	0	0	0	00000000100000000000	00010000000000000000
Pattern 8	0	1.21	1.00	0	1.00	0	1.09	0	0	0	0	0	00010000000000000000	00000000000000000001
Pattern 9	1.00	0	1.00	0	1.09	0	0	0	0	0	0	0	00000000000000000001	00000000000000000001
Pattern 10	1.00	0	1.09	0	0	0	0	0	0	0	0	0	00000000000000000001	00000000000000010000
Pattern 11	1.09	0	0	0	0	0	0	0	0	0	0	0	00000000000000010000	00000000000000000000
Pattern 12	0	0	0	0	0	0	0	0	0	0	0	0	00000000000000000000	00000000000000000000

Figure 5.1: Preprocessing of Amino Acid Sequence into Input Patterns

created, as shown in Pattern 12 in the Figure.

One of the fundamental principles of software engineering is the principle of *modularity*. When the solution to a problem is not designed in a modular manner but is instead *monolithic*, the solution is difficult for people to understand, implement and maintain [64]. The manner in which I have chosen to implement the Chou-Fasman algorithm is modular, building the implementation in phases. The organization of this chapter follows the phases in which the solution was developed. I first describe the implementation, training and testing of two sub-networks, one for finding helices and one for finding strands in the protein sequence. Each of these sub-networks can be used independently of the larger network solution. Next, I describe the synthesis of the two sub-networks. This synthesis involves the resolution of conflicts between the outputs of the two sub-networks as well as the prediction of coils as a default when neither a helix nor a strand is predicted. Finally, I describe various methods for augmenting the network architecture with knowledge about the context in which the prediction is being made along with additional knowledge for further refinement of the predictions.

The solutions to many complex problems are likely to be implemented in such a modular manner. For example, a system for visual processing is likely to have a module for finding edges of objects, another module for finding objects that are rectangular in shape, another module for finding objects that are spherical in shape, and so on. The system then must combine the outputs of the various modules. Finally, the system is likely to have at least one module that pulls the pieces together in a manner that takes the context of the particular situation into account. Although such a visual processing system is more complex than the protein secondary structure prediction system described here, many of the issues and solution principles are likely to be similar.

5.2 Helix Finding Expert Sub-Network

The first experiment to be performed is the creation and training of an expert network to detect helical amino acids in proteins. The topology of the helix finding expert network is shown in Figure 4.7 and is derived from the Chou-Fasman rules for finding helices shown in Chapter 2. In Figure 4.7, hard synapses are represented by solid lines and soft synapses are represented by dashed lines. The weights on all synapses are set initially to 1.0 to determine the effectiveness of the rules before refinement by the data. The network is then trained with a learning rate of .0001 for 2000 iterations. The results of sending the training and test sets through the network before and after training are shown in Figure 5.2.

The first row in Figure 5.2, labeled *Positive*, shows the number of helical amino acids incorrectly predicted to be non-helix over the total number of helical amino acids. The second row, labeled *Negative*, shows the number of non-helical amino acids incorrectly predicted to be helix over the total number of non-helical amino acids. The number of helical amino acids incorrectly predicted to be non-helix increases as the network is trained while the number of non-helical amino acids incorrectly predicted to be helical decreases as the network is trained. In fact, if training continues for a long enough period of time, the network eventually predicts all amino acids to be non-helical because there are 4124 non-helical amino acids (strands plus coils) compared to only 1965 helical amino

	Untrained Net Training Pats	Trained Net Training Pats	Untrained Net Testing Pats	Trained Net Testing Pats
Helix Finding				
Positive	*216/1965*	*640/1965*	*96/849*	*292/849*
Negative	*3054/5029*	*1575/5029*	*1837/2875*	*1045/2875*
Pred Error	*.467544*	*.316700*	*.519066*	*.359023*
Actual Error	*.467544*	*.232770*	*.519066*	*.262498*
Correlation	*.274045*	*.329456*	*.226055*	*.247928*
Strand Finding				
Positive	*336/1135*	*485/1135*	*230/746*	*350/746*
Negative	*2605/5859*	*1735/5859*	*1292/2978*	*912/2978*
Pred Error	*.420503*	*.317415*	*.408700*	*.338883*
Actual Error	*.420503*	*.246736*	*.408700*	*.246535*
Correlation	*.191317*	*.215107*	*.206484*	*.188305*
Nishikawa	**Helix Coefficient: .25** **Strand Coefficient: .19** **Coil Coeficient: .24**			

Figure 5.2: Separate Helix and Strand Finding Expert Network Results

acids. The ΔW averaging technique mitigates this tendency but eventually the non-helices still overwhelm the helices. Therefore, one must be careful to stop the training of the network when its predictive ability on the various types of structures is fairly balanced, in other words, the network does not predict one structure much more accurately than another structure.

In Figure 5.2, the Prediction Error is shown in the third row, labeled *Pred Error*. Prediction Error is the percentage of all amino acids incorrectly predicted. In this experiment, if the amino acid has been determined to actually take on a helical conformation in the folded protein, a correct prediction, in the sense of Prediction Error, occurs if the output of the output node is above 0.6 while if the amino acid is not a helical one, a correct prediction occurred if the output of the output node is below 0.6. I determined the threshold of

0.6 through experimentation. The Actual Error, labeled *Actual Error*, is the difference between the actual output and the ideal output for the network over all the patterns. That is, if the ideal output, I, for a particular pattern is 1.0 and the actual output, O, for that pattern is 0.322, then the actual error for this pattern is $(I - O)^2$, or 0.460. To determine the actual error for the entire network, I summed the actual error for each pattern and divided the sum by the number of patterns, that is,

$$A = \frac{\sum_{i=1}^{N}(I_i - O_i)^2}{N}, \tag{5.1}$$

where A is the actual error and N is the number of patterns. Finally, in the Figure 5.2, the Correlation Coefficient, labeled *Correlation*, shows the correlation between the observed structure of the amino acids in the protein and the predicted structure in the protein (see Section 4.3.7) [63]. Again, the predicted structure is helical if the output value for the output node is above 0.6 and non-helical if it is below 0.6.

5.3 Strand Finding Expert Sub-Network

The next experiment performed is the creation and training of an expert network to detect strand amino acids in proteins. The strand finding expert network is shown in Figure 5.3. Again, the topology of this network is derived from the Chou-Fasman rules for finding strands shown in Chapter 2. The training of this network proceeds in the same manner as that of the helix finding network. Again, the results of sending the training and test sets through the network before and after training are shown in Figure 5.2. The method for determining a correct strand or not strand prediction is the same as that described for the helix finding sub-network.

By comparing the correlation coefficients for the helix and strand finding networks, one can see that helices are easier to predict than strands. Individually, these networks, without benefit of information about helix and strand initiation from scanning the protein in the opposite direction, predict helix and strand amino acids as well as the Chou-Fasman algorithm implemented in Nishikawa [37] and shown for comparison at the bottom of Figure 5.2.

5.4 Synthesis of the Two Sub-Networks

A network for finding helices is of little practical use in predicting the secondary structure of actual proteins without some integration of the network into a larger system for prediction of all secondary structure. The same holds true for the strand finding network. Therefore, the helix finding and strand finding expert network predictions must be combined to create a system which predicts overall secondary structure for proteins. The most difficult part of this task comes in determining how to resolve conflicts between the outputs of the two networks. Figure 5.4 shows the overall scheme for combining the helix and strand finding sub-networks. The details for the various options for implementing the synthesis and refinement sub-networks are discussed below.

Figure 5.5 shows the results of all the combination methods described below. In this figure, the number labeled H represents the correlation coefficient for helices (see Section 4.3.7), which is also sometimes called C_α, S represents the correlation coefficient for strands, also called C_β, and C represents the correlation coefficient for coils, also called C_{coil}. The final column in the figure shows

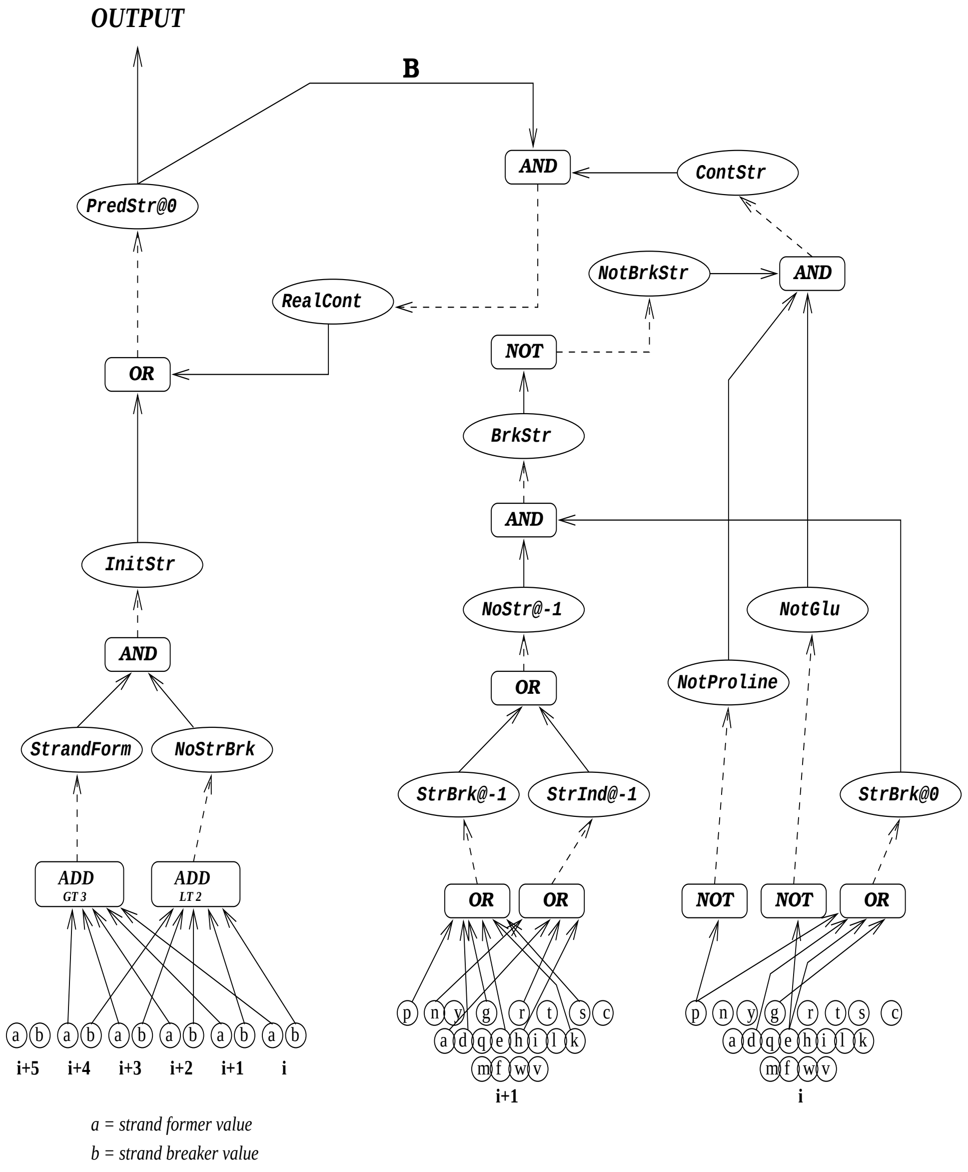

Figure 5.3: Strand Finding Expert Network

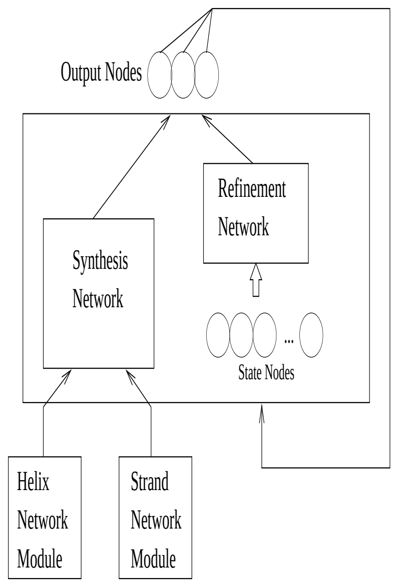

Figure 5.4: Overall Scheme for Prediction Expert Network

the average correlation coefficient of the trained network on the test dataset for all three secondary structure types. For a comparison with a representative implementation of the Chou-Fasman algorithm, the numbers reported in Nishikawa [37] are also shown at the bottom of the figure. The implementation in Nishikawa [37] results in an average correlation coefficient of .226667 over all three secondary structural types. This correlation coefficient is typical of other algorithmic implementations of the Chou-Fasman rules [40] [41].

5.4.1 Simple Combine

The first method for combining the outputs of the helix and strand finding expert networks that I developed is called the *Simple Combine* method. In the simple combine method, the output files of the two networks are read by a C program and the prediction is created in a non-network manner, that is, with no further learning from data occurring. The algorithm utilized in this method

Experiment	Untrained Net Training Pats	Trained Net Training Pats	Untrained Net Testing Pats	Trained Net Testing Pats	Average
Helix Finding	.274	.329	.226	.248	N/A
Strand Finding	.191	.215	.206	.188	N/A
Simple Combine	H: .145 S: .203 C: .187	H: .239 S: .204 C: .231	H: .117 S: .202 C: .223	H: .190 S: .253 C: .293	.245
User Nodes Combine Nodes	H: .247 S: .139 C: 0.0	H: .297 S: .075 C: .254	H: .171 S: .134 C: 0.0	H: .217 S: .095 C: .212	.175
User Nodes No Combine Nodes	H: .247 S: .217 C: .305	H: .247 S: .217 C: .305	H: .171 S: .266 C: .293	H: .171 S: .266 C: .293	.243
No User Nodes No Combine Nodes	H: .061 S: .198 C: .154	H: .259 S: .187 C: .244	H: .071 S: .208 C: .149	H: .238 S: .125 C: .204	.189
Comparison Values from Nishikawa	H: .25 S: .19 C: .24				.227

Figure 5.5: Protein Folding Expert Network Results

```
H = output of helix finding network
S = output of strand finding network

if (H< 0.6) and (S < 0.6)
   predict coil
else
if (H >S )
   predict helix
else
   predict strand
```

Figure 5.6: Simple Combine Algorithm

is shown in Figure 5.6. The threshold of 0.6 was determined experimentally. The idea of predicting the secondary structure with the highest output value has some basis in the original Chou-Fasman rules [38], which states that when $\langle P_\alpha \rangle > \langle P_\beta \rangle$, a helix should be predicted while if $\langle P_\beta \rangle > \langle P_\alpha \rangle$, a strand should be predicted. With an average correlation coefficient of .245055 on the test set after training, the simple combine method compares favorably with the results reported in [37] suggesting that the lack of information from moving along the protein in the opposite direction can be overcome.

Because the work described in this dissertation concerns the addition of knowledge about longer-range amino acid interactions to the Chou-Fasman algorithm, I am interested in a network approach to the synthesis of the outputs of the helix and strand finding sub-networks. The manner in which these network predictions are to be combined is not obvious and I therefore have implemented several options in the synthesis sub-network shown in Figure 5.4.

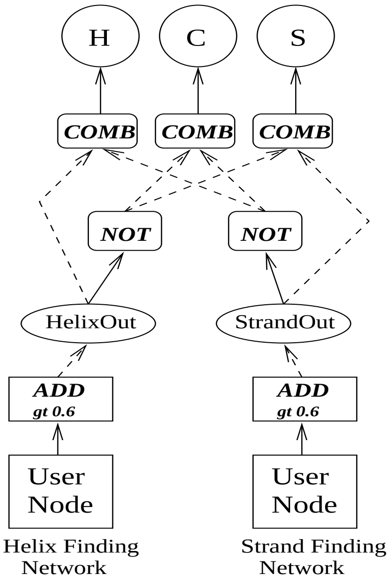

Figure 5.7: Network Using User Nodes and Combine Nodes

5.4.2 User Nodes, Combine Nodes

First, after training the helix and strand finding networks, I created a user node for each of these networks. The user nodes are then incorporated into a larger network scheme, shown in Figure 5.7. The connections that are shown as solid lines in the figure are hard synapses and cannot be learned while those shown as dashed lines are soft synapses and can be learned. The weights on all synapses are initialized to 1.0. The results for this particular synthesis alternative is shown in Figure 5.5 in the row labeled *User Nodes, Combine Nodes*.

Before training, this alternative predicted no coils, indicated by the fact that the correlation coefficient for coils, C, is 0.0 for both the training set and the test set. Figure 5.8 illustrates the reason the untrained network predicts no coils. The column labeled *HelixOut* in Figure 5.8 shows the value output from the *HelixOut* node in Figure 5.7 while the column *StrandOut* corresponds to the

INPUT		OUTPUT		
HelixOut	*StrandOut*	H	C	S
0	0	1	1	1
1	0	1	1	0
0	1	0	1	1
1	1	1	0	1

Figure 5.8: Combination Nodes Inputs and Outputs

StrandOut node output. The columns labeled H, S, and C correspond to the three output nodes in the network shown in Figure 5.7. To see how the values in the figure are derived, refer to the first row of Figure 5.8, where the value from the *HelixOut* node is zero and the value from the *StrandOut* node is also zero. This situation corresponds to a prediction of neither helix by the helix finding network nor strand by the strand finding network. Intuitively, one would expect a prediction of coil from this situation. In this case, however, a strand is predicted. The `CCombineNode` object connected to the helix output node, H in Figure 5.7, receives as input a zero from the *HelixOut* node. The value coming from the `CNotNode` object connected to the *StrandOut* node inputs a one to the H node. The combining function of the `CCombineNode` object combines these values so that the internal state of the `CCombineNode` object is one. The output node, H, because the weight on the synapse between the `CCombineNode` object and the H node is 1.0 and the H node is linear, has an output value of one, as shown in the H column of Figure 5.8. By the same reasoning, the output value of the S node is also one. The `CCombineNode` object connected to the C node receives a one as input from the `CNotNode` object connected to the *HelixOut* node and a one from the `CNotNode` object connected to the *StrandOut* node. The combining function of the `CCombineNode` object starts with an internal state of zero. Each of the input values is combined with the internal state using $newInternalState = oldInternalState + InputValue - (oldInternalState * InputValue)$. After receiving input from the `CNotNode` object connected to the *HelixOut* node, the internal state of this `CCombineNode` object is 1.0. Receiving input from the `CNotNode` object connected to the *StrandOut* node results in $newInternalState = 1.0 + 1.0 - (1.0 * 1.0) = 1.0$. Therefore, the output value of the C node is also one. Since all three output nodes have the same value, a strand is predicted by default. The decision to predict strand in this situation could be changed easily but I chose to predict strand by default because strands are typically more difficult to predict than helices or coils.

After training for 4000 iterations, this network predicts helices and coils reasonably well but is not good at predicting strands, with an average correlation coefficient of .174624. Because of the above analysis concerning the `CCombineNode` objects, I decided to attempt a synthesis of the network outputs without the `CCombineNode` objects. Without the `CCombineNode` objects, more of a distinction between helices, strands and coils in the untrained network can be made.

5.4.3 User Nodes, No Combine Nodes

The results of training the network shown in Figure 5.9 for 1000 iterations can be seen in the row labeled *User Nodes, No Combine Nodes* in Figure 5.5. Before

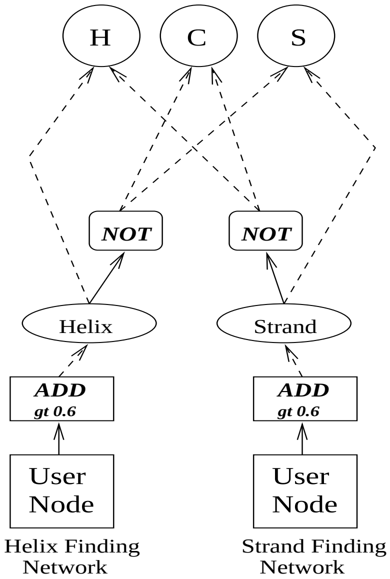

Figure 5.9: Network Using User Nodes and No Combine Nodes

training, this network is relatively good at predicting all three secondary structure types, with an average correlation coefficient of .256421 on the training set. Although the actual error decreases during training of 1000 iterations, the correlation coefficients do not change. For the test set, the trained network implementing this combination method results in an average correlation coefficient of .243338. This value compares reasonably well with the coefficient from the simple combine method and is higher than the average value from the Nishikawa [37] study.

5.4.4 No User Nodes, No Combine Nodes

The next combination method eliminates both the user nodes and the combine nodes. The resulting network is shown in Figure 5.10. This network is created by training the helix and strand finding networks, shown in Figures 4.7 and 5.3,

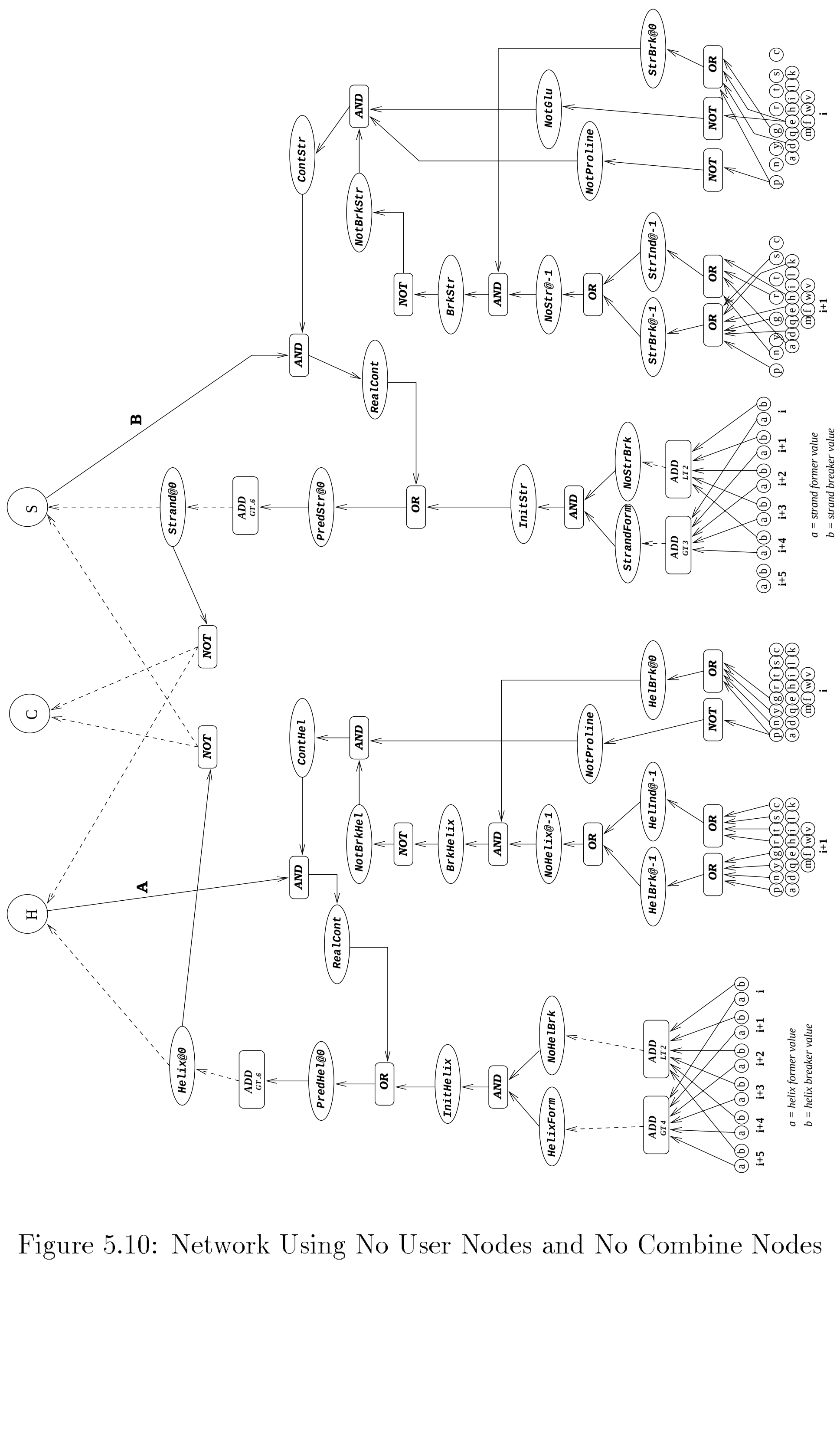

Figure 5.10: Network Using No User Nodes and No Combine Nodes

as described above. Once these sub-networks are trained, their soft synapses are changed to hard synapses to prevent further learning on these connections. In addition, the synapse labeled A in Figure 4.7 is replaced by the synapse labeled A in Figure 5.10. The effect of changing these connections is to change the feedback about the previous prediction from the output of the helix finding sub-network to the output of the helix output node of the entire network. In addition, the synapse labeled B in Figure 5.3 is replaced by the synapse labeled B in Figure 5.10 with the same effect for the strand prediction as described for the helix prediction. The resulting network is then trained using the training set for 1000 iterations. The prediction results of this network are shown in Figure 5.5 in the row labeled *No User Nodes, No Combine Nodes*. On the test set, the trained network has an average correlation coefficient of .189146. This value compares unfavorably with the simple combine method and with the method using user nodes and no combine nodes.

The replacement of the recurrent connections in this network causes the overall prediction accuracy to drop since the sub-networks were trained using feedback from only the sub-networks themselves but the prediction is occurring using feedback from the overall prediction network. In other words, the weights on the synapses were trained expecting one type of feedback but then for prediction, these synapses receive a different kind of feedback. For example, for the first amino acid in the training dataset, the feedback from the helix finding sub-network is -0.0791941, which means definitely do not predict a helix for this amino acid. For this same amino acid in the same context, the feedback from the overall prediction network is 1.03562 because knowledge from the strand finding subnetwork (which has a feedback for prediction of a strand of -0.0585148, meaning definitely do not predict strand) has been added. The weights, now fixed, have seen this context before but with a different feedback value and thus, the predictive accuracy is lowered. Changes in the input or output node for a synapse after learning occurs is therefore not advisable. As a result of the lowered prediction accuracy, subsequent experiments discussed in this dissertation use networks whose connections have not had either their input or their output nodes changed after the initial training of the expert subnetworks and before the combining of the subnetworks into the larger expert network.

5.5 Adding State Nodes

As discussed by many researchers, including Chou and Fasman [21], the Chou-Fasman algorithm takes only short-range amino acid interactions into account during prediction. In this section, I discuss several options for adding knowledge about longer-range amino acid interactions to the expert network encoding the Chou-Fasman rules. The network to which state information is to be added is the network discussed in Subsection 5.4.3, which is the combination network with the highest prediction accuracy.

5.5.1 Experiment 1: State Nodes and Equal Add Node Thresholds

Figure 5.11 shows the first method to incorporate state information into the network for secondary structure prediction. For each of the possible secondary structures, there are three state nodes. These state nodes are similar to the state units in Jordan networks [17]. The outputs of each set of state nodes are gathered together into a `CAddNode` object with a threshold of 1.0. The outputs

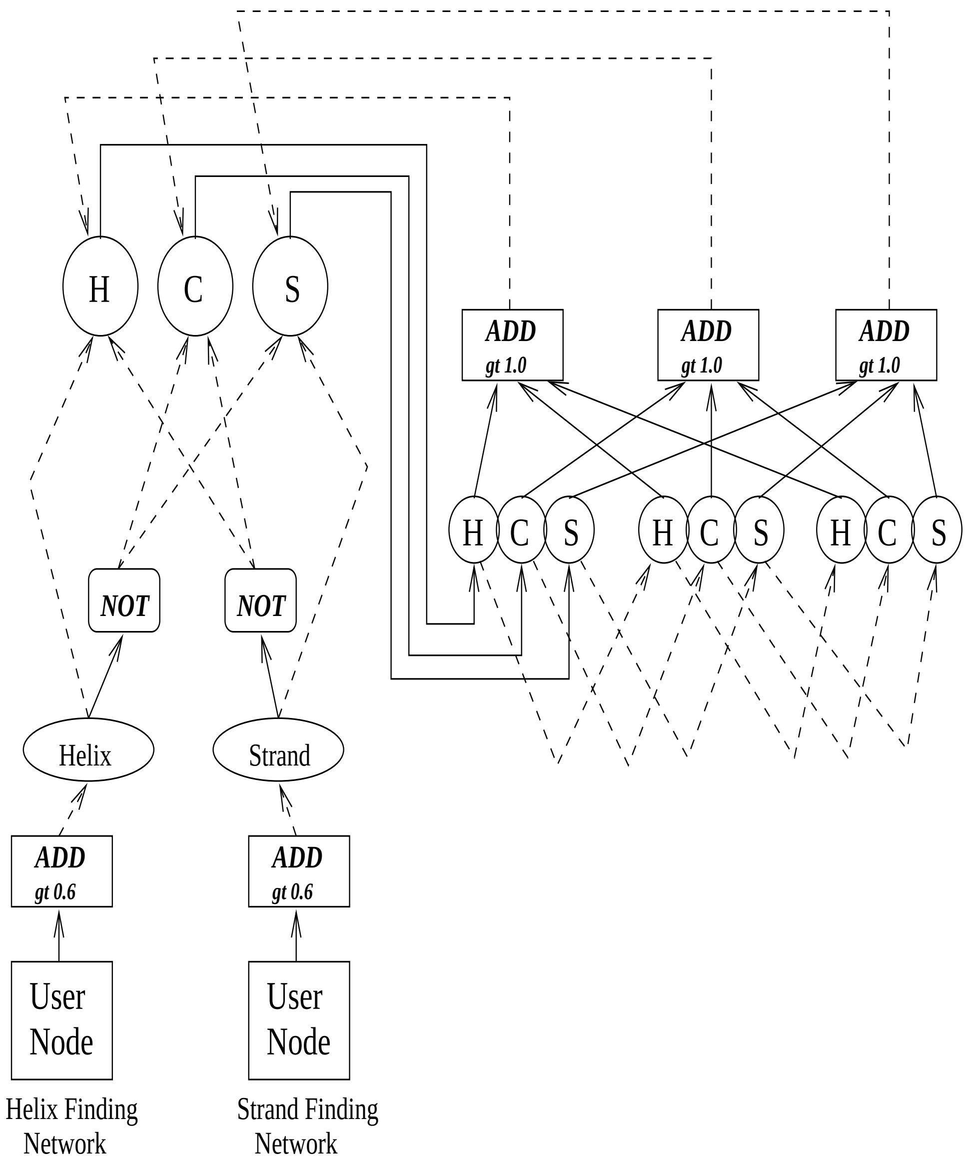

Figure 5.11: Prediction Network Incorporating State Information

of the `CAddNode` objects are 1.0 if the inputs sum to a value greater than 1.0 and 0.0 if not. These output values are passed along a soft synapse into the output node representing the secondary structure corresponding to the inputs to the `CAddNode` object. The output nodes also receive input concerning the current prediction for the current amino acid. The recurrent connection comes from the output nodes of the network to the first state node for each structure. These recurrent connections feed previous prediction information back into the output nodes of the network. The results of activation of this network, trained for 1000 iterations, with the training and test datasets are shown in Figure 5.12 in the row labeled *Experiment 1, 1000 iterations*. The average correlation coefficient (see Section 4.3.7) for the trained network on the test dataset is .241316. I then trained the network for another 1000 iterations, the results of which are seen in Figure 5.12 in the row labeled *Experiment 1, 2000 iterations*. The average correlation coefficient for all structures dropped to 0.0 because after 2000 iterations, this network predicts every amino acid to be coil. The overprediction of coil occurs because there are so many more examples of coils in the training dataset than there are of helices or strands. Recall that the technique of ΔW averaging serves to mitigate this effect but does not eliminate it completely.

5.5.2 Experiment 2: Increasing the Add Node Thresholds

In the second version of the network incorporating state information, I increased the threshold on the ADD nodes from 1.0 to 2.5. The results of activating this network with the training and testing datasets are shown in Figure 5.12 in the

Experiment	Untrained Net Training Pats	Trained Net Training Pats	Untrained Net Testing Pats	Trained Net Testing Pats	Average
Experiment 1 1000 iterations	H: .247 S: .217 C: .305	H: .307 S: .218 C: .281	H: .171 S: .266 C: .293	H: .245 S: .203 C: .275	.241
Experiment 1 2000 iterations	H: .247 S: .217 C: .305	H: 0.0 S: 0.0 C: 0.0	H: .171 S: .266 C: .293	H: 0.0 S: 0.0 C: 0.0	0.0
Experiment 2 500 iterations	H: .247 S: .217 C: .305	H: .247 S: .217 C: .305	H: .171 S: .266 C: .293	H: .171 S: .266 C: .293	.243
Experiment 2 1000 iterations	H: .247 S: .217 C: .305	H: 0.0 S: .179 C: .047	H: .171 S: .266 C: .293	H: 0.0 S: .121 C: .058	.060
Experiment 3	H: .247 S: .140 C: 0.0	H: .307 S: .220 C: .305	H: .171 S: .134 C: 0.0	H: .247 S: .202 C: .293	.247
Experiment 4	H: .247 S: .140 C: 0.0	H: .307 S: .220 C: .305	H: .171 S: .134 C: 0.0	H: .247 S: .202 C: .293	.247
Experiment 5	H: .234 S: .134 C: 0.0	H: .171 S: .266 C: .293	H: .172 S: .129 C: 0.0	H: .231 S: .214 C: .290	.245
Experiment 6 100 iterations	H: .242 S: .143 C: 0.0	H: .315 S: .214 C: .310	H: .169 S: .147 C: 0.0	H: .254 S: .192 C: .289	.246
Experiment 6 500 iterations	H: .242 S: .143 C: 0.0	H: .242 S: .217 C: .310	H: .169 S: .147 C: 0.0	H: .169 S: .266 C: .289	.241
Comparison Values from Nishikawa	H: .25 S: .19 C: .24				

Figure 5.12: Results of Incorporating State Information

two rows labeled *Experiment 2, 500 Iterations* and *Experiment 2, 1000 Iterations*. When trained for 500 iterations, no change in the correlation coefficients occurs when compared to the untrained network correlation coefficient. That is, on the test set, the average correlation coefficient is .243338. When trained for 1000 iterations, the trained network has a correlation coefficient of 0.0 for helices because this network predicts no helices. This fact is an indication that the network has been trained for too many iterations. I then started over, iteratively training the untrained network, and found that the correlation coefficients do not change from the coefficients for the untrained network until after iteration 880. At that point, the correlation coefficient immediately drops from .246832 to 0.0 for helices. Examination of the network output throughout these experiments shows that after each iteration, the value of the output node representing coil increases for each pattern while the value of the output node representing helix decreases. At the moment of the significant drop in the helical correlation coefficient, the output value of the coil output node has become just slightly larger than the output value of the helix output node for the amino acids which had previously been predicted helical. All amino acids that had previously been predicted helical are predicted to be coil after the drop in the helix correlation coefficient occurs. These experiments suggest that perhaps coil is too easy to predict and should be made more difficult. The next experiment represents an attempt to make coil more difficult to predict than helix or strand.

5.5.3 Experiment 3: Requiring More Evidence for Coil Prediction

Figure 5.13 shows the third version of the network that incorporates state information. In this network, because of the observation that the prediction of coil begins to overwhelm all the other predictions during the course of training, I increased the threshold for the `CAddNode` object for the coil state nodes to 6.0 while leaving the thresholds for helices and strands at 2.5. The change is marked in Figure 5.13 with the words **Changed Threshold**. The increased threshold forces a higher level of evidence to be present in order for the network to predict coil. The results of using this network for prediction are shown in Figure 5.12 in the row labeled *Experiment 3*. When untrained, this network does not predict any coils. Therefore, the correlation coefficients in the untrained network are among the worst of any that have been discussed. On the other hand, after 500 training iterations, the correlation coefficients on the training set, with an average of .277284, are better than any other network discussed thus far. On the test set, the trained network has an average correlation coefficient of .247282, again better than any other trained network discussed. Recall that the network without state information which resulted in the highest average correlation coefficient was the network labeled *User Nodes, No Combine Nodes* (see Section 5.4.3), with a coefficient of .243338 on the test set. Although the difference between these two coefficients is small, a rough estimate using Fisher's arcTan transform [65] shows that it is significant at the .075 level.

The fact that the network described in this section has a higher average correlation coefficient than any of the networks without state information suggests that state information is useful in increasing prediction accuracy. Continuing

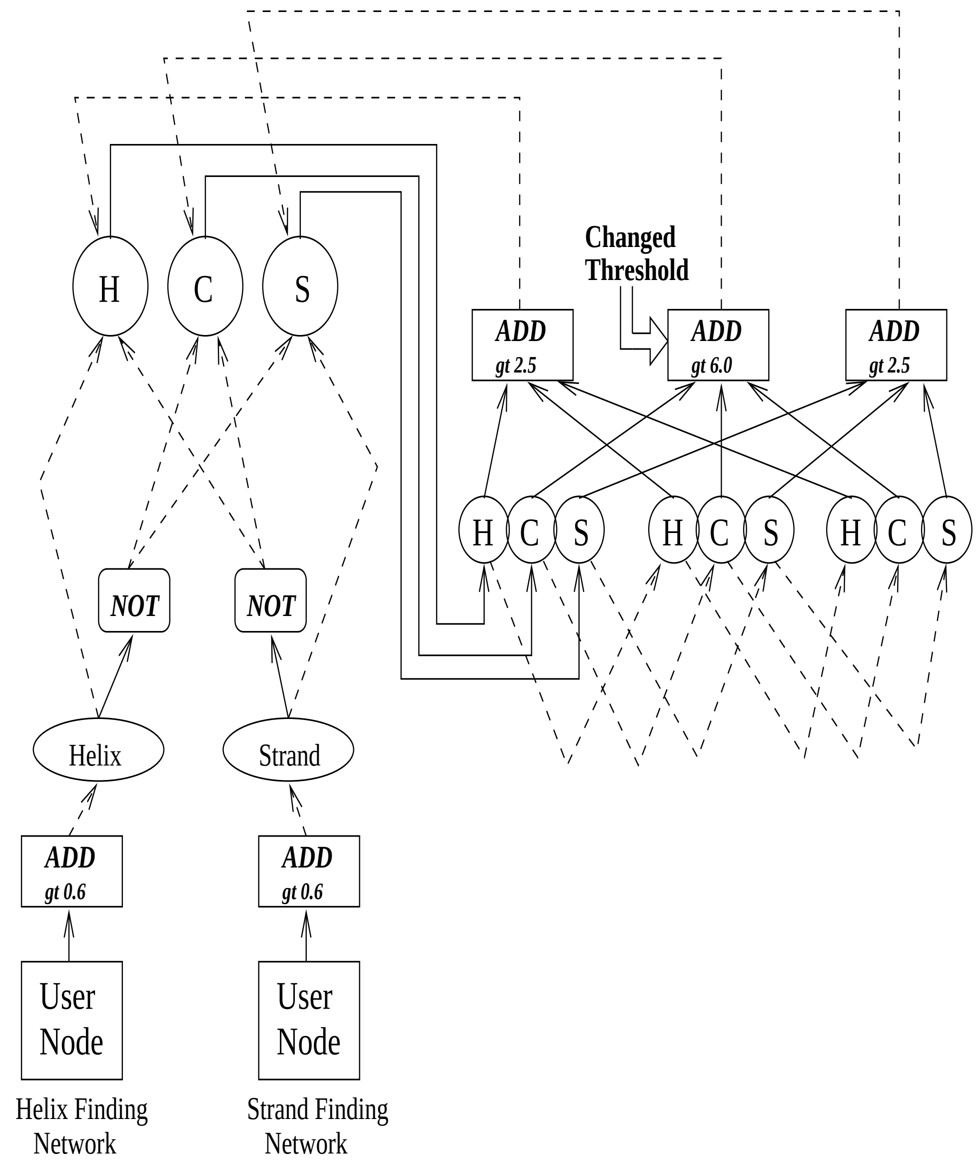

Figure 5.13: Third Prediction Network Incorporating State Information

to train the network for another 500 iterations reveals that the correlation coefficients for the network trained for 1000 iterations are unchanged from the network at 500 iterations.

5.5.4 Experiment 4: Requiring More Evidence for State Node Prediction of Helices

Experiment 3 illustrated that by varying the threshold values for the `CAddNode` objects connected to the state nodes, one can increase the predictive capability of the overall network. In this experiment, the value of the threshold of the `CAddNode` object for the helix state nodes is changed to 4.0, the threshold of the `CAddNode` object for the strand state nodes remains at 2.5 and the threshold of the `CAddNode` object for the coil state nodes remains at 6.0. Again, the change is marked in Figure 5.14 with the words **Changed Threshold**. The semantics of this threshold change are that more evidence for a helix is needed for a helix to be predicted than the amount of evidence required for a strand to be predicted. The results of activating the training and testing datasets through the untrained network and the network trained for 500 iterations are shown in Figure 5.12 in the row labeled *Experiment 4*. These results are identical to the results from the network in Experiment 3.

5.5.5 Experiment 5: Requiring More Evidence for Helix Finding Subnetwork Prediction of Helices

Another method for making helices harder to predict is shown in Figure 5.15. In this experiment, rather than changing the threshold value of one of the `CAddNode`

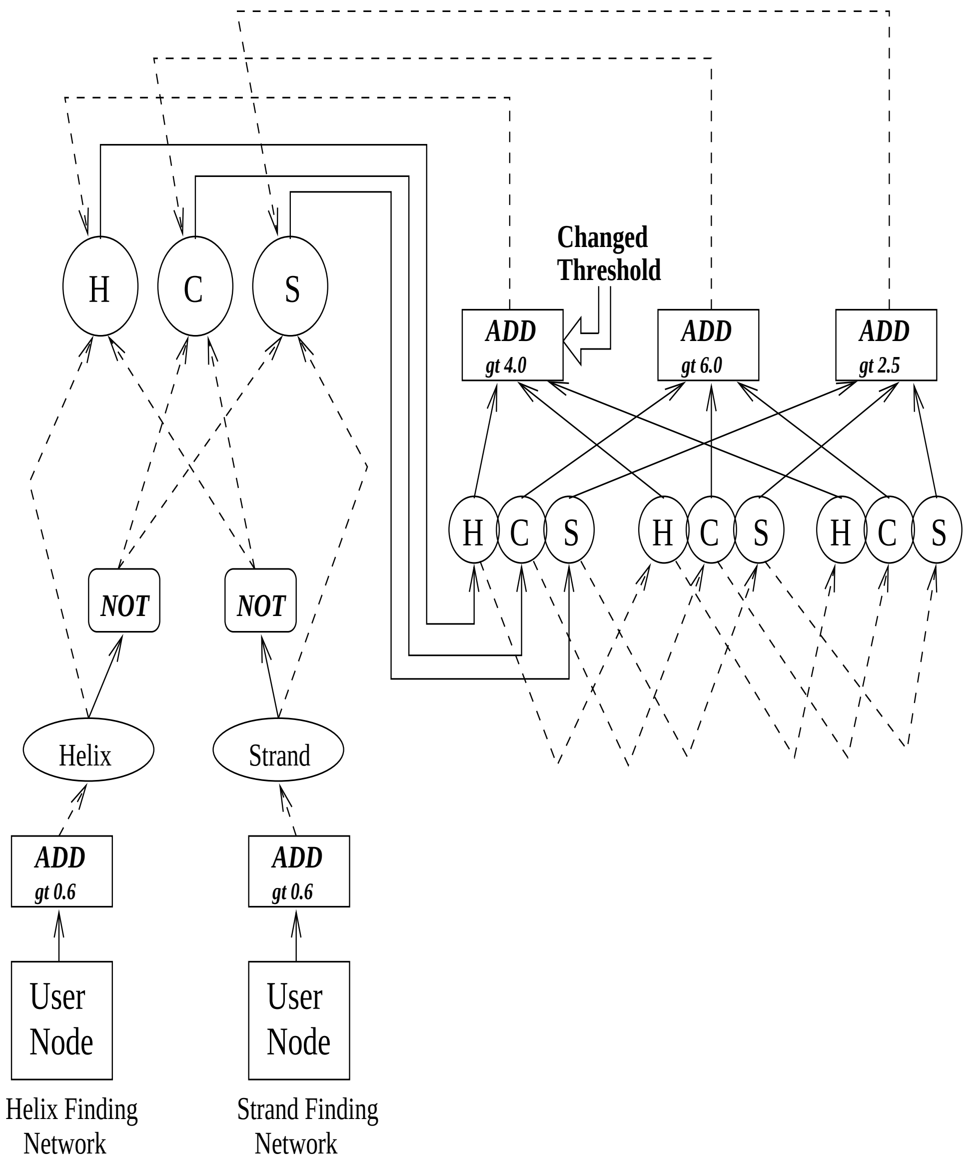

Figure 5.14: Fourth Prediction Network Incorporating State Information

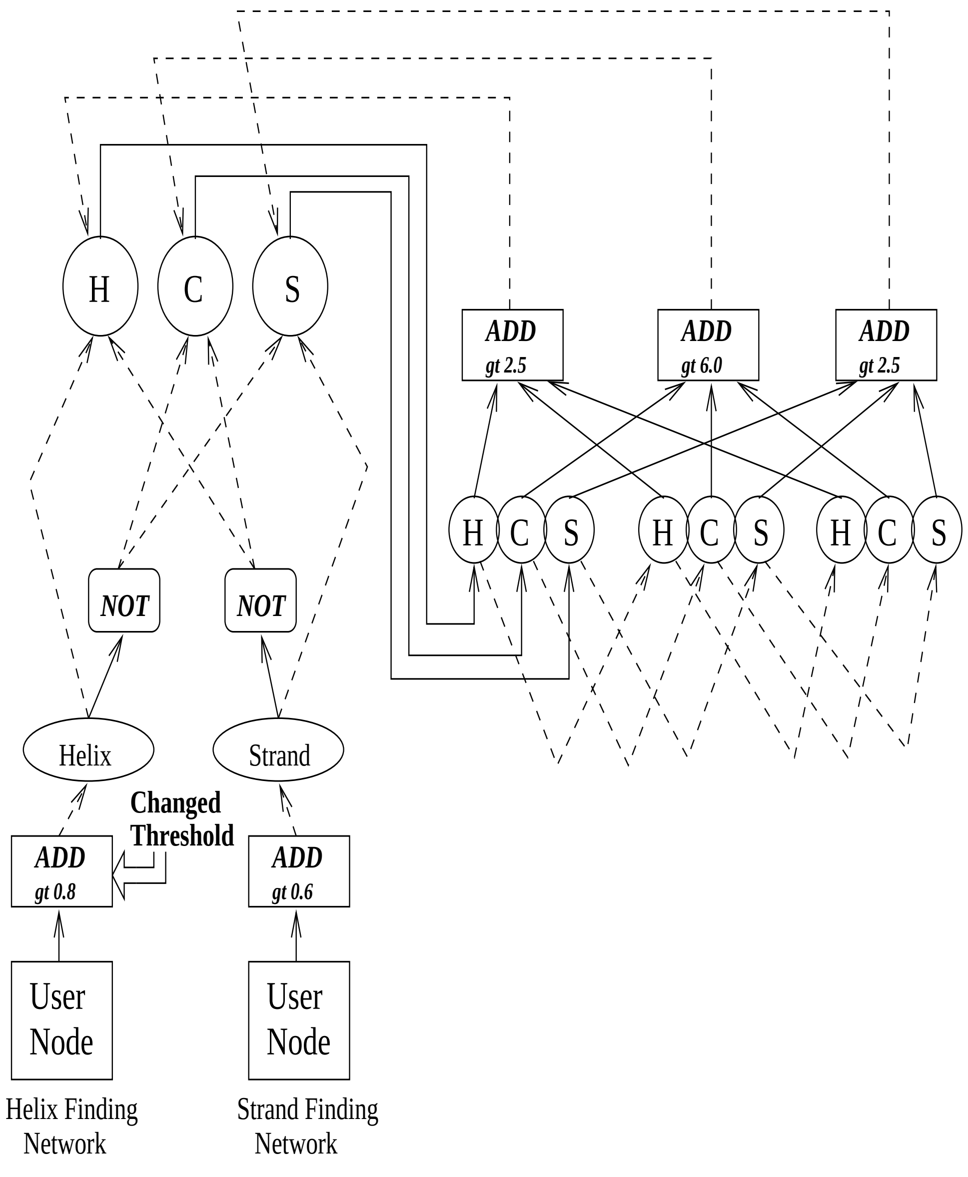

Figure 5.15: Fifth Prediction Network Incorporating State Information

objects connected to the state nodes, the threshold of the `CAddNode` object connected to the output of the helix-finding subnetwork is changed. This threshold value is changed from 0.6 to 0.8 and the network is trained for 500 iterations. The prediction results for this network are shown in Figure 5.12 in the row labeled *Experiment 5*. The trained network has an average correlation coefficient of .245027 on the test dataset.

The network in this experiment has a lower average correlation coefficient on the test dataset than the network used in Experiment 4 although both networks attempt to make helices harder to predict. The difference between the two networks is in semantics. In the network in Experiment 4, less evidence is required to predict a helix from the helix finding subnetwork than that required in this experiment. On the other hand, once a helix has been predicted from the helix finding subnetwork, the network in this experiment keeps knowledge of that helical prediction, whether right or wrong, in its state nodes for a longer time than does the network in Experiment 4. The incorrect predictions of helix remaining in the state nodes for a longer period of time are likely to be the cause of the network in this experiment having a lower average correlation coefficient. The long-term effect of these incorrect predictions is the downside to maintaining information about the state of the problem solution as the solution proceeds.

5.5.6 Experiment 6: Requiring Less Evidence for Helix Finding Subnetwork Prediction of Helices

A final experiment at maintaining state information during the course of the problem solution attempts to make helices easier to predict by changing the `CAddNode` object threshold value discussed in Experiment 5 to 0.4. The resulting network is shown in Figure 5.16. The results of this experiment are shown in Figure 5.12, in the two rows labeled *Experiment 6, 100 iterations* and *Experiment 6, 500 iterations*. The network trained for 100 iterations has an average correlation coefficient of .245509 on the test dataset. The network trained for 500 iterations has a lower average correlation coefficient of .241228. The lower average comes entirely from the fact that as training proceeds, the helix correlation coefficient decreases back to the value it had before any training occurred. The strand correlation coefficient has increased during this time but not quite as much as the helix coefficient has decreased. The coil correlation coefficient remains the same when the training iterations are increased from 100 to 500.

Examination of the prediction for several individual amino acid positions in the test dataset illuminates the reasons for the changes in the correlation coefficients seen in this experiment. Figure 5.17 shows the values of the output nodes for three amino acids from the first protein in the benchmark test dataset. The observed secondary structure for the amino acid at position 4 is coil. For the amino acid at position 7, the observed secondary structure is strand and finally, for the amino acid at position 13, the structure is coil.

In the untrained network, all three amino acids produce the same output values at all three output nodes. In Figure 5.16, the input connections to the

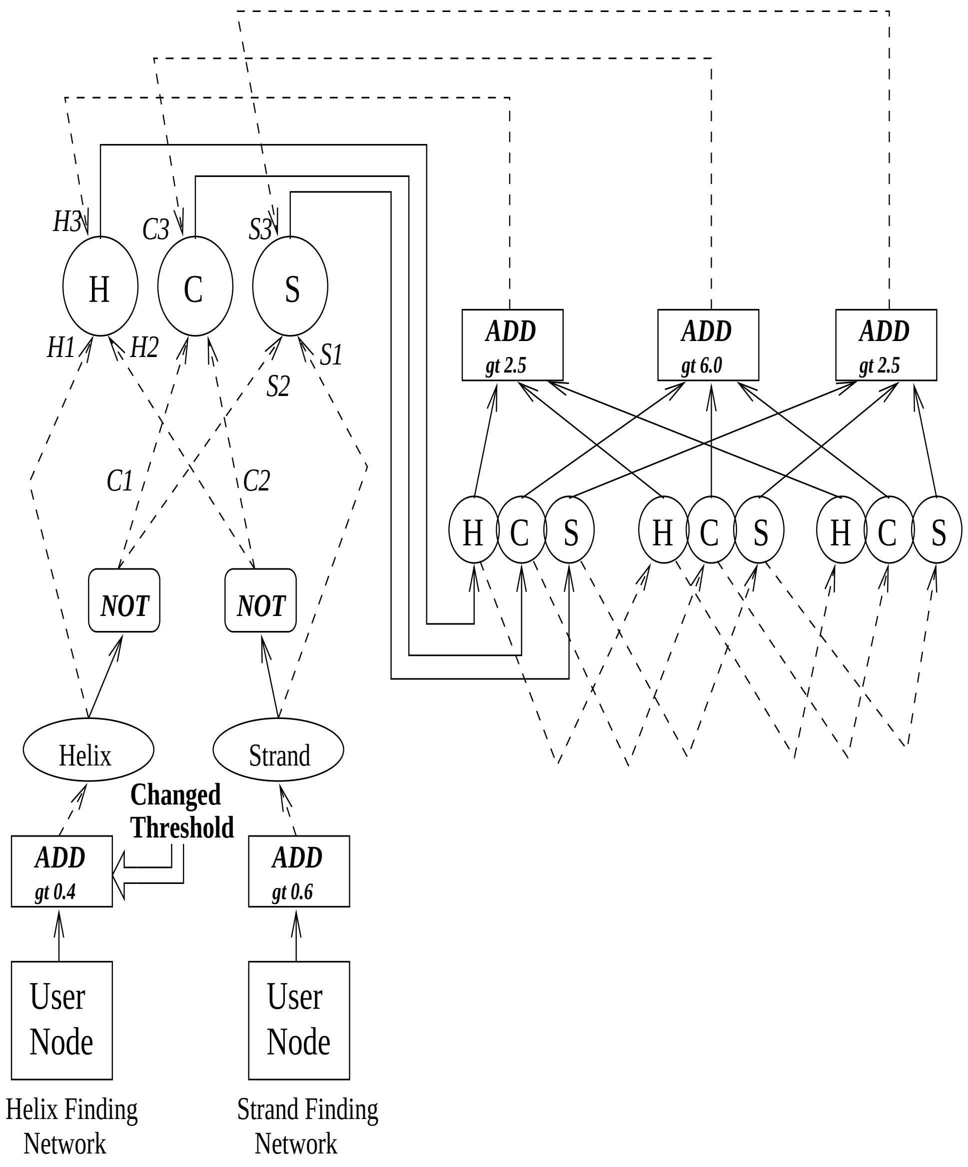

Figure 5.16: Sixth Prediction Network Incorporating State Information

Untrained Network			
	Position 4	Position 7	Position 13
Helix Output	2.0	2.0	2.0
Strand Output	2.0	2.0	2.0
Coil Output	0.0	0.0	0.0
Trained Network - 100 iterations			
	Position 4	Position 7	Position 13
Helix Output	1.96729	1.96729	1.97368
Strand Output	0.994407	0.994407	0.984293
Coil Output	0.130348	0.130348	2.081230
Trained Network - 500 iterations			
	Position 4	Position 7	Position 13
Helix Output	0.899581	0.899581	0.918819
Strand Output	0.977351	0.977351	0.929918
Coil Output	0.529910	0.529910	2.309500

Figure 5.17: Output Values for Individual Amino Acids

helix output node are labeled $H1$, coming through a `CAddNode` object from the prediction of the helix finding subnetwork, $H2$, coming through a `CNotNode` object from the strand finding subnetwork and $H3$, coming through a `CAddNode` object from the state nodes for helices. The three inputs for the strand output node and the coil output node are similarly labeled. For the untrained network, the helix output node has a value of 2.0 for all three amino acid positions shown in Figure 5.17. This value comes from input connection $H1$, with a value of 1.0, suggesting that a helix should be predicted for the current amino acid, plus the input connection $H3$, also with a value of 1.0, suggesting that previous amino acid positions were predicted helical with high enough confidence. Input connection $H2$ has a value of 0.0, suggesting that the strand finding subnetwork also predicts these amino acid positions to be strands. In fact, examination of the strand output node values confirms this suggestion. For all of these positions, strand is predicted because, given the same output value, strand is always chosen over helix.

In the network trained for 100 iterations, some distinctions begin to occur among the output node values for individual positions. Positions 4 and 7, despite the fact that they have differing observed structure, have output node values which are all the same. For positions 4 and 7, input synapse $H1$ has a high value, suggesting that the helix finding subnetwork predicts a helix at the current amino acid position, and the input synapse $S1$ also has a high value suggesting that the strand finding network predicts a strand at the current amino acid position. Both input synapses $H2$ and $S2$, thus, have low values. Input synapse $H3$ has a high value, suggesting that previous amino acids have been predicted to be helical with a high level of confidence but input synapse $S3$ has a low value, suggesting that previous amino acids have not been predicted to be strand with a high level of confidence. An interesting difference between positions 4 and 7 can be seen by examination of the distance between the amino acid and the last amino acid for which the strand finding subnetwork did not predict strand. The amino acids at positions 1 through 3 in the protein have low input synapse $S1$ values while for positions 4 through 7, this value is high. The proximity of position 4 to positions 1 through 3 explains the fact that for position 4, the $S3$ input synapse value is also low, suggesting that previous amino acids were not strongly predicted to be helical. Although position 7 is further away from positions 1 through 3, the effect of these amino acid predictions is still felt at position 7. Position 13 has low $H1$ and $S1$ input synapse values, suggesting that the helix finding subnetwork and the strand finding subnetwork both predict the current amino acid to have some other structure. These low values mean that both the $H2$ and $S2$ input synapses have a high value. The $H3$ input synapse also has a high value, suggesting that previous amino acids were strongly predicted to be helical, but the $S3$ input synapse has a low value suggesting that previous amino acids were not strongly predicted to be strand. The combination of these input values results in a helix output node value close to 2.0 and a strand output value close to 1.0. Because synapses $H1$ and $S1$ both have low values, synapses $C1$ and $C2$ both have high values. Synapse $C3$ has a low value. The combination of these values results in a coil output node value of close to 2.0 but because of the weights on the input synapses for that node, the output value is slightly higher than the value seen for the helix output node. Therefore, a coil is predicted by this network for position 13.

A similar analysis of the output nodes for network trained for 500 iterations explains the output node values for that network, as seen in Figure 5.17. It is interesting to note that for this network, for all three amino acids analyzed, the helix output node value has dropped significantly. By examining the weights on the synapses in the trained network, it is apparent that the network trained for 500 iterations has learned to weight the state information less than other information. By weighting this state information less, the prediction accuracy of the network drops. The weights on synapses $H1$, $H2$, and $H3$ in the trained network are all just about equal, with values between 0.96 and 0.97. The weights on the synapses between the state nodes, however, have widely varying values. Recall that all the weights began at 1.0. In the trained network, the weight on the synapse between the first helix state node and the second helix state node has dropped to 0.54. In the network trained for 100 iterations, the weight on this synapse is 0.83. The benefit of state information in prediction accuracy can be seen from this analysis.

5.6 Results on FKBP-type Proteins

The FKBP-type proteins are those proteins for which various researchers have reported a change in conformation when unfolding the protein in the laboratory [44] [43] [45]. These proteins all have a region in them which is typically predicted to have one conformation by prediction algorithms when in fact they have a different conformation in their observed structure. The area incorrectly predicted actually takes on the predicted conformation when the protein is unfolded in the lab. The change in conformation suggests that local amino acid interactions would cause the protein to form the predicted conformation but long-range interactions force a different conformation. Since most prediction algorithms take only local amino acid interactions into account, these algorithms are unable to correctly predict the conformation of these regions.

FKBP is a protein that has one of the incorrectly predicted regions of amino acids. Amino acids 97-107 are predicted to be helical but are observed to participate in a β-sheet in the folded conformation [43]. When the protein is unfolded in the laboratory, evidence suggests that the interresidue interactions holding the region in the strand conformation are broken, at which time the region folds into a helix. The broken interresidue interactions are made with amino acids that are far from amino acids 97-107 in the primary sequence, that is, they are long-range amino acid interactions. When the local amino acids are allowed to work alone, the region forms the structure predicted by algorithms which take only these local interactions into account during prediction.

Figure 5.18 shows the results of activating the networks described earlier in this chapter with the FKBP-type proteins. The number of iterations for each trained network are shown in parentheses. Note that the helix finding subnetwork does a poor job at correctly predicting helical amino acids and this fact is seen in every expert network which contains the helix finding subnetwork. In fact, every expert network shown in Figure 5.18 has significantly higher strand and coil correlation coefficients than helix correlation coefficient. This lower helix correlation coefficient likely results from the fact that the secondary structure distribution of this set of proteins is significantly different than the distribution found in an average set of proteins. In the test dataset on which the expert networks discussed in this work were trained, the distribution for the amino acid secondary structure is 24% helix, 21% strand and 55% coil. In the FKBP-type protein dataset, on the other hand, the distribution is 16% helix, 41% strand and 43% coil.

Figure 5.19 shows the prediction results on the region in FKBP that Logan has shown to undergo a conformational change during unfolding. The residues 97 through 107 are shown in the second row, labeled *Sequence*. The third row shows that the region is observed to be in a strand conformation. In the second column of the figure, below the observed structure, the predicted structure from a variety of experiments is shown. The three columns labeled *helix*, *strand* and *coil* show the numbers of amino acids participating in each structural type for which there is an error in prediction. The columns labeled *positive* show

Experiment	Untrained Net	Trained Net	Trained Net Average
Helix Finding	.150	.043	N/A
Strand Finding	.276	.301	N/A
Simple Combine	H: .151 S: .240 C: .442	H: .140 S: .282 C: .365	.262
User Nodes No Combine Nodes	H: .142 S: .302 C: .365	H: .142 S: .302 C: .365	.270
State Information Experiment 1 (1000)	H: .142 S: .302 C: .365	H: .128 S: .320 C: .289	.246
State Information Experiment 3 (500)	H: .142 S: -.031 C: 0.0	H: .115 S: .346 C: .365	.275
State Information Experiment 3 (1000)	H: .142 S: -.031 C: 0.0	H: .142 S: .302 C: .365	.270
State Information Experiment 5 (500)	H: .123 S: -.013 C: 0.0	H: .109 S: .346 C: .325	.260
State Information Experiment 6 (100)	H: .155 S: -.060 C: 0.0	H: .101 S: .292 C: .411	.268
State Information Experiment 6 (500)	H: .155 S: -.060 C: 0.0	H: .155 S: .302 C: .411	.286

Figure 5.18: Results on FKBP-type Proteins

Experiment	97 … 107	helix		strand		coil	
sequence	l v f d v e l l k l e						
actual structure	e e e e e e e e e e e	*positive*	*negative*	*positive*	*negative*	*positive*	*negative*
simple combine	e e h h h h h h h h -	**23/40**	**55/219**	**55/107**	**33/152**	**45/112**	**35/147**
unodes,no combs	e e e h h h h h h h -	**24/40**	**50/219**	**51/107**	**35/152**	**45/112**	**35/147**
state info #1 (1000)	e e e - h h h h h h -	**23/40**	**58/219**	**66/107**	**17/152**	**39/112**	**53/147**
state info #3 (500)	e e e h h h h h h h -	**22/40**	**66/219**	**57/107**	**23/152**	**45/112**	**35/147**
state info #3 (1000)	e e e h h h h h h h -	**24/40**	**50/219**	**51/107**	**35/152**	**45/112**	**35/147**
state info #5 (500)	e e e h h h h h h - -	**25/40**	**53/219**	**55/107**	**25/152**	**42/112**	**44/147**
state info #6 (100)	h h h h h h h h h h h	**19/40**	**85/219**	**67/107**	**19/152**	**46/112**	**28/147**
state info #6 (500)	e e e h h h h h h h h	**22/40**	**56/219**	**51/107**	**35/152**	**46/112**	**28/147**

Figure 5.19: Predictions on FKBP Region of Interest

the number of positive examples of the structural type that are predicted to participate in some other structural type. The columns labeled *negative* show the number of negative examples of the structural type that are predicted to participate in that structural type. For example, there are 40 amino acids of structural type *helix* and of these, for the *Simple Combine* method, 23 are predicted to be strand or coil. On the other hand, there are 219 amino acids which are not helical and of these, for the *Simple Combine* method, 55 are incorrectly predicted to be helical.

Experiment 6, with 500 iterations, produces the highest average correlation coefficient on the FKBP-type proteins. This fact is interesting because, as seen in Subsection 5.5.6, this network has learned to ignore information concerning state information and this network performed less accurate prediction than the same network trained for only 100 iterations on the benchmark protein dataset. The network from Experiment 6, trained for 500 iterations, also predicts coils extremely well, with a correlation coefficient of .410892 resulting from a very small number of false positive coils (that is, 28 of the 147 non-coil amino acids incorrectly predicted to be coil). It is also interesting to note that this network has the highest helix correlation coefficient for the FKBP-type proteins of any of the networks discussed in this work and the network is designed to make helices easier to predict (refer to Subsection 5.5.6). As seen in Figure 5.19, all of the networks discussed here predict the region shown to undergo a conformational change to be a primarily helical region, just as do most other prediction algorithms [43].

Examination of the output values for positions 97 through 107 in FKBP activated through the network from Experiment 6, trained for 100 iterations, yields some interesting information. The helix output node has a value for positions 97 through 99 of 1.97, receiving positive input from synapses $H1$, from the helix finding subnetwork, and $H3$, from the state nodes. For these same positions, the strand output node has a value of 0.99, receiving positive input from synapse $S1$, from the strand finding subnetwork. For these positions, the strand output node does not receive positive input from synapse $S3$, from the state nodes, suggesting that the network, although recognizing that the state information stored in the state nodes is important, does not maintain the longer-range state information necessary to predict this region of FKBP to be a strand. At position 100, the strand finding subnetwork begins to predict not strand, giving the strand output node a value for positions 100 through 107 of 0.01. This low value means that the helix output node, in addition to receiving positive input from synapses $H1$ and $H3$ as before, now also receives positive input from synapse $H2$ and has an output value of 2.94.

In the network from Experiment 6, trained for 500 iterations, the helix output node no longer receives positive input from the state nodes for the positions 97 through 99 and thus, has an output value of 0.90, arising strictly from its positive input from synapse $H1$. Recall that this network has learned to ignore the state information maintained in the state nodes. The strand output node also does not receive positive input for these positions from the state nodes and thus has an output value of 0.98. For these three positions, the strand finding subnetwork finds higher evidence for strand than the helix finding subnetwork finds for helix and thus, the overall network, ignoring state information, predicts strand. Beginning at position 100, the strand finding subnetwork no longer finds evidence for a prediction of strand and thus the strand output node has an output value of 0.06 for positions 100 through 107. The lack of evidence for strand provides the network with stronger evidence for the prediction of helix and thus, the helix output node value for positions 100 through 107 is 1.80, which comes from the positive input from synapses $H1$ and $H2$. Although the network trained for 100 iterations and the network trained for 500 iterations both predict helix for positions 100 through 107, it is clear that the network trained for 100 iterations has stronger evidence for the prediction of helix.

The fact that the network trained for 500 iterations has learned to ignore the state information stored by the state nodes combined with the fact that this network has a lower prediction accuracy on the benchmark dataset but higher prediction accuracy on the FKBP-type protein dataset than the network trained for 100 iterations suggests that the FKBP-type protein dataset requires the maintenance of a different type of state information than the benchmark dataset. The FKBP-type proteins undergo conformational changes as a result of their state information while it is unclear whether the state information causes such conformational changes in the benchmark proteins.

The networks described in this dissertation have all been trained using a gradient descent based learning algorithm. Bengio, *et al.* have shown that as the distance between the dependencies grows, gradient descent methods have a more difficult time learning them [66]. The difficulty arises because when a recurrent network stores information in a manner that is resistant to noise, the gradient with respect to past events very quickly becomes small when compared to the gradient with respect to more recent events [66]. In other words, as the time t between two events increases, the derivatives of the cost function with respect to system activations approach 0. This situation is called the problem of *vanishing gradients.* The dependencies found in FKBP and other proteins like it are likely longer-range amino acid interactions which the networks explored here have difficulty capturing.

Adding additional state nodes may help to mitigate the problem of vanishing gradients although situations may arise that exceed the distance capacity of any set number of nodes in the state sequence. A better solution to this problem is to develop a learning algorithm which is not based on gradient descent. Such an algorithm would necessarily weight all previously discovered knowledge equally, since any sort of diminishing weighting would result in events occurring in the distant past eventually losing any impact on current events.

CHAPTER 6
DISCUSSION AND CONCLUSIONS

6.1 Summary

One primary goal of this work has been to design and implement in a modular manner an existing algorithm as an expert network and to investigate the consequences of adding state information to the expert network. The work described in this dissertation first involves the implementation of an expert network development environment that has a graphical user interface, is object-oriented and allows persistence of the developed networks. To implement the environment, I developed a class hierarchy of expert network objects and implemented the hierarchy in C++. Because expert networks differ from traditional neural networks, the class hierarchy for expert networks contains several features not typically found in neural classes. For example, because expert network synapses typically have behavior that is dependent upon the node types they connect, I implemented the synapse as a separate class while most implementations of neural network classes have the synapse as a data member of some other class.

I used the expert network development environment to implement expert networks encoding the Chou-Fasman algorithm rules for protein secondary structure prediction and then augment those rules with information about the state of the problem solution. The state information was added via a small, carefully selected set of recurrent connections in the topology of the expert networks.

The development of these partially recurrent expert networks proceeded one piece at a time, in a modular fashion. The modularity of the solution allows the implementation of expert subnetworks for solving pieces of the problem and then the combination of the subsolutions with resolution of any conflicts that might arise. Finally, methods for augmenting the resolved solution with both state knowledge and other refinement knowledge are described and investigated. Such augmentation has been shown to be useful.

Previous research has shown that combining previously discovered background knowledge with data-driven methods increases the problem-solving capabilities of a system [1] [48] [50] [67]. In addition, systems built from knowledge acquired from data rather than human experts are less costly and more accurate [3]. The expert network technique incorporates data-driven refinement of rules given by human experts, creating a set of rules that more accurately solves the problem at hand. The addition of recurrent connections in these networks allows expert networks to be applied to problems which require the maintenance of state information for adequate solutions. The recurrent connections allow state information to be gleaned from the data with no input from a human expert concerning the distance between the state dependencies in the input patterns.

The results presented in this dissertation show that, for the task of protein secondary structure prediction, the Chou-Fasman algorithm implemented as an expert network without state information has a higher prediction accuracy than the Chou-Fasman algorithm implemented in a strictly algorithmic manner [37], despite the fact that the expert network implementation does not consider all available information before making a prediction. In particular, the algorithm implemented as an expert network only uses information obtained by moving along the protein in one direction rather than both directions as stipulated in the original, non-automated algorithm. The fact that the expert network implementation of the algorithm can achieve prediction accuracy as good as or better than the algorithm implemented by Nishikawa [37] suggests that refinement of the Chou-Fasman rules via data-driven methods is a worthwhile effort.

During the course of this dissertation research, I developed a new expert network class, called `CUserNode`, which allows expert networks to be built and trained in phases. Such incremental building of the network facilitates the development and investigation of the hierarchical symbolic/sub-symbolic architectures typical of expert networks [7]. The expert network encoding of the Chou-Fasman algorithm was implemented in a modular manner, with expert subnetworks for helix finding and strand finding being implemented first. Once the subnetworks were completed, much effort was expended in investigation of various methods for combining the output predictions of the two subnetworks. An appropriate combination required the addition of new expertise to resolve conflicts arising between the outputs of the two subnetwork modules.

Once the helix finding and strand finding subnetwork modules were combined, I investigated the addition of state information to the larger expert network. I found that the state information, as a separate module, also needed to be combined with the previously developed modules and again investigated several combination methods. All of these combination methods required use of previously known knowledge about the problem domain. The state information module increased the prediction accuracy of the network even further. When presented with a set of proteins whose properties were close to what one finds on average in the protein world, the partially recurrent expert networks described in this dissertation performed more accurate prediction than the expert networks without recurrent connections. The analysis described in this dissertation shows that the increase in predictive capability came from the addition of state information rather than the expertise added for module combination.

Because the learning algorithm used in these partially recurrent expert networks is based on gradient descent, the technique has difficulty capturing relationships that span large distances [66]. Therefore, for proteins whose properties differ significantly from what one finds on average, the maintenance of the kind of state information found in the expert networks developed here may inhibit the predictive capacity of the networks.

FKBP is a protein, studied by Logan, *et al.* [43], which undergoes a conformational switch during unfolding in the laboratory conditions. This conformational switch indicates that long-range amino acid interactions play a significant role in determining the final conformation of FKBP. The distribution of helix, strand and coil amino acids in this protein, and others like it, however, differs significantly from the secondary structure distribution found in the proteins in the benchmark protein dataset of Qian and Sejnowski [13]. Because of this very different distribution of secondary structure, the partially recurrent networks described in this dissertation predicted helical amino acids in the FKBP-like proteins with low accuracy. The analysis described in Chapter 5 shows that as the network from experiment 6 began to ignore the kind of state information learned on the training set, its predictive accuracy increased on all three secondary structure types for the FKBP-like protein dataset while the predictive accuracy decreased for the benchmark protein dataset. These results suggest that the kind of state information captured by the partially recurrent expert networks is representative of amino acid interactions that occur at a shorter distance than those found in the FKBP-like proteins, probably because of the limitations found in gradient descent methods. To capture truly long-range interactions, the development of a learning algorithm not based on gradient descent is necessary.

6.2 Future Work for Improved Prediction Accuracy

The learning algorithm implemented for the partially recurrent expert networks attempts to minimize *actual* error rather than *prediction* error (see Section 4.3.3). If the goal of this dissertation had been to solve the protein secondary structure prediction problem, I would have developed a learning algorithm which attempted to minimize prediction error instead. During the training of these networks, a lower actual error often resulted in a higher prediction error because the lowest possible actual error seemed to occur when the network predicted every input pattern to fall into the class with the highest number of training examples. For example, because there are many more non-helical amino acids than helical ones, the helix finding subnetwork had the lowest actual error when it predicted all amino acids to be non-helical. This tendency was mitigated but not eliminated through the use of the ΔW averaging technique. The reason that more highly trained networks had higher prediction error is that as the network was trained, the actual error was minimized but when the prediction accuracy of the network was measured, the prediction error was used.

I did not develop a learning algorithm which minimized prediction error because my goal was to develop a general environment for the development and testing of partially recurrent expert networks. A learning algorithm which minimizes prediction error is very application specific and not as general as I wanted my work to remain. In addition, the development of a learning algorithm which minimizes prediction error requires the violation of the object oriented principle of encapsulation (see Section 4.3.4) and since my goals had more to do with the development environment than the solution to the particular application problem, I chose not to violate this principle.

In addition, as discussed in Nishikawa [37], the Chou-Fasman algorithm, when computerized, suffers from the problem of overprediction of helices and strands. My implementation also overpredicts helices and strands. To mitigate this effect, Chou and Fasman developed additional rules [68] after the publication of the rules used in this dissertation. Again, if my goal were to solve the protein secondary structure prediction problem, I could implement these additional rules. Rules concerning the shortest helix or strand regions allowed could also be useful in increasing the expert network prediction accuracy on the protein secondary structure problem. Finally, the Chou-Fasman algorithm requires information about the protein gathered from scans in both directions to be used. In the networks described in this dissertation, information gathered only in a forward scan of the protein was used in prediction. To use information from both directions, a non-network method of combining the information must be developed. Since my goals involved the investigation of partially recurrent expert network topologies, I did not investigate the development of non-network methodologies.

Finally, the prediction of the secondary structure of proteins like FKBP is likely to benefit from the development of learning algorithms that are not based on gradient descent. Such an algorithm must maintain information about previous activations of the network for long periods of time. The long-range interactions that occur among amino acids in these proteins are stronger than and persist longer than those that can be maintained by gradient descent methods.

6.3 Future Work in Computer Science

The technology for the development and use of expert networks has changed considerably since this tool was conceived. The continued exploration of learning algorithms and topologies for expert networks will continue to bring increased problem solving capability to the tools. The development of supporting knowledge engineering tools, however, is likely to have a more revolutionary impact on the field.

During my investigations, much time was spent on translation of expert knowledge to persistent network architectures and then detailed examination of the changes that occurred in those architectures during training. The development of a knowledge engineering tool that allows the expert knowledge to be entered by a human in a variety of forms and then automatically translates the knowledge to a network architecture would save tremendous time in the solution development phase. Such a tool should also allow the knowledge engineer to watch and query the network during activation and training. This type of examination will save time in the analysis phase of these initiatives.

Additional tools to allow the knowledge engineer to query which patterns cause particular nodes in the network to fire and for visual manipulation of the input and output data would also aid in the iterative knowledge refinement process. The development of the next generation of knowledge engineering is likely to be critically important in the widespread adoption of knowledge-based hybrid tools by a more general population.

6.4 Conclusions

Many researchers recognize that if humans are to solve the remaining difficult artificial intelligence problems, hybrid techniques which combine both human expertise and machine learning are necessary [14]. The work in this dissertation involves three major tasks: the exploration of a set of methodologies for extending an existing hybrid technique by allowing automated discovery of new types of knowledge, the combination of the newly discovered knowledge with previously known knowledge, and the application of these methodologies to difficult problems.

A main objective of this dissertation work has been to investigate methods to allow discovery of knowledge about state via automated data examination and the addition of that state knowledge to expert networks. Such state knowledge is usually unknown to the human expert from whom the original knowledge was gathered and must be gleaned from another source, such as the data. Because many difficult problems are not completely understood by humans with expertise in the problem domain, techniques which allow discovery and use of state knowledge shed much-needed light on these problem domains. Adding the state knowledge to the existing knowledge requires investigation of a variety of methods for combining these two types of knowledge.

In many problems involving some sort of classification, such as the problem of assigning an amino acid to a particular secondary structure class, the development of modules to address a piece of the overall problem is useful. Such a modular approach to system development is one of the fundamental principles of software engineering [64]. I have developed modules for encoding the knowledge obtained from the expert and separate modules for discovering new knowledge about state. When a system is developed in a modular manner, however, conflicts may arise between the outputs from individual modules. I have, therefore, presented tools for developing knowledge-based neural systems in a modular manner and explored methodologies for incorporating domain-specific knowledge concerning module conflict resolution into the overall network topology.

Because some of the modules encode existing domain knowledge while other modules discover knowledge from the data, the method chosen for combining the outputs of these various models influences the resulting classification or prediction ability of the system. Determining the types of and the topologies for nodes to be used in the implementation of the appropriate combination method requires the use of additional knowledge about the problem domain. Adding the same knowledge before or after various previously developed modules changes the semantics of the newly added knowledge. Therefore, expertise in the problem domain is required to determine which combination methods make sense in a particular context.

I have demonstrated that data-driven techniques allow the discovery of a wide variety of types of useful knowledge. The successful use of this knowledge, however, requires the developer of a hybrid system to have extensive *a priori* knowledge about the problem domain. The combination of *a priori* knowledge and discovered knowledge results in systems that perform better than their stand-alone counterparts.

APPENDIX A
CLASS DEFINITIONS

A.1 CBaseObj

```
class CBaseObj
{
   protected:
      // Object ID to allow persistence
      int OID;
      // set OID when constructing object
      void SetOID();
   public:
      // return unique OID for object
      int GetOID();
};
```

A.2 CListNode

```
template <class T>
class CListNode
{
  public:
     // data of this item in list
     T Data;
     // next item in linked list
     CListNode<T> *Next;
     // constructor
     CListNode();
     // destructor
     virtual ~CListNode();
};
```

A.3 CList

```
template <class T>
class CList
{
  protected:
     // first item of list
     CListNode<T> *Head;
     // number of items in list
     int Size;
     // return pointer to node at position
     CListNode<T> * PtrTo(int Pos);
  public:
     // constructor
     CList();
     // copy constructor
     CList(CList<T>& L);
     // destructor
     virtual ~CList();
     // find length of list
     virtual int GetSize();
     // insert an item
     virtual void Insert(int Pos, T& Item);
     // delete an item
     virtual void Delete(int Pos);
     // get an item
     virtual T Get(int Pos);
     // find position of item
     virtual int Find(T Item);
     // require that = operator be overloaded for each type T
     virtual void operator =(const CList<T>& L);
};
```

A.4 CSynapse

```
class CSynapse: public CBaseObj
{
  private:
    // disallow copy constructor use
    CSynapse(CSynapse& S);
  protected:
```

```
  // weight of the synapse
  double mWeight;
  // accumulate weight change
  double mDeltaW;
  // influence factor for backprop
  double mInfluence;
  // input node for this synapse
  CNode* mInNode;
  // output node for this synapse
  CNode* mOutNode;
public:
  // constructor function
  CSynapse();
  // destructor
  virtual ~CSynapse();
  // return pointer to in node
  virtual CNode* mGetInNode();
  // return pointer to out node
  virtual CNode* mGetOutNode();
  // set weight to W
  virtual void mSetWeight(double W);
  // return synapse weight
  virtual double mGetWeight();
  // return weight * synapse's input
  virtual double mGetWeightedOutput();
  // get the synapse type
  virtual char* mGetType();
  // connect input node to synapse
  virtual void mConnectInput(CNode* N);
  // connect output node to synapse
  virtual int mConnectOutput(CNode* N);
  //update synaptic weight
  virtual void mUpdate();
  // accumulate deltas for batch learning
  virtual void mAccumDelta(double LR,double Error,int Divisor);
  // calculate the influence of this synapse on the total error
  virtual void mCalcInfluence();
  // return influence factor
  virtual double mGetInfluence();
  // set the influence factor variable
  virtual void mSetInfluence(double NewInfluence);
```

```
};
```

A.5 CHardSynapse

```
class CHardSynapse: public CSynapse
{
  private:
    // disallow copy constructor use
    CHardSynapse(CHardSynapse& H);
  public:
    // constructor
    CHardSynapse();
    // destructor
    virtual ~CHardSynapse();
    // get synapse type--always returns 'h'
    virtual char* mGetType();
    //connect outnode to synapse
    virtual int mConnectOutput(CNode* N);
    // synaptic weight for hard synapses won't change
    virtual void mUpdate();
    // delta w is always 0 for hard synapses
    virtual void mAccumDelta(double LR,double Error,int Divisor=0);
};
```

A.6 CBias

```
class CBias: public CBaseObj
{
  private:
    // disallow copy constructor use
    CBias(CBias& B);
  protected:
    // trainable bias
    double mWeight;
    // accumulate weight change
    double mDeltaW;
    // contribution of bias to error for node
    double mInfluence;
    // disallow constructor use
    CBias();
  public:
```

```
    // constructor initializes mWeight to W
    CBias(double W);
    // destructor
    virtual ~CBias();
    // return bias mWeight value
    virtual double mGetBias();
    //accumulate delta w
    virtual void mAccumDelta(double LR, double Error, int Divisor);
    // update bias value with accumulated delta w
    virtual void mUpdate();
    // calculate contribution of bias to error
    virtual void mCalcInfluence(double O);
    // return influence value
    virtual double mGetInfluence();
};
```

A.7 CNode

```
class CNode: public CBaseObj
{
  private:
    // disallow copy constructor use
    CNode(CNode& N);
  protected:
    // input, middle, or output layer
    char mNodeLayer[2];
    // reg, op, combine, or user node
    char mNodeType[2];
    // bias for this node
    CBias mBias;
    // internal state computed by combining function
    double mInternalState;
    // output value for node is internal state with
    // firing function applied
    double mOutput;
    // error at this node
    double mError;
    // type of firing function--linear or sigmoid
    char mFireFunction[2];
    // list of output connections from this node
    CList<CSynapse*> mOutConnections;
```

```
  // list of input connections to this node
  CList<CSynapse*> mInConnections;
  // function to compute internal state
  virtual void mCombine(double I);
  // function to compute output
  virtual void mFire();
public:
  // default constructor
  CNode();
  // constructor
  CNode(char type[2], char layer[2],double Bias);
  // destructor
  virtual ~CNode();
  // activation function for this node calls proper
  // mFire function.  N is a dummy variable for all
  // but input nodes
  virtual double mActivate(char update,double N=0.0);
  // derivative of firing function for backprop of error
  virtual double mFireDerivative();
  // calculate error for node. I is dummy variable for
  // all but output nodes.
  virtual double mCalcError(double I=-9999.99);
  // return error value for this node
  virtual double mGetError();
  // return output value
  virtual double mGetOutput();
  // return internal state
  virtual double mGetInternalState();
  // for all input synapses to this node, accumulate delta w's
  virtual void mAccumDeltas(double LR, int Divisor);
  // update weights on all input synapses
  virtual void mUpdateWeights();
  // update the bias value
  virtual void mUpdateBias();
  // add an input or output connection for node
  virtual void mAddConnection(char type[2], CSynapse* C);
  // delete an input connection
  virtual void mDelInConnection(CSynapse* S);
  // delete an output connection
  virtual void mDelOutConnection(CSynapse* S);
  // return node type
```

```
    virtual char* mGetType();
    // return node layer
    virtual char* mGetLayer();
    // return node bias value
    virtual double mGetBias();
    // return number of out connections
    virtual int mNumOutConnections();
    // return number of in connections
    virtual int mNumInConnections();
    // return pointer to synapse #P in out connection list
    virtual CSynapse* mGetOutConnection(int P);
    // return pointer to synapse #P in in connection list
    virtual CSynapse* mGetInConnection(int P);
};
```

A.8 COpNode

```
class COpNode: public CNode
{
  private:
    // disallow copy constructor use
    COpNode(COpNode& N);
  protected:
    // and, or, not, add node
    char mOpNodeType[2];
    // combining function
    virtual void mCombine(double I);
    // firing function
    virtual void mFire();
  public:
    // constructor
    COpNode();
    // destructor
    virtual ~COpNode();
    // return op node type: a, o, n, d
    virtual char* mGetOpNodeType();
    // derivative of firing function
    virtual double mFireDerivative();
    // op node bias value will not be updated
    virtual void mUpdateBias();
};
```

A.9 CAddNode

```
class CAddNode: public COpNode
{
  private:
    // disallow copy constructor use
    CAddNode(CAddNode& N);
  protected:
    // threshold above or below which internal state must be
    // in order for node to fire
    double mThreshold;
    // determines whether comparison of threshold and internal
    // state will be greater or less than
    char mGorL[2];
    // combining function
    virtual void mCombine(double I);
    // firing function
    virtual void mFire();
  public:
    // constructor
    CAddNode();
    // destructor
    virtual ~CAddNode();
    // derivative of firing function
    virtual double mFireDerivative();
    // update threshold value and comparison value
    virtual void mUpdateThresGorL(double T,char* GorL);
    // return threshold value
    virtual double mGetThreshold();
    // return comparison value
    virtual char* mGetGorL();
};
```

A.10 CCombineNode

```
class CCombineNode: public CNode
{
  private:
    // disallow copy constructor use
    CCombineNode(CCombineNode& N);
  protected:
```

```
    // combining function
    virtual void mCombine(double I);
    // firing function
    virtual void mFire();
  public:
    // constructor
    CCombineNode();
    // constructor
    CCombineNode(char type[2], char layer[2],double Bias);
    // destructor
    virtual ~CCombineNode();
    // derivative of firing function
    virtual double mFireDerivative();
};
```

A.11 CAndNode

```
class CAndNode: public COpNode
{
  private:
    // disallow copy constructor use
    CAndNode(CAndNode& N);
  protected:
    // keep track of synapse which sends in the minimum input
    // value so error can be backproped only along that synapse
    int mMinConnection;
    // combining function
    virtual void mCombine(double I);
    // firing function
    virtual void mFire();
  public:
    // constructor
    CAndNode();
    // destructor
    virtual ~CAndNode();
    // derivative of firing function
    virtual double mFireDerivative();
    // accumulate weight changes along appropriate synapses
    virtual void mAccumDeltas(double LR,int Divisor=0);
};
```

A.12 CNotNode

```
class CNotNode: public COpNode
{
  private:
    // disallow copy constructor use
    CNotNode(CNotNode& N);
  protected:
    // combining function
    virtual void mCombine(double I);
    // firing function
    virtual void mFire();
  public:
    // constructor
    CNotNode();
    // destructor
    virtual ~CNotNode();
    // derivative of firing function
    virtual double mFireDerivative();
};
```

A.13 COrNode

```
class COrNode: public COpNode
{
  private:
    // disallow copy constructor use
    COrNode(COrNode& N);
  protected:
    // keep track of synapse sending the maximum value as
    // input so that error can be backproped along this
    // synapse only
    int mMaxConnection;
    // combining function
    virtual void mCombine(double I);
    // firing function
    virtual void mFire();
  public:
    // constructor
    COrNode();
    // destructor
```

```
    virtual ~COrNode();
    // derivative of firing function
    virtual double mFireDerivative();
    // accumulate weight changes for appropriate synapses
    virtual void mAccumDeltas(double LR, int Divisor=0);
};
```

A.14 CPattern

```
template <class T>
class CPattern
{
  private:
    // disallow copy constructor
    CPattern(CPattern<T>& P);
  protected:
    // length of the input pattern
    int mInPatLength;
    // length of the output pattern
    int mOutPatLength;
    // Pattern consisting of objects of type T
    CList<T> mOnePat;
    // protected constructor
    CPattern(int InLen, int OutLen);
    // CPatFile class can use protected constructor
    friend CPatFile<T>;
  public:
    // name of the pattern
    char mPatName[25];
    // constructor
    CPattern();
    // destructor
    ~CPattern();
    // get the input pattern length
    virtual int mGetInLength();
    // get the output pattern length
    virtual int mGetOutLength();
    // return element N of in pattern
    virtual T mGetInElement(int N);
    // return element N of out pattern
    virtual T mGetOutElement(int N);
```

```
    // add in pattern element into position N
    virtual void mAddInElement(int N, T& Item);
    // add out pattern element into position N
    virtual void mAddOutElement(int N, T& Item);
    // overloaded = operator
    virtual void operator = (CPattern<T>& P);
    // overloaded ==
    virtual int operator == (CPattern<T>& P);
    // ostream class is allowed direct access to private members
    friend ostream& operator << (ostream& O, CPattern<T>& L);
    // istream class is allowed direct access to private members
    friend istream& operator >> (istream& I, CPattern<T>& L);
};
```

A.15 CPatFile

```
template <class T>
class CPatFile
{
  private:
    // disallow copy constructor
    CPatFile(CPatFile<T>& F);
  protected:
    // number of patterns in pattern file
    int mNumPatterns;
    // list of patterns in pattern file
    CList<CPattern<T>*> mAllPats;
  public:
    // constructor
    CPatFile();
    // destructor
    ~CPatFile();
    // read in entire file of patterns
    virtual void mReadFile(char* filename);
    // return number of patterns in file
    virtual int mGetNumPatterns();
    // return pattern number N
    virtual CPattern<T>& mGetPattern(int N);
    // add a new pattern to pattern file
    virtual void mAddPattern(int N, CPattern<T>& P);
};
```

A.16 CNetwork

```
// new name for list of pointers to nodes
typedef CList<CNode *> NodeListType;

class CNetwork: public CBaseObj
{
  private:
    // disallow copy constructor use
    CNetwork(CNetwork& N);
  protected:
    // list of all user-defined nodes
    CList<CUserNode*> mUserNodeList;
    // list of all network input nodes
    CList<CNode*> mInNodeList;
    // middle nodes in network
    CList<CNode*> mMidNodeList;
    // list of all network output nodes
    CList<CNode*> mOutNodeList;
    // list of all connections in network
    CList<CSynapse*> mConnectionList;
    // print adjacency matrix representing network
    virtual void mPrintAdjMatrix(CList<NodeListType*>& L);
    // keep middle nodes topologically sorted
    virtual void mReSortMidLayer();
    // find a middle node with no successors
    virtual char mNoSuccessors(NodeListType* N);
    // propagate error backward, updating weights along the way
    virtual void mPropError();
    // accumulate weight changes for batch learning
    virtual void mAccumDeltas(double LR, int Divisor);
  public:
    // constructor
    CNetwork();
    // constructor for fully connected network with InNodes
    // input nodes and OutNodes output nodes and no middle nodes
    CNetwork(int InNodes, int OutNodes);
    // destructor
    virtual ~CNetwork();
    // create network from file filename
    ReadFile(char filename[80]);
    // train network with patterns
```

```
    virtual double mTrainNet(CPatFile<double>& P,int I,double T,
                                  double LR);
    // activate network without learning
    virtual double mActivateNet(CPatFile<double>& P);
    // one pass thru net (can update weights or not)
    virtual double mOnePass(CPattern<double>& P, char update, int i,
                            double LR=0.0);
    // get Nth node in layer
    virtual CNode* mGetNode(char layer[2], char type[2], int N);
    // get pointer to node with unique object identifier ID
    virtual CNode* mNodeID_2_Ptr(int ID);
    // get pointer to synapse with unique ID
    virtual CSynapse* mSynID_2_Ptr(int ID);
    // adding a node to the network
    virtual int mAddNode(char type[2],char layer[2], char optype[2],
                         int& Pos,double Bias,double Threshold,
                         char* GorL);
    // add a synapse between node N1 and node N2
    virtual int mAddConnection(CNode* N1,CNode* N2,char type[2],
                               double W);
    // delete node with object ID NodeID
    virtual int mDelNode(int NodeID);
    // delete synapse with object ID SynID
    virtual int mDelConnection(int SynID);
    // return number of user nodes
    virtual int mGetNumUserNodes();
    // return number of input nodes
    virtual int mGetNumInNodes();
    // return number of middle nodes
    virtual int mGetNumMidNodes();
    // return number of output nodes
    virtual int mGetNumOutNodes();
    // save the network to a file
    virtual void mSaveNet(char filename[80]);
    // clear the network
    virtual void mClearNet();
};
```

A.17 CUserNode

```
class CUserNode: public CNode
```

```
{
  private:
    // disallow copy constructor use
    CUserNode(CUserNode& N);
    // disallow simple constructor use
    CUserNode();
  protected:
    // name of file containing information for construction
    // of mNet data member
    char InFileName[80];
    // name of file containing information for construction
    // of mPats data member
    char PatFileName[80];
    // input patterns for subnetwork mNet
    CPatFile<double> mPats;
    // subnetwork comprising this user node
    CNetwork mNet;
  public:
    // constructor
    CUserNode(char* Netfile, char* Patfile);
    // destructor
    virtual ~CUserNode();
    // activate mNet
    virtual double mActivate(char update, int i);
    // add output connection
    virtual void mAddConnection(CSynapse* C);
    // delete output connection
    virtual void mDelOutConnection(CSynapse* S);
    // return number of output connections
    virtual int mNumOutConnections();
    // return pointer to output synapse #P
    virtual CSynapse* mGetOutConnection(int P);
    // return name of network file
    virtual char* mGetInFileName();
    // return name of pattern file
    virtual char* mGetPatFileName();
};
```

APPENDIX B

FILE FORMAT FOR NETWORK PERSISTENCE

B.1 CNode Objects

B.1.1 Regular Nodes

```
Field Name             Type           Acceptable Values
-------------------    ------------   -----------------
Node Layer             1 character    I,M,O
Node Type              1 character    R
Unique Object ID       1 integer
Bias                   1 double
```

B.1.2 Operation Nodes

Not nodes.

```
Field Name             Type           Acceptable Values
-------------------    ------------   -----------------
Node Layer             1 character    M
Node Type              1 character    O
OpNode Type            1 character    N
Unique Object ID       1 integer
Bias                   1 double
```

And nodes.

```
Field Name             Type           Acceptable Values
-------------------    ------------   -----------------
Node Layer             1 character    M
Node Type              1 character    O
OpNode Type            1 character    A
Unique Object ID       1 integer
Bias                   1 double
```

Or nodes.

Field Name	Type	Acceptable Values
Node Layer	1 character	M
Node Type	1 character	O
OpNode Type	1 character	O
Unique Object ID	1 integer	
Bias	1 double	

Add nodes.

Field Name	Type	Acceptable Values
Node Layer	1 character	M
Node Type	1 character	O
OpNode Type	1 character	D
Threshold	1 integer	
Comparison Value	1 character	G,L
Unique Object ID	1 integer	
Bias	1 double	

Combine Nodes

Field Name	Type	Acceptable Values
Node Layer	1 character	M
Node Type	1 character	M
Unique Object ID	1 integer	
Bias	1 double	

B.1.3 User Nodes

Field Name	Type	Acceptable Values
Node Layer	1 character	I
Node Type	1 character	U
Network File Name	1 string	
Pattern File Name	1 string	
Unique Object ID	1 integer	
Bias	1 double	

B.2 CSynapse Objects

Field Name	Type	Acceptable Values
Connection Indicator	1 character	C
From Node ID	1 integer	
To Node ID	1 integer	
Connection Type	1 character	H,S
Weight	1 double	

REFERENCES

[1] D.C. Kuncicky, S.I. Hruska, and R.C. Lacher, "Hybrid systems: the equivalence of expert system and neural network inference," *International Journal of Expert Systems*, vol. 4, pp. 281–297, 1992.

[2] R.C. Lacher, "Expert networks: Paradigmatic conflict, technological rapprochement," *Minds and Machines*, vol. 3, pp. 53–71, 1993.

[3] Arie Ben-David and Janice Mandel, "Classification accuracy: Machine learning vs. explicit knowledge acquisition," *Machine Learning*, vol. 18, pp. 109–114, 1995.

[4] Larry R. Medsker, *Hybrid Neural Network and Expert Systems*, Boston: Kluwer Academic Publishers, 1994.

[5] David C. Kuncicky, *Isomorphism of Reasoning Systems with Applications to Autonomous Knowledge Acquisition*, Ph.D. dissertation, Florida State University, 1991.

[6] J. Giarratano and G. Riley, *Expert Systems: Principles and Programs*, Boston, MA: PWS-Kent Publishing Co., 1989.

[7] R.C. Lacher and K.D. Nguyen, "Hierarchical architectures for reasoning," in *Computational Architectures for Integrating Neural and Symbolic Processes*, R. Sun and L. Bookman, Eds. Boston: Kluwer Academic Publishers, 1994, pp. 117–150.

[8] R.C. Lacher, S.I. Hruska, and D.C. Kuncicky, "Backpropagation learning in expert networks," *IEEE Transactions on Neural Networks*, vol. 3, pp. 62–72, 1992.

[9] John Hertz, Anders Krogh, and Richard G. Palmer, *Introduction to the Theory of Neural Computation*, Redwood City, California: Addison-Wesley Publishing Company, 1991.

[10] D. Rumelhart, G. Hinton, and R. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, D. Rumelhart and J. McClelland, Eds., vol. 1. Cambridge: MIT Press, 1986, pp. 318–363.

[11] T.J. Sejnowski and C.R. Rosenberg, "Parallel networks that learn to pronounce English text," *Complex Systems*, vol. 1, pp. 145–168, 1987.

[12] D.A. Pomerleau, "ALVINN: An automonous land vehicle in a neural network," in *Advances in Neural Information Processing Systems*, D.S. Touretzky, Ed., vol. I, 1988, pp. 305–313.

[13] N. Qian and T. Sejnowski, "Predicting the secondary structure of globular proteins using neural network models," *Journal of Molecular Biology*, vol. 202, pp. 865–884, 1988.

[14] Larry R. Medsker, *Hybrid Intelligent Systems*, Boston: Kluwer Academic Publishers, 1995.

[15] A. Cleeremans, D. Servan-Schreiber, and J. McClelland, "Finite state automata and simple recurrent networks," *Neural Computation*, vol. 1, pp. 372–381, 1989.

[16] Jeffrey L. Elman, "Distributed representations, simple recurrent networks and grammatical structure," *Machine Learning*, vol. 7, pp. 195–225, 1991.

[17] M.I. Jordan, "Serial order: a parallel, distributed processing approach," in *Advances in Connectionist Theory: Speech*, J.L. Elman and D.E. Rumelhart, Eds. Hillsdale: Erlbaum, 1989.

[18] Jeffrey L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, pp. 179–211, 1990.

[19] Richard Maclin and Jude W. Shavlik, "Using knowledge-based neural networks to improve algorithms: Refining the Chou-Fasman algorithm for protein folding," in *Proceedings of the Tenth National Conference on Artificial Intelligence*, July 1992, pp. 165–170.

[20] Carl Branden and John Tooze, Eds., *Introduction to Protein Structure*, New York: Garland Publishing, Inc., 1991.

[21] Peter Y. Chou and Gerald D. Fasman, "Conformational parameters for amino acids in helical, β-sheet, and random coil regions calculated from proteins," *Biochemistry*, vol. 13, no. 2, pp. 211–222, 1974.

[22] J. Garnier, D.J. Osguthorpe, and B. Robson, "Analysis of the accuracy and implications of simple methods for predicting the secondary structure of globular proteins," *Journal of Molecular Biology*, vol. 120, pp. 873–894, 1978.

[23] V.I. Lim, "Algorithms for prediction of α-helical and β-structural regions in globular proteins," *Journal of Molecular Biology*, vol. 88, pp. 873–894, 1974.

[24] Scott R. Presnell, Bruce I. Cohen, and Fred E. Cohen, "A segment-based approach to protein secondary structure prediction," *Biochemistry*, vol. 31, pp. 983–993, 1992.

[25] Ron Unger and John Moult, "Genetic algorithms for protein folding simulations," *Journal of Molecular Biology*, vol. 231, pp. 75–81, 1993.

[26] Xiru Zhang, Jill P. Mesirov, and David L. Waltz, "Hybrid system for protein secondary structure prediction," *Journal of Molecular Biology*, vol. 225, pp. 1049–1063, 1992.

[27] Richard Maclin and Jude W. Shavlik, "Using knowledge-based neural networks to improve algorithms: refining the Chou-Fasman algorithm for protein folding," *Machine Learning*, vol. 11, pp. 195–215, 1993.

[28] D.G. Kneller, F.E. Cohen, and R. Langridge, "Improvements in protein secondary structure prediction by an enhanced neural network," *Journal of Molecular Biology*, vol. 214, pp. 171–182, 1990.

[29] H.L. Holley and M. Karplus, "Protein secondary structure prediction with a neural network," *Proceedings of the National Academy of Sciences, USA*, vol. 86, pp. 152–156, 1989.

[30] S. Hayward and J.F. Collins, "Limits on α-helix prediction with neural network models," *Proteins*, vol. 14, pp. 372–381, 1992.

[31] Martin Karplus and Eugene Shakhnovich, "Protein folding: theoretical studies of thermodynamics and dynamics," in *Protein Folding*, Thomas E. Creighton, Ed. New York: W.H. Freeman and Company, 1993, pp. 127–195.

[32] Burkhard Rost, Chris Sander, and Reinhard Schneider, "Redefining the goals of protein secondary structure prediction," *Journal of Molecular Biology*, vol. 235, pp. 13–26, 1994.

[33] Larry Hunter, "Molecular biology for computer scientists," in *Artificial Intelligence and Molecular Biology*, Larry Hunter, Ed. Menlo Park, CA/Cambridge, MA: AAAI Press/MIT Press, 1993, pp. 1–46.

[34] C.B. Anfinsen, E. Haber, M. Sela, and F.H. White, "The kinetics of formation of native ribonuclease during oxidation of the reduced polypeptide chain," *Proceedings of the National Academy of Sciences, USA*, vol. 47, pp. 1309–1314, 1961.

[35] Gerald D. Fasman, Ed., *Prediction of Protein Structure and the Principles of Protein Conformation*, New York: Plenum Press, 1989.

[36] Peter Y. Chou and Gerald D. Fasman, "Prediction of protein conformation," *Biochemistry*, vol. 13, no. 2, pp. 222–245, 1974.

[37] Ken Nishikawa, "Assessment of secondary-structure prediction of proteins: comparison of computerized Chou-Fasman method with others," *Biochimica et Biophysica Acta*, vol. 748, pp. 285–299, 1983.

[38] Peter Prevelige, Jr. and Gerald D. Fasman, "Chou-Fasman prediction of secondary structure," in *Prediction of Protein Structure and the Principles of Protein Conformation*, Gerald D. Fasman, Ed. New York: Plenum Press, 1989, pp. 391–416.

[39] Daniel I.A. Cohen, *Introduction to Computer Theory*, New York: John Wiley and Sons, 1991.

[40] W.W. Ralph, T. Webster, and T.F. Smith, "A modified Chou and Fasman protein structure algorithm," *CABIOS*, vol. 3, pp. 211–216, 1987.

[41] N. Rawlings, K. Ashman, and B. Wittman-Leibold, "Computerized version of the Chou and Fasman protein secondary structure predictive method," *International Journal of Peptide Protein Research*, vol. 22, pp. 515–524, 1983.

[42] W. Kabsch and C. Sander, "Dictionary of protein secondary structure: pattern recognition of hydrogen bonded and geometrical features," *Biopolymers*, vol. 22, pp. 2577–2637, 1983.

[43] Timothy Logan, Yves Theriault, and Stephen Fesik, "Structural characterization of the FK506 binding protein unfolded in urea and guanidine hydrochloride," *Journal of Molecular Biology*, vol. 236, pp. 637–648, 1994.

[44] Pei Fan, Clay Bracken, and Jean Baum, "Structural characterization of Monellin in the alcohol-denature evidence for β-sheet to α-helix conversion," *Biochemistry*, vol. 32, pp. 1573–1582, 1993.

[45] Hong Pan, Elisar Barbar, George Barany, and Clare Woodward, "Extensive nonrandom structure in reduced and unfolded bovine pancreatic trypsin inhibitor," *Biochemistry*, vol. 34, pp. 13974–13981, 1995.

[46] J. Reed and V. Kinzel, "A conformational switch is associated with receptor affinity in peptides derived from the CD4-binding domain of gp120 from HIV I," *Biochemistry*, vol. 30, pp. 4521–4528, 1991.

[47] Ouwen Zhang and Julie D. Forman-Kay, "Structural characterization of folded and unfolded states of an SH3 domain equilibrium in aqueous buffer," *Biochemistry*, vol. 34, pp. 6784–6794, 1995.

[48] S.I. Hruska, D. Kuncicky, and R.C. Lacher, "Hybrid learning in expert networks," in *Proceedings of the International Joint Conference on Neural Networks*, vol. II, July 1991, pp. II117–II120.

[49] Alan P. Levis, "*Inferencing and Training in a Development Environment for Table-based Expert Networks*," M.S. thesis, Florida State University, 1995.

[50] Alan P. Levis, Kristin L. Adair, and Susan I. Hruska, "A table-based approach to expert fault diagnosis networks," in *Proceedings of ANNIE'96*, 1996.

[51] A.P. Levis, R.G. Timpany, W.E. Austad, J.R. Elling, J.J. Ferguson, D.A. Klotter, and S.I. Hruska, "Application of knowledge-based network processing to automated GC data interpretation," in *Proceedings of SPIE AeroSense '95*, vol. 2492, April 1995, pp. 294–302.

[52] John K. Ousterhout, *Tcl and the Tk Toolkit*, Reading, MA: Addison-Wesley, 1994.

[53] S. Zdonik and D. Maier, "Fundamentals of object-oriented databases," in *Readings in Object-Oriented Database Systems*, S. Zdonik and D. Maier, Eds. San Mateo: Morgan Kaufman, 1990, pp. 1–32.

[54] Valluru B. Rao and Hayagriva V. Rao, *C++ Neural Networks and Fuzzy Logic*, New York, NY: MIS:Press, 1995.

[55] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Reading, MA: Addison-Wesley, 1974.

[56] G.E. Hinton and T.J. Sejnowski, "Learning and relearning in Boltzmann machines," in *Parallel Distributed Processing*, Rumelhart and McClelland, Eds. Cambridge: MIT Press, 1986.

[57] J.J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences, USA*, vol. 79, pp. 2554–2558, 1982.

[58] L.B. Almeida, "Backpropagation in perceptrons with feedback," in *Neural Computers*, R. Eckmiller and Ch. von der Malsburg, Eds. Berlin: Springer-Verlag, 1988, pp. 199–208.

[59] F.J. Pineda, "Generalization of back-propagation to recurrent neural networks," *Physical Review Letters*, vol. 59, pp. 2229–2232, 1987.

[60] R. Rohwer and B. Forrest, "Training time-dependence in neural networks," in *IEEE First International Conference on Neural Networks*, M. Caudill and C. Butler, Eds., vol. 2, 1987, pp. 701–708.

[61] James A. Freeman, *Simulating Neural Networks with Mathematica*, Reading, MA: Addison-Wesley, 1994.

[62] Frank M. Carrano, *Data Abstraction and Problem Solving with C++: Walls and Mirrors*, Reading, MA: Addison-Wesley, 1995.

[63] B.W. Matthews, "Comparision of the predicted and observed secondary structure of T4 phage lysozyme," *Biochimica and Biophysica Acta*, vol. 405, pp. 442–51, 1975.

[64] Roger S. Pressman, *Software Engineering: A Beginner's Guide*, New York, NY: McGraw-Hill Book Company, 1988.

[65] George Snedecor and William Cochran, *Statistical Methods*, Des Moines, IA: Iowa State University Press, 6 edition, 1967.

[66] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.

[67] Susan I. Hruska and David C. Kuncicky, "Automated knowledge refinement for control chart selection," *Heuristics*, vol. 5, pp. 101–7, 1992.

[68] Peter Y. Chou and Gerald D. Fasman, "Prediction of secondary structure of proteins from their amino acid sequence," *Advanced Enzymology*, vol. 47, pp. 45–148, 1978.

BIOGRAPHICAL SKETCH

Catherine LeBlanc was born in Goffstown, NH. After graduating from Goffstown High School, she attended Dartmouth College, graduating *cum laude* with a BA in computer science. After stints delivering pizzas, developing large software systems and writing grant proposals, she returned to graduate school at Florida State University in 1992. At FSU, she has worked as a teaching assistant and a research assistant in the Department of Computer Science, earning a Master's degree in 1995. She has received numerous teaching awards and served as a Teaching Associate with the Program for Instructional Excellence. She received a University Dissertation Fellowship for the academic year 1996-1997. Following graduation, she will be an Assistant Professor of Computer Science at Keene State College in Keene, NH.