Using the Aggregate Feedforward Neural Network for Rule Extraction

Vincent A. Schmidt and C.L. Philip Chen
Wright State University, Dayton, Ohio

Abstract

The Aggregate Feedforward Neural Network (AFFNN) is a unified connectionist architecture designed to behave as if it contained K distinct neural networks. This architecture specifically learns each attribute in the source dataset as a function of the remaining attributes, a mechanism that promotes using the AFFNN as a platform for knowledge extraction. A decompositional extraction algorithm is created to demonstrate the viability of obtaining rules from the successfully trained AFFNN. A description of this generalized knowledge extraction procedure indicates how knowledge can be extracted from the AFFNN in the form of rules. This technique is applied to AFFNNs trained for well-understood examples (the Monks problem #2 and the Iris problem) as a basis for comparing the results with other connectionist and non-connectionist rule extraction systems and techniques.

Keywords: ANN, Data Mining, AFFNN, Rule Generation

1 Introduction

A combination of technological and scientific advances has fueled a resurgence of interest in using neural networks as platforms for data mining. As a result, neural approaches to data mining are systematically becoming more advanced and robust, while neural platforms are simultaneously gaining diversity. Such research efforts strive to use connectionist models in many aspects of data mining, and various techniques for extracting knowledge from neural networks have been developed.

Mining classification or association rules from quantities of data has traditionally been studied and performed in terms of database tools and algorithms operating on very large datasets, and considerable progress has been made [Zaki(1998)]. However, using massive databases is not a requirement for data mining activities. Small datasets are often useful, and connectionist systems are among the numerous methods currently employed to determine relationships contained in these datasets. Connectionist systems can generally be trained to recognize and classify the data, but are often viewed as black boxes that only reluctantly yield up the knowledge they have learned. Thus, research involving the extraction of knowledge from neural networks remains popular.

A report from the Neurocomputing Research Center in Queensland, Australia classifies connectionist rule extraction into three categories: decompositional, pedagogical, and eclectic [Andrews et al.(1995)Andrews, Diederich, and Tickle]. Decompositional approaches tend to concentrate on the internals of the neural network, focusing on the trained network's links and weights. Rules are established by algorithmically tracing the desired values of the links within the network. Common decompositional methods move from the output nodes back through the nodes in the hidden layer, integrating these results with contributions from the input nodes. Decompositional methods are popular because they provide full visibility into the network, lending confidence to the extracted results, and tend to be relatively straightforward to apply.

Pedagogical approaches treat the trained connectionist system as a black box. The system is used as a mechanism for mapping inputs directly to the outputs, with no heed given to the trained network weights. The report by Andrews et al. suggests that the trained neural network is used to generate examples for other symbolic learning algorithms. The eclectic category is reserved for hybrid systems, using both decompositional and pedagogical mechanisms for rule generation.

Several popular and important methods for performing rule extraction on various connectionist systems are summarized by Andrews et al., including:

- SUBSET (Decompositional) — Examine individual links meeting exceeding a node's bias such that other link values will not impact the results; write each of these links as a rule. Examine pairs of links, then triples, etc. until all sets have been examined. If some link prevents the solution from being reached, the rule is written as "if NOT n" for that link.

- M-of-N (Decompositional) — Generate rules in the form "if (M of the following N antecedents are true) then ..." Groups of similarly-weighted links are formed, with link weights all set to the average of the group values. Groups that do not impact node activation are eliminated, and the node biases are adjusted to reflect the new group values. The resultant group and bias data is the source of the rules generated. It is reported that these rules are more robust with respect to previously unseen examples. Ideas applying the M-of-N method were originally used with a KBANN system [Towell and Shavlik(1993)]. Setiono also proposes an algorithm for extracting M-of-N rules from feedforward neural networks [Setiono(2000)].

- Validity Interval Analysis (VIA) (Pedagogical) — Extract rules that map inputs directly to outputs by generating arbitrary intervals to all nodes within the network, then iteratively refining the intervals according to activation values. These intervals become either consistent or inconsistent with network weights and biases. Thrun, the creator of VIA, indicates this approach is similar to sensitivity analysis because it examines the network outputs while varying the inputs.

- RULENEG (Pedagogical) — Extract conjunctive rules using propositional calculus, where each rule expression is the disjunction of conjunctions. Training data is coupled with the trained neural network to iteratively build at most one conjunctive rule for each input pattern.

And, more recent developments than those included in the report by Andrews:

- RX (Decompositional) — Combinations of discretized output link values are tested to determine when the desired outputs are produced. Inputs supporting these values are substituted, resulting in a series of rules describing the outputs in terms of the inputs [Setiono and Liu(1995), Lu et al.(1995b)Lu, Setiono, and Liu, Lu et al.(1995a)Lu, Setiono, and Liu].

- RG (Decompositional) — Data (clusters of activation values) are grouped in terms of class. Rules are constructed by iteratively finding the best selection of attributes that differentiate the target class from the remaining classes of data until all data elements have been considered. This process is performed for finding the relationships of network inputs to hidden nodes, and also for determining the relationships of hidden nodes to output nodes [Setiono and Liu(1996)].

- TREPAN (Pedagogical) — Constructs a decision tree from the neural network by growing the tree according to decisions promoted by network nodes. The tree is grown until one of several stopping criteria are met [Craven and Shavlik(1997), Craven and Shavlik(1999)].

It is evident that there are many approaches to rule extraction for connectionist systems; traditional wisdom suggests there will always be a place for a variety of solutions. A couple of these systems warrant additional comments before introducing the aggregate model.

NeuroRule is a well-specified system for performing rule extraction on feedforward connectionist systems [Lu et al.(1995b)Lu, Setiono, and Liu]. NeuroRule trains a neural network containing a single hidden layer using input cases that are typically preprocessed using thermometer encoding. Hyperbolic tangent activation functions are used at the hidden layer with a basic sigmoid function at the output layer. Training is performed using a cross-entropy function with a penalty component to keep weights small [Setiono(1997)]. The authors also propose a simple pruning scheme to reduce the initial fully-connected network into a more manageable system with few links.

The fully trained NeuroRule system extracts rules using the authors' RX algorithm, an approach that discretizes activation values to determine classification rules. This decompositional extraction algorithm closely couples rule extraction with network pruning to obtain a reasonably small and accurate set of representative rules. The network is pruned and retrained until a maximum tolerable error is reached, then the RX approach is used to discretize link values, working from the output node(s) back to the inputs. NeuroRule has also been used with the RG algorithm.

KBANN is among the earliest connectionist systems supporting rule extraction, including the use of M-of-N rules [Towell and Shavlik(1993)]. KBANN is a knowledge-based neural network developed and enhanced by Shavlik, Towell, Craven and others. The KBANN systems creates neural networks by building a topology and setting link weights. This is accomplished by translating rules directly to the network architecture, then applying training examples for system learning. Continued research using the KBANN has produced several interesting extensions, including TopGen and DAID [Towell and Shavlik(1992), Opitz and Shavlik(1995)].

The KBANN algorithm tends to produce "deep" networks (containing multiple hidden layers), which can lengthen training times and increase network complexity. Towell and Shavlik address this issue with the creation of the DAID algorithm. The DAID preprocessor sits between the domain rules and the network translator, suggesting additional links that may strengthen relationships found during training. DAID's sole responsibility is the generation of these links. This is important because it is easier for the KBANN to eliminate useless links than to add new links. Links added by DAID can improve network generality and robustness to noise, since the KBANN may contain a better subset of network links. This could also decrease network complexity, allowing the KBANN a better selection of links to eliminate.

The TopGen (Topology Generator) algorithm by Opitz and Shavlik specializes in adding nodes to the KBANN, versus the links added by DAID [Opitz and Shavlik(1995)]. TopGen is a heuristic algorithm that searches the KBANN for nodes with high error rates, where the network experiences difficulty in generalization. The strategic addition of nodes eliminates or decreases the number of false positives and false negatives in the network without disrupting the domain theory originally encoded by KBANN. Of course, the network must be retrained following the addition of new nodes.

Once the KBANN is created and appropriately trained, the TREPAN technique concentrates on generating hypotheses by extracting decision trees from the trained neural network model [Craven and Shavlik(1997), Craven and Shavlik(1999)]. TREPAN creates the tree best-first because it progressively refines the tree based on the entire network, not using an analysis of the network structure or weights like other algorithms. A subset of training instances are applied as queries against the trained network to guide tree creation. Nodes are split as the tree continues to grow until one meeting one of the stopping criteria: maximum tree size, reasonable validation, or node coverage using statistical testing.

TREPAN's creators point out that the algorithm can be applied to a wide variety of neural networks and does not require special network training algorithms or architectures. In some cases, the algorithm can also be applied to structures besides neural networks. The user also has control over the size and granularity of the extracted tree. Finally, a variant of TREPAN can be used to extract finite automata from recurrent neural networks.

Another useful model is the Aggregate Feedforward Neural Network (AFFNN), an architecture designed by Schmidt and Chen, introducing a new mechanism for designing and training neural net-

works [Schmidt and Chen(2002)]. This unique connectionist system strives to learn the relationships of each attribute with respect to the remaining attributes in the system. The AFFNN is trained by simultaneously presenting all K attributes as network inputs. All K attributes are also provided as network outputs, where each output is based solely on the remaining attributes. At the completion of successful training, the network is able to provide the values of every attribute as a function of the remaining attributes. The AFFNN topology consists of a fully connected feedforward network with a single hidden layer, and can be trained using traditional backpropagation algorithms and their variants. In fact, the AFFNN does not have any system-specific training or error requirements; almost any combination of training and error functions can be used. The AFFNN does need a suite of support functions, consisting of a set of preprocessing and postprocessing operations and a specialized performance function wedge that operates immediately before the standard error function is called.

Since this system learns relationships for all attributes within one unified network network, it is reasonable to believe that the AFFNN provides a novel basis for rule extraction. The main thrust of this paper describes one mechanism by which the knowledge contained within a trained AFFNN can be extracted and expressed as a collection of rules. The decompositional generalized knowledge extraction technique discussed here is capable of finding descriptive rules based on AFFNN connection weights and representing them in a format acceptable for use with code generation methods. This extraction procedure has been designed, implemented, and tested by the authors as a proof of concept, demonstrating that the AFFNN is a practical and useful platform for rule extraction. The mechanics of extracting rules from academic examples are provided, along with comparisons to other systems.

Background on the key aspects of the AFFNN and its support system is briefly reviewed in the next section, followed by an examination of a practical method for extracting rules from the trained network. Remarks about the continuation of this research effort, including comments about about the AFFNN's strengths and weaknesses, are given in the conclusion.

2 The Aggregate Feedforward Neural Network (AFFNN)

The AFFNN is best described as a unified connectionist model that behaves as if it were K individual networks. Alternatively, consider the AFFNN as a collection of K parallel, interconnected neural networks that are treated as a single neural network system. Either way, the AFFNN itself is a fully-connected neural model with a single hidden layer and the same number of nodes in the input and output layers.

The premise of the AFFNN is that a single feedforward neural network can be constructed in such a way that, for any given attribute, each attribute can be expressed as a function of the remaining attributes, when such functions exist. Given K attributes, a single network can be trained to learn the same functions that would generally require designing and training K completely independent networks.

2.1 Defining the Aggregate Feedforward Neural Network

The data used with the AFFNN system determines network topology. Once the data is prepared, cleaned, and discretized, if needed, a template vector G is created to reflect the encoding used by the data cases. The length of G ($|G|$) defines the number of nodes in the AFFNN's input and output layers, but the number of nodes in the hidden layer is determined empirically. Layers are fully connected, from input layer to hidden layer to output layer.

Figure 1 illustrates the fundamental AFFNN structure. All attributes are treated as inputs to the AFFNN. Since these attributes cannot generally be used in their raw form, they are commonly encoded into a simple binary representation using a binning or thermometer-based scheme. Each attribute requires one or more network inputs to be properly represented. The template vector G captures the

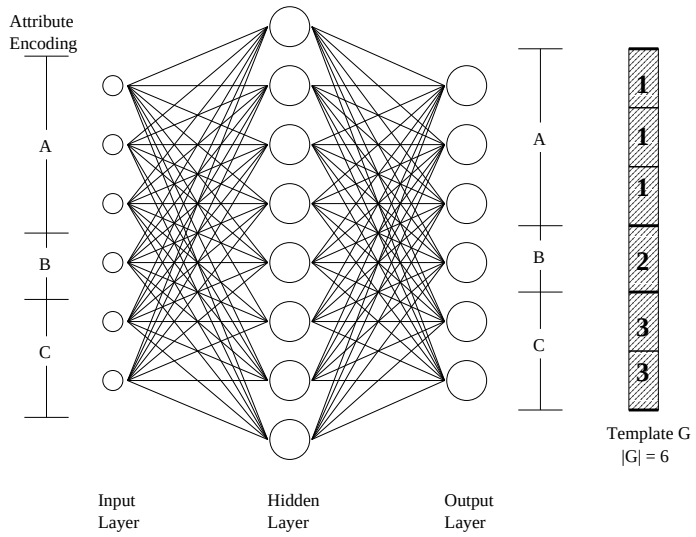


Figure 1: Example AFFNN

input vector positions used to encode each attribute. In the figure, suppose an arbitrary dataset with three attributes A, B, and C are encoded in a way that they use 3, 1, and 2 input nodes respectively for a total of six required inputs. The template G at the right of the figure reflects the encoded positions of all attributes: the three cells containing "1" represent the encoded positions of the first attribute, A; the cell containing "2" is the encoded position of the second attribute, B; the last two positions are reserved for the third attribute, C. Since $|G| = 6$, the AFFNN contains six nodes in the input and output layers, which are fully connected to the nodes in the hidden layer.

The newly created AFFNN can use virtually any standard error function and supervised training algorithm the user specifies, but relies on a support system for proper operation. This system includes a small amount of administrative data coupled with a unique combination of preprocessing and post-processing support functions and a specialized performance function wedge ("AGPF"). These functions allow the AFFNN to use many standard feedforward training and performance functions without modification. They also prevent the AFFNN from becoming autoassociative, even though all input attributes are also network outputs. System features are examined here in turn.

Figure 2 illustrates the general AFFNN system architecture. The template vector G ($|G| = |V|$), shown in the figure, denotes the sections of input vector V belonging to each attribute, allowing G to be used to quickly and efficiently mask attributes. The preprocessing function uses G to reproduce a single input vector V containing K attributes as a series of K input vectors, $V_{1*}..V_{K*}$, where each vector V_{i*} in the series masks a single attribute $i \in (1..K)$ with zeros. The shaded portions of the vectors at the left of Figure 2 indicate the masked sections of each V_{i*}. Masking an attribute in this manner prevents it from directly contributing to the network solution, so the trained AFFNN's output is a function of the specific input vector V_{i*}:

$$O_{i+} = f_{net}(V_{i*}) \tag{1}$$

where $f_{net}()$ is the function learned and performed by the AFFNN. The preprocessor extends the entire set of T input cases using this method to yield a total of $T * K$ AFFNN inputs, with each extended input vector processed according to Equation 1.

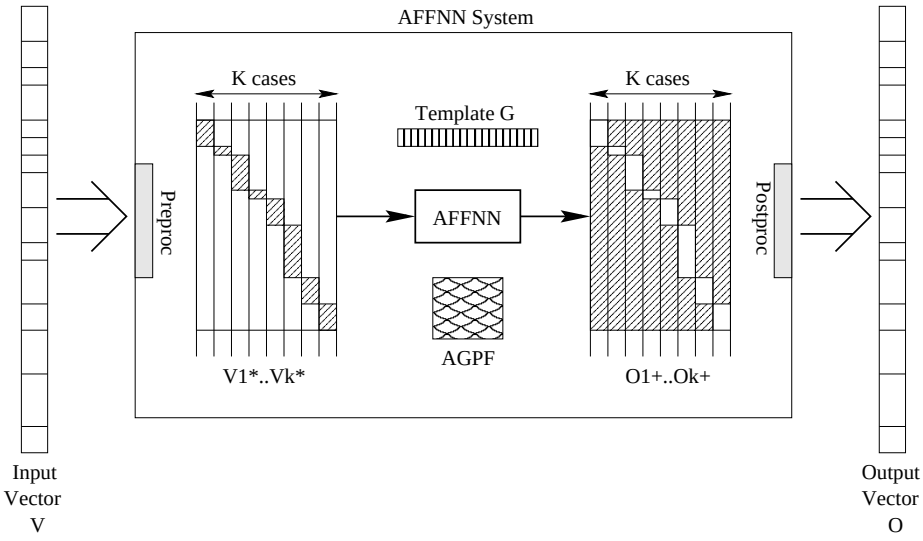


Figure 2: The AFFNN System

Target vectors must also be provided since the AFFNN is trained with supervised algorithms. K target vectors are generated for each original input vector V in the same way the extended input vectors are created, but with the template mask in G reversed: extended output vectors O_{i+} only contain values in the locations corresponding to the masked positions in V_{i*}. These extended output vectors are only used during network training, when target values are required. Since the extended input (source) and output (target) vectors are masked with 0's based on the same original input vector, the relationship between these vectors can be expressed by:

$$\forall i : V_{i^*} = V \cdot G_{i^*} \tag{2}$$

$$\forall i : O_{i+} = V \cdot G_{i+} \tag{3}$$

$$\forall i : V = V_{i^*} + O_{i+} \tag{4}$$

where G_{i^*} and G_{i+} are mask vectors based on the template vector G, with G_{i^*} containing 0's at elements corresponding to attribute i and 1's elsewhere, and G_{i+} containing 1's at elements corresponding to attribute i and 0's elsewhere. Note relationships 3 and 4 only hold for the training data; target vectors are not computed for general AFFNN use after training has been performed.

When $T * K$ input cases are applied to the AFFNN, $T * K$ output cases are produced. The post-processing step merges these intermediate system outputs into T cases, corresponding to the original T input cases. Any specific output vector O_{i+} only contains valid data for attribute i at positions indicated by G_{i+}, and G_{i+} for a specific i is mutually exclusive of all other positions in the output vector. This allows the results $O_{1+}...O_{K+}$ to be combined into a single output vector O by masking and summing each O_{i+} at G_{i+}, visually represented in Figure 3:

$$O = \sum_{i=1}^{K} (f_{net}(V_{i^*}) \cdot G_{i+}) \tag{5}$$

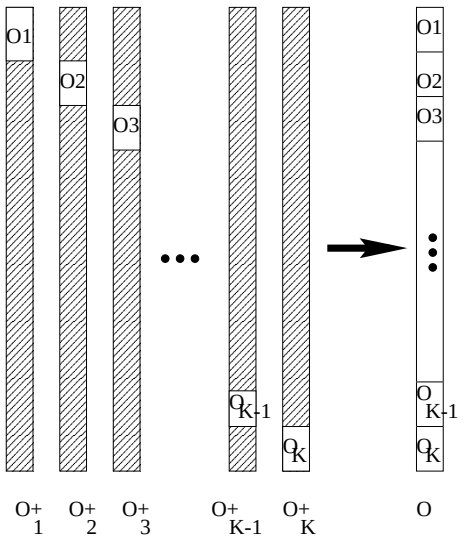


Figure 3: Combining Outputs

where $f_{net}(V_{i^*})$ represents the actual output of the AFFNN for a specific V_{i^*}. Such exclusivity enables the simultaneous training of all K attributes, and for any specific attribute i in O, the relation $O_{i^+} = f_{net}(V_{i^*})$ expressed in Equation 1 still holds.

Note that the postprocessing operation is the reverse of preprocessing, collecting $T * K$ output vectors from the AFFNN and combining each set of K vectors into a single output vector O. Resultant output vectors are valid for all K attributes in the vectors. That is, observing some G_i-masked portion of O for any i will yield a result according to Equation 1, guaranteeing the values of attribute i were computed by the AFFNN only based on the other $K - 1$ attributes[1].

2.2 Training the Aggregate Feedforward Neural Network

The AFFNN is trained like other connectionist systems, by methodically presenting all encoded and preprocessed training cases to the network. By extending the input and output vectors, each input vector V in the dataset is presented to the network as V_{i^*} for every value in $i = 1..K$. This effectively cycles the training through all attributes represented in V. Extended input and output vectors are used with a supervised batch training algorithm, but the key to training with these extended vectors lies in the definition of the AGPF performance function wedge. Figure 4 illustrates the movement of a single extended input vector V_{i^*} through the AFFNN during network training and demonstrates how the AGPF wedge uses G_i to modify O_a, the raw (actual) AFFNN output.

As Figure 4 shows, presenting an arbitrary (extended) training case V_{i^*} to the AFFNN results in an actual output vector O_a containing (or converging toward) the expected target output vector X_{i^+}, which contains the target component X_i. The problem is, while all of the target vector X_{i^+} is cleared except for the desired component X_i, the actual output vector O_a contains "garbage," non-zero values outside of the component X_i. Since these excess values don't matter to the training, it is important that they not adversely impact training by "falsely" contributing to neural network training error.

[1] The accuracy of these outputs will be bounded by the accuracy of the trained network.

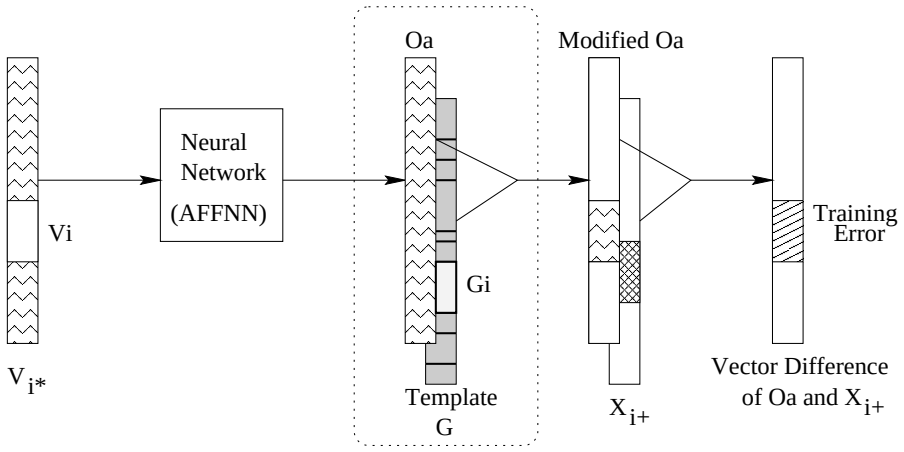


Figure 4: Single Input

The AFFNN performance function wedge alleviates this problem by using the template vector G to clear all components of O_a except for the G_i components (corresponding to the desired components X_i) prior to passing O_a forward for error evaluation:

$$O_a \leftarrow O_a \cdot G_{i+} \tag{6}$$

The resulting modified actual output vector is compared with the expected target vector X_{i+} in the computation of error using MSE or other appropriate error functions. This procedure is performed on every output case, and the cleaned outputs are evaluated by the network's true error function in determining training error, in turn feeding the supervised training algorithm. All of training is completed this way.

In reality, this process is done for the entire array of output cases, since training occurs in batch mode. Administrative data included in the AFFNN definition includes matrices configured to enable the template vector mask to be quickly and efficiently applied along the length of the entire actual extended output array. The raw "corrected" network outputs are a continuous array of $T * K$ cases, which are postprocessed back to the original T cases before leaving the AFFNN system.

Training commences when the prepared input cases are provided to the AFFNN system preprocessor, and proceeds with the performance function wedge as described. The fully trained AFFNN is able to process previously unseen inputs just like other connectionist models, as long as the input cases are preprocessed and extended using the same techniques as the training data.

2.3 Visualizing the Trained AFFNN

It is possible to view the trained AFFNN as a collection of physically separate networks that "coincidentally" have many of the same weights. This technique, demonstrated in Figure 5 (as a de-aggregation of the AFFNN example in Figure 1), emphasizes the similarity among the aggregate network structure while simultaneously calling attention to the distinct differences in the way the individual portions of the AFFNN are used, depending on the output attribute desired and its corresponding input vector V_{i*}. These "independent" networks form the supporting basis for rule extraction, discussed in the next section.

The solid nodes (circles) and links (lines) in each figure reflect the path taken by the data when considering the training and/or solution for each specific attribute. For example, Figure 5(a) only uses

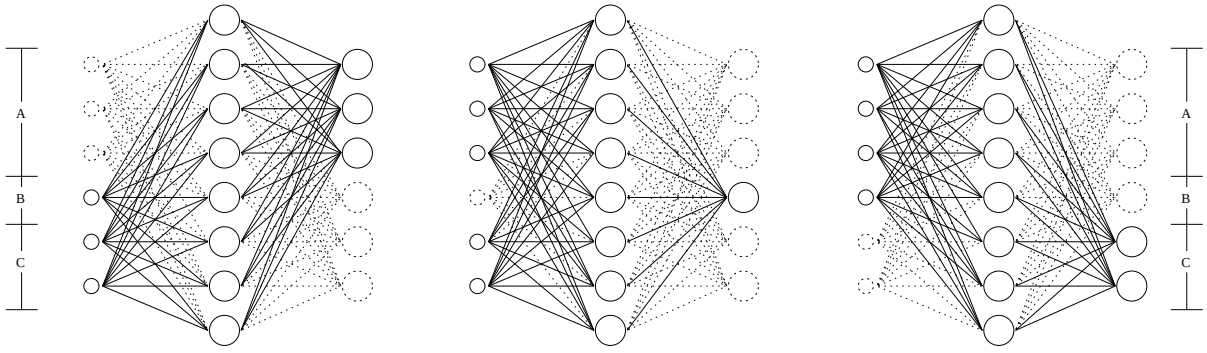


Figure 5: De-Aggregating the AFFNN

input data from attributes B and C to produce outputs for attribute A, explicitly ignoring inputs from attribute A. This usage corresponds to $O_{1+} = f_{net}(V_{1*})$, where attribute A is indicated by the encoding "1," as shown in Figure 1. The B and C attribute data becomes the input to all nodes in the hidden layer, and the results are only passed to the outputs designated for attribute A[2]. Figure 5(b) operates in the same manner, ignoring attribute B inputs for obtaining attribute B outputs; similar methods are employed when attribute C in Figure 5(c).

It is important to note that an attribute j is "reused" in every V_{i*}, $i \neq j$. For example, inputs from attribute C (i=3) contribute when determining results for both attributes A (V_{1*}) and B (V_{2*}) (Figures 5(a) and 5(b)). The weights leading from input attribute C to the hidden layer nodes are identical in both cases, since they are the exact same links in the full AFFNN. It is these weights the AFFNN is challenged to discover during training.

3 Using the AFFNN for Rule Extraction

The previous section briefly described how the AFFNN works in preparation for discussing rule extraction. The ultimate goal is to find rules describing the functions provided by the network's output nodes. One way to achieve this goal is to use a basic decompositional extraction recipe. Such approaches tend to follow the same basic schema, although a large number of variations exist:

1. Select an output node,
2. Find the clusters of values leading from the hidden layer to the selected output (clustering algorithms abound),
3. Determine which cluster values support the desired output,
4. Compute the network inputs producing the necessary cluster values, and
5. Substitute the results to obtain the final solution.

[2]The results from the hidden layer are actually passed to all output nodes, but only the outputs corresponding to attribute A are valid when V_{1*} is the network input. All other attribute outputs are invalid by definition.

The fully trained AFFNN contains a complete picture of the the relationships among the data elements. The objective of rule extraction is to coax the network into revealing the knowledge it contains. This section explains how a decompositional technique can be used to generate rules using the AFFNN as an extraction platform.

The authors designed Algorithm 1 specifically for the AFFNN following the example of typical decompositional methods. Like RX, this algorithm begins at the final network layer and works in reverse for a single node in the output layer. Unlike RX, however, each expression derived at the output layer is a collection of terms, treated as an atomic unit that is either satisfied or discarded, depending on the potential support provided at the input nodes.

Algorithm 1 Generalized Knowledge Extraction

Given:

- Output node N_j for which rules are desired
- Range of values R of N_j for which extracted rules should be valid
- One set L_i corresponding to each node H_i in the hidden layer, where L_i is the set of clusters of values traveling along the links from the hidden node to node N_j

Compute:

1. Create an empty candidate expression set E
2. For every distinct set S of cluster values, selecting one value from each L_i:
   (a) Compute the output x of N_j when values in S are used
   (b) If x is in range R, then S is a reasonable "expression"; add set S to expression list E
3. Simplify the candidate expressions in E
4. Create an empty result expression set F
5. For each candidate expression C in E:
   (a) Find the set of all input nodes P (and their values) that satisfy all terms in C
   (b) If P is non-empty then add P to F
6. Set F contains information describing the inputs required to satisfy the desired range R of node N_j

Algorithm Returns:

- Set F, enumerating rules for range R of node N_j in terms of input values

The objective is to find rules describing the functions at the output nodes. For a given output node, contributing hidden layer nodes are found, and the contributing input layer nodes are computed for each contributing hidden layer node. This data is combined into terms, which are used to derive rules. The rules are turned into an algorithm, which is easy to read, validate, and use. This algorithm also serves as a product, able to perform classifications apart from the trained neural network.

These ideas are more easily seen in the example of Figure 6, a relabeled replication of Figure 5(c). When examining the output attribute C for this AFFNN, only the output nodes labeled c_1 and c_2 are

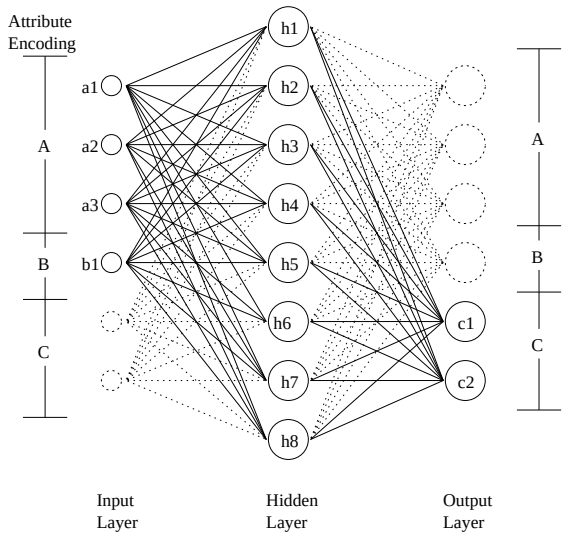


Figure 6: AFFNN Interpreted for Attribute C

useful. The solid lines from nodes $h_1..h_8$ are the links of interest from all nodes in the hidden layer to the output nodes c_1 and c_2. There are "expressions" involving these hidden nodes that describe their contributions to the outputs of c_1, and also of c_2. Expressions describe output node functions in terms of hidden node values, and are expressed as sets of numbers obtained from the clusters associated with each hidden node.

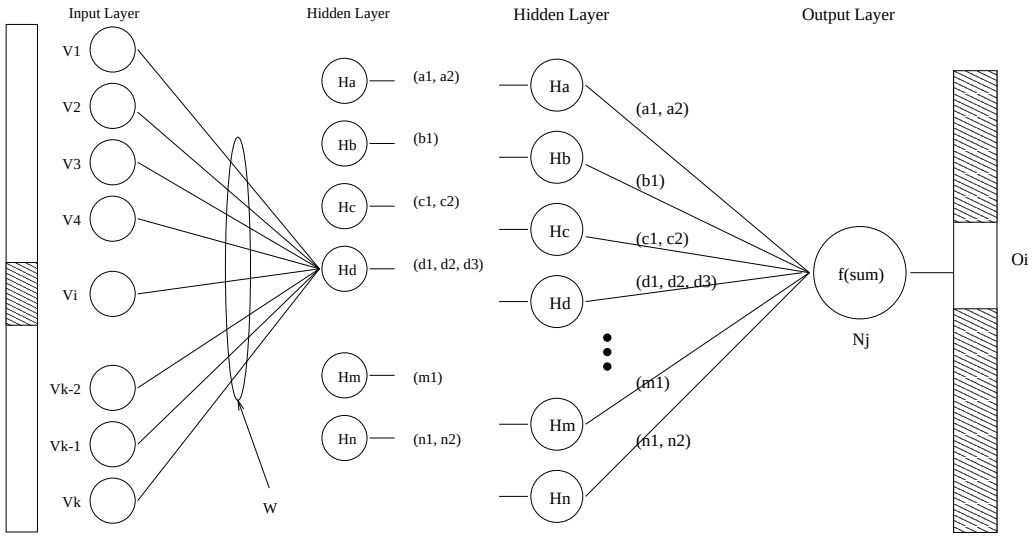
For any specific value of hidden node h_i, some combination of inputs connected to h_i satisfies the h_i "term" of the expression. (Input nodes corresponding to c_1 and c_2 cannot contribute to any h_i when outputs for attribute C are being considered, by definition of the AFFNN. This is indicated by the dashed lines leaving attribute C inputs in the figure.) Once the contributing inputs are identified, a series of rules can be developed describing the relationship from network input to output.

Figures 7(a) and 7(b) illustrate the concepts identified in Algorithm 1 for a generic AFFNN. Figure 7(b) represents the nodes of the hidden layer feeding a single node N_j of the output layer. During training, it becomes evident that a cluster of values is associated with each link, as shown in the figure. Certain combinations of the values on these links will produce the desired output node values. Each unique combination of hidden node cluster values makes up an expression, where there are $\prod_i |H_i|$ total expressions when $|H_i|$ represents the number of cluster values for hidden node H_i. The exhaustive process of testing all combinations of these values corresponds to steps 1–3 of the algorithm. For examples, one valid expression might be:

$$(a1 \;\; b1 \;\; c2 \;\; d1 \;\; \ldots \;\; m1 \;\; n2) \tag{7}$$

indicating that if the corresponding nodes in the hidden layer produce values belonging to these clusters, the desired value of N_j will be obtained. A valid expression is an expression producing the desired output at the specified output node, and is a referred to as a candidate expression. The final list of candidate expressions enumerates all combinations of hidden node values producing outputs in the expected range for the specified output node.

Only candidate expressions undergo additional testing, with their terms individually checked against the possible values from the input layer. Testing these expressions is an iterative process similar to the method already introduced for initially finding expressions. Since candidate expression terms are simply



(a) Exposing Expression Terms (b) Finding Output Expressions

Figure 7: Finding Expressions

the outputs of the nodes in the hidden layer, it is possible to determine the set of inputs producing the desired hidden node output. All terms of an expression must be tested. An expression is true if, and only if, there is a set of inputs that produce the desired hidden node values for all terms in the expression.

Figure 7(a) depicts the input nodes leading to a single node in the hidden layer. This figure depicts the the links leading from the nodes in the input layer to the nodes in the hidden layer. Clusters of values travel along these links much like the clusters of values traveling on links from the hidden layer to the output layer. Any combination of these values leading to the desired term is collected into a new set S_m of contributing input expressions for hidden node H_i. (Recall, the cluster values of input nodes belonging to the same attribute as output node N_j are not permitted to be used for testing terms, according to the definition of the AFFNN.) After contributing inputs for all terms have been identified, the intersection of all sets ($\bigcap_m S_m$) defines the set of inputs supporting all terms of the expression. Said another way, a variety of inputs may support an individual term, but all terms must be supported for the entire expression to be supported; thus, the inputs satisfying the expression are exactly those inputs satisfying all individual terms in the expression. Exposing expression terms as described coincides with algorithm steps 4 and following. The resulting expressions define the output with respect to the original input values.

Simple collections of *if-then* rules can be almost trivially generated once the final expressions are found. The quality of these rules depends almost entirely on the quality of the expressions, which in turn heavily rely on the accuracy of the system from which the expressions are derived. As an example, an expression suggesting that attributes A_1 and B_2 and C_1 are required to satisfy the solution set can be written as the rule:

```
if (A1 and B2 and C1) then return true;
```

where A1, B2, and C1 represent specific values of the attributes A, B, and C. An entire series of similarly generated rules reflects the extracted expressions, providing an equivalent solution that can be verified by subject-matter experts.

One convenient side-effect of such rules is that they can also be generated directly into a macro language, allowing them to be executed as code. Creating rules in the form of code provides a strict structure that assists in forcing the generation algorithms to write concise, expressive, and compliant rules. The AFFNN uses MATLAB as a notation for expressing rules. The generated rules are executable MATLAB code, capable of independently processing the same datasets used with the AFFNN.

4 Comparative Results

There are many practical methods for extracting rules from data, as well as an abundance of reasonably high quality, publicly available datasets. Two problem domains (datasets) and a sample of published techniques are given here as evidence that extracting rules from the AFFNN is both sound and practical. The domains selected are popular (well-known among researchers) and have special characteristics that make them valuable for testing the AFFNN and demonstrating that it is useful as a platform for rule extraction. Before presenting the AFFNN-based solutions, a rationale for the selection of each problem domain is presented. Comparisons to other techniques are given in each section below, as applicable.

Problem M_2 of the MONK's problems is an excellent example for the AFFNN to solve. It is a well-defined problem with a known solution, and it is a simple matter to exhaustively test all cases of the solution. Problem M_2 is small, making a very reasonable proof-of-concept example for use during algorithm development and testing. It is also reasonable easy to train the AFFNN for M_2, and the training is not extensive. This means that algorithmic modifications can be made and tested quickly using M_2.

Fisher's "Iris Plants Database"[3] is probably the most well-known classification dataset in existence. The goal is to determine the class of iris plant from four continuous-valued attributes. There are 50 examples of each class: Iris Setosa, Iris Versicolor, and Iris Virginica. According to published information, the first class is linearly separable from the others two, which are not linearly separable from one-another.

All of the AFFNN solutions were generated using custom MATLAB (v5.3r11) models built by the authors using the MATLAB Neural Network Toolbox [Demuth and Beale(1998)]. Model execution and testing was performed on a Linux-based dual Pentium-II 300 MHz machine with 128Mb RAM and 1Gb swap space.

4.1 The MONK's Problems

The three MONK's problems are fabricated datasets in which robots are described using seven (7) attributes, listed in Table 1 [Thrun et al.(1991)Thrun, Bala, Bloedorn, Bratko, Cestnik, Cheng, De Jong, Džeroski, Fahlman, Fisher, Hamann, Kaufman, Keller, Kononenko, Kreuziger, Michalski, Mitchell, Pachowicz, Reich, Vafaie, Van de Welde, Wenzel, Wnek, and Zhang]. Attributes $x_1..x_6$ are the original attributes describing the problem set, and attribute x_0 was inserted to indicate whether or not the case belongs to the solution set. For each of the three problems, a robot belongs to the solution set if it meets the problem constraints.

The original problem statement requires that only a predefined subset of the 432 combinations of attributes $x_1..x_6$ is given as training data, with the entire dataset used for testing. The defined training subset consists of 169 training cases, 40% of the problem space, containing both positive and negative examples with no noise in the data. The second problem (M_2) is described as:

[3]Fisher,R.A. "The use of multiple measurements in taxonomic problems" Annual Eugenics, 7, Part II, 179-188 (1936).

Table 1: Attributes of the MONK's Problems

Variable	Description	Possible Values
x_0	in solution set?	no, yes
x_1	head shape	round, square, octagon
x_2	body shape	round, square, octagon
x_3	is smiling?	yes, no
x_4	holding	sword, balloon, flag
x_5	jacket color	red, yellow, green, blue
x_6	has tie?	yes, no

Problem M_2: (exactly two of the six attributes $x_1..x_6$ have their *first* value)

4.1.1 The AFFNN Solution

There are nineteen (19) total attributes if all options of $x_0..x_6$ are counted. One AFFNN solving Problem M_2 is a 19-12-19 network with sigmoid activation functions in the hidden layer and linear activation functions in the output layer. The template vector for the encoding of these boolean-valued attributes is:

$$G = [1\,\underline{1}\,2\,2\,2\,3\,3\,3\,4\,4\,5\,5\,5\,6\,6\,6\,6\,7\,7]^T \tag{8}$$

where the encoding of each attribute is indicated by the corresponding number in G. The underscored element in G is the encoded position of attribute 1 representing $x_0 = yes$ ("member of the solution set"). This position corresponds to the 2nd node in the output layer, the node for which M_2 rules are generated.

Following 1000 epochs of training in 360 seconds, the number of clustered elements for each of the 12 hidden nodes was, respectively, { 1, 1, 4, 1, 4, 3, 4, 1, 1, 19, 1, 1 }. This results in 3648 ($\prod_{i=1}^{12} |H_i|$) unique expressions, the set of all combinations cluster values. After following Algorithm 1, the list of those expressions generating the desired output of $N_2 = 1$ contains 14 candidate expressions after minimization. The output node $N_2 = 1$ when any one of the candidate expressions is true.

Automatically generating *if-then* rules for these expressions produces 15 combinations of inputs that satisfy all of the terms of at least one of the candidate expressions. Therefore, output N_2 is satisfied if any of these final 15 rules are satisfied. These rules, generated to reflect the original unencoded attributes, basically follow the form:

```
if (
     (x1 <= 1) and
     (x2 <= 1) and
     (x3 > 1) and
     (x4 > 1) and
     (x5 > 1) and
     (x6 > 1)
) then result=true;
```

According to this example rule, if attributes x1 and x2 (head_shape and body_shape) have their initial values and all other attribute do not, the output is true. This rule demonstrates the desired result, and the generation algorithm created this and 14 additional similar rules.

For the minimal set of rules satisfying M_2, all of $x_1..x_6 \neq 1$ except for:

(x1==1 && x2==1)	(x2==1 && x3==1)	(x3==1 && x4==1)	(x4==1 && x5==1)
(x1==1 && x3==1)	(x2==1 && x4==1)	(x3==1 && x5==1)	(x4==1 && x6==1)
(x1==1 && x4==1)	(x2==1 && x5==1)	(x3==1 && x6==1)	
(x1==1 && x5==1)	(x2==1 && x6==1)		(x5==1 && x6 ==1)
(x1==1 && x6==1)			

Figure 8: Minimal Set of Rules Satisfying M_2

Table 2: Solutions to the M_2

Algorithm Name & Type	Type	M_2 % Correct	Comments
mFOIL	Inductive Learning	69.2%	FOIL-based
ID3	Inductive Learning	67.9%	Decision Trees
Backpropagation	Connectionist	100.0%	17-2-1 Network
AFFNN	Connectionist	100.0%	19-12-19 Network

Problem M_2 is small, and the set of expressions satisfying the solution set can be easily enumerated, shown in Figure 8. The minimal rule set for M_2 is only 15 rules. The autogenerated AFFNN rule set of 15 rules exactly corresponds to the rules listed in Figure 8, and correctly classifies 100% of the test data.

4.1.2 Comparative Results

Thrun prepared a report comparing many algorithms for learning the MONK's problems, where researchers advocating their methods contributed their own results to this collaborative report [Thrun et al.(1991)Thrun, Bala, Bloedorn, Bratko, Cestnik, Cheng, De Jong, Džeroski, Fahlman, Fisher, Hamann, Kaufman, Keller, Kononenko, Kreuziger, Michalski, Mitchell, Pachowicz, Reich, Vafaie, Van de Welde, Wenzel, Wnek, and Zhang]. A small subset of the reported problem M_2 solutions is reproduced in Table 2, with data from the AFFNN added.

In fairness, many of the systems in Thrun's report performed much better on problems M_1 and M_3, but those replicated in this table show M_2 results that were typical of most systems; connectionist and cascade correlation systems tended to perform well, where most others were in the 65-75% arena for problem M_2. The mFOIL algorithm, generating 19 Prolog clauses, performed poorly due to a low number of training examples. The ID3 example is based on the classical Top-down Induction of Decision Trees by Quinlan, selecting the "best" partition of the training examples. (Quinlan's various decision-tree-based algorithms continue to be the measuring rod for much rule-building research.) ID3 produced a decision tree with 66 nodes and 110 leaves.

Thrun reported the training time of his own 17-2-1 neural network to be in the neighborhood of 10-30 seconds on a SUN SparcStation, and approximately 5 seconds on a Connection Machine CM-2. In comparison, the authors were able to conduct an independent test replicating this network with a training time of just over 6 seconds on a dual Pentium-II 300MHz machine. Thrun's network did not produce any "rules," but did classify all test examples accurately. The trained AFFNN correctly classified all test examples, in addition to using the rule extraction algorithm to produce the minimum set of exactly 15 *if-then* rules shown in Figure 8. The extracted rules correctly classified 100% of the data with no misclassifications.

4.2 The Iris Problem

This dataset consists of fifty examples each of three different types of iris plants (Iris Setosa, Iris Versicolor, or Iris Virginica), where each sample contains four distinct measurements: sepal length, sepal width, petal length, and petal width. The goal for classifiers is to determine the type of iris plant based on these four attributes. Results presented here are based on the corrected dataset, where samples #35 and #38 have been corrected. The full dataset, including corrections, can be obtained from the UCI Machine Learning Repository [Murphy and Aha(1992)].

4.2.1 The AFFNN Solution

An AFFNN with a 21-9-21 configuration took 89 seconds to train 1000 epochs on this dataset, where the odd entries in the original data were used for training and the even entries for testing. Training utilized all five attributes (sepal length, sepal width, petal length, petal width, and iris class), with the first four being thermometer-encoded using 4, 3, 9, and 2 inputs, and the class attribute binned into 3 inputs. The trained AFFNN classified the Iris Setosa to 100% accuracy in the training set and 99% in the test set; the Iris Versicolor to 100% accuracy in the training set and 99% in the test set; and the Iris Virginica to 95% accuracy in the training set and 93% in the test set.

The hidden nodes contained between 1 and 9 clustered values, resulting in a total of 17280 expressions. The automated rule generation algorithm applied to the fifth attribute (iris class) produced 10 rules classifying the Iris Setosa to 100% accuracy in both the training and test sets, 5 rules classifying the Iris Versicolor to 91% and 88% accuracy in the training and test set respectively, and 10 rules classifying the Iris Virginica to 95% and 89% accuracy in the training and test sets. Rule extraction time, including generation of the executable rule files, was under 30 seconds in all cases.

At the same time the AFFNN learned the relationships of the three iris classes, it also learned to classify the Petal Width attribute as a function of all other attributes with 99% accuracy in the test set. When considering this output, hidden nodes contained from 1 to 5 clusters yielding 9375 expressions, and the resulting 15 rules performed with 96% and 89% accuracy on the training and test sets, respectively. The AFFNN also learned Sepal Length at 72% accuracy with respect to the other four attributes, Sepal Width at 80%, and Petal Length at 75% on the test set. This additional learning demonstrates the utility and functionality of the AFFNN model, showing the ability of the AFFNN to learn multiple relationships within the same network.

4.2.2 Comparative Results

Engelbrecht and Viktor use the Iris Problem in an evaluation of their approach to use sensitivity analysis for locating decision boundaries within neural networks [Engelbrecht and Viktor(1999)]. The method they propose applies first-order derivatives to the neural network outputs with respect to the input patterns for finding the decision boundaries. This technique is used in conjunction with their ANNSER rule extraction algorithm to find the rules describing iris data in 4-2-3 neural network containing sigmoid activation functions. Inputs to this system were scaled to the range of [-1, 1], and pruning reduced the network to a 2-2-3 system with an accuracy of 95.9% on the test set. (This system randomly selected 105 cases for the training set and used the remaining 45 cases in the test set.) The ANNSER rule extraction algorithm produced rules:

Rule_1: if petal_length $<$ 19.5 and petal_width $<$ 6.5 then Setosa

Rule_2: if petal_length $>$ 49.5 and petal_width $>$ 16.5 then Virginica

Rule_3: if petal_length $<$ 49.5 then Versicolor

Rule_4: if petal_width $<$ 18.5 then Versicolor

The test accuracy of this rule set is reported to be 95.9% with individual rules ranging from an accuracy of 93.9% to 100%.

Setiono and Liu present interesting and useful results using a version of the RX algorithm [Setiono and Liu(1995)]. Their feedforward 39-3-3 neural network uses discretized inputs, where sepal length, sepal width, petal length, and petal width are thermometer-encoded into 16, 9, 7, and 6 inputs, respectively. The final pruned network is a 4-2-3 net, and their extraction algorithm produces the rules:

> Rule 1: if petal_length <= 1.9 then Iris Setosa
>
> Rule 2: if petal_length <= 4.9 and petal_width <= 1.6 then Iris Versicolor
>
> Default: Iris Virginica

These compact rules correctly classify 98.67% of the training set and 97.33% of the test set, where rules in the odd positions in the original dataset were used for training, and the even numbered rules for testing. The authors did not indicate the training or rule extraction time for this model. Similar results are obtained using RG [Setiono and Liu(1996)].

The AFFNN system performance was very competitive, with the network correctly classifying test data at 99%, 99%, and 93% accuracy for the Iris Setosa, Versicolor, and Virginica. The rule extraction mechanism introduced as a proof-of-concept generated a separate set of rules from this 21-9-21 AFFNN describing each class of iris flower, with rule results ranging in accuracy from 88% to 100% as reported above. The rule extraction algorithm uses a straight forward decompositional approach intended to demonstrate how knowledge can be obtained from the AFFNN, and does not claim to generate a minimum number of rules for any given dataset.

The rules generated for the Iris Problem are very promising, although the number of extracted rules is somewhat greater than those produced by Setiono. This is largely due to the greater number of hidden nodes in the AFFNN. Another factor is the performance of the AFFNN's discretization algorithm and where it "chose" to partition the inputs. (This discretization algorithm yielded 21 more coarsely divided network inputs for the AFFNN, versus the 39 inputs used in Setiono's example.) Pruning and retraining the AFFNN might also reduce the number of rules and improve their accuracy; pruning algorithms have not yet been introduced into the AFFNN design.

5 Conclusions

Exploratory data analysis often suggests looking at the same data along several different axes, investigating the impact of data attributes on a single output attribute. Using a neural approach for individually analyzing K attributes of a dataset would generally require K distinct neural networks, each with one attribute as an output and all other attributes as inputs. Examining all of these relationships typically means training K individual networks, each with its own unique set of weights and, possibly, its own topology. The Aggregate Feedforward Neural Network architecture developed by the authors allows all attributes to be simultaneously trained using the same physical network. This is an important characteristic for rule extraction, allowing rules to be produced for any of the network outputs without requiring separate networks to be trained.

One distinct advantage of the AFFNN is that it is more of a technique, a methodology, than a topology. The AFFNN is designed to be extended and modified without loss of generality, especially with respect to these features:

1. Usable with any suitable supervised training algorithm — There is no implicit reliance on backpropagation algorithms or their variants.

2. No reliance on specific performance functions — MSE is common, but cross-entropy functions like those used in NeuroRule, or any other reasonable functions can be applied.

3. Add hidden layers — Fully or partially connected hidden layers, best applied after the initial fully connected hidden layer, may be added as long as the selected training algorithm supports them.

4. Results can be de-aggregated — The user satisfied with the training performed on a particular attribute can "lift" the corresponding de-aggregated network from the AFFNN, as shown in Figure 5. The resulting network can be used as-is, or additional training and pruning can be applied in an effort to improve results further.

5. Not tied to specific rule extraction — An original decompositional algorithm demonstrates the ability to extract rules from the AFFNN, but any practical extraction algorithm could be applied.

6. Rule notation — Automatically generated rules are expressed using MATLAB as a convenient notation. These rules can independently process the same datasets as the AFFNN. It is not necessary to use this notation, however; any suitable rule base or notation can be generated from the expressions extracted from the AFFNN.

The simple and original decompositional rule extraction mechanism developed by the authors for use with the AFFNN demonstrates one reasonable approach for extracting learned relationships from within this system. Although the extraction mechanism does not guarantee a minimal rule set, it does generate a small and understandable collection of portable rules. The examples of AFFNN training and rule extraction documented herein show that this connectionist model is a viable architecture that could be particularly well-suited to data mining activities, especially when coupled with practical rule extraction techniques.

Despite these successes, there are also specific drawbacks to using the AFFNN, including training time and convergence. The AFFNN will take longer to train than a "targeted" neural network because it will be a larger network (with more input, output, *and* hidden nodes) than a similar network trained for a single attribute. In most cases, this larger network will require more training epochs, and perhaps even more data, to train with the same accuracy as a more specialized network. This is clearly the case when comparing Thrun's neural solution to the M_2 problem; his compact 17-2-1 network trains in less time and fewer epochs than the AFFNN solution.

Convergence is a very important issue, and studies are currently under way to define the conditions under which the AFFNN is known to converge. This architecture will not always converge, especially if there are not enough hidden nodes to support the functional requirements of the network. There will also be cases where the dataset itself contributes to non-convergence, most notably when data is poorly partitioned according to the anticipated rule set.

The AFFNN is a new research effort. Many current questions are still unanswered, and more questions arise as the research effort continues. Intermediate results appear to be promising, and developments continue to be made. Clearly the AFFNN is an interesting architecture that lends itself to data mining activities. The results presented in this report using these small benchmarking problems merely qualify the AFFNN for continued research.

References

[Andrews et al.(1995)Andrews, Diederich, and Tickle] Robert Andrews, Joachim Diederich, and Alan B. Tickle. A survey and critique of techniques for extracting rules from trained artificial neural networks. Technical report, Queensland University of Technology, Queensland, Australia, 1995.

[Craven and Shavlik(1999)] Mark Craven and Jude Shavlik. Rule extraction: Where do we go from here? Technical report, University of Wisconsin, 1999. Machine Learning Research Group Working Paper 99-1.

[Craven and Shavlik(1997)] Mark W. Craven and Jude W Shavlik. Using neural networks for data mining. *Submitted to the Future Generation Computer Systems special issue on Data Mining*, 1997.

[Demuth and Beale(1998)] Howard Demuth and Mark Beale. *Neural Network Toolbox User's Guide*. Natick, third edition, 1998.

[Engelbrecht and Viktor(1999)] A. P. Engelbrecht and H. L. Viktor. Rule improvement through decision boundary detection using sensitivity analysis. In *IWANN (2)*, pages 78–84, Alicante, Spain, 1999. URL `citeseer.nj.nec.com/engelbrecht99rule.html`.

[Lu et al.(1995a)Lu, Setiono, and Liu] Hongjun Lu, Rudy Setiono, and Huan Liu. Effective data mining using neural networks. In *Proceedings of the 21st VLDB Conference*, Zurich, Swizerland, 1995a.

[Lu et al.(1995b)Lu, Setiono, and Liu] Hongjun Lu, Rudy Setiono, and Huan Liu. NeuroRule: A connectionist approach to data mining. In *Proceedings of the 21st VLDB Conference*, Zurich, Swizerland, 1995b.

[Murphy and Aha(1992)] P. M. Murphy and D. W. Aha. UCI repository of machine learning databases. Department of Information and Computer Sciences, 1992. University of California, Irvine.

[Opitz and Shavlik(1995)] David W. Opitz and W. Shavlik, Jude. Dynamically adding symbolically meaningful nodes to knowledge-based neural networks. *Knowledge-Based Systems*, 8(6):301–311, 1995.

[Schmidt and Chen(2002)] Vincent A. Schmidt and C. L. Philip Chen. Defining an aggregate feedforward neural network. Submitted to ANNIE02 Conference, 2002.

[Setiono(1997)] Rudy Setiono. A penalty function for pruning feedforward neural networks. *Neural Computation*, 9(1):185–204, 1997.

[Setiono(2000)] Rudy Setiono. Extracting M-of-N rules from trained neural networks. *IEEE Transactions on Neural Networks*, 11(2):512–519, 2000.

[Setiono and Liu(1995)] Rudy Setiono and Huan Liu. Understanding neural networks via rule extraction. In *Proceedings of the 14th International Conference on Artificial Intelligence*, pages 480–485, Montreal, Canada, 1995.

[Setiono and Liu(1996)] Rudy Setiono and Huan Liu. Symbolic representation of neural networks. *IEEE Computer*, 29(3):71–77, 1996.

[Thrun et al.(1991)Thrun, Bala, Bloedorn, Bratko, Cestnik, Cheng, De Jong, Džeroski, Fahlman, Fisher, Hamann, Kau
S.B. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. De Jong, S. Džeroski, S.E. Fahlman, D. Fisher, R. Hamann, K. Kaufman, S. Keller, I. Kononenko, J. Kreuziger, R.S. Michalski, T. Mitchell, P. Pachowicz, Y. Reich, H. Vafaie, W. Van de Welde, W. Wenzel, J. Wnek, and J. Zhang. The MONK's problems: A performance comparison of different algorithms. Technical report, Carnegie Mellon University, 1991. CMU-CS-91-197.

[Towell and Shavlik(1992)] Geoffrey G. Towell and Jude W. Shavlik. Using symbolic learning to improve knowledge-based neural networks. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 177–182, San Jose, CA, 1992. AAAI/MIT Press.

[Towell and Shavlik(1993)] Geoffrey G. Towell and Jude W. Shavlik. Extracting refined rules from knowledge-based neural networks. *Machine Learning*, 13:71–101, 1993.

[Zaki(1998)] Mohammed Javeed Zaki. *Scalable Data Mining for Rules*. PhD thesis, University of Rochester, Rochester, NY, 1998.