Dynamically Adding Symbolically Meaningful Nodes to Knowledge-Based Neural Networks

David W. Opitz and **Jude W. Shavlik**

Computer Sciences Department
University of Wisconsin – Madison
1210 W. Dayton St.
Madison, WI 53706
{opitz, shavlik}@cs.wisc.edu

Abstract

Traditional connectionist theory-refinement systems map the dependencies of a domain-specific rule base into a neural network, and then refine this network using neural learning techniques. Most of these systems, however, lack the ability to refine their network's topology and are thus unable to add new rules to the (reformulated) rule base. Therefore, on domain theories that are lacking rules, generalization is poor, and training can corrupt the original rules, even those that were initially correct. We present TopGen, an extension to the KBANN algorithm, that heuristically searches for possible expansions to KBANN's network. TopGen does this by dynamically adding hidden nodes to the neural representation of the domain theory, in a manner analogous to adding rules and conjuncts to the symbolic rule base. Experiments indicate that our method is able to heuristically find effective places to add nodes to the knowledge bases of four real-world problems, as well as an artificial chess domain. The experiments also verify that new nodes must be added in an intelligent manner. Our algorithm showed statistically significant improvements over KBANN in all five domains.

Keywords: network-growing algorithm
theory refinement
the KBANN algorithm
computational biology

Submitted to the journal *Knowledge-Based Systems*.

Dynamically Adding Symbolically Meaningful Nodes to Knowledge-Based Neural Networks

David W. Opitz and **Jude W. Shavlik**

1 Introduction

The task of *theory refinement* is to improve approximately correct, domain-specific inference rules (commonly called a *domain theory*) using a set of data (Ginsberg, 1990; Pazzani & Kibler, 1992; Ourston & Mooney, 1994; Towell & Shavlik, 1994). In this paper, we concentrate on connectionist theory-refinement systems, since they have been shown to frequently generalize[1] better than many other inductive-learning and theory-refinement systems (Fu, 1989; Towell, 1991; Lacher et al., 1992; Tresp et al., 1992; Mahoney & Mooney, 1993). Most of these systems, however, suffer in that they do not refine the topology of the network they create. We address this problem by presenting a new approach to connectionist theory refinement, particularly focusing on the task of automatically adding nodes to a "knowledge-based" neural network.

KBANN (Towell & Shavlik, 1994) is an example of a connectionist theory-refinement system; it translates a set of propositional rules into a neural network, thereby determining the network's topology. It then applies backpropagation to refine these reformulated rules. KBANN has been shown to generalize to previously unseen examples better than many other inductive learning algorithms (Towell, 1991). Towell and Shavlik (1994) attribute KBANN's superiority over other symbolic systems to both its underlying learning algorithm (i.e., backpropagation) and its effective use of domain-specific knowledge.

However, connectionist theory-refinement systems that do not alter their network's topology, such as KBANN, are restricted in the types of refinements they can make to the domain theory. KBANN, for instance, essentially only adds and subtracts antecedents of existing rules and is thus unable to add new symbolic rules to the rule set. In fact, Towell and Shavlik (1992) showed that, while KBANN is reasonably insensitive to extra rules in a domain theory, its ability to generalize degrades significantly as rules are removed from a theory. (We further empirically verify this in Section 2).

In addition, with sparse domain theories, KBANN needs to significantly alter the original rules in order to account for the training data. While it is clearly important to classify the examples as accurately as possible, changes to the initial domain theory should be kept to a minimum because the domain theory presumably contains useful information, even if it is not completely correct. Also, large changes to the domain theory can greatly complicate rule extraction following training (Towell & Shavlik, 1993).

Hence, our goal is to expand, during the training phase, knowledge-based neural networks — networks whose topology is determined as a result of the direct mapping of the

[1] We use *generalize* to mean accuracy on examples not seen during training.

dependencies of a domain theory — so that they are able to learn the training examples without needlessly corrupting their initial rules.

The TopGen (Topology Generator) algorithm, the subject of this paper, heuristically searches through the space of possible expansions of a knowledge-based network, guided by the symbolic domain theory, the network, and the training data. It does this by adding hidden nodes to the neural representation of the domain theory. More specifically, it first uses a symbolic interpretation of the trained network to help locate primary errors of the network, and then adds nodes to this network in a manner analogous to adding rules and conjuncts to a propositional rule base. We show that adding hidden nodes in this fashion synergistically combines the strengths of refining the rules symbolically with the strengths of refining them with backpropagation. Thus, TopGen mainly differs from other network-growing algorithms (Fahlman & Lebiere, 1989; Mezard & Nadal, 1989; Frean, 1990) in that it is specifically designed for knowledge-based neural networks.

TopGen uses beam search, rather than a faster hill-climbing algorithm, because for many domains an expert who created the domain theory is willing to wait for weeks, or even months, for an improved theory. Thus, given the rapid growth in computing power, it seems wise to search more of the hypothesis space to find a good network topology. Finding such a topology allows better generalization, provides the network with the ability to learn without overly corrupting the initial set of rules, and increases the interpretability of the network so that efficient rules may be extracted. Section 5 presents evidence for these claims.

The rest of this article is organized as follows. In the next section, we give a brief review of KBANN. We present the details of the TopGen algorithm in Section 3. This is followed by an example of how TopGen works. In Section 5 we present results from four real-world, Human Genome domains and controlled studies on an artificial domain. Finally, we discuss these results, as well as future and related work.

2 Review of KBANN

In this section, we first review how KBANN works. We then discuss the hypothesized limitations of KBANN and empirically verify these limitations.

2.1 The KBANN Algorithm

KBANN translates a set of propositional rules, representing what is initially known about a domain, into a neural network (see Figure 1). Figure 1a shows a Prolog-like rule set that defines membership in category a. Figure 1b represents the hierarchical structure of these rules, with solid lines representing necessary dependencies and dotted lines representing prohibitory dependencies. Figure 1c represents the resulting network created from this translation. KBANN creates nodes $b1$ and $b2$ in Figure 1c to handle the two rules deriving b in the rule set. The thin lines in Figure 1c represent low-weighted links that KBANN adds to allow refinement of these rules during backpropagation training. Biases are set so that nodes representing disjuncts have an output near 1 only when at least one of

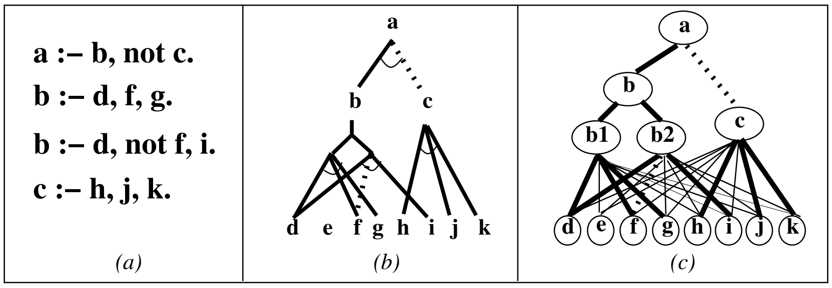


Figure 1: KBANN's translation of a knowledge base into a neural network. Panel (a) shows a sample propositional rule set, panel (b) shows this rule set's corresponding dependency tree, and panel (c) shows the resulting network created from KBANN's translation.

their high-weighted antecedents is correct, while nodes representing conjuncts must have all of their high-weighted antecedents correct (i.e., near 1 for positive links and near 0 for negative links). Otherwise activations are near 0. Refer to Towell (1991) or Towell and Shavlik (1994) for more details.

KBANN uses training instances to refine the network links. This training alters the antecedents of existing rules; however, KBANN does not have the capability of inducing new rules. For example, KBANN is unable to add a new rule for inferring b.

2.2 Limitations of KBANN

Towell (1991) showed, with real-world empirical evidence (the DNA promoter domain), that the generalization performance of KBANN suffers when given "impoverished" domain theories – theories that are missing rules or antecedents needed to adequatedly learn the true concept. While real-world domains are clearly useful when exploring the utility of an algorithm, they are difficult to use in closely controlled studies that examine different aspects of an algorithm. An artificial domain, however, allows one to know the relationship between the theory provided to the learning system and the correct domain theory. Thus we repeat Towell's experiments using an artificial domain, so that we can more confidently determine how much KBANN suffers when given impoverished domains.

Our artificial domain is derived from the game of chess. The concept to be learned is those board configurations where moving a king one space forward is illegal (i.e., the king would be in check). To make the domain tractable, we only consider a 4x5 subset of the chess board (shown in Figure 2). The king is currently in position $c1$ and the player is considering moving it to position $c2$ (which is currently empty) and thus wants to know if this is a legal move. Pieces considered in this domain are a queen, a rook, a bishop, and a knight for both sides. Each instance is generated by randomly placing a subset of these pieces on the remaining board positions. For example, a queen of the opposing team occupying position $b4$, and a bishop from the player's team occupying position $d3$ would comprise one instance of a legal move. (Refer to Appendix A for a detailed description of the rule set defining this domain.)

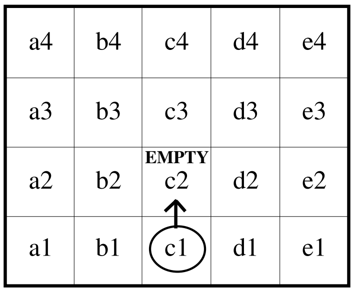


Figure 2: The 4x5 subset of a chess board considered in our domain. The king is currently on position $c1$, and will be moved to position $c2$ if this is a legal move. This move is illegal if the king would be in check on position $c2$.

In order to investigate the types of corrections that KBANN is effective at, we ran experiments where we perturbed the correct domain theory in various ways, and gave these incorrect domain theories to KBANN. We perturbed the domain theory in four different ways: (a) adding an antecedent to a rule, (b) deleting an antecedent from a rule, (c) adding a rule, or (d) deleting a rule. This perturbation was done by scanning each antecedent (rule), and probabilistically deciding whether to add another antecedent (rule), delete it, or leave it alone.

We created a new rule by first setting its consequent to be the consequent of the scanned rule. We then added from two to four antecedents, since the number of antecedents of each rule in the correct rule base also ranged from two to four. We added antecedents to both newly created and existing rules by randomly selecting from the consequents below the rule[2] as well as the input features. When deleting rules, we considered only rules whose consequent was not that of the final conclusion. Finally, when we deleted antecedents, if a rule's consequent does not appear in another rule's antecedent list (and is not the final conclusion), then this rule was deleted as well. We do this because it is unlikely that a domain expert would provide undefined intermediate conclusions in a domain theory.

Figure 3 shows the test-set error (i.e., error measured on a set of examples not seen by the learning algorithm) that results from perturbing the chess domain in each of these four ways. This error is the average of five runs of five-fold cross validation. Each five-fold cross validation is run by first splitting the data into five equal sets. Then, five times one set is held out while the remaining four sets are used to train the system. The held-out set is used to measure test-set performance of KBANN's final concept, thus giving an estimate of its generalization performance. We ran multiple (i.e., five) cross-validation experiments in order to dampen fluctuations due to the random partitioning of the data, and to judge the statistical significance of the results.

Figures 3a and 3b show that KBANN is effective at correcting spurious additions to the

[2] A consequent is "below" a rule if it has a longer path to the target node in the rule base's dependency tree (e.g., see Figure 1b).

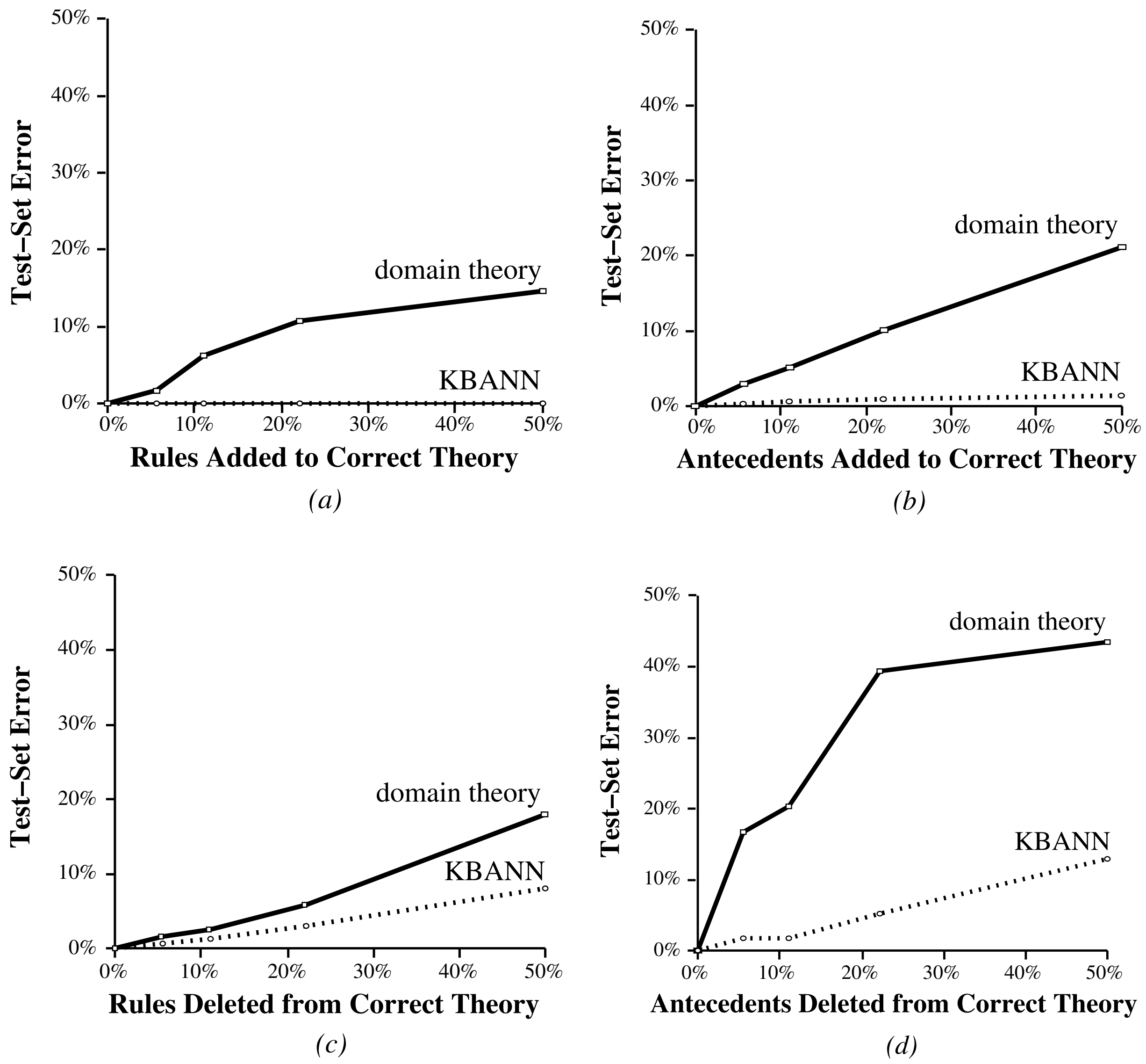


Figure 3: Impact on generalization of perturbing four different ways the correct chess-like domain theory. In all graphs, the top line (labelled *domain theory*), is the test-set error of the domain theory given to KBANN, while the bottom line (labelled KBANN) is the test-set error of KBANN after backpropagation training. The test-set error was obtained from five runs of five-fold cross validation.

domain theory; however, Figures 3c and 3d show that KBANN's generalization degrades rapidly when rules and antecedents are deleted from the domain theory. KBANN's high error rate in Figure 3d is partially due to the fact that when we deleted antecedents, we also deleted rules whose consequent no longer appeared in another rule's antecedent list (unless it was the final conclusion `illegal-move`). As a point of comparison, 26% of the rules were deleted when 50% of the antecedents were removed. Recall that being able to introduce new rules to a knowledge-based neural network is the focus of this paper.

3 The TopGen Algorithm

Our new algorithm, TopGen, introduces new rules by heuristically searching through the space of possible ways of adding nodes to the network, trying to find the network that best refines the initial domain theory (as measured using "validation sets"[3]). Briefly, TopGen looks for nodes in the network with high error rates, and then adds new nodes to these parts of the network.

Table 1 summarizes the beam-search-based TopGen algorithm. TopGen uses two validation sets, one to evaluate the different network topologies, and one to help decide where new nodes should be added (it also uses the second validation set to decide when to stop training individual networks). TopGen uses KBANN's rule-to-network translation algorithm to create an initial guess for the network's topology. This network is trained using backpropagation (Rumelhart et al., 1986) and is placed on an OPEN list. In each cycle, TopGen takes the best network (as measured by `validation-set-2`) from the OPEN list, decides possible ways to add new nodes, trains these new networks, and places them on the OPEN list. This process is repeated until reaching either (a) a `validation-set-2` accuracy of 100% or (b) a previously set time limit. Upon completion, TopGen reports the best network it found.

3.1 *Where* Nodes Are Added

TopGen must first find nodes in the network with high error rates.[4] It does this by scoring each node (which corresponds to a rule in a symbolic domain theory) using examples from `validation-set-1`. By using examples from this validation set, TopGen adds nodes on the basis of where the network fails to generalize, not where it fails to memorize the training set.

TopGen makes the empirically verified assumption that almost all of the nodes in a trained knowledge-based network are either fully active or inactive. By making this assumption, each non-input unit in a TopGen network can be treated as a step function (or

[3]TopGen splits the training set into a set of training instances and two sets of validation instances. Note that the instances in these three sets are separate from a *test* set that experimenters commonly set aside to test the generalization performance of a learning algorithm.

[4]It is important to note that methods for finding nodes with high error rates are just heuristics to help guide the search of where to add new nodes, thus TopGen is able to backtrack if a "bad" choice is made.

Table 1: The TopGen Algorithm

TopGen:

GOAL: Search for the best network describing the domain theory and training examples.

1. Disjointly separate the training examples into a training set and two validation sets (`validation-set-1` and `validation-set-2`).
2. Train, using backpropagation, the initial network produced by KBANN's rules-to-network translation and put on *OPEN* list.
3. Until *stopping criterion* reached:
   (a) Remove best network, according to `validation-set-2`, from *OPEN* list.
   (b) Use **ScoreEachNode** to determine *N* best places to expand topology.
   (c) Create *N* new networks, train and put on *OPEN* list.
   (d) Prune *OPEN* list to length *M*.
4. Output the best network seen so far according to `validation-set-2`.

ScoreEachNode:

GOAL: Use the error on `validation-set-1` to suggest good ways to add new nodes.

1. Temporarily use the threshold activation function at each node.
2. Score each node in the network as follows:
   (a) Set each node's `correctable-false-negative` and `correctable-false-positive` counters to 0.
   (b) For each misclassified example in `validation-set-1`, cycle through each node and determine if modifying the output of that node will correctly classify the example, incrementing the counters when appropriate.
3. Use the counters to order possible node corrections. High `correctable-false-negative` counts suggest adding a disjunct, while high `correctable-false-positive` counts suggest adding a conjunct.

a Boolean rule) so that errors have an all-or-nothing aspect, thus concentrating topology refinement on misclassified examples, not on erroneous portions of *individual* examples. Towell (1991), as well as self-inspection of our networks, has shown this to be a valid assumption.

TopGen keeps two counters for each node, one for false negatives and one for false positives, defined with respect to each individual node's output, not the final output. An example is considered a false negative if it is incorrectly classified as a negative example, while a false positive is one incorrectly classified as a positive example. TopGen increments counters by recording how often changing the "Boolean" value of a node's output leads to a misclassified example being properly classified. That is, if a node is active for an erroneous example and changing its output to be inactive results in correct classification for the example, then the node's false-positives counter is incremented. TopGen increments a node's false-negatives counter in a similar fashion. By checking for single points of failure, TopGen looks for rules that are *near misses* (Winston, 1975).

After the counter values have been determined, TopGen sorts these counters in descending order, while breaking ties by preferring nodes farthest from the output node. TopGen then creates N new networks, where each network contains a single correction (as determined by the first N sorted counters). Section 3.2 details *how* the nodes are added, based on the counter type (i.e., a false-negative or false-positive counter) and node type (i.e., an AND or OR node).

We also tried other approaches for blaming nodes for error, but they did not work as well on our test beds. One such method is to propagate errors back by starting at the final conclusion and recursively considering an antecedent of a rule to be incorrect if both (a) its consequent is incorrect and (b) the antecedent does not match its "target." We approximate targets by starting with the output node, then recursively considering a node to have the same target as its parent, if the weight connecting them is positive, or the opposite target, if this weight is negative. While this method works for symbolic rules, TopGen suffers under this method because its antecedents are weighted. Antecedents with small-weighted links are counted as much as antecedents with large-weighted links. Because of this, we also tried using the backpropagated error to blame nodes, however backpropagated error becomes too diffuse in networks having many layers, such as the ones often created by TopGen.

3.2 *How* Nodes Are Added

Once we estimate where we should add new nodes, we need to know *how* to add these nodes. TopGen makes the assumption that when training one of its networks, the meaning of a node does not shift significantly. Making this assumption allows us to alter the network in a fashion similar to refining symbolic rules. Towell (1991) showed that making a similar assumption about KBANN networks was valid.

Figure 4 illustrates the possible ways TopGen adds nodes to a TopGen network. In a symbolic rule base that uses negation-by-failure, we can decrease false negatives by either dropping antecedents from existing rules or adding new rules to the rule base. We showed

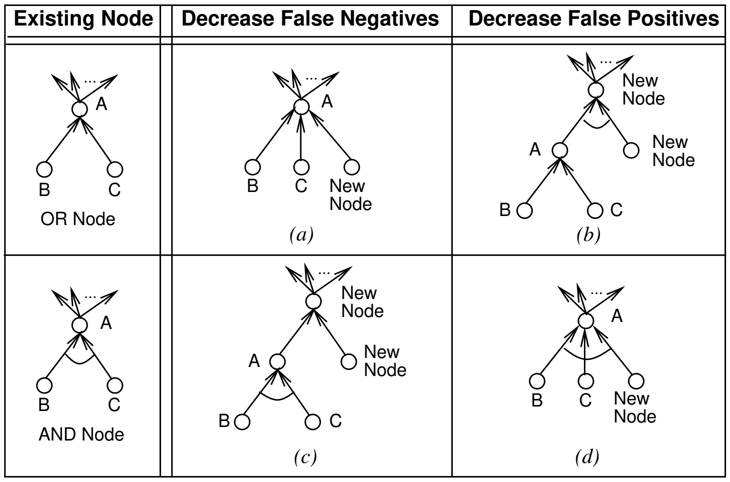


Figure 4: Possible ways to add new nodes to a knowledge-based neural network (arcs indicate AND nodes). To decrease false negatives, we wish to broaden the applicability of the node. Conversely, to decrease false positives, we wish to further constrain the node.

(in Section 2) that while KBANN is effective at removing antecedents from existing rules, it is unable to add new rules. Therefore, TopGen adds nodes, intended for decreasing false negatives, in a fashion that is analogous to adding a new rule to the rule base. If the existing node is an OR node, TopGen adds a new node as its child (see Figure 4a), and fully connects this new node to the input nodes. If the existing node is an AND node, TopGen creates a new OR node that is the parent of the original AND node and another new node that TopGen fully connects to the inputs (see Figure 4c); TopGen moves the outgoing links of the original node (A in Figure 4c) to become the outgoing links of the new OR node.

In a symbolic rule base, we can decrease false positives by either adding antecedents to existing rules or removing rules from the rule base. In Section 2, we showed that KBANN can effectively remove rules, but it is less effective at adding antecedents to rules and is unable to invent (constructively induce) new terms as antecedents. Thus TopGen adds new nodes, intended to decrease false positives, in a fashion that is analogous to adding new constructively induced antecedents to the network. Figures 4b and 4d shows how this is done (analogous to Figures 4a and 4c explained above). These mechanisms allow TopGen to add to the rule base rules whose consequents were previously undefined.

TopGen handles nodes that are neither AND nor OR nodes by deciding if such a node is "closer" to an AND node or an OR node. To be an OR node, the node's bias must be slightly greater than the sum of its *negative* incoming links; alternately, to be an AND node, the bias must be slightly less than the sum of its *positive* incoming links. Therefore, if a node's bias is closer to the summed negative weights than to the summed positive weights it is considered an OR node; otherwise, it is considered an AND node. TopGen classifies nodes it previously added in this manner, when deciding how to add more nodes to them at a later time.

3.3 Additional Algorithmic Details

After new nodes are added, TopGen must retrain the network. While we want the new weights to account for most of the error, we also want the old weights to change if necessary. That is, we want the older weights to retain what they have previously learned, while at the same time move in accordance with the change in error caused by adding the new node. In order to address this issue, TopGen multiplies the learning rates of existing weights by a constant amount (we use 0.5) every time new nodes are added, producing an exponential decay of learning rates. (In backpropagation, each inter-node link can have its own learning rate.)

We also do not want to change the domain theory unless there is considerable evidence that it is incorrect. That is, there is a trade-off between changing the domain theory and disregarding the misclassified training examples as noise. To help address this, TopGen uses a variant of weight decay (Hinton, 1986). Weights that are part of the original domain theory decay toward their initial value, while other weights decay toward zero.

Our weight-decay term, then, decays weights as a function of their distance from their initial value and is a slight variant of the term proposed by Rumelhart in 1987 (Weigend et al., 1990). The idea of our weight decay is to add, to the usual cost function, a term that measures the distance of each weight from its initial value:

$$Cost = \sum_{k \in T} (target_k - output_k)^2 + \lambda \sum_{i \in C} \frac{(\omega_i - \omega_{init_i})^2}{1 + (\omega_i - \omega_{init_i})^2}$$

The first term sums over all training examples T, while the second term sums over all connections C. The tradeoff between performance and distance from initial values is weighted by λ (we use $\lambda = 0.01$).

4 Example of TopGen

Assume that the correct version of Figure 1a's domain theory also includes the rule:

$$b \text{ :- } \neg\, d,\ e,\ g.$$

Although we trained the KBANN network shown in Figure 1c with all possible examples of the correct concept, it was unable to completely learn when a is true. TopGen, however, was able to learn this concept; it begins by training the KBANN network, obtaining little improvement to the original rule base. It then proceeds by using misclassified examples from `validation-set-1` to find places where adding nodes could be beneficial, as explained below.

Consider the following positive example of category a, which is incorrectly classified by the domain theory:

$$not\ d \wedge e \wedge f \wedge g \wedge not\ h \wedge not\ i \wedge j \wedge k$$

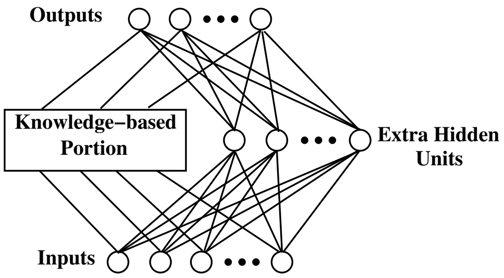


Figure 5: Topology of the networks used by Strawman.

While node c (from Figure 1c) is correctly false in this example, node b is incorrectly false. B is false since both $b1$ and $b2$ are false. If b had been true, this example would have been correctly classified (since c is correct), so TopGen increments the `correctable-false-negative` counter for b. TopGen also increments the counters of $b1$, $b2$, and a, using similar arguments.

Nodes a, b, $b1$, and $b2$ will all have high `correctable-false-negative` counters after all the examples are processed. Given these high counts, TopGen considers adding OR nodes to nodes a, $b1$, and $b2$, as done in Figure 4c, and also considers adding another disjunct, analogous to Figure 4a, to node b. Any one of these four corrections allows the network to learn the target concept. Since TopGen breaks ties by preferring nodes farthest from the output node, it prefers $b1$ or $b2$.

5 Experimental Results

In this section we test TopGen on five domains: the artificial chess-related domain introduced in Section 2, and four real-world problems from the Human Genome Project.

5.1 The Chess Subproblem

In Section 2, we showed that KBANN's generalization suffered when it was given a domain theory where rules or antecedents were deleted from the correct theory. In this section, we test TopGen's performance on these two cases to see how well it addresses these limitations of KBANN.

To help test the efficiency of TopGen's approach for choosing where to add hidden nodes, we also compare its performance with a simple approach (referred to as *Strawman* hereafter) that adds one layer of fully connected hidden nodes "off to the side" of the KBANN network. Figure 5 shows the topology of such a network. The topology of the original KBANN network remains intact, while we add extra hidden nodes in a fully connected fashion between the input and output nodes. If a domain theory is impoverished, it is reasonable to hypothesize that simply adding nodes in this fashion would increase performance. Strawman trains 50 different networks (using weight decay), ranging from

0 to 49 extra hidden nodes and, like TopGen, uses a validation set to choose the best network.

Our initial experiment addresses the generalization ability of TopGen when given domain theories that are missing rules or antecedents. Figure 6 reports the test-set error when we randomly delete rules and antecedents from the chess domain theory (see Section 2 for details on how we generated these domain theories). The plotted results are the averages of five runs of five-fold cross-validation.

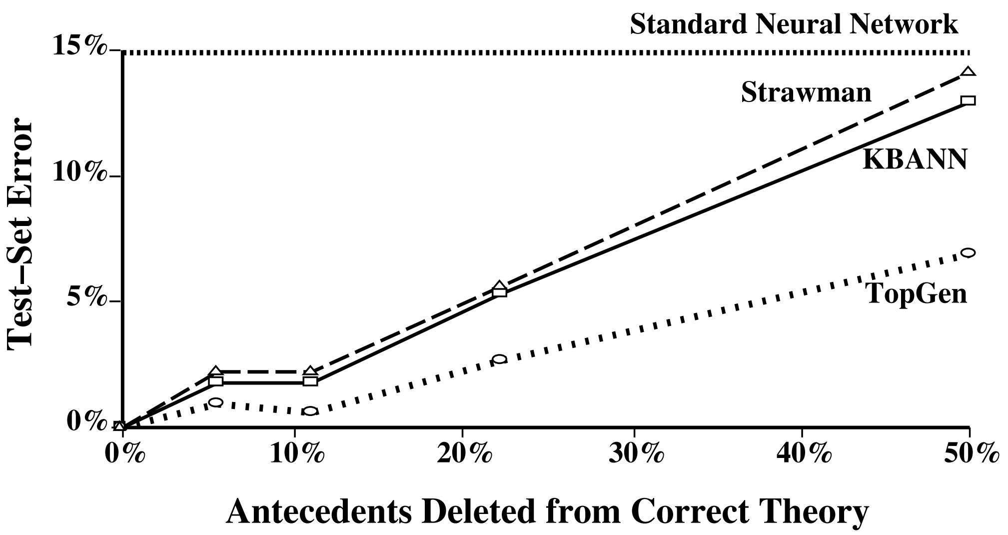
The top horizontal line in each graph results from a fully connected, single-layer, feed-forward neural network. For each fold, we trained various networks containing up to 100 hidden nodes and used a validation set to choose the best network. This line is horizontal because the networks do not use any of the domain theories. Graphing this line gives a point of reference for the generalization performance of a standard neural network; thus, comparisons to knowledge-based neural networks show the utility of being able to exploit the domain theory.

The next two curves in each graph report KBANN's and Strawman's performance; notice Strawman produced almost no improvement over KBANN in either case. Finally, TopGen, the bottom curve in each graph, had a significant increase in accuracy, having an error rate of about half that of both KBANN and Strawman. Like Strawman, TopGen considered 50 networks during its search.

When 50% of the rules or antecedents were deleted, one-tailed, two-sample *t*-tests indicate that the difference between TopGen and KBANN (t=11.14 when deleting rule, t=13.54 when deleting antecedents) and the difference between TopGen and Strawman (t=10.93 when deleting rules, t=14.78 when deleting antecedents) are significant at the 99.5% confidence level. As a point of comparison, when 50% of the rules were deleted, TopGen added 22.4 nodes on average, while Strawman added 22.2 nodes; when 50% of the antecedents were deleted, TopGen added 21.2 nodes, while Strawman added 5.1 nodes.

As stated earlier, it is important to correctly classify the examples while deviating from the initial domain theory as little as possible. Because the domain theory may have been inductively generated from past experiences, we are concerned with semantic distance, rather than syntactic distance, when deciding how far a learning algorithm has deviated from the initial domain theory. Also, syntactic distance is difficult to measure, especially if the learning algorithm generates rules in a different form than the initial domain theory. However, we can estimate semantic distance by considering only those examples in the test set that the original domain theory classifies correctly. Error on these examples indicates how much the learning algorithm has corrupted *correct* portions of the domain theory.

Figure 7 shows accuracy on the portion of the test set where the original domain theory is correct. When the initial domain theory has few missing rules or antecedents (less than 15%), neither TopGen, KBANN, nor Strawman overly corrupt this domain theory in order to compensate for these missing rules; however, as more rules or antecedents are deleted, both KBANN and Strawman corrupt their domain theory much more than TopGen does. Two-sample, one-tailed *t*-tests indicate that TopGen differs from both



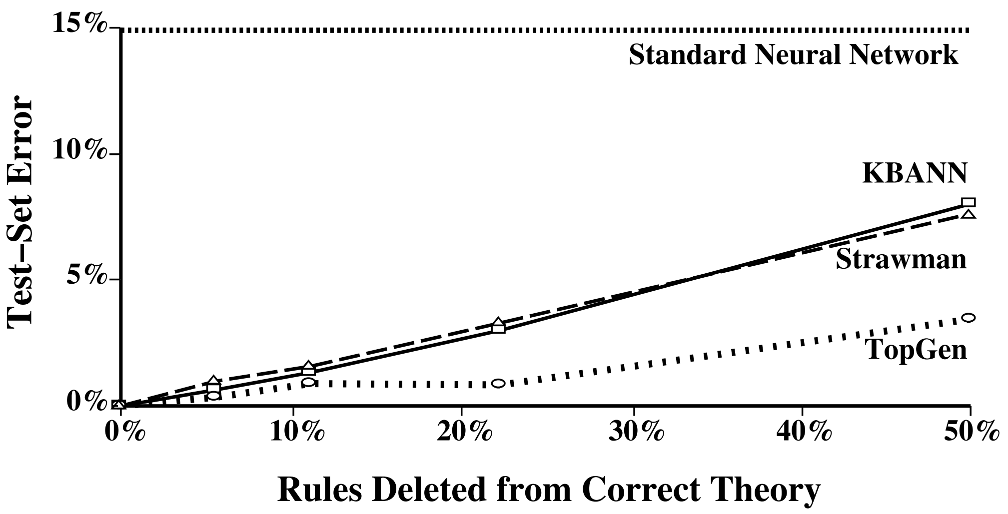


Figure 6: Test-set error of four learners on the chess problem. In both graphs, the Y-axis is the mean test-set error obtained from five runs of five-fold cross validation.

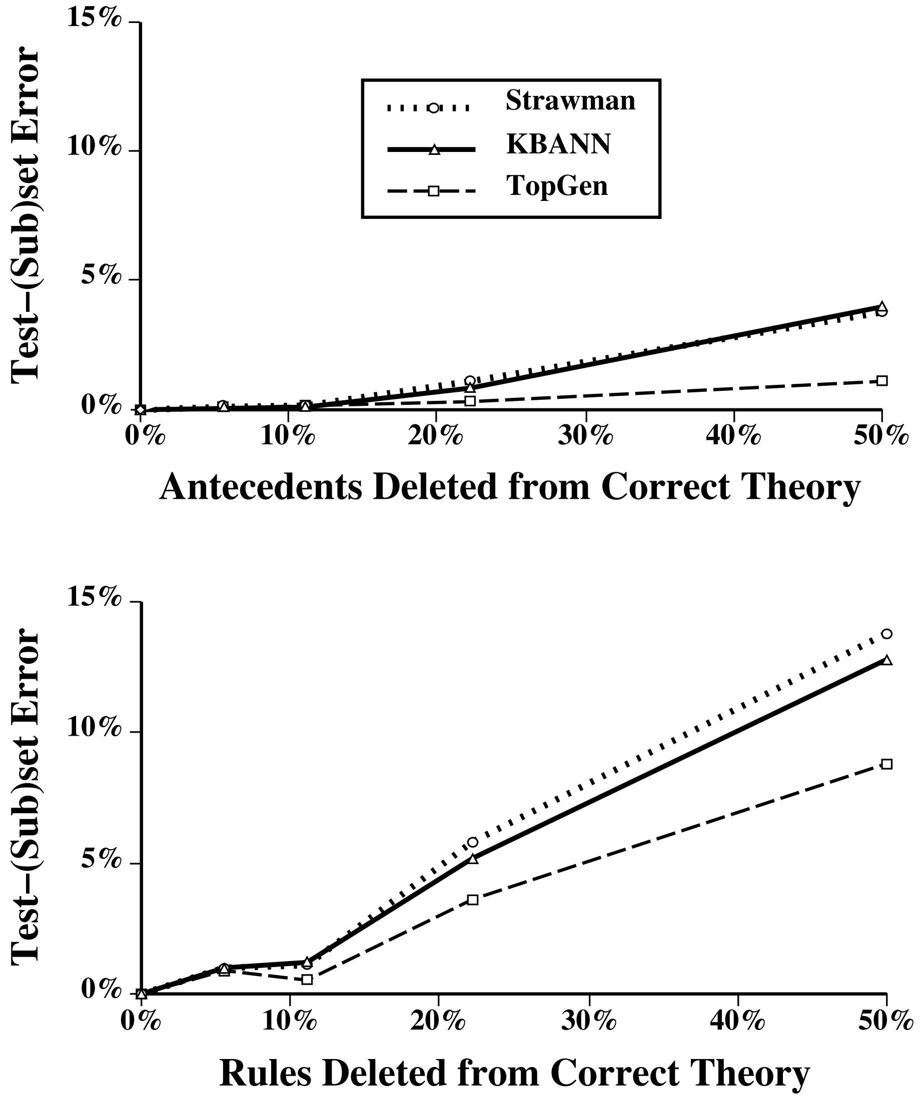


Figure 7: Error rate of the various learners on the subset of the test set where the corrupted domain theory is correct. In both graphs, the Y-axis is the mean test-set error obtained from five runs of five-fold cross validation.

Table 2: Total number of nodes added (on average).

Domain	TopGen	Strawman
RBS	8.2	8.0
Splice Junction	4.0	5.2
Promoters	4.4	5.0
Terminators	9.4	1.2

KBANN and Strawman at the 99.5% confidence level when 50% of the rules or antecedents are deleted.

5.2 Four Human-Genome Domains

We also ran TopGen on four problems from the Human Genome Project. Each of these problems aid in locating genes in DNA sequences. The first domain, *promoter recognition*, contains 234 positive examples, 4,921 negative examples, and 17 rules. (Note that this data set and domain theory are a larger version of the one that appears in Towell, 1991 and Towell and Shavlik, 1994). The second domain, *splice-junction determination*, contains 3,190 examples distributed among three classes, and 23 rules (Noordewier et al., 1990). The third domain, *transcription termination sites*, contains 142 positive examples, 5,178 negative examples, and 60 rules. Finally, the last domain, *ribosome binding sites* (RBS), contains 366 positive examples, 1,511 negative examples, and 17 rules. See Craven and Shavlik (1994) for a description of these tasks.

Our experiment addresses the test-set accuracy of TopGen on these domains. The results in Figure 8 show that TopGen generalizes better than does both KBANN and Strawman. These results are averages of five runs of five-fold cross-validation, where TopGen and Strawman consider 20 networks during their search. Two-sample, one-tailed t-tests indicate TopGen differs from both KBANN and Strawman at the 97.5% confidence level on all four domains, except with Strawman on the promoter domain.

Table 2 shows that TopGen and Strawman added about the same number of nodes on all domains, except the terminator data set. On this data set, adding nodes off to the side of the KBANN network, in the style of Strawman, usually decreases accuracy. Therefore, whenever Strawman picked a network other than the KBANN network, its generalization usually decreased. Even with Strawman's difficulty on this domain, TopGen was still able to effectively add nodes to increase performance.

6 Discussion and Future Work

Towell (1991) has shown that KBANN generalizes better than many machine learning algorithms on the promoters and splice-junctions domains, including purely symbolic approaches to theory refinement. Yet, even though a domain expert (M. Noordewier)

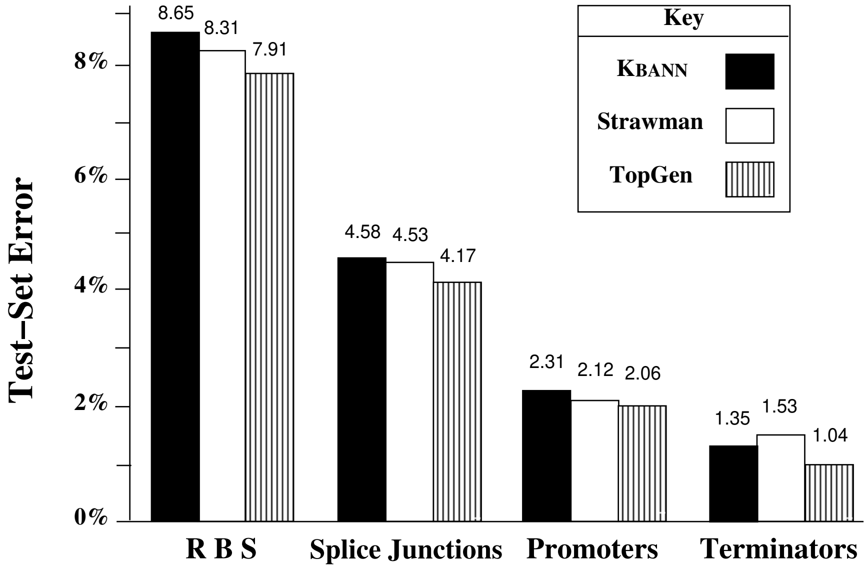


Figure 8: Error rates on four Human Genome problems.

believed the four Human Genome domain theories were large enough for KBANN to adequately learn the concepts, TopGen is able to effectively add new nodes to the corresponding network. The effectiveness of adding nodes in a manner similar to reducing error in a symbolic rule base, is verified with comparisons to a naive approach for adding nodes. If a KBANN network, resulting from an impoverished domain theory, suffered only in terms of capacity, then adding nodes between the input and output nodes would have been just as effective as TopGen's approach to adding nodes. The difference between TopGen and this naive approach (Strawman) is particularly pronounced on the terminator data set.

TopGen has a longer runtime than KBANN;[5] however, we believe this is a prudent investment, because for many domains an expert is most concerned about generalization, and is willing to wait for an extended period of time for an increase in predictive accuracy. Thus, given the rapid increase in computing power, we believe it is important to trade off the expense of large numbers of computer cycles for gains in predictive accuracy. A future plan of ours is to implement a parallel version of TopGen. In doing so, we hope to increase the number of networks considered, from 20 networks (the maximum number for the Genome results presented in this paper) to over a thousand networks, and in the process, obtain even better results.

As we increase the number of networks considered, we also want to increase the variety of networks considered during the search. This is important in tasks where the domain theory is far from the true concept. Currently we only consider expansions to the KBANN network; however, we plan to use techniques such as genetic algorithms (Holland, 1975)

[5]TopGen's runtime is dominated by the training process of each network; thus, TopGen's runtime is longer than KBANN's by approximately the number of networks it considers during its search.

or simulated annealing (Kirkpatrick et al., 1983) to help broaden the types of networks considered during the search.

Also, since we are searching through many candidate networks, it is important to be able to recognize the networks that are likely to generalize the best. We currently use a validation set; however, a validation set can be a poor estimate of the true error (MacKay, 1992). Also, as we increase the number of networks considered, TopGen may start selecting networks that overfit the validation set. Future work, then, is to investigate selection methods that do not use a validation set, which would also allow us to use all the training instances to train the networks. Such techniques include minimum-description-length methods (Rissanen, 1983), Generalized Prediction Error (Moody, 1991), and Bayesian methods (MacKay, 1992).

Future work also includes using a rule-extraction algorithm (Fu, 1991; McMillan et al., 1992; Towell & Shavlik, 1993) to measure the interpretability of a refined TopGen network. We hypothesize that TopGen builds networks that are more interpretable than naive approaches of adding nodes, such as the approach taken by Strawman. Trained KBANN networks are interpretable because (a) the meaning of its nodes does not significantly shift during training and (b) almost all the nodes are either fully active or inactive (Towell & Shavlik, 1993). Not only does TopGen add nodes in a symbolic fashion, it adds them in a fashion that does not violate these two assumptions.

Other future work includes extensively testing other approaches for locating error. Even though this is only a heuristic to help guide the search, a good heuristic will allow more efficient search of the hypothesis space. Methods of using the back-propagated error as well as symbolic techniques for determining error have been tested, but did not improve performance, for reasons explained in Section 3.1. A future research direction includes trying variants of these techniques.

A final research direction includes testing new ways of adding nodes to the network. Newly added nodes are currently fully connected to all the input nodes. Other possible approaches include: adding them to only a portion of the inputs, adding them to nodes that have a high correlation with the error, or adding them to the next "layer" of nodes.

7 Related Work

The most obvious related work is the KBANN system (Towell, 1991), described in detail earlier in this paper. Fletcher and Obradovic (1993) also present an approach that adds nodes to a KBANN network. Their approach constructs a single layer of nodes, fully connected between the input and output units, off to the side of KBANN, in a style similar to Strawman. Their approach differs from Strawman mainly in the training of the network, as well as the fact that only one network is considered during their search. Their new hidden units are determined using a variant of Baum and Lang's (1991) constructive algorithm. Baum and Lang's algorithm first divides the problem-domain space with hyperplanes, thereby determining the number of new hidden units to be added. Then the weights between the new hidden units and the inputs units are determined by these hyperplanes. Finally, the weights between the hidden-unit layer and the output layer are learned by the perceptron algorithm. Fletcher and Obradovic's approach does not change the weights of the KBANN portion of the network, so modifications to the initial rule base are solely left to the constructed hidden nodes.

The DAID algorithm (Towell & Shavlik, 1992), another extension to KBANN, uses the domain knowledge to help train the KBANN network. Because KBANN is more effective at dropping antecedents than adding them, DAID tries to find potentially-useful inputs features not mentioned in the domain theory. DAID backs-up errors to the lowest level of the domain theory, computes correlations with the features, and increases the weight value of potentially useful features. In summary, DAID tries to locate low-level *links* with errors, while TopGen searches for *nodes* with errors.

Connectionist theory-refinement systems have also been developed to refine rule bases that are not propositional in nature. For instance, a number of systems have been proposed for revising certainty-factor rule bases (Fu, 1989; Lacher et al., 1992; Mahoney & Mooney, 1993), finite-state automata (Omlin & Giles, 1992; Maclin & Shavlik, 1993), and mathematical equations (Roscheisen et al., 1991; Scott et al., 1992). Most of these systems also work by first translating the domain knowledge into a network, then modifying its weights with neural learning; however, of these, only RAPTURE (Mahoney & Mooney, 1993) is able modify its architecture during training. It does this by using the Upstart algorithm (Frean, 1990) to add new nodes to the network.

Additional related work includes purely symbolic theory-refinement systems. Systems such as EITHER (Ourston & Mooney, 1994) and RTLS (Ginsberg, 1990) are also propositional in nature. These systems differ from TopGen, in that their approaches are purely symbolic. Even though TopGen adds nodes in a manner analogous to how a symbolic system adds antecedents and rules, its underlying learning algorithm is "connectionist." EITHER, for example, uses ID3 for its induction component. Towell (1991) showed that KBANN was superior to EITHER on a DNA task, and as reported in Section 5, TopGen outperforms KBANN.

A final area related to TopGen is network-growing algorithms, such as Cascade Correlation (Fahlman & Lebiere, 1989), the Tiling algorithm (Mezard & Nadal, 1989), and the Upstart algorithm (Frean, 1990). The most obvious difference between TopGen and these algorithms is that TopGen uses domain knowledge and symbolic rule-refinement techniques to help determine the network's topology. A second difference is that these other algorithms restructure their network based solely on training set error, while TopGen uses a separate validation set. Finally, TopGen uses beam search, rather than hill climbing, when determining where to add nodes.

8 Conclusion

Although KBANN has previously been shown to be an effective theory-refinement algorithm, it suffers because it is unable to add new nodes (rules) during training. When domain theories are sparse, KBANN (a) generalizes poorly and (b) significantly alters correct rules in order to account for the training data. Our new algorithm, TopGen, heuristically searches through the space of possible expansions of the original network, guided by the symbolic domain theory, the network, and the training data. It does this by adding hidden nodes to the neural representation of the domain theory, in a manner analogous to adding rules and conjuncts to the symbolic rule base.

Experiments indicate that our method is able to heuristically find effective places to add nodes to the knowledge bases of four real-world problems, as well as an artificial chess domain. Our TopGen algorithm showed statistically significant improvements over KBANN in all five domains, and over a strawman approach in four domains. Hence TopGen is successful in overcoming KBANN's limitation of not being able to dynamically add new nodes. In doing so, our system increases KBANN's ability to generalize and learn a concept without needlessly corrupting the initial rules (which promises to increase the comprehensibility of rules extracted from a trained network). Thus, our system further increases the applicability of neural learning to problems having a substantial body of preexisting knowledge.

9 Acknowledgement

This work was supported by Department of Energy grant DE-FG02-91ER61129, Office of Naval Research grant N00014-93-1-0998, and National Science Foundation grant IRI-9002413. We would also like to thank Michiel Noordewier of Rutgers for creating the domain theories and data sets used in this paper.

References

Baum, E. & Lang, K. (1991). Constructing hidden units using examples and queries. In Lippmann, R., Moody, J., & Touretzky, D., editors, *Advances in Neural Information Processing Systems (volume 3)*, (pp. 904–910), San Mateo, CA. Morgan Kaufmann.

Craven, M. W. & Shavlik, J. W. (1994). Machine learning approaches to gene recognition. *IEEE Expert*, 9(2):2–10.

Fahlman, S. & Lebiere, C. (1989). The cascade-correlation learning architecture. In Touretzky, D., editor, *Advances in Neural Information Processing Systems (volume 2)*, (pp. 524–532), San Mateo, CA. Morgan Kaufmann.

Fletcher, J. & Obradovic, Z. (1993). Combining prior symbolic knowledge and constructive neural network learning. *Connection Science*, 5(4):365–375.

Frean, M. (1990). The upstart algorithm: A method for constructing and training feedforward neural networks. *Neural Computation*, 2(2):198–209.

Fu, L. (1989). Integration of neural heuristics into knowledge-based inference. *Connection Science*, 1(3):325–340.

Fu, L. (1991). Rule learning by searching on adapted nets. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, (pp. 590–595), Anaheim, CA. AAAI/MIT Press.

Ginsberg, A. (1990). Theory reduction, theory revision, and retranslation. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, (pp. 777–782), Boston, MA. AAAI/MIT Press.

Hinton, G. E. (1986). Learning distributed representations of concepts. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, (pp. 1–12), Amherst, MA. Erlbaum.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI.

Kirkpatrick, S., Gelatt, C., & Vecchi, M. (1983). Optimization by simulated annealing. *Science*, 220:671–680.

Lacher, R., Hruska, S., & Kuncicky, D. (1992). Back-propagation learning in expert networks. *IEEE Transactions on Neural Networks*, 3(1):62–72.

MacKay, D. J. (1992). A practical Bayesian framework for backpropagation networks. *Neural Computation*, 4(3):448–472.

Maclin, R. & Shavlik, J. (1993). Using knowledge-based neural networks to improve algorithms: Refining the Chou-Fasman algorithm for protein folding. *Machine Learning*, 11(2,3):195–215.

Mahoney, J. J. & Mooney, R. J. (1993). Combining connectionist and symbolic learning to refine certainty-factor rule-bases. *Connection Science*, 5(3,4):339–364.

McMillan, C., Mozer, M. C., & Smolensky, P. (1992). Rule induction through integrated symbolic and subsymbolic pr ocessing. In Moody, J., Hanson, S., & Lippmann, R., editors, *Advances in Neural Information Processing Systems (volume 4)*, (pp. 969–976), San Mateo, CA. Morgan Kaufmann.

Mezard, M. & Nadal, J.-P. (1989). Learning in feedforward layered networks: The tiling algorithm. *Journal of Physics A*, 22:2191–2204.

Moody, J. (1991). The *effective* number of parameters: An analysis of generalization and regularization in nonlinear learning systems. In Moody, J., Hanson, S., & Lippmann, R., editors, *Advances in Neural Information Processing Systems (volume 4)*, (pp. 847–854), San Mateo, CA. Morgan Kaufmann.

Noordewier, M. O., Towell, G. G., & Shavlik, J. W. (1990). Training knowledge-based neural networks to recognize genes in DNA sequences. In Lippmann, R., Moody, J., & Touretzky, D., editors, *Advances in Neural Information Processing Systems (volume 3)*, (pp. 530–536), San Mateo, CA. Morgan Kaufmann.

Omlin, C. & Giles, C. (1992). Training second-order recurrent neural networks using hints. In *Proceedings of the Ninth International Conference on Machine Learning*, (pp. 361–366), Aberdeen, Scotland. Morgan Kaufmann.

Ourston, D. & Mooney, R. (1994). Theory refinement combining analytical and empirical methods. *Artificial Intelligence*, 66(2):273–309.

Pazzani, M. & Kibler, D. (1992). The utility of knowledge in inductive learning. *Machine Learning*, 9(1):57–94.

Rissanen, J. (1983). A universal prior for integers and estimation by minimum description length. *Annals of Statistics*, 11(2):416–431.

Roscheisen, M., Hofmann, R., & Tresp, V. (1991). Neural control for rolling mills: Incorporating domain theories to overcome data deficiency. In Moody, J., Hanson, S., & Lippmann, R., editors, *Advances in Neural Information Processing Systems (volume 4)*, (pp. 659–666), San Mateo, CA. Morgan Kaufmann.

Rumelhart, D., Hinton, G., & Williams, R. (1986). Learning internal representations by error propagation. In Rumelhart, D. & McClelland, J., editors, *Parallel Distributed Processing: Explorations in the microstructure of cognition. Volume 1: Foundations.* MIT Press, Cambridge, MA.

Scott, G., Shavlik, J., & Ray, W. H. (1992). Refining PID controllers using neural networks. *Neural Computation*, 5(4):746–757.

Towell, G. (1991). *Symbolic Knowledge and Neural Networks: Insertion, Refinement, and Extraction.* PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, WI.

Towell, G. & Shavlik, J. (1992). Using symbolic learning to improve knowledge-based neural networks. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, (pp. 177–182), San Jose, CA. AAAI/MIT Press.

Towell, G. & Shavlik, J. (1993). Extracting refined rules from knowledge-based neural networks. *Machine Learning*, 13(1):71–101.

Towell, G. & Shavlik, J. (1994). Knowledge-based artificial neural networks. *Artificial Intelligence*, 70.

Tresp, V., Hollatz, J., & Ahmad, S. (1992). Network structuring and training using rule-based knowledge. In Moody, J., Hanson, S., & Lippmann, R., editors, *Advances in Neural Information Processing Systems (volume 5)*, (pp. 871–878), San Mateo, CA. Morgan Kaufmann.

Weigend, A., Rumelhart, D., & Huberman, B. (1990). Generalization by weight-elimination with application to forecasting. In Lippmann, R., Moody, J., & Touretzky, D., editors, *Advances in Neural Information Processing Systems (volume 3)*, (pp. 875–882), San Mateo, CA. Morgan Kaufmann.

Winston, P. (1975). Learning structural descriptions from examples. In Winston, P., editor, *The Psychology of Computer Vision*, (pp. 157–210), New York. McGraw-Hill.

A Correct Domain Theory for the Chess Domain

The following rules define chess-board configurations, for a 4x5 board (see Figure 2), where moving a king one space forward is illegal (i.e., the king would be in check). There are five input features for each position on the board: one each for the four kinds of pieces (a queen, a rook, a bishop, and a knight), and one indicating the color of the piece. The rules are broken into three parts: (1) in check diagonally from the queen or bishop, (2) in check horizontally or vertically from a queen or rook, and (3) in check from a knight. Some rules require a space to be empty between a piece's position and position *c2* (where the player wants to move the king). For instance, if a queen of the opposite color is at position *c4*, then space *c3* must be empty in order for it to be able to put the king in check.

```
illegal_move :- diagonal_check.
illegal_move :- parallel_check.
illegal_move :- knight_check.

diagonal_check :- [queen(a4) OR bishop(a4)], opposite_color(a4), empty(b3).
diagonal_check :- [queen(e4) OR bishop(e4)], opposite_color(e4), empty(d3).
diagonal_check :- [queen(b3) OR bishop(b3)], opposite_color(b3).
diagonal_check :- [queen(d3) OR bishop(d3)], opposite_color(d3).
diagonal_check :- [queen(b1) OR bishop(b1)], opposite_color(b1).
diagonal_check :- [queen(d1) OR bishop(d1)], opposite_color(d1).

parallel_check :- [queen(c4) OR rook(c4)], opposite_color(c4), empty(c3).
parallel_check :- [queen(c3) OR rook(c3)], opposite_color(c3).
parallel_check :- [queen(a2) OR rook(a2)], opposite_color(a2), empty(b2).
parallel_check :- [queen(b2) OR rook(b2)], opposite_color(b2).
parallel_check :- [queen(d2) OR rook(d2)], opposite_color(d2).
parallel_check :- [queen(e2) OR rook(e2)], opposite_color(e2), empty(d2).

knight_check :- knight(b4), opposite_color(b4).
knight_check :- knight(d4), opposite_color(d4).
knight_check :- knight(a3), opposite_color(a3).
knight_check :- knight(e3), opposite_color(e3).
knight_check :- knight(a1), opposite_color(a1).
knight_check :- knight(e1), opposite_color(e1).

empty(b3) :- not queen(b3), not rook(b3), not bishop(b3), not knight(b3).
empty(c3) :- not queen(c3), not rook(c3), not bishop(c3), not knight(c3).
empty(d3) :- not queen(d3), not rook(d3), not bishop(d3), not knight(d3).
empty(b2) :- not queen(b2), not rook(b2), not bishop(b2), not knight(b2).
empty(d2) :- not queen(d2), not rook(d2), not bishop(d2), not knight(d2).
```