Appears in *Connection Science*, 8 (3-4), 1996.

Actively Searching for an Effective Neural-Network Ensemble

David W. Opitz[*]
Computer Science Department
University of Minnesota
10 University Drive
Duluth, MN 55812
opitz@cs.umt.edu

218-726-6149
Fax: 218-726-8240

Jude W. Shavlik
Computer Sciences Department
University of Wisconsin
1210 W. Dayton St.
Madison, WI 53706
shavlik@cs.wisc.edu

608-262-7784

Abstract

A neural-network ensemble is a very successful technique where the outputs of a set of separately trained neural network are combined to form one unified prediction. An effective ensemble should consist of a set of networks that are not only highly correct, but ones that make their errors on different parts of the input space as well; however, most existing techniques only indirectly address the problem of creating such a set. We present an algorithm called ADDEMUP that uses genetic algorithms to explicitly search for a highly diverse set of accurate trained networks. ADDEMUP works by first creating an initial population, then uses genetic operators to continually create new networks, keeping the set of networks that are highly accurate while disagreeing with each other as much as possible. Experiments on four real-world domains show that ADDEMUP is able to generate a set of trained networks that is more accurate than several existing ensemble approaches. Experiments also show that ADDEMUP is able to effectively incorporate prior knowledge, if available, to improve the quality of its ensemble.

*Currently at: Department of Computer Science; University of Montana; Missoula, MT 59812

1 Introduction

Many researchers have shown that simply combining the output of many predictors can generate more accurate predictions than that of any of the individual predictors (Clemen, 1989; Wolpert, 1992; Zhang et al., 1992). In particular, combining separately trained neural networks (commonly referred to as a neural-network *ensemble*) has been demonstrated to be particularly successful (Alpaydin, 1993; Drucker et al., 1994; Hansen & Salamon, 1990; Hashem et al., 1994; Krogh & Vedelsby, 1995; Maclin & Shavlik, 1995; Perrone, 1992). Both theoretical (Hansen & Salamon, 1990; Krogh & Vedelsby, 1995) and empirical (Hashem et al., 1994; Maclin & Shavlik, 1995) work has shown that a good ensemble is one where the individual networks are both accurate and make their errors on different parts of the input space; however, most previous work has either focussed on combining the output of multiple trained networks or only indirectly addressed how one should generate a good set of networks. We present an algorithm, Addemup (Accurate anD Diverse Ensemble-Maker giving United Predictions), that uses genetic algorithms to *generate* a population of neural networks that are highly accurate, while at the same time having minimal overlap on where they make their errors.

Traditional ensemble techniques generate their networks by randomly trying different topologies, initial weight settings, parameters settings, or use only a part of the training set (Alpaydin, 1993; Hansen & Salamon, 1990; Krogh & Vedelsby, 1995; Maclin & Shavlik, 1995) in the hopes of producing networks that disagree on where they make their errors (we henceforth refer to *diversity* as the measure of this disagreement). We propose instead to actively *search* for a good set of networks. The key idea behind our approach is to consider many networks and keep a subset of the networks that minimizes our objective function consisting of both an accuracy and a diversity term. Since genetic algorithms are effective in their use of global information (Holland, 1975; Goldberg, 1989), they allow us to consider a wide variety of networks during our search and are thus a logical choice for our search method. Also, in many domains we care more about generalization[1] performance than we do about generating a solution quickly. This, coupled with the fact that computing power is rapidly growing, motivates us to effectively utilize available CPU cycles by continually considering networks to possibly place in our ensemble.

Addemup proceeds by first creating an initial set of networks, then continually produces new individuals by using the genetic operators of crossover and mutation. It defines the overall fitness of an individual to be a combination of accuracy and diversity. Thus Addemup keeps as its population a set of highly fit individuals that will be highly accurate, while making their mistakes in a different part of the input space. In addition, it actively tries to generate good candidates by emphasizing the current population's erroneous examples during backpropagation training.

In this paper, we investigate using Addemup with both "standard" neural networks and *knowledge-based neural networks* (KNNs). KNNs are networks whose topologies are determined as a result of the direct mapping of a set of background rules that represent what we currently know about our task (which we hereafter refer to as a *domain theory*). Trained KNNs have been shown (Opitz, 1995; Towell & Shavlik, 1994) to frequently generalize better than many other inductive-learning techniques such as standard neural networks. While KNNs that are derived from the same set of rules may tend to agree, using KNNs allows one to have in his or her ensemble highly correct networks. In fact, experiments reported herein demonstrate that Addemup is able use KNNs to generate a more effective ensemble of networks than a wide

[1] As is typical, we use *generalization* to mean accuracy on examples not seen during training.

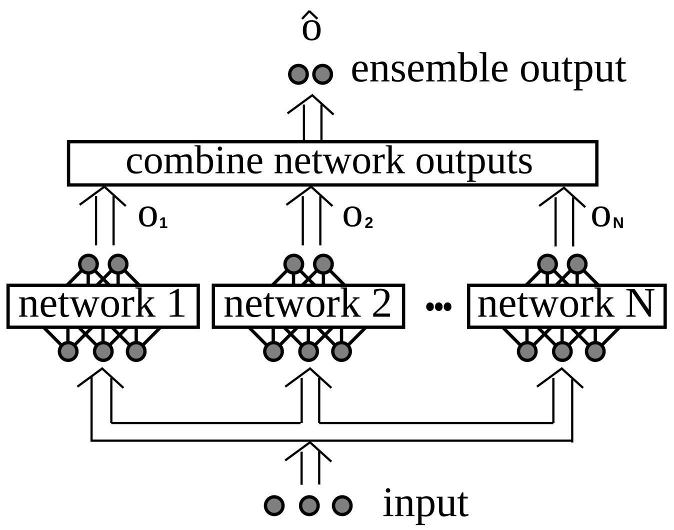

Figure 1: A neural-network ensemble.

variety of other ensemble techniques.

The rest of this article is organized as follows. We start by explaining the importance of an accurate and diverse ensemble. Next we present our new algorithm. We then present experimental results from four real-world domains. Finally, we discuss these results and review additional related work before concluding.

2 Neural Network Ensembles

Figure 1 illustrates the basic framework of a neural-network ensemble. Each network in the ensemble (network 1 through network N in this case) is first trained using the training instances. Then, for each example, the predicted output of each of these networks (o_i in Figure 1) is combined to produce the output of the ensemble ($\hat{o}$ in Figure 1). Many researchers (Alpaydin, 1993; Breiman, 1996b; Hashem et al., 1994; Krogh & Vedelsby, 1995; Lincoln & Skrzypek, 1989) have demonstrated the effectiveness of combining schemes that are simply the weighted average of the networks (i.e., $\hat{o} = \sum_{i \in N} w_i \cdot o_i$ and $\sum_{i \in N} w_i = 1$), and this is the type of ensemble on which we focus in this article, though we briefly review alternative methods in Section 6.

Combining the output of several networks is useful only if there is disagreement on some inputs. Obviously, combining several identical networks produces no gain. Hansen and Salamon (1990) proved that for a neural-network ensemble, if the average error rate for an example is less than 50% and the networks in the ensemble are independent in the production of their errors, the expected error for that example can be reduced to zero as the number of networks combined goes to infinity; however, such assumptions rarely hold in practice.

Krogh and Vedelsby (1995) later proved that the ensemble error can be divided into a term measuring the average generalization error of each individual network and a term called diversity that measures the disagreement among the networks.[2] Formally, they define the diversity term,

[2]Krogh and Vedelsby (1995) refer to diversity as *ambiguity*.

d_i, of network i on input x to be:

$$d_i(x) \equiv [o_i(x) - \hat{o}(x)]^2. \tag{1}$$

The quadratic error of network i and of the ensemble are, respectively:

$$\epsilon_i(x) \equiv [o_i(x) - f(x)]^2, \tag{2}$$

$$e(x) \equiv [\hat{o}(x) - f(x)]^2, \tag{3}$$

where $f(x)$ is the target value for input x. If we define $\hat{E}$, E_i, and D_i to be the averages, over the input distribution, of $e(x)$, $\epsilon(x)$, and $d(x)$ respectively, then the ensemble's generalization error can be shown to consist of two distinct portions:

$$\hat{E} = \bar{E} - \bar{D}, \tag{4}$$

where $\bar{E}$ $(= \sum_i w_i E_i)$ is the weighted average of the individual networks' generalization error and $\bar{D}$ $(= \sum_i w_i D_i)$ is the weighted average of the diversity among these networks. What the equation shows then, is that an ideal ensemble consists of highly correct networks that disagree as much as possible. *Creating such a set of networks is the focus of this article.*

3 The ADDEMUP Algorithm

In this section, we start by giving ADDEMUP's top-level design which describes how it searches for an effective ensemble. This is followed by the details of the particular instantiation of ADDEMUP we use in this article. Namely, we describe how we incorporate prior knowledge into neural networks, then describe how we use genetic algorithms to create new candidate networks for our ensemble.

3.1 ADDEMUP's Top-Level Design

Table 1 summarizes our algorithm, ADDEMUP, that uses genetic algorithms to generate a set of neural networks that are accurate and diverse in their predictions. ADDEMUP starts by creating and training its initial population of networks. It then creates new networks by using standard genetic operators, such as crossover and mutation.[3] ADDEMUP trains these new individuals, emphasizing examples that are misclassified by the current population, as explained below. It adds new networks to the population and then scores each population member with respect to its prediction accuracy and diversity. ADDEMUP normalizes these scores and then defines the fitness of each population member to be:

$$\mathit{Fitness}_i = \mathit{Accuracy}_i + \lambda\ \mathit{Diversity}_i = (1 - E_i) + \lambda\ D_i, \tag{5}$$

where λ defines the tradeoff between accuracy and diversity. Finally, ADDEMUP prunes the population to the N most-fit members, which it defines to be its current ensemble, then repeats this process.

We define our accuracy term, $1 - E_i$, to be network i's validation-set accuracy (or training-set accuracy if a validation set is not used), and we use Equation 1 over this validation set to

[3]One may use any search mechanism during this step. We compare our genetic algorithms approach with a simulated annealing version in Section 4.

Table 1: The ADDEMUP algorithm.

GOAL: Genetically create an accurate and diverse ensemble of networks.

1. Create and train the initial population of networks (see Section 3.2).
2. Until a stopping criterion is reached:
   (a) Use genetic operators to create new networks (see Section 3.2).
   (b) Train the new networks using Equation 6 and add them to the population.
   (c) Measure the diversity of each network with respect to the current population (see Equation 1).
   (d) Normalize the accuracy scores and the diversity scores of the individual networks.
   (e) Calculate the fitness of each population member (see Equation 5).
   (f) Prune the population to the N fittest networks.
   (g) Adjust λ (see the text for an explanation).
   (h) This population of networks compose the current ensemble. Combine the output of these networks according to Equation 7.

calculate our diversity term, D_i. We then separately normalize each term so that the values range from 0 to 1. Normalizing both terms allows λ to have the same meaning across domains.

Since it is not always clear at what value one should set λ, we have therefore developed some rules for automatically adjusting λ. First, we never change λ if the ensemble error $\hat{E}$ is decreasing while we consider new networks; otherwise we change λ if one of following two things happen: (a) the population error $\bar{E}$ is not increasing and the population diversity $\bar{D}$ is decreasing; diversity seems to be under emphasized and we increase λ, or (b) $\bar{E}$ is increasing and $\bar{D}$ is not decreasing; diversity seems to be over-emphasized and we decrease λ. (We started λ at 0.1 for the experiments in this article. The amount λ changes is 10% of its current value.)

A useful network to add to an ensemble is one that correctly classifies as many examples as possible, while making its mistakes primarily on examples that most of the current population members correctly classify. We address this during backpropagation training by multiplying the usual error function by a term that measures the combined population error on that example:

$$Cost = \sum_{k \in T} \left| \frac{t(k) - \hat{o}(k)}{\hat{E}} \right|^{\frac{\lambda}{\lambda+1}} [t(k) - o(k)]^2, \tag{6}$$

where $t(k)$ is the target and $o(k)$ is the network activation for example k in the training set T. Notice that since the network is not yet a member of the ensemble, $\hat{o}(k)$ and $\hat{E}$ are not dependent on this network; our new term is thus a constant when calculating the derivatives during backpropagation. We normalize $t(k) - \hat{o}(k)$ by the current ensemble error $\hat{E}$ so that the *average* value of our new term is around 1 regardless of the correctness of the ensemble. This is especially important with highly accurate populations, since $t(k) - \hat{o}(k)$ will be close to 0 for most examples, and the network would only get trained on a few examples. The exponent $\frac{\lambda}{\lambda+1}$ represents the ratio of importance of the diversity term in the fitness function. For instance, if

λ is close to 0, diversity is not considered important and the network is trained with the usual cost function; however, if λ is large, diversity is considered important and our new term in the cost function takes on more importance.

We combine the predictions of the networks by taking a weighted sum of the output of each network, where each weight is based on the validation-set accuracy of the network. Thus we define our weights for combining the networks as follows:

$$w_i = \frac{1 - E_i}{\sum_k (1 - E_k)}. \tag{7}$$

While simply averaging the outputs can generate a good composite model (Clemen, 1989), we include the predicted accuracy in our weights since one should believe accurate models more than inaccurate ones. We also tried more complicated models, such as emphasizing confident activations (i.e., activations near 0 or 1), but they did not improve the results on our testbeds. One possible explanation is that optimizing the combining weights can easily lead to overfitting (Sollich & Krogh, 1996). We use validation-set accuracy, instead of Breiman's J-fold partitioning (Breiman, 1996b) since, during crossover, new networks are created from two existing networks which may have come from different folds. Therefore it is desirable to have each network use the same validation set.

3.2 Creating and Crossing-Over Knowledge-Based Neural Networks

Steps 1 and 2a in Table 1 specify that new networks need to be created. The algorithm we use for generating these new networks is the Regent algorithm (Opitz & Shavlik, 1994). Regent uses genetic algorithms to search through the space of possible neural network topologies. Regent is specifically designed for KNNs, though it applies to standard neural networks as well. Before presenting the exact details of these steps, we discuss (a) how we generate KNNs, and (b) Regent's genetic operators for refining the topology of these networks.

An empirically successful algorithm for creating KNNs is the Kbann algorithm (Towell & Shavlik, 1994). Kbann translates a set of propositional rules into a neural network, then refines the resulting KNN's weights using backpropagation. Figure 2 illustrates this translation process. Figure 2a shows a Prolog-like rule set that defines membership in category a. Figure 2b represents the hierarchical structure of these rules, with solid lines representing necessary dependencies and dotted lines representing prohibitory dependencies. Figure 2c represents the resulting network created from this translation. Kbann creates nodes $b1$ and $b2$ in Figure 2c to handle the two rules defining b in the rule set. Biases are set to represent the appropriate AND or OR structure of each corresponding node. The thin lines in Figure 2c are lightly-weighted links that Kbann adds to allow refinement of these rules during backpropagation training.

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 third rule for inferring b. Thus Kbann suffers when given domain theories that are missing rules needed to adequately learn the true concept (Opitz & Shavlik, 1993; Towell & Shavlik, 1994). Regent addresses this limitation by searching for refinements to a KNN's topology. It does this by using (a) the domain theory to help create an initial population and (b) crossover and mutation operators specifically designed for knowledge-based networks.

Regent attempts to create an initial population of networks that comes from the same domain theory and yet is diverse. It does this by randomly perturbing the Kbann-generated network at various nodes, thus creating diversity about the domain theory. Briefly, Regent

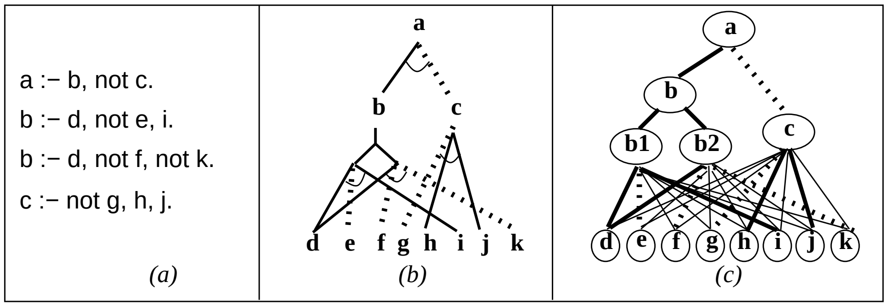

Figure 2: Translation of a knowledge base into a neural network.

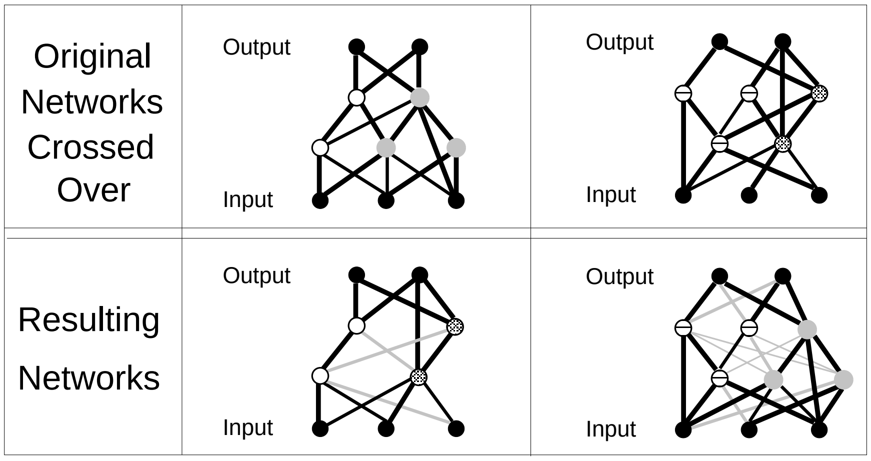

Figure 3: REGENT's method for crossing over two networks. The hidden nodes in each original network are divided into the sets `A` and `B`; the nodes in the two `A` sets form one new network, while the two `B` sets form another.

perturbs a node by either (a) deleting it, or (b) applying its mutation operator (which we explain below).

REGENT crosses over two networks by first dividing the nodes in each parent network into two sets, `A` and `B`, then combining the nodes in each set to form two new networks (i.e., the nodes in the two `A` sets form one network, while the nodes in the two `B` sets form another). Figure 3 illustrates this crossover with an example. REGENT probabilistically divides the nodes into sets so that nodes that are connected by heavily weighted links tend to belong to the same set. This helps to minimize the destruction of the rule structure of the crossed-over networks, since nodes belonging to the same syntactic rule are connected by heavily weighted links. Thus, REGENT's crossover operator produces new networks by crossing-over rules, rather than simply crossing-over nodes.

REGENT's mutation operator adds diversity to the population by adding new nodes to one member of the population. The mutation operator proceeds by estimating where errors are in the network, then adds new nodes in response to these estimates. The operator judges where errors are in a network by using training examples to increment two counters for each node,

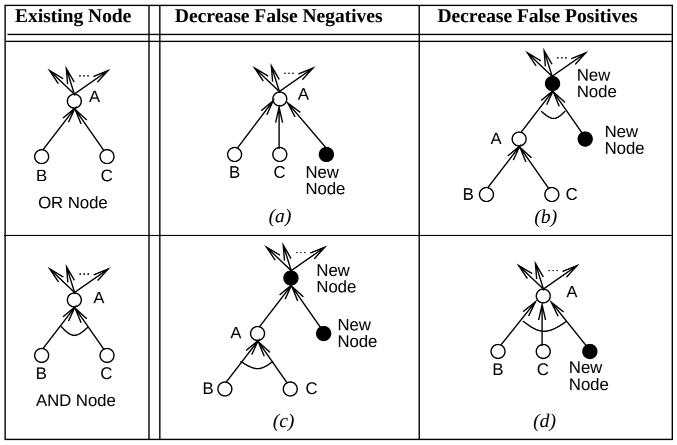

Figure 4: How the mutation operator adds new nodes to knowledge-based networks. Arcs indicate AND nodes.

one for false negatives and one for false positives.[4] Thus the mutation operator adds diversity to a population, while still maintaining a directed, heuristic-search technique for improving the quality of an individual.

Figure 4 shows the possible ways that REGENT's mutation operator adds nodes to KNNs. In a symbolic rulebase that uses negation-by-failure, one can decrease false negatives by either dropping antecedents from existing rules or adding new rules to the rulebase. Since gradient-based training is effective at removing antecedents from existing rules (Towell & Shavlik, 1994), the mutation operator adds nodes, intended to decrease false negatives, in a fashion that is analogous to adding a new rule to the rulebase (see Figure 4a,c). The mutation operator decreases false positives by creating new antecedents for the node (see Figure 4b,d). In doing so, this operator is able to add rules, whose consequents were previously undefined, to the rulebase (something, as discussed above, gradient-based training is incapable of doing).

For the results in this article, ADDEMUP uses, as its step 1 in Table 1, REGENT's method for creating its initial population, and, as its step 2a in Table 1, REGENT's genetic operators to create new networks. These steps also apply to standard neural networks if no domain-specific knowledge is available; however in order to properly use the genetic operators presented above, we need to create networks whose node structure is analogous to dependencies found in symbolic rule bases. We do this by first randomly picking the number of hidden nodes to include in a network; we repeatedly add hidden nodes to the network being constructed by randomly selecting an existing output or hidden node, then adding new nodes to this node using one of the four methods shown in Figure 4.

4 Experimental Study

We ran ADDEMUP on NYNEX's MAX problem set (Provost & Danyluk, 1995) and on three problems from the Human Genome Project that aid in locating genes in DNA sequences (recognizing

[4]A node's false-positive counter is incremented if changing its activation to 0 causes the network to correct an erroneous output. Counters for false negatives are defined analogously.

promoters, *splice-junctions*, and *ribosome-binding sites - RBS*). MAX is an expert system that was designed by NYNEX to diagnose the location of customer-reported telephone problems. The inputs in this case are an electronic profile of the telephone loop and the task it to learn where in this loop the problem occurs. In each of the DNA programs, the input is a short segment of DNA nucleotides (about 100 elements long) and the task is learn to predict if this DNA subsequence contains a biologically important site.

Each of these domains is accompanied by a set of approximately correct rules describing what is currently known about the task (see Opitz, 1995, or Opitz and Shavlik, 1994, for more details). The DNA domains are available at the University of Wisconsin Machine Learning (UW-ML) site via the World Wide Web (`ftp://ftp.cs.wisc.edu/machine-learning/shavlik-group/datasets/`) or anonymous ftp (`ftp.cs.wisc.edu`, then `cd` to `machine-learning/ shavlik-group/datasets`). Due to proprietary reasons, the NYNEX problem set is not publicly available.

Our experiments in this article measure the test-set error of ADDEMUP on these four real-world datasets. All results presented are from a ten-fold cross validation. Within each fold, algorithms that need a validation set held out 10% of the training instances for that set. Each ensemble consists of 20 networks, and the REGENT and ADDEMUP algorithms considered 250 networks during their genetic search.

4.1 Generalization Ability of ADDEMUP

In this subsection, we divide our experiments into two classes: (a) the algorithms randomly create the topology of their networks, and (b) they utilize the domain theory to create their networks (i.e., they use KNNs). As stated earlier, using KNNs allows one to have in his or her ensemble highly correct networks that tend to agree. The alternative of randomly generating the network topologies thus trades off the overall accuracy of each single network for more disagreement between the networks.

As points of comparison, we include the results of running (a) Breiman's et al. (1996a) *Bagging* algorithm, and (b) a simulated annealing (Aarts & Korst, 1989) version of ADDEMUP. Bagging is a "bootstrap" (Efron & Tibshirani, 1993) ensemble method that trains each network in the ensemble with a different partition of the training set. It generates each partition by randomly drawing, with replacement, N examples from the training set, where N is the size of the training set. Breiman (1996a) showed that Bagging is effective on "unstable" learning algorithms where small changes in the training set result in large changes in predictions. Earlier, Breiman (1984) studied instability, and claimed that neural networks and decision trees are unstable, while k-nearest-neighbor methods are stable. We also tried other ensemble approaches, such as randomly creating varying multi-layer network topologies and initial weight settings, but Bagging did significantly better on all datasets (by 15-25% on all three DNA domains).

Our simulated annealing (SA) version of ADDEMUP substituted the genetic operators in step 2a of Table 1 with an SA operator. Our SA operator works by altering a random member of the current population either by (a) using TopGen (i.e., REGENT's mutation operator) to heuristically refine the network's topology or (b) REGENT's algorithm for randomly altering the topology of a network when creating its initial population. The probability of randomly altering the topology (i.e., operator b above) versus systematically altering the topology (i.e., operator a above) decreases with the temperature of the system according to the Bolzmann distribution. For these experiments, temperature started at 270 and decayed 5% after each alteration; therefore, the probability of randomly altering the topology was close to 1 initially, and close to 0 at the end of each run.

Table 2: Test-set error from a ten-fold cross validation. Table (a) shows the results from running the learners without the domain theory; Table (b) shows the results of running the learners with the domain theory. Pairwise, one-tailed t-tests indicate that ADDEMUP-`GA` in Table (b) differs from the other algorithms (other than ADDEMUP-SA in Table b) in both tables at the 95% confidence level, except with REGENT-`combined` in the splice-junction domain.

Standard neural networks (no domain theory used)				
	Promoters	Splice Junction	RBS	NYNEX
`best-network`	6.6%	7.8%	10.7%	37.0%
`Bagging`	4.6%	4.5%	9.5%	35.7%
ADDEMUP-`SA`	4.6%	4.9%	9.3%	35.2%
ADDEMUP-`GA`	4.5%	4.9%	9.0%	34.9%

(a)

Knowledge-based neural networks (domain theory used)				
	Promoters	Splice Junction	RBS	NYNEX
KBANN	6.2%	5.3%	9.4%	35.8%
`Bagging-KNN`	4.2%	4.5%	8.5%	35.6%
REGENT-`best-network`	4.4%	4.1%	8.8%	35.9%
REGENT-`combined`	3.9%	3.9%	8.2%	35.6%
ADDEMUP-`SA`	3.7%	4.0%	7.8%	35.5%
ADDEMUP-`GA`	3.0%	3.6%	7.5%	34.7%

(b)

4.1.1 Generating Non-KNN Ensembles

Table 2a presents the results from the case where the learners randomly create the topology of their networks (i.e., they do not use the domain theory). Table 2a's first row, `best-network`, results from a single-layer neural network where, for each fold, we trained 20 networks (uniformly) containing between 0 and 100 hidden nodes and used a validation set to choose the best network. The next row, `Bagging`, contains the results of applying the Bagging algorithm to 20 standard, single-hidden-layer networks, where the number of hidden nodes is randomly set between 0 and 100 for each network. The results confirm Breiman's prediction that Bagging would be effective with non-KNNs because of the "instability" of standard neural networks. That is, a slightly different training set can produce large alterations in the predictions of the networks, thereby leading to an effective ensemble.

The bottom two rows of Table 2a contain the results of the SA and GA versions of ADDEMUP where, in both cases, their initial population (of size 20) is randomly generated using REGENT's method for creating networks when no domain theory is present (refer to Opitz, 1995, for more details). Even though both versions of ADDEMUP train each network with the same training set, it still produces results comparable to Bagging. The results show that - on these domains - combining the output of multiple trained networks generalizes better than trying to pick the single-best network. Pairwise, one-tailed t-tests indicate that `Bagging` and both

Addemup versions differ from `best-network` at the 95% confidence level on all four domains; however, while Addemup-GA produces better results than both Addemup-SA and `Bagging`, this difference is not significant at this level.

4.1.2 Generating KNN Ensembles

While the previous section shows the general power of a neural-network ensemble, Table 2b demonstrates Addemup's ability to utilize prior knowledge. Again, each ensemble contains 20 networks. The first row of Table 2b contains the generalization results of the Kbann algorithm, while the next row, `Bagging-KNN`, contains the results of the ensemble where each individual network in the ensemble is the Kbann network trained on a different partition of the training set. Even though each of these networks start with the same topology and "large" initial weight settings (i.e., the weights resulting from the domain theory), small changes in the training set still produce significant changes in predictions. Also notice that on all datasets, `Bagging-KNN` is as good as or better than running Bagging on randomly generated networks (i.e., `Bagging` in Table 2a).

The next two rows result from the Regent algorithm. The first row, Regent-`best-network`, contains the results from the single best network output by Regent, while the next row, Regent-`combined`, contains the results of simply combining, using Equation 7, the networks in Regent's final population. Opitz and Shavlik (1994) showed the effectiveness of Regent-`bestnetwork`, and comparing it with the results in Table 2a reaffirms this belief. Notice that simply combining the networks of Regent's final population (Regent-`combined`) decreases the test-set error over the single-best network picked by Regent.

The final two rows present the results from the two versions of Addemup. While Addemup-`SA` produces better results than `Bagging-KNN`, it only produces slightly better results overall than Regent-`combined`. Addemup-`GA`, however, *is* able to generate a more effective ensemble than the other learners. Addemup-`GA` mainly differs from Regent-`combined` in two ways: (a) its fitness function (i.e., Equation 5) takes into account diversity rather than just network accuracy, and (b) it trains new networks by emphasizing the erroneous examples of the current ensemble. Therefore, comparing Addemup-`GA` with Regent-`combined` directly test Addemup's diversity-achieving heuristics. Also, since genetic algorithms are effective at global optimizations, they are more effective at generating diverse ensembles than our simulated annealing approach. (For the rest of this article, we concentrate only on the genetic algorithm version of Addemup.)

4.2 Lesion Study of ADDEMUP

We also performed a lesion study[5] on Addemup's two main diversity-promoting components: (a) its fitness function (i.e., Equation 5) and (b) its reweighting of each training example based on ensemble error (i.e., Equation 6). Table 3 contains the results for this lesion study. The first row, Regent-`combined`, is a repeat from Table 2b, where we simply combined the networks of Regent's final population. The next two rows are "lesions" of Addemup. The first, Addemup-`weighted-examples`, is Addemup with only reweighting the examples during training, while the second, Addemup-`fitness`, is Addemup with only its new fitness function. The

[5] A *lesion study* is one where components of an algorithm are individually disabled to ascertain their contribution to the full algorithm's performance (Kibler & Langley, 1988).

Table 3: Test-set error on the lesion studies of Addemup. Due to the inherent similarity of each algorithm and the lengthy run-times limiting the number of runs to a ten-fold cross-validation, the difference between the lesions of Addemup is not significant at the 95% confidence level.

	Promoters	Splice Junction	RBS
Regent-`combined`	3.9%	3.9%	8.2%
Addemup-`weighted-examples`	3.8%	3.8%	7.8%
Addemup-`fitness`	3.1%	3.7%	7.4%
Addemup-`both`	2.9%	3.6%	7.5%

final row of the table, Addemup-`both`, is Addemup with both its fitness function and its reweighting mechanism (i.e., a repeat of Addemup from Table 2b).

The results show that, while reweighting the examples during training usually helps, Addemup gets most of its generalization power from its fitness function. Reweighting examples during training helps create new networks that make their mistakes on a different part of the input space than the current ensemble; however, these networks might not be as correct as training on each example evenly, and thus might be deleted from the population without an appropriate fitness function that takes into account diversity.

5 Discussion and Future Work

The results in Table 2 show that combining the output of multiple trained networks generalizes better than trying to pick the single-best network, verifying the conclusions of previous work (Alpaydin, 1993; Breiman, 1996a; Hansen & Salamon, 1990; Hashem et al., 1994; Krogh & Vedelsby, 1995; Lincoln & Skrzypek, 1989; Maclin & Shavlik, 1995; Mani, 1991; Perrone, 1992). When generating KNN ensembles, since every network in the population comes from the same set of rules, we expect each network to be similar. Thus the magnitude of the improvements of the KNN ensembles, especially Kbann-`Bagging` and Regent-`combined`, comes as a bit of a surprise. Regent, however, does create some diversity during its genetic search to ensure a broad consideration of the concept space (Goldberg, 1989; Holland, 1975). It does this by randomly perturbing the topology of each knowledge-based neural network in the initial population and it also encourages diversity when creating new networks during the search through its mutation operator.

While Regent *encourages* diversity in its population, it does not *actively* search for a highly diverse population like Addemup. In fact the single best network produced by Addemup (5.1% error rate on the promoter domain, 5.3% on the splice-junction domain, and 9.1% on the RBS domain) is distinctively worse than Regent's single best network (4.4%, 4.1%, and 8.8% on the three respective domains). Thus, while excessive diversity does not allow the population to find and improve the *single* best network, the results in Table 2b show that more diversity is needed when generating an effective ensemble. There are two main reasons why we think the results of Addemup in Table 2b are especially encouraging: (a) by comparing Addemup with Regent-`combined`, we explicitly test the quality of our fitness function and demonstrate its effectiveness, and (b) Addemup is able to effectively utilize background knowledge to decrease the error of the individual networks in its ensemble, while still being able to create enough diversity among them so as to improve the overall quality of the ensemble.

Our first planned extension to ADDEMUP is to investigate new methods for *creating* networks that are diverse in their predictions. While ADDEMUP currently tries to generate such networks by reweighting the error of each example, the lesion study showed that ADDEMUP gets most of its increase in generalization from its fitness function. One alternative we plan to try is the Bagging algorithm. We plan to use bootstrapping to assign each new population member's training examples. Moreover, rather than just randomly picking these training instances, we plan to investigate the utility of more intelligently picking this learning set. For instance, one could emphasize picking examples the current ensemble misclassifies.

Future work also includes investigating intelligent methods for setting the combining weights. Currently, ADDEMUP combines each network in the ensemble by taking the weighted average of the output of each network, where each weight is set to the validation-set accuracy of the network. One approach we plan to implement is a proposed method by Krogh and Vedelsby (1995) that tries to optimally find the settings that minimize the ensemble generalization error in Equation 4. They do this by turning the constraints into a quadratic optimization problem. Thus, while ADDEMUP searches for a *set* of networks that minimize Equation 4, this approach searches for a way to optimally *combine* the set for this equation.

The framework of ADDEMUP and the theory it builds upon can be applied to any inductive learner, not just neural networks. Future work then, is to investigate applying ADDEMUP to these other learning algorithms as well. With genetic programming (Koza, 1992), for instance, we could translate perturbations of the domain theory into a set of dependency trees (see Figure 2b), then continually create new candidate trees via crossover and mutation. Finally, we would keep the set of trees that are a good fit for our objective function containing both an accuracy and diversity term. By implementing ADDEMUP on a learner that creates its concepts faster than training a neural network, we can more extensively study various issues such as finding good ways to change the tradeoff between accuracy and diversity, investigating the value of normalizing the accuracy and diversity terms, and finding the appropriate size of an ensemble.

6 Additional Related Work

As mentioned before, the idea of using an ensemble of networks rather than the single best network has been proposed by several people. We presented a framework for these systems along with a theory of what makes an effective ensemble in Section 2. Lincoln and Skrzypek (1989), Mani (1991) and the forecasting literature (Clemen, 1989; Granger, 1989) indicate that a simple averaging of the predictors generates a very good composite model; however, many later researchers (Alpaydin, 1993; Breiman, 1996b; Hashem et al., 1994; Perrone, 1992; Wolpert, 1992; Zhang et al., 1992) have further improved generalization with voting schemes that are complex combinations of each predictor's output. One must be careful in this case, since optimizing the combining weights can easily lead to the problem of overfitting which simple averaging seems to avoid (Sollich & Krogh, 1996).

Most approaches do not *actively* try to generate highly correct networks that disagree as much as possible. These approaches either *randomly* create their networks (Hansen & Salamon, 1990; Lincoln & Skrzypek, 1989), or *indirectly* try to create diverse networks by training each network with dissimilar learning parameters (Alpaydin, 1993), different network architectures (Hashem et al., 1994), various initial weight settings (Maclin & Shavlik, 1995), or separate partitions of the training set (Breiman, 1996a; Krogh & Vedelsby, 1995). Unlike Addemup however, these approaches do not directly address *how* to generate such networks that are optimized for the ensemble as a whole.

One method that does actively create members for its ensemble, however, is the *Boosting* algorithm (Shapire, 1990). Boosting converts any learner that is guaranteed to always perform slightly better than random guessing into one that achieves arbitrarily high accuracy. Drucker et al. (1992) applied Boosting to neural networks to improve their error rate on a handwritten-digit-recognition task. A problem with the Boosting algorithm, however, is that with a finite amount of training examples, unless the first network has very poor performance, there may not be enough examples to generate a second or third training set. For instance, if a Kbann network is trained with 3,000 examples from one of the DNA tasks and it reaches 95% correct, you would need 30,000 examples to find an appropriate training set for the second network. Even more examples would be needed to generate a third training set.

Recently, Drucker and Cortes (1996) applied a new boosting algorithm, termed AdaBoost (Freund & Shapire, 1995), to decision trees. This algorithm builds an ensemble one member at a time, where each new member randomly picks examples from the original training set with higher probability assigned to those patterns the current ensemble classifies incorrectly. (Bagging, on the other hand, randomly picks each example with equal probability.) One potential problem with this approach is that, since it continually picks training sets consisting mostly of incorrect examples, the new members added to the ensemble are likely to be less correct than earlier methods. As stated earlier, an effective ensemble must not only consist of members who disagree, but ones that are accurate as well. While Addemup also emphasizes incorrectly classified examples, it still trains on *all* examples, and then includes a new network into its ensemble only if it is also highly accurate.

An alternate approach to the ensemble framework is to train individual networks on a subtask, and to then combine these predictions with a "gating" function that depends on the input. Jacobs et al.'s (1991) adaptive mixtures of local experts, Baxt's (1992) method for identifying myocardial infarction, and Nowlan and Sejnowski's (1992) visual model all train networks to learn specific subtasks. The key idea of these techniques is that a decomposition of the problem into specific subtasks might lead to more efficient representations and training (Hampshire & Waibel, 1989).

Once a problem is broken into subtasks, the resulting solutions need to be combined. Jacobs et al. (1991) propose having the gating function be a network that *learns* how to allocate examples to the experts. Thus the gating network allocates each example to one or more experts, and the backpropagated errors and resulting weight changes are then restricted to these networks (and the gating function). Tresp and Taniguchi (1995) propose a method for determining the gating function *after* the problem has been decomposed and the experts trained. Their gating function is an input-dependent, linear-weighting function that is determined by a combination of the networks' diversity on the current input with the likelihood that these networks have seen data "near" that input.

Although the mixtures of experts and ensemble paradigms seem very similar, they are in fact quite distinct from a statistical point of view. The mixtures-of-experts model makes the assumption that a single expert is responsible for each example. In this case, each expert is a model of a region of the input space, and the job of the gating function is to decide from which model the data point originates. Since each network in the ensemble approach learns the whole task rather than just some subtask and thus makes no such mutual exclusivity assumption, ensembles are appropriate when no one model is highly likely to be correct for any one point in our input space.

7 Conclusions

Previous work with neural-network ensembles have shown them to be an effective technique if the predictors in the ensemble are both highly correct and disagree with each other as much as possible. Our new algorithm, ADDEMUP, uses genetic algorithms to search for a correct and diverse population of neural networks to be used in the ensemble. It does this by collecting the set of networks that best fits an objective function that measures both the accuracy of the network and the disagreement of that network with respect to the other members of the set. ADDEMUP tries to actively generate quality networks during its search by emphasizing the current ensemble's erroneous examples during backpropagation training.

Since ADDEMUP continually considers new networks to include in its ensemble, it can be viewed as an "anytime" learning algorithm. Such a learning algorithm should produce a good concept quickly, then continue to search concept space, reporting the new "best" concept whenever one is found (Opitz & Shavlik, 1994). This is important since, for most domains, an expert is willing to wait for weeks, or even months, if a learning system can produce an improved concept.

Experiments demonstrate that our method is able to find an effective set of networks for our ensemble. Experiments also show that ADDEMUP is able to effectively incorporate prior knowledge, if available, to improve the quality of this ensemble. In fact, when using domain-specific rules, our algorithm showed statistically significant improvements over (a) the single best network seen during the search, (b) a previously proposed ensemble method called Bagging (Breiman, 1996a), and (c) a similar algorithm whose objective function is simply the validation-set correctness of the network. In summary, ADDEMUP is successful in generating a set of neural networks that work well together in producing an accurate prediction.

Acknowledgement

This work was supported by Office of Naval Research grant N00014-93-1-0998. This is an extended version of a paper published in *Advances in Neural Information Processing Systems 8*, D. Touretzky, M. Mozer and M. Hasselmo, eds., (pp. 535-541), MIT Press: Cambridge, MA, 1996.

References

Aarts, E. & Korst, J. (1989). *Simulated Annealing and Bolzmann Machines.* Wiley.

Alpaydin, E. (1993). Multiple networks for function learning. In *Proceedings of the 1993 IEEE International Conference on Neural Networks (volume I)*, (pp. 27–32), San Fransisco.

Baxt, W. (1992). Improving the accuracy of an artificial neural network using multiple differently trained networks. *Neural Computation*, 4:772–780.

Breiman, L. (1996a). Bagging predictors. *Machine Learning*, 24(2):123–140.

Breiman, L. (1996b). Stacked regressions. *Machine Learning*, 24(1):49–64.

Breiman, L., Friedman, J., Olshen, R., & Stone, C. (1984). *Classification and Regression Trees.* Wadsworth and Brooks, Monterey, CA.

Clemen, R. (1989). Combining forecasts: A review and annotated bibliography. *International Journal of Forecasting*, 5:559–583.

Drucker, H. & Cortes, C. (1996). Boosting decision trees. In Touretsky, D., Mozer, M., & Hasselmo, M., editors, *Advances in Neural Information Processing Systems (volume 8)*, Cambridge, MA. MIT Press.

Drucker, H., Cortes, C., Jackel, L., LeCun, Y., & Vapnik, V. (1994). Boosting and other machine learning algorithms. In *Proceedings of the Eleventh International Conference on Machine Learning*, (pp. 53–61), New Brunswick, NJ. Morgan Kaufmann.

Drucker, H., Schapire, R., & Simard, P. (1992). Improving performance in neural networks using a boosting algorithm. In Hanson, J., Cowan, J., & Giles, C., editors, *Advances in Neural Information Processing Systems (volume 5)*, (pp. 42–49), Palo Alto, CA. Morgan Kaufmann.

Efron, B. & Tibshirani, R. (1993). *An Introduction to the Bootstrap.* Chapman and Hall, New York.

Freund, Y. & Shapire, R. (1995). A decision-theoretic generalization of on-line learning and an application to boosting. In *Proceedings of the Second European Conference on Computational Learning.*

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, MA.

Granger, C. (1989). Combining forecasts: Twenty years later. *Journal of Forecasting*, 8:167–173.

Hampshire, J. & Waibel, A. (1989). The meta-pi network: Building distributed knowledge representations for robust pattern recognition. Technical Report TR CMU-CS-89-166, CMU, Pittsburgh, PA.

Hansen, L. & Salamon, P. (1990). Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12:993–1001.

Hashem, S., Schmeiser, B., & Yih, Y. (1994). Optimal linear combinations of neural networks: An overview. In *Proceedings of the 1994 IEEE International Conference on Neural Networks*, Orlando, FL.

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

Jacobs, R., Jordan, M., Nowlan, S., & Hinton, G. (1991). Adaptive mixtures of local experts. *Neural Computation*, 3:79–87.

Kibler, D. & Langley, P. (1988). Machine learning as an experimental science. In *Proceedings of the Third European Working Session on Learning*, (pp. 1–12), Edinburgh, UK.

Koza, J. (1992). *Genetic Programming.* MIT Press, Cambridge, MA.

Krogh, A. & Vedelsby, J. (1995). Neural network ensembles, cross validation, and active learning. In Tesauro, G., Touretzky, D., & Leen, T., editors, *Advances in Neural Information Processing Systems (volume 7)*, Cambridge, MA. MIT Press.

Lincoln, W. & Skrzypek, J. (1989). Synergy of clustering multiple back propagation networks. In Touretzky, D., editor, *Advances in Neural Information Processing Systems (volume 2)*, (pp. 650–659), San Mateo, CA. Morgan Kaufmann.

Maclin, R. & Shavlik, J. (1995). Combining the predictions of multiple classifiers: Using competitive learning to initialize neural networks. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Montreal, Canada.

Mani, G. (1991). Lowering variance of decisions by using artificial neural network portfolios. *Neural Computation*, 3:484–486.

Nowlan, S. & Sejnowski, T. (1992). Filter selection model for generating visual motion signals. In Hanson, S., Cowan, J., & Giles, C., editors, *Advances in Neural Information Processing Systems (volume 5)*, (pp. 369–376), San Mateo, CA. Morgan Kaufmann.

Opitz, D. (1995). *An Anytime Approach to Connectionist Theory Refinement: Refining the Topologies of Knowledge-Based Neural Networks.* PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, WI.

Opitz, D. & Shavlik, J. (1993). Heuristically expanding knowledge-based neural networks. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, (pp. 1360–1365), Chambery, France. Morgan Kaufmann.

Opitz, D. & Shavlik, J. (1994). Using genetic search to refine knowledge-based neural networks. In *Proceedings of the Eleventh International Conference on Machine Learning*, (pp. 208–216), New Brunswick, NJ. Morgan Kaufmann.

Perrone, M. (1992). A soft-competitive splitting rule for adaptive tree-structured neural networks. In *Proceedings of the International Joint Conference on Neural Networks*, (pp. 689–693), Baltimore, MD.

Provost, F. & Danyluk, A. (1995). Learning from bad data. In *Workshop on Applying Machine Learning in Practice, held at the Twelfth International Conference on Machine Learning*, Tahoe City, CA.

Shapire, R. (1990). The strength of weak learnability. *Machine Learning*, 5:197–227.

Sollich, P. & Krogh, A. (1996). Learning with ensembles: How over-fitting can be useful. In Touretsky, D., Mozer, M., & Hasselmo, M., editors, *Advances in Neural Information Processing Systems (volume 8)*, Cambridge, MA. MIT Press.

Towell, G. & Shavlik, J. (1994). Knowledge-based artificial neural networks. *Artificial Intelligence*, 70(1,2):119–165.

Tresp, V. & Taniguchi, M. (1995). Combining estimators using non-constant weighting functions. In Tesauro, G., Touretzky, D., & Leen, T., editors, *Advances in Neural Information Processing Systems (volume 7)*, Cambridge, MA. MIT Press.

Wolpert, D. (1992). Stacked generalization. *Neural Networks*, 5:241–259.

Zhang, X., Mesirov, J., & Waltz, D. (1992). Hybrid system for protein secondary structure prediction. *Journal of Molecular Biology*, 225:1049–1063.