Machine Learning Research Group Working Paper 91-1

EXPERIMENTAL ANALYSIS OF ASPECTS OF THE CASCADE-CORRELATION LEARNING ARCHITECTURE [†]

Charles S. Squires, Jr.
Jude W. Shavlik

Computer Sciences Department
University of Wisconsin- Madison
1210 West Dayton Street
Madison, WI 53706
squires@cs.wisc.edu

Abstract

The *cascade-correlation* method of Fahlman and Lebiere is an inventive mix of powerful, largely independent features that work together to incrementally construct a neural network. We identify aspects of their learning algorithm upon which the cascade-correlation approach does not fundamentally depend, and analyze their strengths and weaknesses. We use four problems from the literature to empirically study these features. Evidence indicates that, apart from the cascaded network topology, the most powerful algorithmic feature of cascade-correlation is *patience*. *Correlation* is sometimes a valuable heuristic also, while the use of *quickprop* learning rules and the *freezing* of trained weights are of less value, and can often be detrimental to effective learning.

(Submitted to *Neural Networks*, August 1991.)

[†]This work was partially supported by NSF Grant IRI-9002413 and ONR Grant N00014-90-J-1941 and was inspired by research conducted with Nancy Perugini and Gerald Tesauro at IBM T.J. Watson Labs (Hawthorne, NY) during the summer of 1990.

Experimental Analysis of Aspects of the Cascade-Correlation Learning Architecture

1. Introduction

A major problem in artificial neural networks is designing a network with the appropriate number of hidden units to learn a given concept. If a network has too few hidden units, it will not have the computational power to learn the concept well. Given too many hidden units, it will overfit the training dataset and generalize poorly to novel examples.

With standard backpropagation [9], a feed-forward network topology must be specified *a priori*. To alleviate this constraint, researchers have developed algorithms which modify network topology during learning. Some of these algorithms are *constructive* in that they start with a small network and add hidden units and links as learning progresses. Others start with a large network, and gradually *prune* it. The *cascade-correlation* method of Fahlman and Lebiere [4] is one of the most influential constructive algorithms. Like cascade-correlation, the *self organizing neural network* algorithm of Tenorio and Lee [10] can construct multi-layer networks. A host of other algorithms, including Ash's *dynamic node creation* [1], construct single-layer networks. Popular pruning algorithms include *skeletonization* by Mozer and Smolensky [8] and *optimal brain damage* by Le Cun, Denker, and Solla [6].

Given the myriad of published approaches, it is necessary to carefully analyze their strengths and weaknesses, especially when applied to realistic learning problems. We report a detailed analysis of the Fahlman and Lebiere approach. We break down their method into four major components and, using four learning problems, empirically study each component. Two experimental approaches are taken: either leave out the component or replace it by an alternative. Our results suggest how one might best apply the cascade-correlation method to real-world problems. They also indicate that some aspects of the approach may well prove useful in other neural-network algorithms.

2. Cascade-Correlation: Background

For this paper, we divide cascade-correlation into two distinct components: its *architecture* and its *algorithm*. By its *architecture*, we mean the class of artificial neural network topologies and the control mechanisms by which computation is done within these networks. By its *algorithm*, we mean the specific learning rules and heuristics applied.

The cascade-correlation architecture is a subset of the class of standard feed-forward networks described by Rumelhart [9]. As shown in the example in Figure 1, the architecture is specialized in two ways. First, the outputs of the network receive activation (i.e., have connections) from all input units and all hidden units. Furthermore, hidden units are not organized into distinct layers, but rather are cascaded; i.e., they are arranged in a sequence wherein each hidden unit receives activation from all input units and all hidden units earlier in the sequence.

The cascade-correlation architecture supports a variety of learning algorithms, but the algorithm specifically described by Fahlman and Lebiere [4], outlined in Table 1, works as follows. An initial, minimal network under the cascade-correlation architecture (a network that has only the input and output units required by the task at hand) is trained completely using the *quickprop* algorithm [3]. This step is an efficient, single-layer, error-minimization procedure.

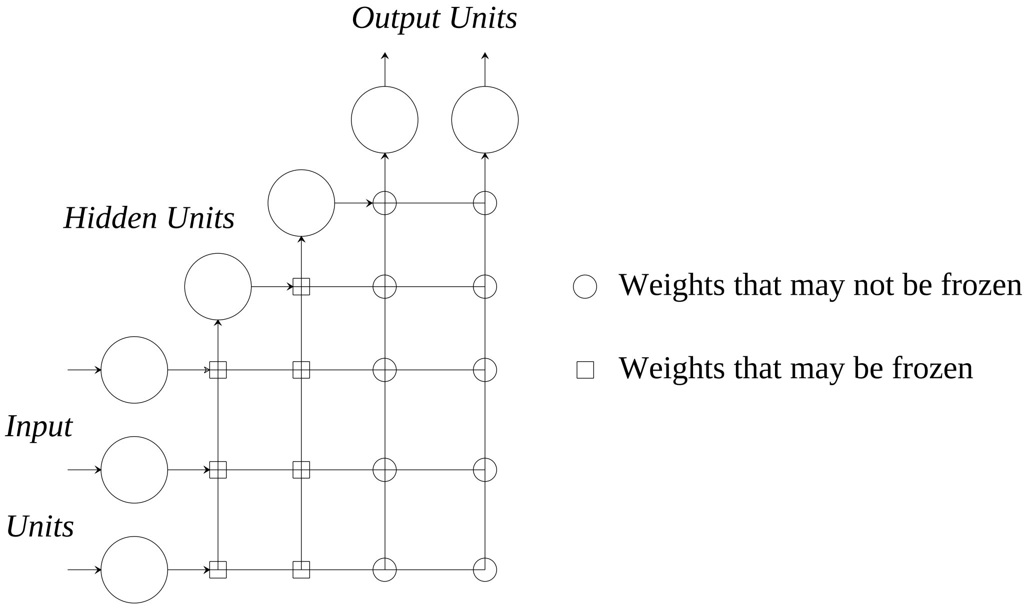

Figure 1: The Cascade-Correlation Architecture. A cascaded network with three input, two hidden, and two output units is shown. Activation is propagated to the right along horizontal lines and up along vertical lines. Individual links between units are indicated by a pair of crossing lines.

Training continues until some maximum number of epochs is reached or a *patience* criterion is met. *Patience* criteria are of the form: "If for the past *n* consecutive epochs, error has not changed by at least *p* percent, stop training."

If after the initial training phase, the network has not achieved the desired correctness, and the network has not reached its maximum allowed size, a new hidden unit is installed. The new hidden unit is drawn from a pool of competing candidate units whose input link weights are adjusted to correlate the unit's activation to the error at the output units, as described in more detail later. The intent is that by correlating to the error, the hidden unit will discover an important missing feature. We will call this correlation phase *input training.* It is an efficient, single-layer, correlation-maximization procedure, also subject to epoch limits and *patience*.

Once a candidate unit is chosen and installed, it's input weights are *frozen* ("matured" and "tenured" are more recent, sexy metaphors). Then, all the links from the input and hidden units to the output units are trained to minimize error in a phase called *output training*. Output training is repeated with the new unit installed. Since weights on links into hidden units are frozen, output training is still single-layer, so it is efficient, and learning speeds slow down only gradually as the network grows.

Table 1: High-level Summary of the Cascade-Correlation Algorithm.

```
Start with a minimal network; /* One with no hidden units. */
Train it;                     /* Perform output training.† */

While (more hidden units may be installed)
  and (not at 100% correctness) do

   /* The input training phase.† */
   Allocate a pool of candidate units;
   Add an edge from each input and hidden unit to each
      candidate unit;
   Train the weights into the candidate units;

   Install the best candidate unit as a new hidden unit;
   /* The weights on the edges into this unit are now frozen. */

   /* The output training phase.† */
   Add an edge from the new hidden unit to each output;
   Retrain all weights into the output units;

end;

/* † Each training phase ends when the patience criteria are
     met or some epoch count limit is reached. */
```

---

3. The Cascade-Correlation Workbench

Cascade-correlation's learning algorithm has four major aspects that are effectively independent within the framework of the architecture:

- QUICKPROP learning rules
- PATIENCE
- error CORRELATION
- weight FREEZING

First, the use of the *quickprop* learning rule is not required. In fact, one could use any learning rule. In this paper, we compare the use of backpropagation [9] with the use of quickprop within the cascade-correlation architecture.

Another optional feature is *patience*. To determine the merit of *patience*, we compare training phases using *patience* to training phases which are simply allowed to run up to some epoch limit.

Correlation of new hidden units' activations to the error at the output units is a heuristic for discovering new important features or terms. When the algorithm installs a unit biased by *correlation*, the new, larger network should start off at a good point on the error landscape. We compare the installation of correlated units with units whose input weights are randomly chosen.

The *freezing* of weights on links which feed into hidden units, once they are trained, reduces all learning to the single-layer case. Moreover, this *freezing* is justified since, after input training, the unit recognizes some important derived feature, and one would like to insure that that feature is not forgotten. We compare learning using *freezing* to learning without *freezing*.

The *cascade-correlation workbench* is our own C implementation of the cascade-correlation architecture, equipped with switches which toggle these four features. Additionally, the set of parameters is kept at a minimum. The system makes the following assumptions which generally hurt performance, but make the comparison between the algorithmic features clearer. Output training (error minimization) weight updates are after every training example presentation. The backprop and quickprop rules do not use a momentum term. The size of the candidate pool of new hidden units is always one. Random numbers are always in the range [−1,+1] and are equally distributed in that range. Units all have sigmoidal activation functions. Values are not cached [4]. Runtime is measured in neither epochs nor milliseconds, but rather in the more meaningful metric of link crossings (i.e., during input and output training phases, whenever a signal is propagated across a link and multiplied by the link weight, the count is incremented). Complexity of computing local gradient terms is not included in the time metric.

4. The Learning Rules

In our cascade-correlation workbench, units have sigmoidal activation functions and activations can vary from -0.5 (false) to +0.5 (true). Given this activation function, derivations of the basic equations used in both output training and input training, and common to both backprop and quickprop, are shown in Appendix A. In this section, we show how the learning rules are applied in the cascade-correlation workbench, and how backprop-based learning differs from quickprop-based learning.

Given a link's error gradient S, we can compute the weight change Δw at time t. The difference between input training and output training lies in the equations defining S (Appendix A). Backprop and quickprop differ in the definition of Δw. In backprop, a simple gradient descent, given by Equation 1, is used.

$$\Delta w_{i,j}(t) = \varepsilon S_{i,j}(t) \tag{1}$$

When the network's output and the teacher (desired output) differ, the difference (or error) is propagated back through the network, stopping at input units and frozen links. Additionally, it is necessary to have a threshold θ, which is a value such that if, at any unit, $|ERROR| \leq \theta$, the *ERROR* term is effectively treated as zero, preventing error from being propagated back any further, and avoiding "inconsequential" weight changes. This θ value is implemented in the cascade-correlation workbench, since without it, we found that quickprop suffers from numerical instability and computes Δw terms that are so large that floating point overflows occur quickly. With correct tuning of θ, this was avoided entirely. Unfortunately, this is unfair to backprop, which does not suffer from this kind of numerical instability, but to make comparisons meaningful, we apply the θ parameter to backprop as well as quickprop, effectively limiting backprop's performance.

Although time appears in Equation 1, backprop does not depend on time. We include it only to be consistent with quickprop, which uses Equation 2 to compute weight changes.

$$\Delta w_{i,j}(t) = \frac{S_{i,j}(t)}{S_{i,j}(t-1) - S_{i,j}(t)} \Delta w_{i,j}(t-1) + \varepsilon S_{i,j}(t) \qquad (2)$$

Quickprop uses the difference of first partials at consecutive times t and $t - 1$ to approximate the second partial derivative. Thus, quickprop efficiently approximates a second-order Newtonian method. Equation 2 does have certain special caveats however. First, $|\Delta w(t)|$ is never allowed to exceed $\mu |\Delta w(t-1)|$, where μ is some constant parameter. This prevents erroneous approximation steps from making single large weight changes which could undo many epochs of learning. Still, the θ parameter is required since μ alone does not prevent weight changes from running amok. Secondly, the gradient-descent term $\varepsilon S(t)$ is only added when the signs of $S(t)$ and $S(t-1)$ are the same. The addition of this term also helps bootstrap quickprop by effectively computing an iteration of backprop at the beginning, or whenever change dies down to zero.

5. Data and Measurements

To have some confidence that our conclusions are as domain-independent as possible, we examine four learning problems of increasing degrees of difficulty:

- MAJORITY7
- PARITY6
- HEART DISEASE
- TIME SERIES

As an example of a "trivial" learning task, we selected MAJORITY7. There are seven inputs, each assigned a value of either ± 0.5. The desired output is the same as the majority of the inputs. We chose 64 training and 64 test patterns at random.

For an "easy" problem, we chose PARITY6. Given six inputs, the desired output is +0.5 if the number of +0.5 inputs is odd, and is -0.5 otherwise. We randomly divided the data into 32 training and 32 test patterns.

For "medium" complexity, we selected the HEART DISEASE diagnosis problem presented in [2]. This noisy, real-world problem maps 25 inputs onto two outputs. The input units encode eight nominally-valued (i.e., possible values from an unordered, finite set) and six numerically-valued features. The output units separately represent the two possible categories: healthy and diseased. We used 195 training and 101 test patterns, divided randomly.

Finally, as a "hard" problem, we chose TIME SERIES (the Mackey-Glass time series benchmark first used by Lapedes [5] and again by Moody [7]), where four inputs are mapped to one output. We randomly split the dataset into a 500 training and 500 test patterns.

For each problem, we roughly tuned the network parameters. The parameters are shown in Appendix B. Then, we shuffled the datasets into ten permutations and split each permutation into a training data set and a test data set in the same proportions as above. We assigned each permutation a distinct random number generator seed used to select initial random biases in the cascade-correlation workbench to assure that our results are not anomalies of biases in the initial edge weights.

We collected the following statistics at the end of each output training phase: hidden units installed, mean-squared error on the training set, mean-squared error on the test set, and cumulative training time in link crossings. We performed pairwise *t*-tests on the results to assure that differences were statistically significant with a confidence level of at least 95%.

6. Experiments and Discussion

In this section, we discuss the four experiments we conducted to analyze the independent effects of four features of the cascade-correlation learning algorithm: the learning rule chosen (quickprop vs. backprop), the use of *patience*, *freezing* of input weights, and *correlation* of new hidden units' activations to the error at the outputs.

Quickprop versus Backprop: Results

To compare quickprop to backprop, we ran each problem with all other features true to the original spirit of cascade-correlation; that is, *patience, freezing,* and *correlation* are all turned on. Surprisingly, we find that quickprop is not generally quicker than backprop. There was a high degree of sensitivity to the training data and random biases, but on average, backprop ran significantly faster in the MAJORITY7 and TIME SERIES domains. In the HEART DISEASE domain, the time difference is not statistically significant. In PARITY6, results are inconclusive, with quickprop running faster in the later stages of learning. The runtimes on the four domains are illustrated in Figure 2.

On individual runs, quickprop sometimes generalized better than backprop, but on average, backprop's generalization as measured by test set error was significantly better than quickprop's. The one exception in the MAJORITY7 domain, where learning is so easy that both methods generalize extremely well. Figure 3 shows how well the networks under each algorithm learned the training set, and how well they generalized.

Quickprop versus Backprop: Discussion

While the results are problem-dependent, there is evidence that for general problems, quickprop *can* converge significantly faster than backprop, but to less acceptable error levels, because backprop may be a better general gradient-descent algorithm. However, backprop performs better on average within the cascade-correlation system, both in terms of run-time and generalization.

In our experiments, the θ parameter also imposes unnecessary limits on backprop performance. The limits θ imposes on quickprop cannot be improved due to its numerical instability; therefore, backprop's margin of superiority is probably greater than our results indicate.

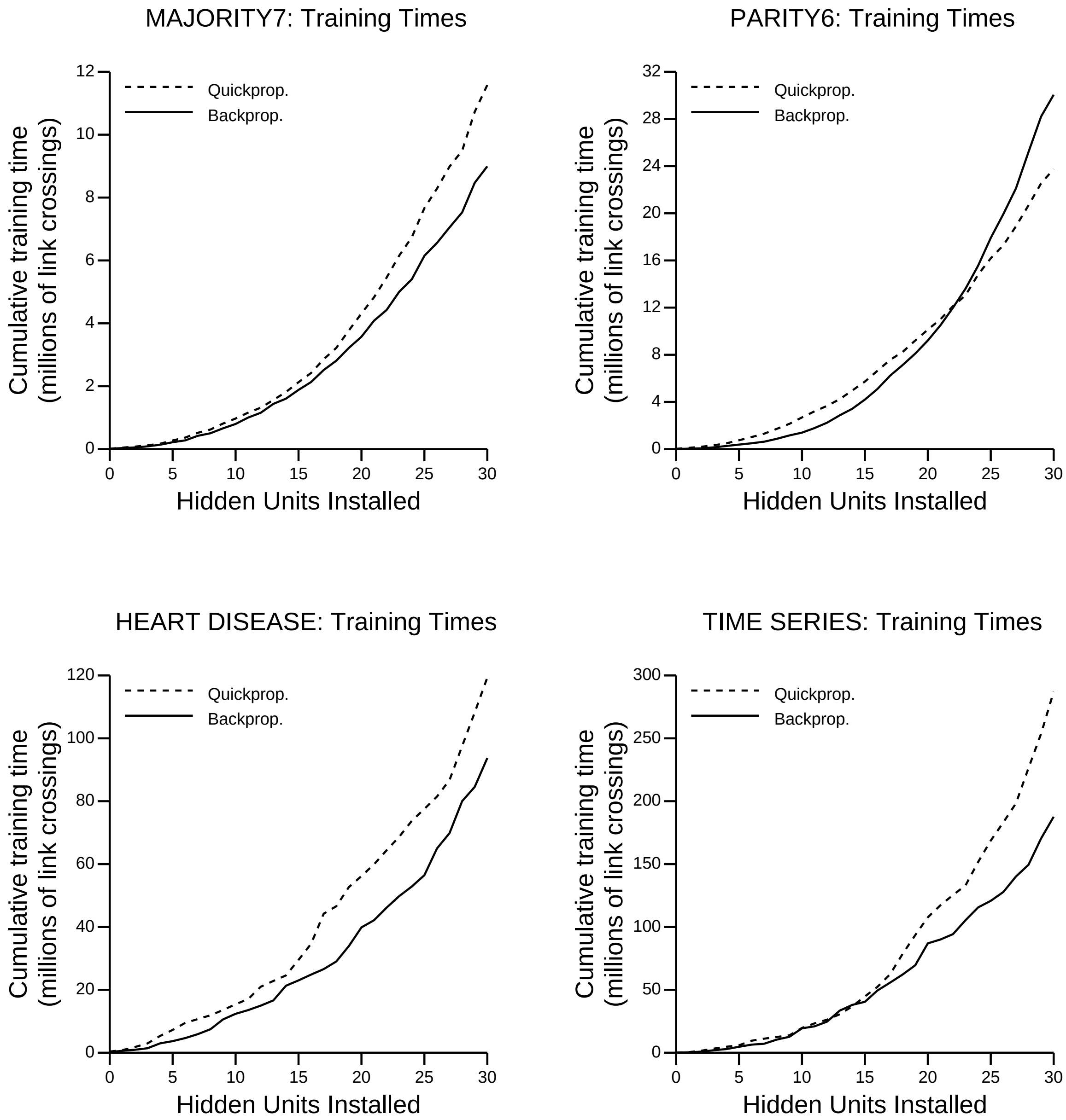

*Figure 2: Runtime using **quickprop** vs. **backprop.*** This figure shows runtimes of cascade-correlation produced using *patience*, *correlation*, and *freezing* of input weights. The learning rule is varied: quickprop vs. backprop. Figure 3 shows the corresponding error curves.

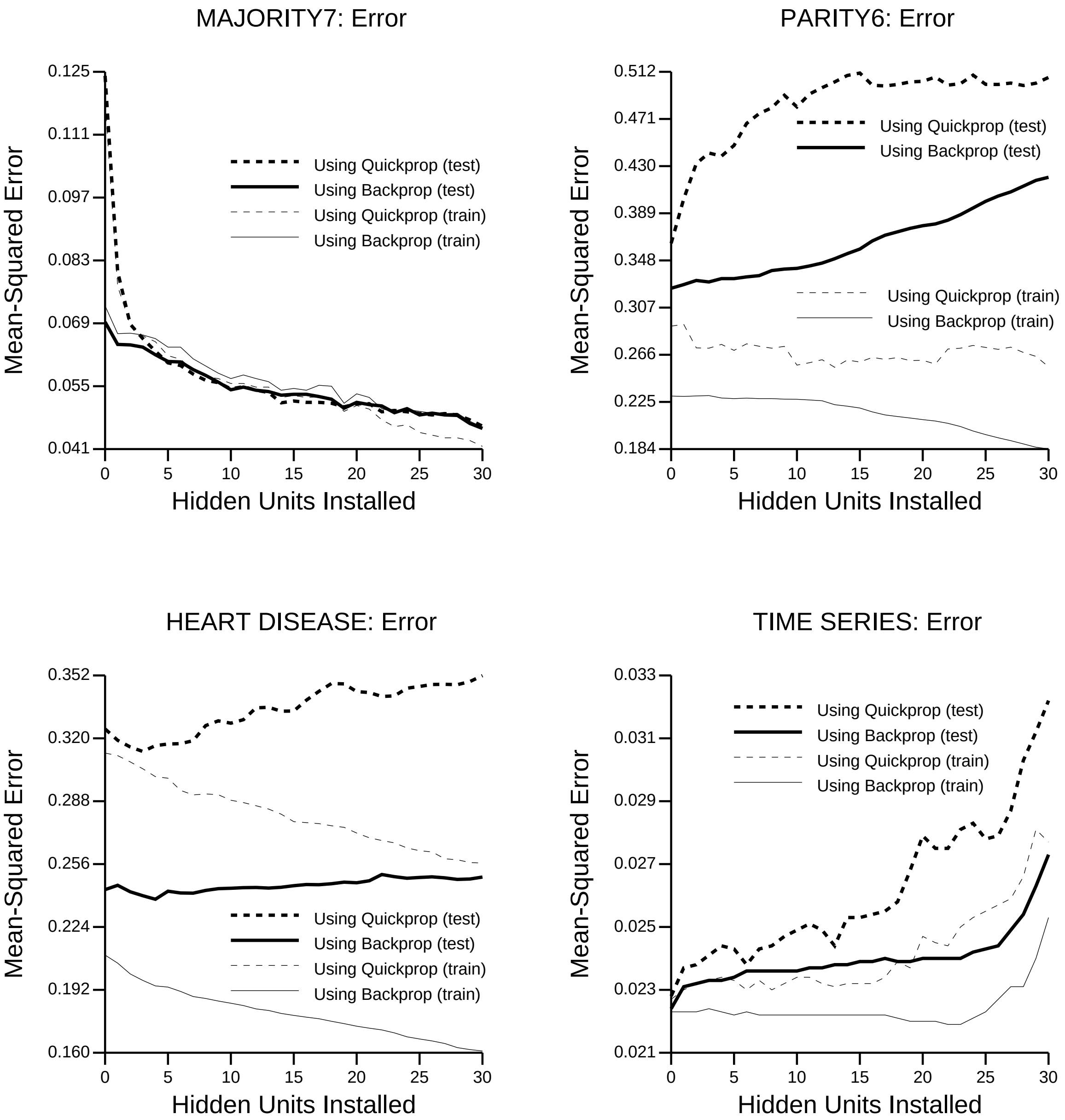

*Figure 3: Generalization using **quickprop** vs. **backprop.*** This figure shows learning and generalization curves produced using *patience, correlation*, and *freezing* of input weights. The learning rule is varied: quickprop vs. backprop. Figure 2 shows the corresponding runtime curves.

Patience: Results

Patience is important in Fahlman and Lebiere's implementation of cascade-correlation. It helps end each learning phase when progress grows excessively slow. This is confirmed by our quickprop-based, *correlating*, *freezing* runs with *patience* turned on and off. More noteworthy, however, is that *patience* is a virtue with backprop as well. Figure 4 shows that *patience* can save a substantial amount of learning time compared to running to a reasonable number of epochs determined a priori. Figure 5 shows that even though so much less time is spent learning, better generalization is achieved using *patience.* Generalization is significantly better in all domains tested except for MAJORITY7, where *patience* results in a generalization that is neither significantly better nor worse than without patience.

Patience: Discussion

Not surprisingly, *patience* saves time since it cuts off each learning phase early. As a result of the early termination of learning, the error on the training set is higher. What is surprising, however, is that the timing of this cut-off produces significantly better generalization. So, without patience, the network memorizes the training set better, but this leads to overfitting the training data, yielding inferior generalization to the test set.

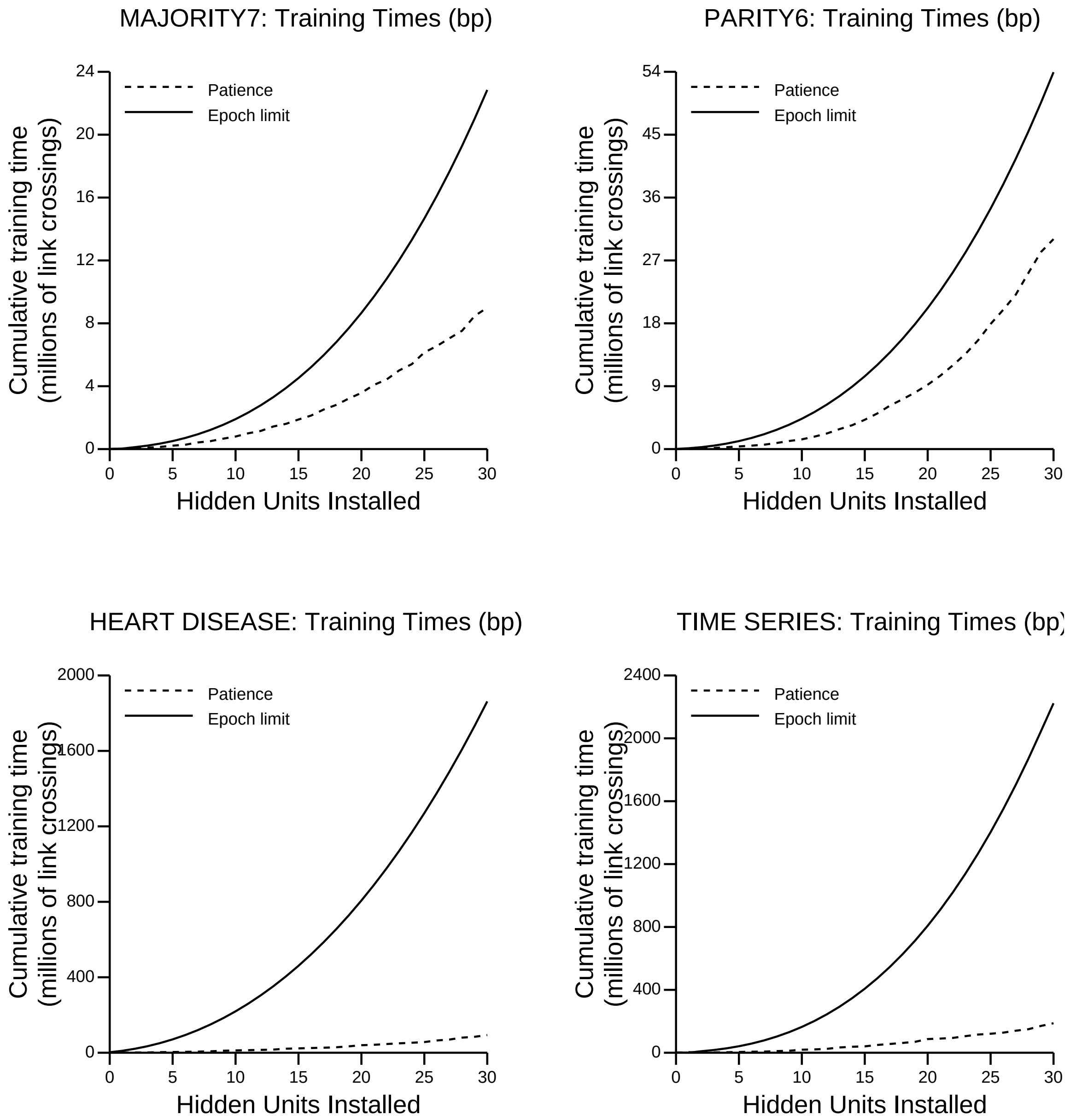

Figure 4: Learning time savings due to ***patience.*** This figure shows runtimes of cascade-correlation produced using the backprop learning rule, *correlation*, and *freezing* of input weights. In some runs, *patience* is used. In others, learning phases run for a fixed number of epochs. Figure 5 shows the corresponding error curves.

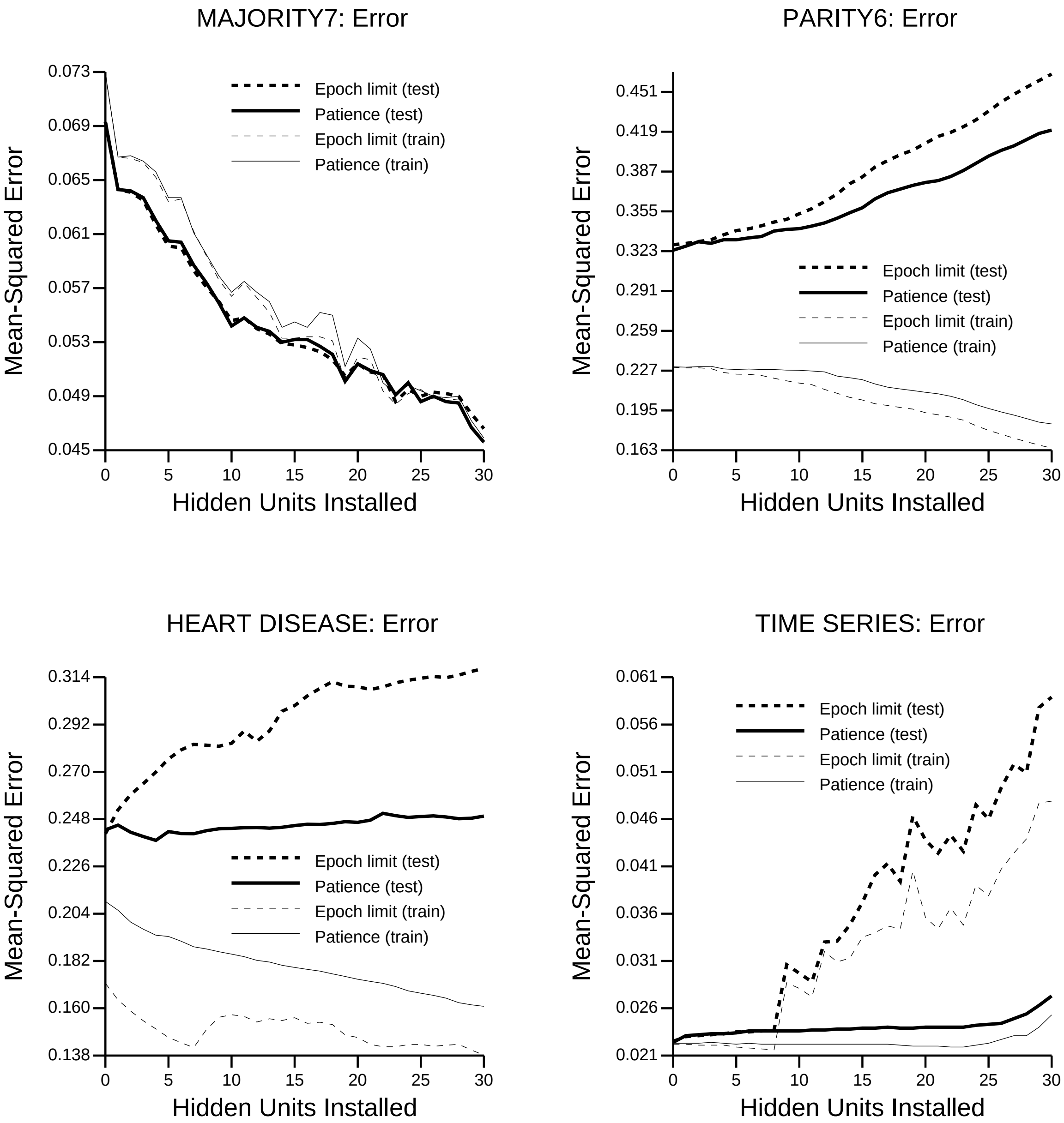

Figure 5: Generalization due to ***patience.*** This figure shows learning and generalization curves produced using the backprop learning rule, *correlation*, and *freezing* of input weights. In some runs, *patience* is used. In others, learning phases run for a fixed number of epochs. Figure 4 shows the corresponding runtime curves.

Freezing: Results

Fahlman and Lebiere identify *freezing* of trained input weights as a major strength of cascade-correlation [4]. The reason seems simple. With *freezing*, the number of weights to train in consecutive phases of output training (or input training) increases by only one each time a unit is added to the network. Avoiding the complexity of multi-layer learning, these learning phases should scale up linearly. Of course, this is not really the case, since individual learning phases can take a highly variable amount of time. Figure 6 shows the time saved due to *freezing*. Rather than the order of magnitude of savings expected, we see more modest savings.

Figure 7 does not report variance. In fact the time savings in the HEART DISEASE domain is statistically insignificant, and the difference on PARITY6 and TIME SERIES is significant but highly variable, with some individual permutations taking substantially longer with *freezing* than without. The result for MAJORITY7 turns out to be the most significant result as the run-times have comparatively low variances.

Freezing results in no significant change in generalization in any of the four domains studied; however, in two domains, *freezing* significantly limited the ability of the network to learn the training set. In MAJORITY7 and TIME SERIES, there was no significant difference in training set error. Figure 7 shows the error curves.

Freezing: Discussion

Freezing impaired the ability of the network to learn training examples. Informally, this is because *freezing* can lock the network into a state from where it is impossible to converge on the training set error levels reachable without *freezing*. On a positive note, the data suggests that *freezing* saves some learning time without negative impact to generalization.

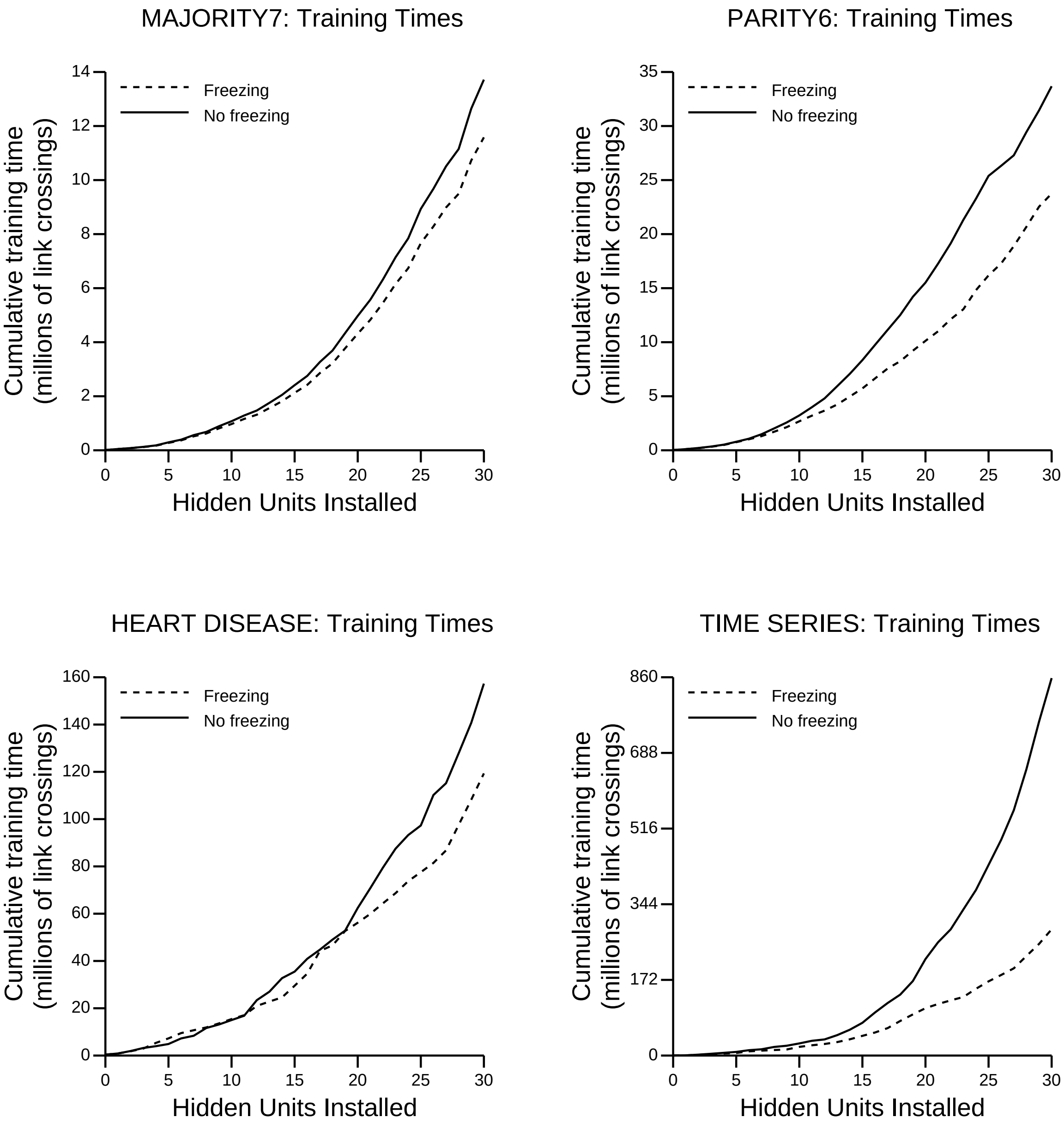

Figure 6: Learning time savings due to ***freezing.*** This figure shows runtimes of cascade-correlation produced using the quickprop learning rule, *patience*, and *correlation*. Without *freezing*, output learning amounts to multi-layer quickprop and further adjusts the input weights. Figure 7 shows the corresponding error curves.

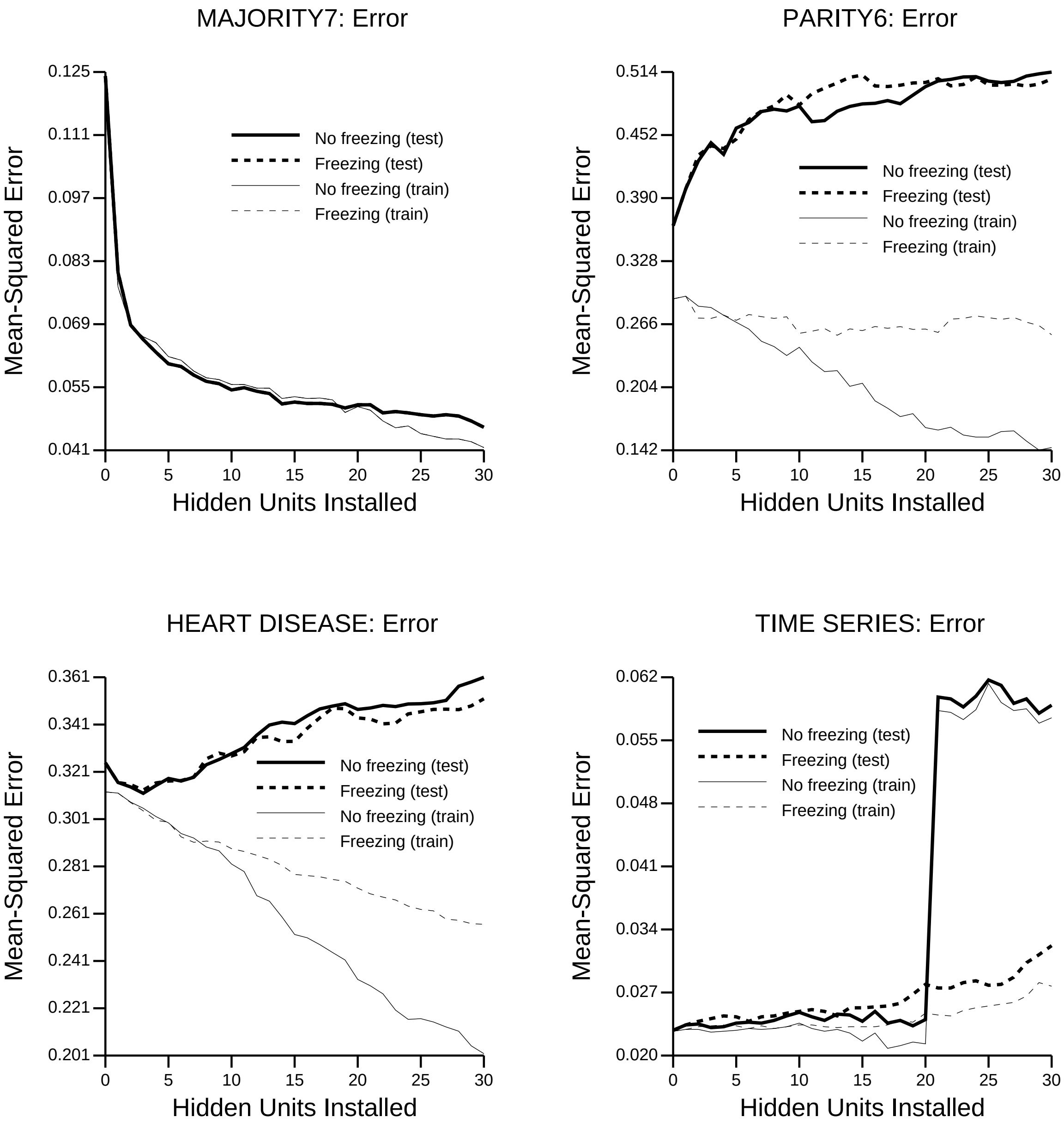

Figure 7: Generalization due to ***freezing.*** This figure shows learning and generalization curves produced using the quickprop learning rule, *patience*, and *correlation*. Without *freezing*, output learning amounts to multi-layer quickprop and further adjusts the input weights. Figure 6 shows the corresponding runtime curves. On the MAJORITY7 problem, *freezing* did not effect the training set error or generalization. Also, the jump in error levels at the 21st unit using no *freezing* in the TIME SERIES problem is an oddity for which we do not yet have an explanation.

Correlation: Results

Maximizing the *correlation* of the activations of new hidden units to the current error at the outputs should help these new units learn important high-level features or high-order terms. This *correlation* is done during input learning, which is expensive. Figure 8 shows comparative run-times with and without *correlation*. In all four domains, *correlation* introduces a significant learning time cost.

The additional learning time might be acceptable if the bias due to *correlation* improves learning and generalization over installing hidden units with randomized input weights. In all four domains, *correlation* resulted in significantly better learning of the training data, except beyond the twentieth hidden unit added in the TIME SERIES domain. However, this did not result in uniformly better generalization. *Correlation* significantly improved generalization in the MAJORITY7 domain but substantially hurt generalization in the PARITY6 and TIME SERIES domains. There was no significant difference in generalization in the HEART DISEASE domain. This data is shown in Figure 9.

Correlation proved very time-consuming. Some of this is due to the lack of caching in our cascade-correlation workbench implementation, but caching is unrealistic for large datasets in any case. For larger pools of hidden units, the cost would be even greater. Is *correlation* worth the price? *Correlation* does not seem to promote good generalization to novel examples.

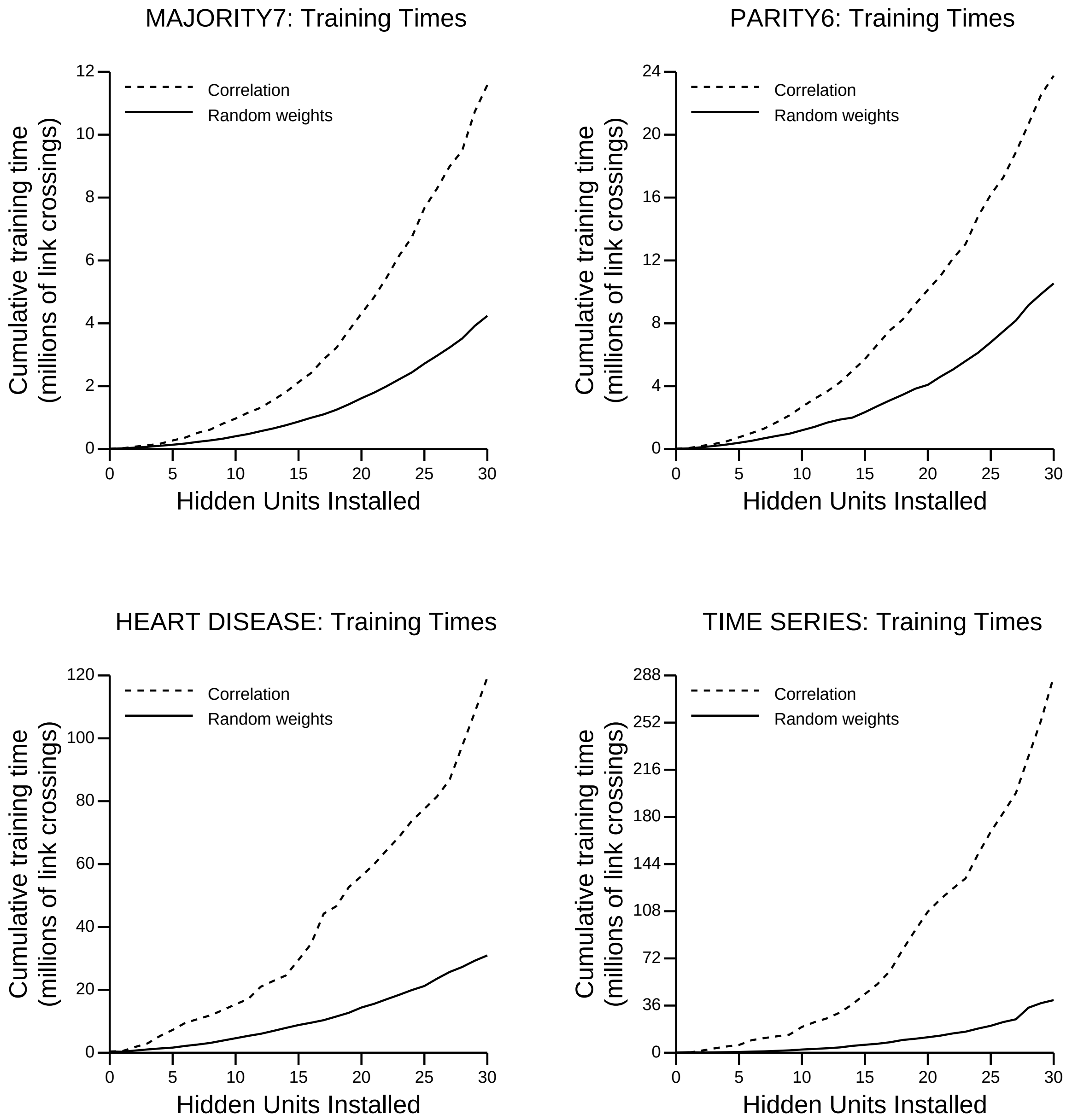

Figure 8: Learning time cost due to ***correlation.*** This figure shows runtimes of cascade-correlation produced using the quickprop learning rule, *patience*, and *freezing*. In some runs, new hidden units are installed using *correlation*. In others, they are given random initial input weights. Figure 9 shows the corresponding error curves.

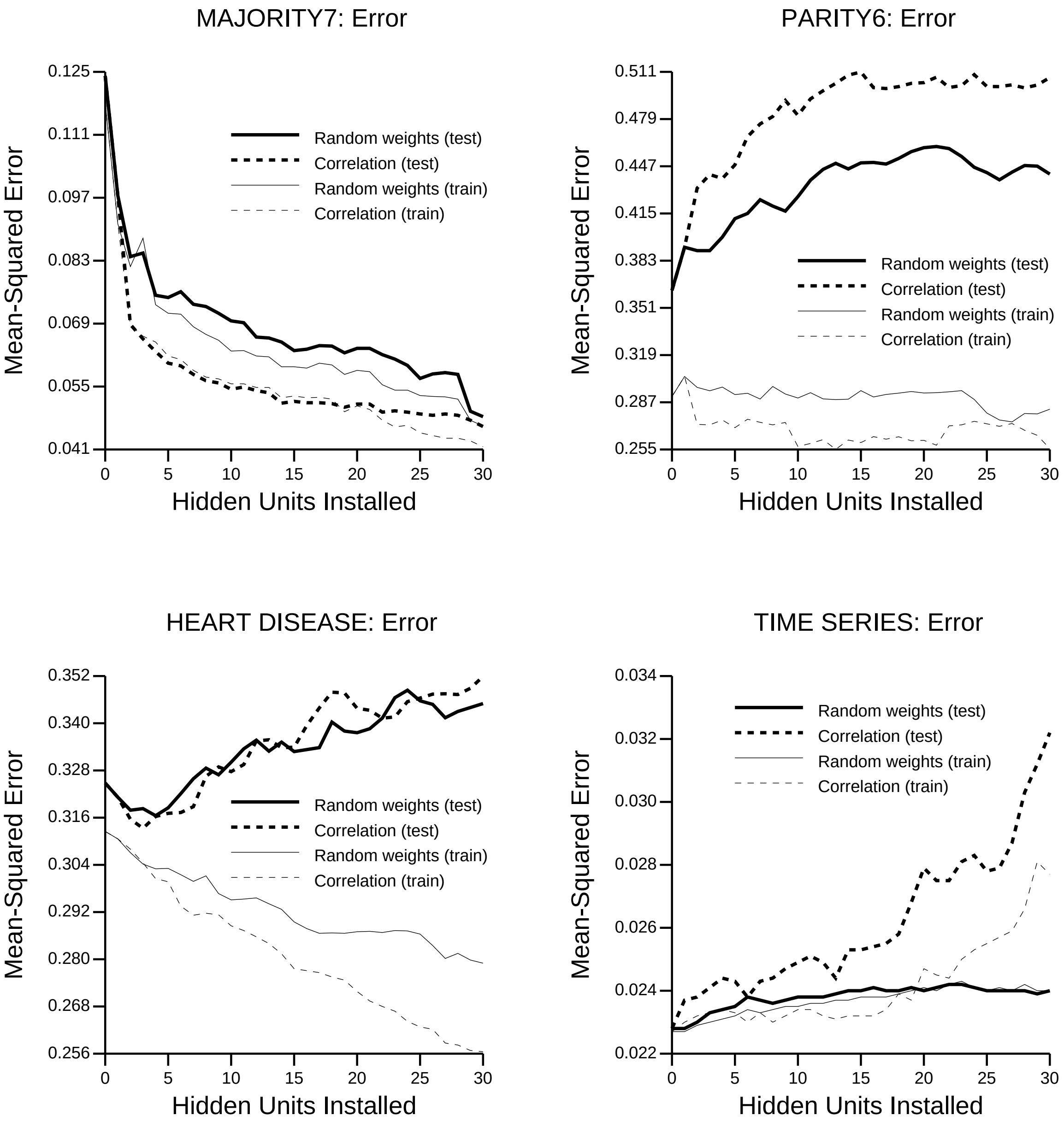

Figure 9: Generalization using ***correlation.*** This figure shows learning and generalization curves produced using the quickprop learning rule, *patience*, and *freezing*. In some runs, new hidden units are installed using *correlation*. In others, they are given random initial input weights. Figure 8 shows the corresponding runtime curves.

7. Conclusion

Evaluating algorithmic features of Fahlman and Lebiere's cascade-correlation learning architecture individually, we present a richer understanding of the costs and benefits of each one.

One result is that quickprop is not always quicker than backprop. Also, any speedup that might be gained from quickprop is more than likely offset by its numerical instability. Finally, and most significantly, backprop generalizes better. Performance on novel examples is arguably the most important aspect of a learning algorithm. Comparing backprop and quickprop outside of the cascade-correlation architecture may yield additional noteworthy results.

Patience is the most remarkable feature of Fahlman and Lebiere's system. The cost in training-set error is minimal and generalization is improved. Moreover, the time savings is significant. It is a good way to estimate when to terminate one phase of a learning algorithm and move to the next. As a well-principled, powerful heuristic which simultaneously improves runtime and generalization, *patience* is probably well-suited to any neural-network learning algorithm and architecture.

Patience might also be extended to operate at multiple levels. For example, if *patience* factors can be used to end individual learning phases, perhaps a more global *patience* factor could be used to decide when enough hidden units have been added to the network, and the entire network-building system should stop. Similarly, *patience* could operate at lower levels to adjust system parameters automatically.

Freezing of weights does not impact the order of magnitude of the growth in learning time as cascade-correlation adds hidden units. The runtime benefit is marginal, and it has potential pitfalls in terms of correctness and compactness of representation.

Correlation is a good heuristic for biasing new hidden units. Unfortunately, it is computationally expensive without caching, a substantial time/space trade-off. It helps learn the training set better with fewer nodes added to the network, but its value with respect to generalization is unclear.

We hope our experiments provide data that proves useful in the design of the next generation of neural-network learning algorithms.

Appendix A: Basic Equations

In this section, we review the basic equations used in backprop-style learning, which includes quickprop. The equations are designed to describe both backprop and quickprop in a common language using a common notation for easy comparison. Additionally, we present the fundamental equations for *correlation* in the same notation, showing that the gradient-descent principles of error minimization and correlation maximization are the same.

The derivations which follow are consistent with both quickprop [3] and backprop [9], starting with our chosen activation function, given by A.1.

$$f(x) = \frac{1}{1+e^{-x}} - 0.5 \tag{A.1}$$

At time t (time is only important in quickprop— see section 4), the activation at a node i, $a_i(t)$, is the value of the activation function applied to the weighted sum of the inputs, given by A.2.

$$a_i(t) = f(\sum_j w_{i,j}(t) a_j(t)) \tag{A.2}$$

To perform gradient descent, the function f must be differentiable. The sigmoidal function in A.1 is nice, since its derivative, shown in A.3, is a simple function of the activation value itself.

$$a'_i(t) = f'(\sum_j w_{i,j}(t)\, a_j(t)) = 0.25 - a_i^2(t) \tag{A.3}$$

For output training, in both quickprop and backprop, error is first calculated according to A.4 at each outputs unit i by taking the difference between its desired activation τ (provided by a hypothetical teacher) and its activation a.

$$ERROR_i(t) = \tau_i(t) - a_i(t) \tag{A.4}$$

Then, a term δ is computed via the rule in A.5. The rule is the same for hidden and output units.

$$\delta_i(t) = ERROR_i(t)\, a'_i(t) \tag{A.5}$$

The error gradient (partial derivative of the error with respect to the weight) is computed for each link from a node j to a node i according to A.6:

$$S_{i,j}(t) = \delta_i(t)\, a_j(t) \tag{A.6}$$

Error at each hidden unit i is computed using A.7 in lieu of A.4.

$$ERROR_i(t) = \sum_k \delta_k(t)\, w_{k,i}(t) \tag{A.7}$$

Input training maximizes correlation. This can be done using either backprop or quickprop rules. The correlation at candidate unit c of its activations a the errors at an output unit o is defined in A.8. All patterns p must be presented to calculate this term.

$$CORREL_o = \sum_p (a_c^p - \overline{a}_c)(ERROR_o^p - \overline{ERROR}_o) \tag{A.8}$$

The bar terms refer to averages over all presentations p. The term to maximize in input training is the sum of the absolute values of these correlations, given by A.9.

$$\overline{CORREL} = \sum_o |CORREL_o| \tag{A.9}$$

$\overline{CORREL}$ is differentiable, and its partial with respect to the weight on a link from j to i is given by A.10. Whether using backprop or quickprop, we can maximize $\overline{CORREL}$ by simply using A.10 in place of A.6, where σ_o is the sign of $CORREL_o$ (± 1) from A.8.

$$S_{i,j}(t) = \sum_o \sigma_o \sum_p (ERROR_o^p - \overline{ERROR}_o)\, a'^p_i\, a^p_j \tag{A.10}$$

Appendix B: Selected Parameter Values

The following table shows the system parameter settings used for each domain in the experiments described in this paper. We generally chose parameter values by trial-and-error hand-tuning to achieve locally optimal performance within the constraints of the cascade-correlation workbench.

Problem	**θ**	**ε**	**μ**	**Max Epochs PerPhase**	**Max Hidden Units**	**Patience**	
						n	***p***
MAJORITY7	0.35	0.25	1.6	20	30	4	0.01
PARITY6	0.25	0.25	1.6	100	30	5	0.001
HEART DISEASE	0.05	0.25	1.6	250	30	5	0.01
TIME SERIES	0.05	0.25	1.6	300	30	5	0.01

θ is an error threshold parameter. When error is being propagated backwards through the neural network, an error level $\leq \theta$ is treated as zero error. θ was chosen for each problem such that additional lowering of the value resulted in run-time errors using the quickprop learning rule.

ε is the backprop learning rate parameter, used in the learning rules for both backprop and quickprop. We used 0.25 for all four problems. This assures that error does decrease as learning progresses.

μ is the quickprop parameter which limits how large a weight update can be with respect to the previous weight update. Fahlman recommends 1.6 in his paper on quickprop [3]. This seemed to work well on all four problems.

MaxEpochsPerPhase is the absolute limit on how many times the training data set may be presented in a single phase of input or output training. For each problem, a value was chosen above which significant improvement in error does not seem to occur for most learning phases.

MaxHiddenUnits is the absolute limit on the number of hidden units which may be installed in the network. Thirty was used for all four problems.

n and ***p*** are the *patience* parameters which may help terminate a learning phase before the maximum number of epochs per phase is reached. After each epoch, the program checks to see if, for the past ***n*** consecutive epochs, error has changed by a factor of at least ***p***. If not, then the current learning phase ends. For each problem, a sufficiently large value for ***n*** and a sufficiently small value for ***p*** were chosen that seemed to save learning time without a substantial increase in error.

References

[1] Ash, T. (1989). *Dynamic Node Creation in Backpropagation Networks.* ICS Report #8901, Institute for Cognitive Science, University of California at San Diego, La Jolla, CA.

[2] Detrano, R. (unpublished manuscript). *International Application of a New Probability Algorithm for the Diagnosis of Coronary Artery Disease.* V. A. Medical Center, Long Beach, CA.

[3] Fahlman, S.E. (1988). "Faster-Learning Variations on Back-Propagation: An Empirical Study" in *Proceedings of the 1988 Connectionist Models Summer School*, Morgan Kaufmann, pp. 38-51.

[4] Fahlman, S.E. and Lebiere, C. (1990). "The Cascade-Correlation Learning Architecture" in Touretzky, D.S. (ed.), *Neural Information Processing Systems 2*, Morgan Kaufmann, pp. 524-532.

[5] Lapedes, A.S. and Farber, R. (1987). *Nonlinear Signal Processing Using Neural Networks: Prediction and System Modeling.* Technical Report, Los Alamos National Laboratory, Los Alamos, NM.

[6] Le Cun, Y., Denker, J.S., and Solla, S.A. (1990). "Optimal Brain Damage" in Touretzky, D.S. (ed.), *Neural Information Processing Systems 2*, Morgan Kaufmann, pp. 598-605.

[7] Moody, J. and Darken, C.J. (1989). "Fast Learning in Networks of Locally-Tuned Processing Units" in *Neural Computation 1*, MIT Press, pp. 281-294.

[8] Mozer, M.C. and Smolensky, P. (1989). "Skeletonization: A Technique for Trimming the Fat from a Network via Relevance Assessment" in Touretzky, D.S. (ed.), *Neural Information Processing Systems 1*, Morgan Kaufmann, pp. 107-115.

[9] Rumelhart, D.E., Hinton, G.E., and Williams, R.J. (1986). "Learning Internal Representations by Error Propagation" in Rumelhart, D.E. and McClelland, J.L., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, MIT Press, pp. 318-362.

[10] Tenorio, M.F. and Lee, W. (1989). "Self Organizing Neural Networks for the Identification Problem" in Touretzky, D.S. (ed.), *Neural Information Processing Systems 1*, Morgan Kaufmann, pp. 57-64.