Genetic Cascade Learning for Neural Networks

N. Karunanithi, R. Das and D. Whitley
Computer Science Department
Colorado State University, Fort Collins, CO 80523.
(E-mail: `karunani/das/whitley@cs.colostate.edu`)

Abstract

Genetic Cascade learning is a new constructive algorithm for connectionist learning which combines genetic algorithms and the architectural feature of the Cascade-Correlation learning algorithm. Like the Cascade-Correlation learning architecture, this new algorithm also starts with a minimal network and dynamically builds a suitable cascade structure by training and installing one hidden unit at a time until the problem is successfully learned. This step-wise constructive algorithm exhibits more scalability than existing genetic algorithms and is free of the *competing conventions* problem which results from the fact that functionally equivalent networks may have different assignments of functionality to individual hidden units. Initial tests of Genetic Cascade learning are carried out on a difficult supervised learning problem as well as a reinforcement learning control problem.

1 Problem Statement

Genetic Algorithms have been applied in connectionist learning both to optimize weights of a fixed size network and to find the topology of the network (Harp, Samad, & Guha 1989, and Miller, Todd, & Hegde 1989). However genetic algorithms do not exhibit good scalability and fast convergence when compared with algorithms such as *Cascade-Correlation learning architecture* (Fahlman & Lebiere 1990).

When genetic algorithms are used to train multilayer networks they may exhibit poor convergence due to a fundamental problem we have previous called the *structural/functional mapping problem* (Whitley et al., 1989; 1990); Radcliffe (1990) calls this the *permutation problem* and more recently it has come to be called the *competing conventions* problem. This problem arises, regardless of whether a binary encoding or a real-valued encoding is used, because genetic algorithms operate on a population of solution strings and at any given generation it is possible to have multiple representations of functionally equivalent networks within the population. Recombining strings with different representations typically will produce offsprings that are ineffective. For example, consider two neural nets with two hidden nodes each. Both may successfully learn the same training problem, but the

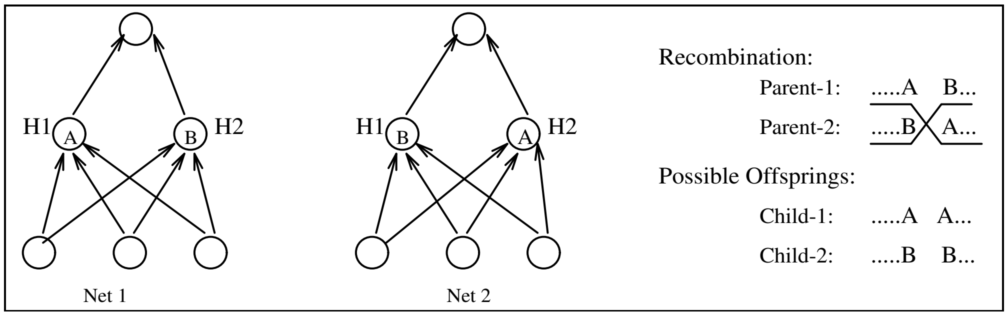

Figure 1: The "competing conventions problem". Assume that each hidden unit performs a distinct function. With 2 hidden units, these functions can be labeled A and B. Thus Net1 and Net2 represent two possible solutions. Performing crossover on these parents can result in disfunctional offsprings.

two solutions may not be compatible because one net learns to compute function A in hidden node H1, and function B in hidden node H2, while the other learns just the opposite mapping. Figure 1 gives an example of this problem. Attempts to reduce the severity of this problem have relied on using smaller population sizes and more aggressive mutation rates (Whitley, Starkweather & Bogart 1989). However these methods do not eliminate the *competing conventions* problem, but rather only make its impact less significant.

In this paper we present a solution to the *competing conventions* problem. *Genetic Cascade learning* combines genetic algorithms and the features of the Cascade-Correlation learning algorithm introduced by Fahlman and Lebiere (1990). These two algorithms are fundamentally different in terms of how they optimize network weights because the genetic algorithm uses neither correlation information nor any gradient information. Genetic Cascade learning is applied to both a difficult supervised learning task, the 2-spirals problem, and a reinforcement learning task, the control of an inverted pendulum.

2 Background

2.1 Cascade-Correlation Learning Architecture

The Cascade-Correlation learning architecture combines two important ideas in its learning method. First the *cascade architecture* adds hidden layers to the network one at a time. Second, a *learning algorithm* creates and installs each individual hidden unit using correlation information to directly train the hidden unit. The Cascade-Correlation algorithm in effect "grows" a connectionist network. An overall outline of this algorithm is as follows.

1. **Initialize the Network**: Create a minimal network consisting of only input and output units. Establish links from the input layer to the output layer and initialize them with random values.
2. **Train Output Layer**: Adjust weights feeding the output units using error propagation. If the learning is complete (i.e, the error is below a preset limit) then stop; else if the error has not been reduced significantly for *patience* number of consecutive epochs

or the *timeout* (i.e., the maximum number of epochs allowed) has been reached, then go to the next step.

3. **Initialize Candidate Units**: Create and initialize a pool of candidate units with learnable weights from all input units, all pre-existing hidden units and the bias unit.

4. **Train Candidate Units**: Adjust each candidate unit's weights using gradient ascent so as to maximize the correlation between activation of the candidate unit and the residual error of the network. If the correlation has not improved significantly for *patience* number of consecutive epochs or the *timeout* has been reached, then go to the next step; else repeat step 4.

5. **Install a new Hidden Unit**: Select the candidate unit with the highest correlation value and install it in the network as a new hidden unit by establishing links to the output units. Now "freeze" its incoming weights and initialize the newly established outgoing weights with the negative of the correlation values. Go to step 2.

Two important features of this algorithm are: i) it trains only one layer of weights at any time and 2) it trains and adds one hidden unit at a time. These features are important to the use of genetic algorithms because they can be exploited not only to improve scalability but also to eliminate the *competing conventions* problem among hidden units.

2.2 Genetic Algorithms

Our experiments use a version of the genetic algorithm we refer to as the GENITOR algorithm. The mechanics of the algorithm are as follows. A population of strings is randomly generated. Each string is evaluated and the population is sorted by ranking strings in terms of their evaluations. A random selection function with a linear bias towards the higher ranked strings is used to stochastically pick two parents for recombinations. The parents are recombined so as to produce a single offspring (or, if you will, two offspring are generated and 1 is randomly discarded). After the offspring is evaluated it replaces the lowest ranked string in the population and is inserted into its appropriate rank location. The differences between this and a standard genetic algorithm, as well as comparative performance data on small neural network optimization problems and other problems is given in (Whitley & Kauth 1988) as well as (Whitley & Hanson 1989).

Several researchers have attempted to apply genetic algorithms to neural network weight optimization problems. The most straightforward way of doing this is to encode each weight in the network as binary substrings; the entire binary encoding would be a string composed of these concatenated substrings. Genetic algorithms which rely on recombination of binary encodings and limited mutation can easily solve small neural net problems, but we have not been able to scale this kind of algorithm to handle larger problems. Some basic changes in the GENITOR algorithm resulted in the ability to handle relatively large neural net training problems (Whitley, Starkweather & Bogart 1990). Our results are similar to those reported by Montana and Davis (1989). First, the problem encoding is real-valued instead of binary. This means that each parameter (weight) is represented by a single real value and that recombination can only occur between weights. Second, a much higher level of mutation is used; traditional genetic algorithms are largely driven by recombination, not

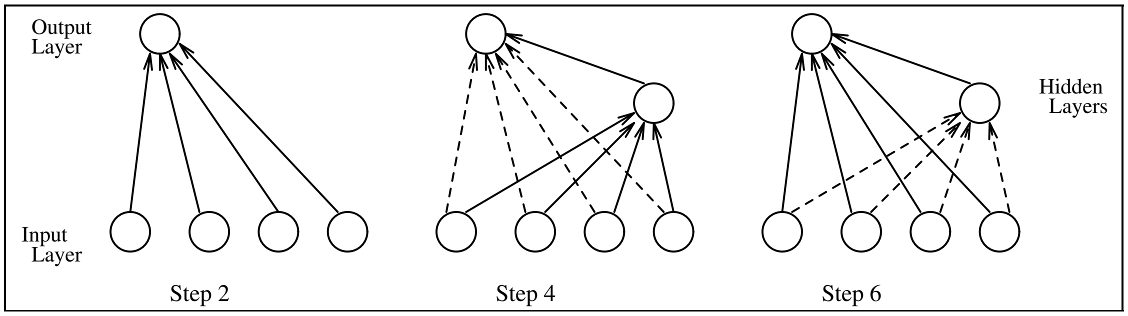

Figure 2: Three weight-updation steps in Genetic Cascade learning. The solid lines represent connections that are actively trained at the step indicated. The dashed lines represent weights which are not trained at the particular time step indicated.

mutation. Third, a small population is used (e.g. 50 individuals). The use of small populations reduces the exploration of the multiple (representationally dissimilar) solutions for the same net. The stronger reliance on mutation may also help to avoid this problem since no recombination is involved when mutation occurs. Although the implementation details are not so different from conventional genetic algorithms, the empirical evidence suggests that the result is a type of stochastic hill-climbing algorithm which we refer to as a "genetic hill-climber" (Whitley et al., 1991). It should be noted that most of the existing theory for genetic algorithms does not necessarily apply to this algorithm. This algorithm has solved some large supervised learning problems (a neural network with approximately 500 weighted connections for a difficult signal detection problem) in approximately the same amount of time as back-propagation (Whitley, Starkweather & Bogart 1990), but it is still much slower than the Cascade-Correlation algorithm.

3 Genetic Cascade Learning

While the genetic hill-climber has been moderately successful for training neural networks, it side-steps the *competing conventions* problem rather than solving it. On supervised learning problems, we have also found that it does not scale as well and does not produce as reliable generalization as the Cascade-Correlation learning architecture. One advantage of the genetic hill-climbing algorithm for training neural nets, however, is that it can optimize weights without using gradient information. Given these considerations, we have investigated a Genetic Cascade learning algorithm. One principle which guided the development of this algorithm was that the algorithm should be general enough to apply to both supervised learning problems and reinforcement learning problems. As a result, it uses neither gradient information nor correlation information. The algorithm is as follows.

1. **Initialize the Network**: Create a minimal network consisting of only input and output units.
2. **Train Output Layer**: Create and initialize a random population corresponding to a single layer of weights feeding the output layer and train them using GENITOR. If the learning is complete then stop; else if the error has not been reduced significantly for

patience number of consecutive evaluations or the *timeout* (i.e., the maximum number of evaluations allowed) has been reached, then go to the next step.

3. **Initialize Candidate Units**: Create and initialize a random population sized to fit the problem. Each string in the population represents weights linking a candidate unit to input units, all pre-existing hidden units, output units and the bias unit.

4. **Train Candidate Units**: Adjust candidate units' weights using GENITOR. If the learning is complete then stop; else if the error has not been reduced significantly for *patience* number of consecutive evaluations or the *timeout* has been reached, then go to the next step.

5. **Install a New Hidden Unit**: Select the candidate unit which is at the top of the pool and install it in the network as a new hidden unit. Now freeze its incoming weights and go to the next step.

6. **DELTA Train Output Layer**: Create and initialize a random population with *Delta Values* sized to fit the links feeding the output layer and train them using GENITOR. If the learning is complete or if the maximum number of hidden units have been added then stop; else add the *Delta Values* of the gene at the top of the pool to the values of the output layer weights and go to step 3.

Figure 2 illustrates steps 2, 4 and 6. Note that connections which directly feed into output nodes are continually retrained each time step 6 is repeated. However those connections that feed into hidden units are trained only once and are then frozen.

Note that we are really running the genetic hill-climber multiple times to solve several different optimization subproblems. A few important differences between the Genetic Cascade learning and the Cascade-Correlation learning are as follow: 1) The parameters that decide stagnation(*patience*) and *timeout* are similar in both algorithms; however their values are much larger in the Genetic Cascade learning than in the Cascade-Correlation algorithm. 2) When initializing the population we use a different range for the random number generator depending on whether it is for the output layer or for the hidden units. We initialize the population with random values over the following ranges: +1.0 to -1.0 for the output layer, +4.0 to -4.0 for the hidden units and +2.0 to -2.0 for the *Delta Values*. 3) To retrain the output layer we use *Delta Values* instead of completely new random values. The values encoded on the strings manipulated by the genetic algorithm are added to the already existing weights that feed into the output units. These delta values therefore represent an adjustment to the existing weight vector feeding the output units, and thus are defined to have a smaller range. 4) When we train a candidate unit we use a population size of 50. 5) Weights are adjusted by GENITOR and hence there is no need for gradient values. 6) In our approach the training of candidate units includes the weights connected to the output units; it is therefore possible for the algorithm to terminate while training a hidden unit.

4 Results

The Genetic Cascaded networks had no difficulty in learning problems such as exclusive-or and adding two 2-bit numbers. We chose as our primary benchmark the "two-spirals" problem, a problem that is very difficult to learn using traditional back-propagation. Fahlman also used the two-spirals problem as a benchmark for the Cascade-Correlation algorithm. Another problem used in our study is a pole balancing and cart centering problem also known as the inverted pendulum problem. We use this as our "reinforcement learning" problem because it has been widely studied (Anderson 1989) and because genetic algorithms have previously been used to solve this problem with fixed networks (Whitley et al., 1991; Weiland 1991).

4.1 The Two-Spiral Problem

For the two-spirals problem, the network has two continuous valued inputs and one output. The training set consists of 192 pairs of X,Y coordinate values. The input patterns are arranged in two interlocking spirals that circumscribe the origin three times. Exactly half of the input patterns are selected from one spiral with a +1 output, whereas the training patterns from the other spiral results in a -1 output.

In general this problem is very hard to solve using traditional back-propagation. Lang and Witbrock (1988) report obtaining a solution to the two-spirals problem using back-propagation on a network with three hidden layers, each with five hidden units. A salient feature of this particular network is that each unit receives connection from every unit in every earlier layer, not just from the immediately preceding layer.

Fahlman and Lebiere (1990) show that the Cascade-Correlation learning is able to solve the two-spiral problem rapidly and consistently. In our experiments using Cascade-Correlation we find that the number of hidden units required to solve the two-spirals problem varies from 13 to 24 with an average of 18.6 and a median of 18.

To test the Genetic Cascade learning, for each set of parameter settings we performed a set of forty experiments. Two values of the *timeout* parameter (5,000 and 10,000 evaluations) and two values of the *patience* parameter (1000 and 2000 evaluations) were tested. The maximum number of hidden units was limited to 40. In each of the four sets of forty experiments, we found that the Genetic Cascaded network learned more than 96% of the training patterns. In 50% of all experiments, the networks learned 100% of the training patterns. Our tests show that higher success rates can be achieved if the parameters are tuned carefully. To solve the two-spirals problem with the timeout parameter equal to 5000 evaluation, the networks required 30 hidden units on an average. The average of the total number of evaluations with timeout equal to 5000 evaluations was 300,000. Overall, the number of hidden units used by the Genetic Cascade nets varied from 16 to 40.

In Figure 3, we have plotted average of the total sum squared error over forty experiments as a function of the number of hidden units (*and not total training time*). In these plots the upper curve corresponds to the Genetic Cascade learning and the lower one for the Cascade-Correlation algorithm. Each vertical bar in these graphs represents ± 1 standard deviation over the average of the sum squared error. We see that initially, when the size of the network is small, the performance of the Genetic Cascaded networks is roughly comparable to the Cascade-Correlation architecture. However, as the total sum squared error is reduced, the number of training patterns to be learned also decreased. Here, we see a marked difference in the performance of the two approaches. Unlike the Cascade-Correlation algorithm, which

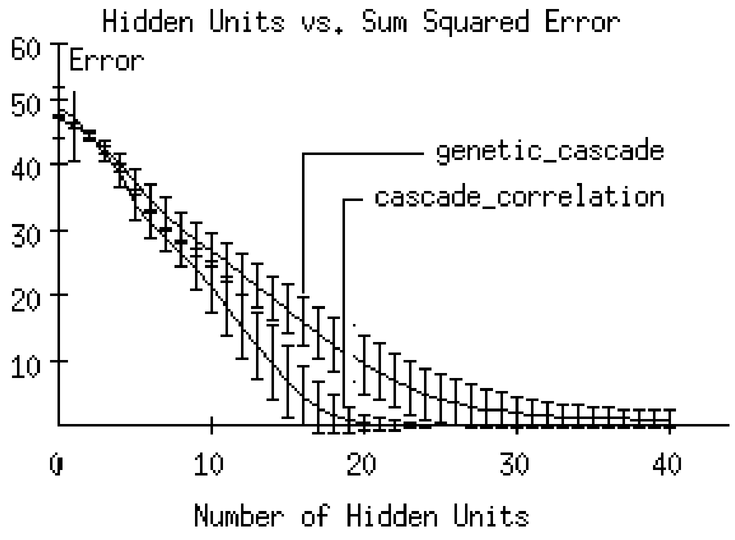

Figure 3: A comparison of Cascade-Correlation learning and Genetic Cascade learning. The graph plots sum squared error as a function of the number of hidden units installed.

can use correlation information to focus in on residual errors, the Genetic Cascaded learning has no way of focusing on these training patterns. Thus, it uses more hidden units than the Cascade-Correlation algorithm. The Genetic Cascade learning also takes longer to train each individual unit when compared to gradient methods.

Figure 4 illustrates the "receptive fields" of a net trained by the Genetic Cascaded learning. After each hidden unit is added, the resulting network is tested for 25,921 (161x161) input patterns and the corresponding output is recorded. For plotting purposes, each of the 25,921 pixels is colored black if the output is less than zero; else the pixel is colored white. Plots are arranged adjacently in a left to right, top to bottom fashion. The top left plot displays the output response of the network before the first hidden unit is added, while the bottom right plot presents the output after the 24th hidden unit is added. A more detailed inspection of the training patterns that are learned after the addition of a particular hidden unit show a great deal of similarity in the way the Genetic Cascade learning and the Cascade-Correlation algorithm solve the two-spiral problem. In other words, both approaches learn to correctly classify the same training patterns initially. This similarity decreases only when the receptive field becomes irregular.

The final network obtained in the above case failed to learn two of the 192 training patterns. Our purpose in presenting the results from such a network is to delineate some of the distinctive characteristics of the Genetic Cascaded approach. In Figure 4, we can see that there is very little change in the receptive field after the addition of the 21st hidden unit (units 21 to 25 form the last row in Figure 4). In fact, for the next three hidden units (22 to 24), the network fails to correctly classify the same two patterns out of the 192 training patterns. The Cascade-Correlation algorithm has no difficulty in such a situation, since the new hidden unit is chosen only if its activation pattern correlates with the remaining error. Since the genetic algorithm does not rely on the correlation between the error and the hidden

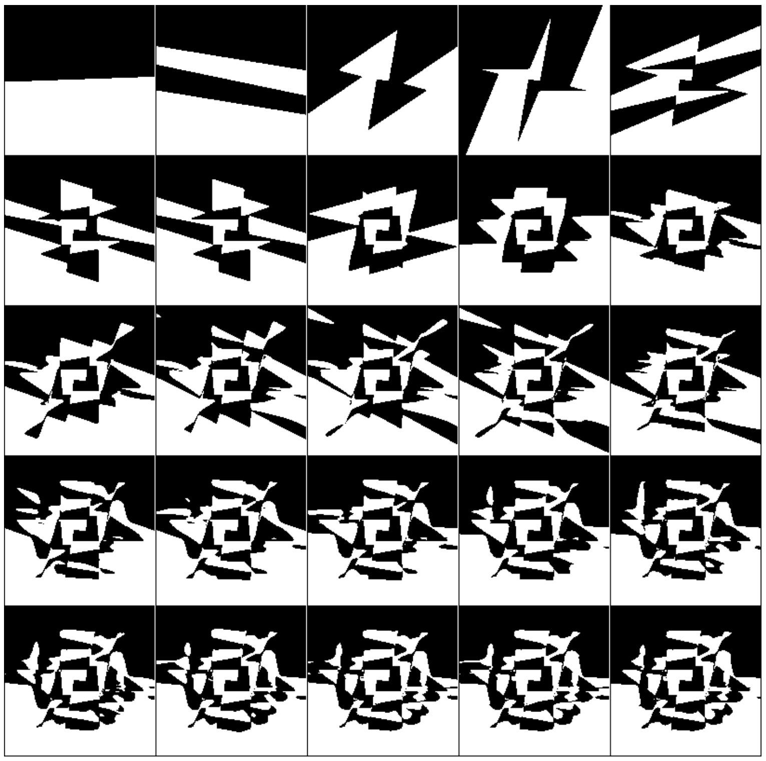

Figure 4: Receptive Field of the Genetic Cascade learning for a typical run

unit's activation it is unable to focus on the two patterns and eliminate the remaining error.

4.2 The Pole-balancing Problem

The inverted pendulum problem is a well-studied problem which involves controlling an inherently unstable mechanical system of a cart and a pole that is constrained to move within a vertical plane. The objective is to keep the pole balanced and to avoid track boundaries. At any point in time, the available state information includes the angle of the pole, θ, and the angular velocity of the pole, $\dot{\theta}$, as well as the position of the cart, ρ, and the velocity of the cart, $\dot{\rho}$. Using θ, $\dot{\theta}$, ρ and $\dot{\rho}$ as inputs, the output of the neural network is one of two actions applied to the cart at each time step: full-push left or full-push right, as in bang-bang control. In case of the *deterministic* action policy, the cart is pushed to the left if the output of the network is less than 0.5, else it is pushed to the right. For the *probabilistic* action policy the output of the network represents the probability with which the cart is pushed to the right. For details on the equations of motions for this system refer to Anderson (1989). The control problem is approximately linear for $\theta \leq 12°$ and non-linear otherwise. We performed experiments that attempt to balance the pole with θ ranging up to $\pm 35°$. A failure signal is generated either when the pole falls beyond θ (12° or 35°) from vertical or when the cart runs into the ends of the track at ± 2.4 meters from center.

Following Anderson's work (1989), we stopped learning when a network was found that was able to maintain the system without generating a failure signal for 120,000 time steps (40 minutes of simulated time). The "fitness" of a string is based on the accumulated number of steps until failure for a single initial state vector. The *timeout* limits used for hidden unit

PERFORMANCE OF GENETIC CASCADE LEARNING FOR THE INVERTED PENDULULM										
	Maximum Pole Angle = $\pm 12°$					Maximum Pole Angle = $\pm 35°$				
	Number of Successes = 21(70.0%)					Number of Successes = 25(83.3%)				
	Min	Max	Mean	Median	STD	Min	Max	Mean	Median	STD
Hidden Units	0	10	5.05	6	3.49	1	10	4.96	4	2.76
Evaluations(x10^3)	9.7	377.2	192.1	209.0	123.7	54.7	358.7	195.0	173.1	103.9
Hidden Unit Training Timeout = 25,000 Evaluations										
Output Layer Training Timeout = 35,000 Evaluations										
Ratio of Crossover Vs Mutation ≈ 0.92						Patience = 10,000 Evaluations				

Table 1: Performance of Genetic Cascade learning for 12 and 35 degree problems averaged over 30 experiments. The number of successes represents the number of experiments in which the algorithm was able to evolve a network architecture that successfully balanced the pole for 120,000 time steps. An experiment is considered failure if the algorithms fails to balance the pole after installing 10 hidden units. The population size was 50 in these experiments.

training and the output layer training were set at 25,000 and 35,000 respectively. If the algorithm was not making significant progress for 10,000 successive evaluations then it was considered stagnant. (We used the same value for *patience* to train both hidden layer units and the output layer.) We measured the learning efficiency of our algorithm in terms of the number of hidden units and the number of evaluations required to balance. Table 1 shows the performance of the Genetic Cascade learning for 12° and 35° problems.

The minimum number of hidden units required to balance the pole for the 12° problem was *zero* which is consistent with the fact that the problem is approximately linear. For the 35° problem the smallest network found by the algorithm had only one hidden unit which suggests that the problem cannot be solved using a single layer network. This minimal architecture is equivalent to the minimal architecture reported by Whitley et al. (1991). The number of experiments in which the algorithm solved the 35° problem was higher than that of the 12° problem. We conjecture that once the pole is in a vertical position the algorithm was not able to keep it consistently within $\pm 12°$ because it often made moves that overshoot those limits. However, when the failure angle was increased to $\pm 35°$, the algorithm had more flexibility and was able to generate moves that kept the pole balanced. In all experiments, the total number of evaluations needed to solve the problem increased approximately linearly with the number of hidden units added to the network.

The preliminary learning experiments reported here were based on the deterministic action policy. With the deterministic action policy the ability of the algorithm to evolve networks, that could balance the pole for 120,000 time steps, is influenced by the initial state vector used to evaluate a network. However, in future we plan to improve the algorithm so that it can evolve networks based on a tougher learning criteria suggested by Whitley et al. (1991).

In general the results of using Genetic Cascade learning on this reinforcement problems

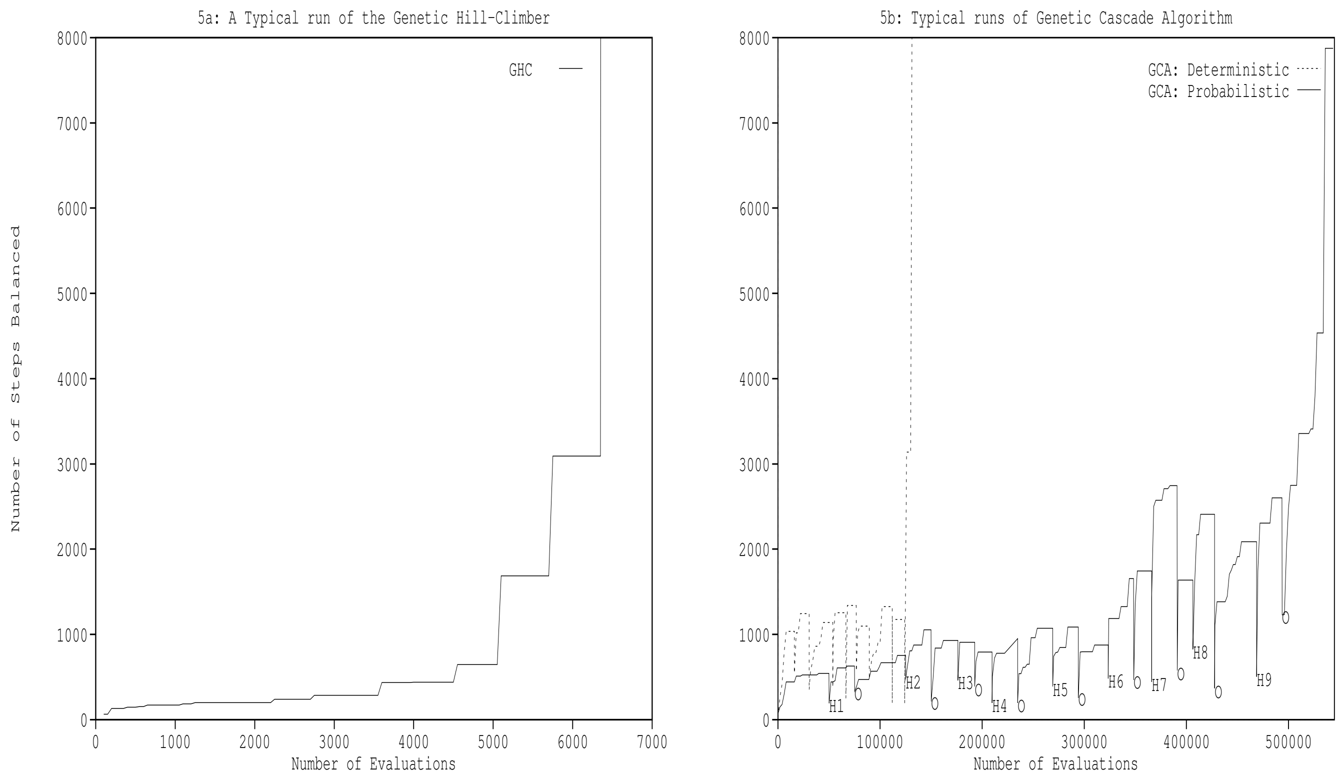

Figure 5: Figure 5a shows a typical run of the Genetic Hill-Climber (GHC). For comparison, a typical run of the Genetic Cascade Algorithm (GCA) is shown in Figure 5b. In these graphs the number of steps balanced by the best network in the population is plotted against the total number of evaluations.

are inferior to results using genetic hill-climbing with a fixed neural network architecture (Whitley et al., 1991). The fixed architecture learns after approximately 5000 total attempts and exhibits generalization which compares favorably both in terms of learning rates and generalization with the *Adaptive Heuristic Critic,* an approach to reinforcement learning based on temporal difference methods (Sutton 1988; Anderson 1989). A typical run for the 35° problem from both the genetic hill-climber and the Genetic Cascade learning are illustrated in Figure 5. In Figure 5a the genetic hill-climber used the probabilistic action policy and it was able to successfully balance the pole for 120,000 time steps. Figure 5b shows the performance of the Genetic Cascade learning for both the deterministic and the probabilistic action policies. With the deterministic action policy the final network was able to successfully balance the pole, whereas the best performance obtained with the probabilistic action policy was close to 8000 steps.

As shown in Figure 5a the performance of the genetic hill-climber increases monotonically since the best network always stays in the population. On the other hand, in Genetic Cascade learning the best network is modified every time when a new candidate unit is added. Such incremental modifications do not necessarily improve the performance of the network because of sparse nature of the feedback in this reinforcement learning problem. This results in non-monotonic improvement in the performance as illustrated in Figure 5b. The sharp dips in the performance of the Genetic Cascade learning occurs because the population of potential hidden units and that of delta values are initialized with random values (steps 3 and 6 of the Genetic Cascade Learning respectively).

The use of *cascade* style algorithms has never been explored for reinforcement learning problems, and it is possible that freezing weights (as occurs in these cascade algorithms) results in features to be prematurely fixed because of the sparse feedback. This limitation does not arise in a fixed architecture since any weight can be modified at any stage of the search process.

5 Conclusions

The experiments with Genetic Cascade learning for supervised learning and reinforcement learning presented in this paper are preliminary in nature. We have empirically tested this this new algorithm by applying it to the 2-spiral problem and the pole balancing problem. These are not necessarily the best or most appropriate problems for Genetic Cascade learning, but have served to provide feedback for algorithm development. There are still many issues that need to be addressed.

1. *Speed*: The computational time needed to solve the 2-spiral problem with Genetic Cascade learning was much greater than that of Cascade-Correlation algorithm. When optimizing individual hidden units the genetic search is slower than algorithms such as quick-propagation because it does not use any gradient information. The Genetic Cascade learning also made less effective use of individual hidden units in later stages of learning because it was unable to focus its search within a narrowed residual error surface. It is possible to correct these problems by either using correlation information, or by introducing other focusing mechanisms.
2. *Alternate Encodings*: At present the Genetic Cascade learning uses a real-valued encoding. Since the algorithm always deals with a small number of weights at any given stage one should be able to use binary encodings. Experiments with more traditional genetic algorithms should also be more successful since the *competing conventions* problem as it relates to multiple hidden units no longer exists.
3. *Higher Order Networks*: Since the Genetic Cascade learning does not require error gradient information one should be able to use it to train networks with complex transfer units, such as Sigma-Pi units, Product units, Splines or other types of units with higher order polynomial activation functions. As we have previous argued (Whitley et al., 1991), genetic algorithms have the most to contribute to neural network learning for applications where gradient information is unavailable or is otherwise difficult to obtain; reinforcement learning is one such application. However, if there is indeed a basic conflict between the goals of reinforcement learning and the use of cascade architectures, then the application of Genetic Cascade learning to networks with higher-order polynomial activation units may represent the best avenue for short term research for exploring possible applications of Genetic Cascade learning.

Acknowledgements:
This research was supported in part by NSF grant IRI-9010546.

References

[1] Anderson, C. W. (1989). "Learning to Control an Inverted Pendulum Using Neural Networks", *IEEE Control Systems Magazine,* 9, 31-37.

[2] Fahlman, S., & Lebiere, C. (1990). "The Cascade-Correlation Learning Architecture", *CMU-CS-90-100, Carnegie Mellon University.*

[3] Harp, S., Samad, T., & Guha A. (1989). "Towards the Genetic Synthesis of Neural Networks", *Proc. of the 3rd Int. Conf. on Genetic Algorithm*, Morgan Kauffman, pp 360-369.

[4] Lang, K. J., & Witbrock, M.J. (1988). "Learning to Tell Two Spiral Apart", *Proc. of the 1988 Connectionist Models Summer School*, Morgan Kauffman.

[5] Miller, G., Todd, P., & Hegde. S. (1989). "Designing Neural Networks using Genetic Algorithms", *Proc. of the 3rd Int. Conf. on Genetic Algorithm*, Morgan Kauffman, pp 379-384.

[6] Montana, D., & Davis, L. (1989). "Training Feedforward Neural Networks Using Genetic Algorithms", *IJCAI*, 1, pp 762-767.

[7] Radcliffe, N.J. (1990). *Genetic neural networks on MIMD computers.* Unpublished doctoral dissertation, University of Edinburgh, Edinburgh, Scotland.

[8] Sutton, R. (1988). "Learning to Predict by the Methods of Temporal Differences", *Machine Learning,* 3, pp 9-44.

[9] Weiland, A. (1991). "Evolving Neural Network Controllers for Unstable Systems", *IJCNN-91 Seattle*, Vol-II, pp 667-673.

[10] Whitley, D., & Kauth, K. (1988). "GENITOR: A Different Genetic Algorithm", *Proc. of the 1988 Rocky Mountain Conf. on Artificial Intelligence,* Denver, pp 188-130.

[11] Whitley, D., & Hanson, T. (1989). "Optimizing Neural Networks Using Faster, More Accurate Genetic Search", *Proc. of the 3rd Int. Conf. on Genetic Algorithm*, Morgan Kauffman, pp 391-396.

[12] Whitley, D., Starkweather, T., & Bogart, C. (1990). "Genetic Algorithms and Neural Networks: Optimizing Connections and Connectivity", *Parallel Computing*, 14, pp 347-361.

[13] Whitley, D., Dominic, S., & Das, R. (1991). "Genetic Reinforcement Learning with Multilayer Neural Networks", *Proc. of the 4th Int. Conf. on Genetic Algorithm*, In R. Belew and L. Booker, (Eds.), Morgan Kauffman, pp 562-569.