Designing Neural Networks using Genetic Algorithms

Geoffrey F. Miller **Peter M. Todd**
Psychology Department
Stanford University, Stanford, CA 94305
geoffrey@, todd@psych.stanford.edu

Shailesh U. Hegde
Department of Computation and Neural Systems
California Institute of Technology, Pasadena, CA 91125
shailesh@aurel.caltech.edu

Abstract

We present a genetic algorithm method that evolves neural network architectures for specific tasks. Each network architecture is represented as a connection constraint matrix mapped directly into a bit-string genotype. Modified standard genetic operators act on populations of these genotypes to produce network architectures with higher fitnesses over successive generations. Architecture fitness is assessed by training particular network instantiations and recording their final performance error. Three applications of this method to simple network mapping tasks are discussed.

1 Introduction

A new computational paradigm is gaining popularity throughout diverse fields ranging from psychology and cognitive science to signal processing and pattern recognition. This paradigm, parallel distributed processing (PDP), replaces the single powerful processor of the tradition von Neumann computer with a network of simple interconnected processing units. The network's computational power emerges from the collective activity of these units operating in parallel (Rumelhart and McClelland, 1986). PDP systems are often called neural networks, based on features shared with sets of real biological neurons. Central among these similarities are the ability to learn mappings from a set of inputs to a set of outputs based on training examples, and the ability to generalize beyond the particular examples learned. These useful abilities have brought a surge of neural network research activity in the past few years. New learning algorithms have been developed, mathematical foundations have become deeper and broader, and network models have been trained and applied to solve difficult problems in a variety of domains. But one aspect of current neural network research remains a bottleneck that could seriously impair progress in the coming years if left unaddressed. This bottleneck is the problem of network design.

1.1 The Problem of Network Design

The process of developing a neural network model for a particular application typically includes the following four stages. First, a researcher selects a problem domain, such as visual pattern recognition or language processing, based on his or her theoretical, empirical, or applied interests. Next, a network architecture is designed for learning tasks from the application domain. This architecture forms the skeletal structure of the network: the number of units used, their organization into layers or modules, the connections between them, and other structural parameters. Third, given a network with this architecture and some chosen task, a gradient descent learning algorithm such as error back-propagation trains the network by converging on appropriate connection weights. Finally, the researcher evaluates the trained network according to objective performance measures such as ability to solve the specified task, speed of learning, and generalization ability. This whole process can be repeated until the desired results are obtained.

Although reasonable methods exist for executing the other three stages, the network design stage remains something of a black art. Few rigorously established design principles exist, so the researcher must depend on personal experience with previous designs and on the informal heuristics of the neural network research community (e.g. 'the harder the problem, the more hidden units you need'). To circumvent the problems associated with intuitive network design by humans, this paper presents an automated evolutionary design method based on genetic algorithms.

1.2 Reasons for Automating Network Design

Designing neural networks is hard for humans. Even small networks can behave in ways that defy comprehension; large, multi-layer, nonlinear networks can be downright mystifying. Many of the basic principles governing information processing in such networks (such as parallel constraint satisfaction and distributed representation) are hard to understand and even harder to exploit in trying to design useful new network architectures. As a result, most neural network research employs only a few standard architecture types, e.g. layered feed-forward designs or simple recurrent schemes. Those seeking radically new architectures cast off into uncharted darkness.

Standard engineering design techniques founder on neural networks. The complex distributed interaction among network units usually makes even the divide-and-conquer technique of modular design inapplicable. Furthermore, this complexity seems to preclude concocting direct analytic design methods.

The prospects get even dimmer. Even if we find a design sufficient for a particular task, how can we be certain that we didn't miss a much-preferable solution? How can we optimize network designs given complex combinations of performance criteria, such as learning speed, compactness, generalization ability, and noise-resistance? And how can we determine the proper design modifications required to effect some desired change in network function?

At present, none of these questions have principled answers. The only way to negotiate these dilemmas has been to throw large amounts of human time and effort at them. As network applications continue to grow in number, size, and complexity, this human-engineering approach must begin to break down. The problem of network design requires a more efficient, automated solution.

1.3 Reasons for Using Genetic Search

The problem of network design comes down to searching for an architecture which performs best on some specified task according to some explicit performance criteria. This process in turn can be viewed as searching the surface defined by levels of trained network performance above the space of possible neural network architectures. Since the number of possible units and connections is unbounded, the surface is infinitely large. Since changes in the number of units or connections must be discrete, and can have a discontinuous effect on the network's performance, the surface is undifferentiable. The mapping from network design to network performance after learning is indirect, strongly epistatic, and dependent on initial conditions (e.g. random starting weights), so the surface is complex and noisy. Structurally similar networks can show very different information processing capabilities, so the surface is deceptive; conversely, structurally dissimilar networks can show very similar capabilities, so the surface is multimodal. We seek an automated method for searching this vast, undifferentiable, epistatic, complex, noisy, deceptive, multimodal surface.

Enumerative search methods are sure to bog down in the combinatorially explosive space of network architectures. Random search methods are no better than enumerative methods in the long run, so are equally unlikely to find useful designs. Gradient descent search methods will also fail because they require a differentiable surface with well-defined slopes, and because they are poor at searching complex, deceptive surfaces with many local minima. Heuristic knowledge-guided search by a human designer is, for reasons discussed earlier, likely to be inefficient, misdirected, slow, and costly.

Holland's (1975) schema theorem indicated the general utility of genetic search for large, complex, deceptive problem spaces. Thus, in contrast to the above search techniques, genetic algorithms might allow fast, robust evolution of genotypes specifying useful network architectures. Therefore, we propose using genetic algorithms as the appropriate evolutionary search technique to automate neural network design.

1.4 Previous Evolutionary Approaches

Evolutionary design of cognitive systems has had a long and sporadic history. One early method called *evolutionary programming* (Fogel, Owens, and Walsh, 1966) attempted to evolve finite state machines that would predict the next state of a world given previously witnessed states, using a mutation operator yielding a nonregressive random walk search. More recently, another mutational approach has been applied to individual systems learning in simple environments (Dress, 1987). Mutation and fitness-based reproduction in competing populations of automata have also been explored (Bergman and Kerszberg, 1987).

Genetic algorithms have been applied to neural networks recently in two main ways. First, there have been attempts to use genetic search instead of learning to find appropriate connection weights in fixed architectures. For example, Miller (1988), Whitley and Hanson (1989), and Montana and Davis (1989) compared genetic search to gradient-descent learning for particular network designs and problem domains, but the results have been ambiguous. Alternatively, genetic algorithms have been used to find network architectures themselves, which are then trained and evaluated using some learning procedure. Guha, Harp, and Samad (1988; also these proceedings) used an architecture representation based on groups of units with probabilistic projections between them. Todd (1988) introduced the approach described here.

2 The Genetic Algorithm: Overview

Our method of automating neural network architecture design combines two adaptive processes: genetic search through the network architecture space, and backpropagation learning in individual networks to evaluate the selected architectures. Thus, in our method, as in real biological systems, cycles of learning in individuals are nested within cycles of evolution in populations. Each learning cycle presents an individual neural network--an instantiation of a particular network architecture--with the set of input-output pairs defining the task. The backpropagation learning algorithm then compares the network's actual outputs with the desired outputs, and modifies the network's connection weights so that it performs the desired input/output mapping task more accurately. Each evolution cycle processes one population of network designs according to their associated fitness values (computed during the learning cycles) to yield an offspring population of more highly adapted network designs.

2.1 Network Representation Scheme

Different network representation strategies can be categorized according to their degree of *developmental specification*: the specificity of the mapping from genotype to phenotype. Weak specification representation schemes use relatively abstract genetic 'blueprints' that must be translated through some 'developmental machinery' to yield a network phenotype (e.g. Guha, Harp, & Samad, 1988). Such schemes may be good at capturing the architectural regularities of large networks rather efficiently. However, they necessarily involve either severe constraints on the network search space, or stochastic specification of individual con-

nections. For example, a weak specification scheme could represent whole network layers in single genes, facilitating the recombination and evaluation of large, highly regular networks, but precluding detailed connection design.

Strong specification schemes, which interpret genes more directly as individual connections, are good at capturing the connectivity patterns within smaller networks very precisely and deterministically. Such a scheme could facilitate the rapid evolution of finely optimized, compact architectures. A variety of moderate specification schemes are also possible.

We chose a strong specification scheme to gain greater resistance to human design biases for crisply articulated network layers, localist representations, and easily interpretable processing strategies, all of which can creep into weak specification schemes. A strong specification scheme may facilitate the rapid generation and optimization of tightly pruned, interesting designs that no one has hit upon before. We hope that the inspection of such streamlined designs will hone our intuitions about what weak specification schemes might work well for larger network designs.

In our particular strong specification scheme (Figure 1), we represent the architecture of a network of N units by a *connectivity constraint matrix*, C, of dimension Nx(N+1). The first N columns of matrix C specify the constraints on the connections between the N units, while the final (N+1) column contains the constraints for the threshold biases of each unit. (A bias can be thought of as a connection to an extra unit which is always "on" with a value of 1.0.)

Each entry $C_{i,j}$ in the matrix C is a member of the *connectivity constraint set*, S, and indicates the nature of the constraint on the connection from unit j to unit i (or on unit i's bias if j=N+1). Thus, column i of C represents the constraints on the the fan-out of connections from unit i. Similarly, row j represents the constraints on the fan-in of connections to unit j.

The elements in S specify different types of connection constraints: for instance, connection weights could be fixed at some constant numeric value (indicated by a specific value entered in C), learnable to any value (indicated by an L), learnable but restricted to positive values (L^+), or learnable but restricted to negative values (L^-). In our current implementation, S has only two elements: 0 (fixed at zero: no connection) and L (learnable). An example of a constraint matrix C for a network with 5 units (with 2 input units and a single output unit) is shown in Figure 1, along with the actual corresponding architecture.

We convert the network architectures, specified here in connection constraint matrix form, to a bit-string genotype as shown in Figure 1. Since the constraint set S has been restricted for now to only two values (0 and L), each entry in C is coded by a single bit. Successive rows in C are then concatenated to form a bit-string of length N*(N+1), which enters the genotype population to be processed by the genetic operators.

2.2 Genetic Operators

In our current implementation the crossover operator selects a random row number i from 1 to N and swaps all the entries in that row of C between two parents. We use this form of crossover because it is easy to implement in the bit-string genotype, and because each row of C specifies a functional building block composed of a single unit. Each such building block incorporates both the unit's bias and its fan-in connections (i.e. its 'receptive field'), thereby specifying all the information that unit receives, and thus what sort of function it can compute.

We are currently exploring other forms of crossover, such as swapping multiple rows, single or multiple columns, cross-shaped regions, or rectangular submatrices of C. The main goal is to develop crossover operators which work with our strong specification scheme to swap functionally cohesive portions of network structure. Note that our specification scheme ensures that all of these crossover operators are well-defined, always producing valid new network architectures. This eliminates the extra offspring-checking procedures required by some weak specification schemes.

Our standard mutation operator goes through each entry in C and randomly chooses a new constraint (0 or L) with some (low) specified probability. Note that any given matrix C can be transformed to any other C by a series of mutations. Thus, mutation alone suffices to explore the space of C matrices, ensuring that our genetic search can cover the entire N-unit network architecture space.

Finally, our system also uses fitness-proportionate reproduction, including Grefenstette's (1987) fitness scaling routine, based on the fitness evaluation procedure described next.

2.3 Fitness Evaluation

Performance measures for evaluating the fitness of alternate network architectures should be chosen carefully to reflect the design criteria judged important for a given network ap-

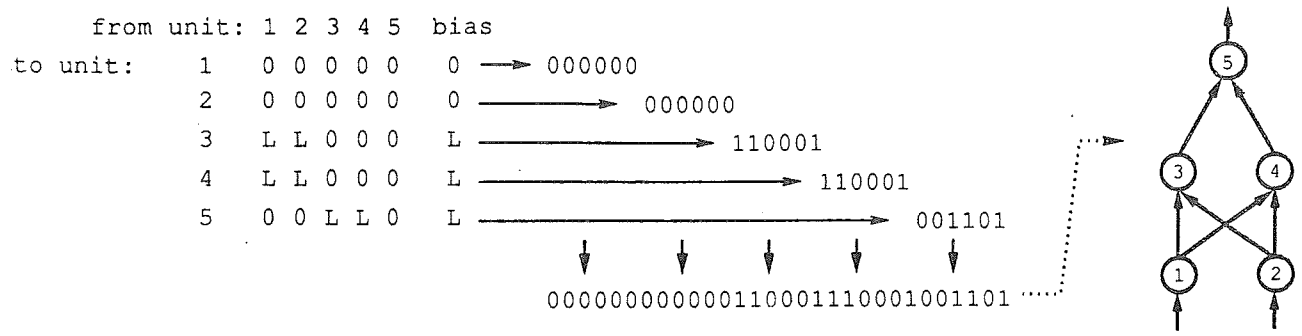

Figure 1. The conversion process from connectivity constraint matrix C at left, to bit-string genotype, center, to network architecture phenotype, right.

plication. Our major criterion has been ability to successfully learn the input-output mappings specified by each task. A more complex weighted fitness function could include further criteria, such as generalization ability or number of connections used in the network.

Our current method for evaluating a particular architecture's fitness proceeds as follows. First, a particular instantiation of the architecture is created with learnable connections where the matrix C has an entry L, and no connections where the entry is 0. Learnable connections are initialized with small random weights. (We currently ignore any connections to input units, and any feedback connections specified in the genotype, since including these would complicate the learning process considerably. Thus for now we restrict ourselves to feedforward networks.)

This initial network instantiation is then trained for a certain number of epochs (cycles through all the training pairs for the current task) using the back-propagation learning rule. The total sum squared error (TSSE) of each network's performance at the last epoch ultimately determines the network's fitness. Since low TSSEs correspond to better learning of the task at hand, raw fitness is computed by subtracting the actual TSSE from a constant TSSE representing chance performance on the given task.

3 Empirical Method

We merged Grefenstette's (1987) genetic algorithm system, *GENESIS 4.5*, with Rumelhart and McClelland's (1988) back-propagation system for training multilayer networks, *bp*, to yield a system called *Innervator* for evolving neural network architectures. Innervator was named for the innovative, evolutionarily constrained process of *innervation* whereby neural connectivity patterns develop in individual biological nervous systems.

All empirical studies were run on Sun 4/260 workstations. Typical runs to train and evaluate a single generation of 50 5-unit network architectures (such as in the XOR problem to be described shortly), with 1000 learning epochs of 4 training patterns each per network, required about 10 minutes of computer time. Thus, it often took less than an hour to evolve successful new network architectures for the relatively small problem domains we investigated.

Unless otherwise noted, all Innervator system runs described here used a population size of 50, a crossover rate of 0.6, and a bitwise mutation rate of 0.005. The "elitist" reproduction strategy was used, ensuring that the best individual in a given generation was kept in the next. Because the network training evaluation process is noisy (depending on the initial random weights of each architecture instantiation), each architecture was evaluated every time it appeared in the population, rather than carrying over a previously-found fitness value. Finally, network training parameters must vary according to the application task, so they are described separately for each task in the following sections.

4 Empirical Results

Since our strong architectural specification scheme is particularly suited for evolving highly specialized network structures with exact patterns of connections, we began applying Innervator to some relatively small tasks where such structural precision should be useful. Our goal was to see whether our genetic algorithm could discover successful architectural solutions for each task, and whether the population would converge to these solutions. Our results were as follows.

4.1 Run 1: XOR Problem

The much-studied exclusive-OR (XOR) Boolean function served as our first application task. The XOR function maps two binary inputs to a single binary output as follows: 00→0, 01→1, 10→1, 11→0. This is the simplest Boolean function which is not linearly separable--i.e. cannot be solved by a simple mapping directly from the inputs to the output--and so requires the use of extra "hidden units" to learn the task (Minsky and Papert, 1969). The standard network architectures for learning the XOR problem contain 4 or 5 units (see Figure 2a,b). Since we were curious which known or novel compact architectures Innervator might discover, we set it to search for XOR architectures with 5 or fewer units.

For the network training portion of the architecture evaluation, we used the following parameters: the learning rate was set at 0.4, the momentum (used to smooth out learning) was set at 0.8, and the number of training epochs (cycles through all 4 training patterns) was set at 1000. These values were chosen because they were found sufficient for properly-designed XOR architectures to learn the mapping.

The results were encouraging. In the initial population, the best network architecture achieved a fitness of 0.75 (out of a maximum of 1.0), indicating that it was not sufficient to fully solve the XOR task. After one generation, though, the maximum fitness rose to 0.99--thus just a single generation of reproduction, crossover, and mutation sufficed to discover an architecture which accurately learned the XOR task. Within 10 generations (500 evaluations), the population converged on a single successful architecture.

Populations in several different runs converged upon the network architecture shown in Figure 2c. It is essentially the standard 5-unit XOR network architecture, but with an extra connection from one input unit to the output unit. After training, all of the connections in this architecture are used (i.e. have substantial weights). The extra connection from input to output may allow faster weight adjustment and symmetry-breaking, leading to faster learning of the XOR task.

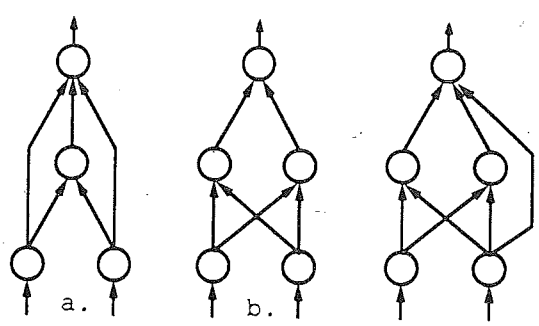

Figure 2. Architectural solutions for the XOR problem. a. Standard 4-unit architecture. b. Standard 5-unit architecture. c. Typical discovered 5-unit architecture.

4.2 Run 2: Four-Quadrant Problem

Our second application was a more advanced version of the XOR problem known as the two-dimensional XOR or four-quadrant problem. As in the simple XOR problem, two inputs are mapped to a single binary output, but here the inputs are both real numbers ranging from 0.0 to 1.0 inclusive. The two inputs can be considered the x and y coordinates of a point in the unit square. The mapping to be learned is from all points in the lower left and upper right quarters of this square to an output of 0.0, and from all points in the other two quarters to 1.0, as shown in Figure 3a.

Two types of architectural solutions to the four-quadrant problem have been discussed. First, a one-hidden-layer feedforward network can solve this problem to arbitrary precision, given enough units in the hidden layer (Figure 3b). But with two hidden layers, a second, smaller solution becomes possible (Figure 3c). The mapping can be learned with just two units in the first hidden layer, which convert all real inputs to 0's and 1's, and two units in the second, which compute the standard XOR function on these converted values. We hoped that, when Innervator was constrained to search for architectures with 7 or fewer units, it would discover the more accurate (and in some ways more complex) two-hidden-layer solution.

Networks in this run were trained for 200 epochs, with each epoch cycling through a fixed set of 500 randomly generated x-y input pairs. A lower learning rate of 0.05 was used for this more complex task, with a higher momentum of 0.9.

Some successful architectures appeared in the initial population. Other solutions appeared in successive generations, and the population as a whole slowly increased in average performance over 10 generations. The architectures discovered, though, were not what we expected. Rather than clean three- or four-layer structures, these networks tended to be asymmetric, unlayered, and somewhat convoluted. (e.g. Figure 3d). As a result, it is quite difficult to analyze exactly how such networks perform the four-quadrant mapping. One common feature of these evolved structures, though, as for Innervator's XOR network architecture presented earlier, is the presence of direct connections from the input units to the output unit. Again we believe that such connections, by partly increasing the directness of the mapping computed, help to speed up learning in these networks.

Innervator's discovery of unexpected, difficult-to-interpret architectures can be considered a validation of our approach, rather than a drawback. The layered architectures previously proposed for the four-quadrant problem conform to the biases of the humans who designed them; by removing these biases from Innervator, new, and perhaps better, architectures were found.

4.3 Run 3: Pattern Copying

The fact that Innervator found architectural solutions to the previous problems in very few generations suggests that genetic recombination may not have played a very important role in its search. To more rigorously test Innervator's search ability (specifically, our row-wise crossover operator), we applied it to a task where mutation alone would take many generations to find a solution architecture. The task we selected is simple pattern copying, where the binary pattern presented to the input units must be copied exactly to the output units. All that this copying requires is a connection directly from input 1 to output 1, from input 2 to output 2, and so on, for as many inputs and outputs as there are in the network. No hidden units are needed for this task.

We used 10 input units and 10 output units in this experiment, thereby requiring that 10 direct connections be present in the network architecture for the mapping to be properly learned. Since each entry in the connectivity constraint matrix C can be either 0 or L, there is a 0.5 chance that any given connection will be learnable (have constraint L) in each randomly generated architecture of the initial population. The chance that all 10 necessary connections will appear in a single architecture is 0.5^10, or 1/1024; since the initial population contains 50 randomly generated architectures, the overall chance that an appropriate one will appear there is 50*1/1024, or about 1 in 20. By extension, mutation alone would take 20 generations on average to find a solution architecture.

Since pattern copying is a simple task learnable by a two-layer network architecture without encountering local minima, we can use a very high learning rate, 1.0, and a momentum of 0.9. The high learning rate in turn means that few epochs are needed. Each network in this experiment was trained for 5 epochs on all 1024 10-bit patterns.

As expected, no solutions were generated in the initial populations of several runs (e.g. see the graph of maximum

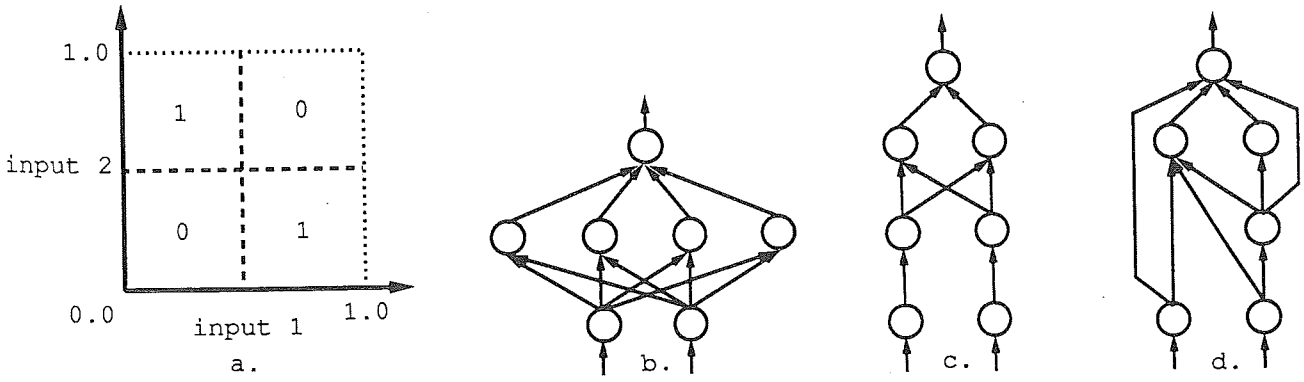

Figure 3. The four-quadrant problem. a. 2-d mapping to be learned. b. Standard 3-layer architectural solution. c. Standard 4-layer architectural solution. d. Typical discovered architectural solution.

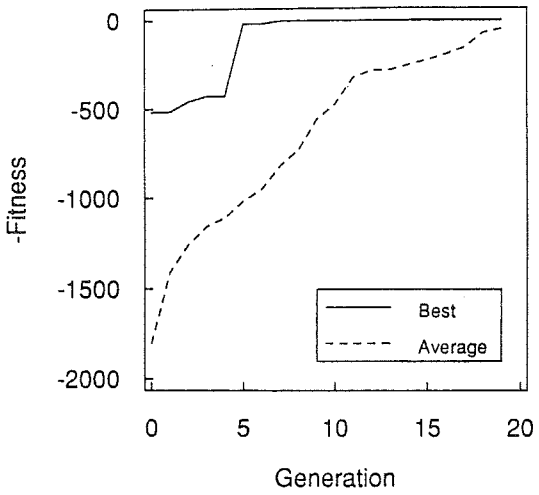

Figure 4. Innervator's genetic search performance on the pattern copying task: best and average fitnesses from an example run. Chance fitness = 2560; ceiling fitness = 0; inverse fitnesses are plotted for ease of comparison.

fitness for one example run in Figure 4). A successful solution was typically found within 6 generations, and the populations typically converged very strongly within 15 generations. This performance beats the 20-generation expected search time using random architecture generation by mutation. These results lend validation to our row-swapping crossover operator as a method for reshuffling appropriate network building blocks throughout the population. More generally, Innervator's performance on this pattern copying task suggests that it could perform powerful searches through the space of network architectures for other applications.

5 Conclusions

The *Innervator* system rapidly evolves neural network architectures capable of learning to perform simple mapping problems. We are currently testing it on larger, more difficult domains, such as position-invariant pattern recognition and visual stereoscopic depth perception. Our strong specification scheme does allow the development of highly specific, often unexpected designs. However, we believe that alternate representation schemes, genetic operators, fitness evaluation methods, and genetic algorithm parameters could yield still faster, more robust neural network evolution across a variety of domains. We are investigating a number of options, including weaker specification schemes incorporating a connectivity development stage before learning; operators that handle functional groups of units simultaneously; and fitness measures including network size costs, to help eliminate unused connections.

The results in this paper represent the early stages of a long-term research program aimed at developing a powerful, flexible computer system for simulating the adaptive processes of evolution, development, learning, and information-processing for neural networks in complex virtual environments. Such a system would have applications in biological, neurological, and psychological modelling, as well as the engineering and design applications emphasized in this report. Our immediate goal has been to free the network design process from the constraints of human biases, and to discover new forms of neural network architectures applicable to a variety of domains. The automation of network architecture search by genetic algorithms seems the best way to achieve this timely goal.

References

Bergman, A., & Kerszberg, M. (1987). Breeding intelligent automata. In *Proceedings of the IEEE First International Conference on Neural Networks*. San Diego: SOS Printing.

Dress, W.B. (1987). Darwinian optimization of synthetic neural systems. In *Proceedings of the IEEE First International Conference on Neural Networks*. San Diego: SOS Printing.

Fogel, L.J., Owens, A.J., & Walsh, M.J. (1966). *Artificial intelligence through simulated evolution*. New York: John Wiley & Sons.

Guha, A., Harp, S.A., & Samad, T. (1988). *Genetic synthesis of neural networks*. Honeywell Corporate Systems Development Division. Technical report CSDD-88-I4852-CC-1.

McClelland, J.L., & Rumelhart, D.E. (1988). *Explorations in parallel distributed processing: A handbook of models, programs, and exercises*. Cambridge, MA: MIT Press/Bradford Books.

Miller, G.F. (1988). *Evolution and learning in adaptive networks*. Psychology Department, Stanford University. Unpublished manuscript.

Minsky, M., & Papert, S. (1969). *Perceptrons*. Cambridge, MA: MIT Press.

Montana, D.J., & Davis, L. (1989). *Training feedforward neural networks using genetic algorithms*. BBN Systems and Technologies, Inc. Technical report.

Rumelhart, D.E., & McClelland, J.L. (Eds.) (1986). *Parallel distributed processing: Explorations in the microstructure of cognition*. Cambridge, MA: MIT Press/Bradford Books.

Todd, P.M. (1988). *Evolutionary methods for connectionist architectures*. Psychology Department, Stanford University. Unpublished manuscript.

Whitley, D., & Hanson, T. (1989). *The GENITOR algorithm: Using genetic recombination to optimize neural networks*. Computer Science Department, Colorado State University. Technical report.

This research was supported by National Science Foundation Graduate Fellowships. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.