Adding learning to the cellular development of neural networks: Evolution and the Baldwin effect.

Frédéric Gruau*
Départment de Recherche Fondamentale
Matiere Condensée, CEA CENG
SP2M PSC BP 85X 38041 Grenoble
France
gruau@drfmc.ceng.cea.fr

Darrell Whitley
Computer Science Department
Colorado State University
Fort Collins, CO 80523
USA
whitley@cs.colostate.edu

Abstract

A grammar tree is used to encode a cellular developmental process that can generate whole families of Boolean neural networks for computing parity and symmetry. The development process resembles biological cell division. A genetic algorithm is used to find a grammar tree that yields both architecture and weights specifying a particular neural network for solving specific Boolean functions. The current study particularly focuses on the addition of learning to the development process and the evolution of grammar trees. Three ways of adding learning to development process are explored. Two of these exploit the *Baldwin effect* by changing the fitness landscape without using Lamarckian evolution. The third strategy is Lamarckian in nature. Results for these three modes of combining learning with genetic search are compared against genetic search without learning. Our results suggest that merely using learning to change the fitness landscape can be as effective as Lamarckian strategies at improving search.

1 Combining genetic algorithms and neural networks

There is now a relatively large number of papers that deal with various combinations of genetic algorithms and neural networks. A survey of this work is given by Schaffer, Whitley and Eshelman (1992) in an introduction to the proceeding of a workshop on *Combinations of Genetic Algorithms and Neural Networks.* Genetic algorithms have been used to do weight training for supervised learning and for reinforcement learning applications. Genetic algorithms have also been used to select training data and to interpret the output behavior of neural networks. Finally, genetic algorithms have also been applied to the problem of finding

*or: Laboratoire de l'informatique du parallélisme Ecole Normale supérieure de Lyon, 46 Allée d'Italie 69007 Lyon, France

neural network architectures. An architecture specification indicates how many hidden units a network should have and how these units should be connected.

Typically the encodings for neural net architectures assume that the number of hidden units is bounded; the genetic algorithm can then be used to determine what combinations of weights or hidden units yield improved computational behavior given a finite range of architectures. The resulting networks have usually been trained using back propagation. A common fitness measurement is the sum squared error (Miller et al. 1989; Belew et al., 1990; Schaffer et al, 1990). Whitley et al. (1990) use a genetic algorithm to find network topologies that enhance learning speed; they also explore how to create selective pressure toward smaller nets and to reduce training time.

Still, the difficulty with directly optimizing a network architecture is the high cost of each evaluation, which typically involves *training* the neural network. If we must run a back-propagation algorithm (or some improved, faster form of gradient descent) for each evaluation, the cost of the numerous evaluations needed to find improved network architectures quickly becomes computationally prohibitive. The computation cost can be so high as to make genetic algorithms impractical except for optimizing small topologies. For example, if an evaluation function for a modest-sized neural network architecture on a complex problem involves one hour of computation time (which maybe unrealistically low), then it requires *one year* to do only 9,000 architecture evaluations during genetic search.

There are new directions being explored for optimizing neural network architectures. Grammar based architecture descriptions (Kitano, 1990; Mjolsness, et al., 1989; Gruau, 1992a; 1992b) may display better scaling properties. By optimizing grammars that generate network architectures instead of directly optimizing architectures, this research hopes to achieve better scalability, and in some sense, reusability of network architectures. The goal of this research, then, is to find rules for generating networks which will be useful for defining architectures for some general class of problems. In particular, this would allow developers to define grammar based architectures on smaller problems and then use these architecture descriptions as building blocks when attacking larger problems.

In the current study the genetic algorithm is used to find both the Boolean weights and the architecture for neural networks that learn Boolean functions. A *grammar tree* is used which is both more compact and flexible than direct codings or previously developed grammatical representations. The grammar tree is used to encode a cellular developmental process that more closely resembles biological cell division than other neural network encodings. This representation, referred to as a *cellular encoding*, has been described by Gruau (Gruau, 1992a; 1992b).

The current study particularly focuses on the addition of learning to the development process and the evolution of grammar trees. Learning is used to speed up the search performed by the genetic algorithm. We compare various ways of combining learning and the genetic algorithm. Specifically, the following four modes of learning are explored: 1) using a basic genetic algorithm without learning, 2) using simple learning that merely changes the fitness function by making weight changes on small networks, 3) using a more complex form of learning that changes weights on small networks, then "reuses" these weight changes in the development of larger nets, and 4) using a form of Lamarckian evolution where learned behavior is coded back onto the chromosome of an individual so that it is passed along to its offspring during genetic reproduction.

It may seems odd that strategy 2 and 3 could potentially speed up genetic search. Both strategies exploit the *Baldwin effect* (Hinton and Nolan, 1987; Ackley and Littman, 1991; Baldwin, 1896; Belew, 1989). As indicated, learning is used to change the fitness of an individual without actually changing the genetic code. However, any form of learning that "improves" on inherited traits also changes the fitness landscape associated with the objective function so as to reward solutions that are close to a more highly fit solution found via learning (Hinton and Nolan, 1987).

Section 2 of this paper reviews the literature on grammar based descriptions of neural networks and also presents the details of the cellular encoding and development scheme developed by Gruau (1992a). It also describes the genetic algorithm used in these experiments, and the *genetic programming* paradigm developed by Koza (Koza and Rice, 1991; Koza, 1992) which is used to generate and recombine grammar trees. Section 3 discusses the role of learning in evolution and the various forms of learning and combined learning-evolutionary search strategies to be used in these experiments. Section 4 described the experiments and results.

2 Background

Grammars such as Lindenmayer systems, or L-systems, were developed to provide both a model and a mathematical theory of plant development. These recursive grammars are in effect context free grammars, except that in L-systems the rewriting rules are applied in parallel so as to simultaneously replace multiple occurrences of the nonterminal symbols in a particular string. This use of parallel rewrites is designed to mimic biological cell divisions where several cells may be undergoing similar division processes in parallel. It has also been shown that context free L-Systems can generate languages that cannot be generated by normal sequential context-free grammars. By applying the rewrite rules of an L-system grammar in different ways, or by stochastically applying rewrite rules, one can generate surprisingly complex structures that have a remarkable resemblance to different plant structures (Prusinkiewicz and Lindenmayer, 1992).

When optimizing neural network architectures we would like to develop grammatical descriptions of neural networks that have some of the properties of L-systems. In particular, the same set of rules might be used to generate an entire family of related networks.

Kitano (1990) uses a grammar to generate a family of matrices of size 2^k. The elements of the matrix are characters in a finite alphabet. In order to develop matrix M_{k+1} each character of the matrix M_k is replaced by a 2×2 matrix. A predefined convention allows one to deduce a connectivity matrix from each matrix M_k of the family. This connectivity matrix describes the architecture of a neural net. In a more recent study, Kitano (1992) shows that it is also possible to deduce weight values from M_k.

One drawback with Kitano's representation scheme is that an $m \times m$ matrix must be developed for a network of n neurons, where m is the smallest power of 2 bigger than n.

A smaller $n \times n$ matrix is then extracted from the larger $m \times m$ matrix for developing the neural network. In the worst case, $n = m/2 + 1$ and one uses only $(0.5 - 1/m)^2$ of the bits (approximately 25%). In order to get an acyclic graph for a feed forward network, one must consider only the upper right triangle, which further decreases the efficiency of the encoding.

Mjolness et al. (1989) defines a recursive equation for a matrix from which he computes a family of integer matrices and then a family of weighted neural nets; in this sense, the approach is similar to that used by Kitano. The search space is defined over the set of equation coefficients. Mjolness uses simulated annealing instead of the genetic algorithm to search this space. Unlike Kitano, Mjolness is not restricted to only using matrices of size 2^k. However, the size of the neural networks has to be determined by hand.

Gruau (1992a) directly develops a family of neural nets and avoids the need for the matrix representation. Instead, development rules are applied to the cells of a network. Each cell has a copy of a chromosome that encodes the development process. Each cell reads the chromosome at a different position. Depending on what it reads, a cell can divide, change some internal parameters, and finally become a neuron. The resulting language can describe networks in a more elegant and compact way, and the representation can be readily recombined by the genetic algorithm. Various properties of cellular encoding have been formalized and proved by Gruau (1992b). Gruau used a genetic algorithm to recombine grammar trees representing cellular encodings and showed that neural networks for the parity problem and symmetry problem could be found. Furthermore, the grammar trees are recursive encodings that generate whole families of networks that compute parity or symmetry. In this way, once small parity or symmetry problems are solved, the grammars can generate solutions to parity and symmetry problems of arbitrary size.

2.1 Review of cellular development

The chromosome is represented as a *grammar tree* with ordered branches whose nodes are labeled with character symbols. A cell is a node of an oriented *network graph* with ordered connections. Each cell carries a duplicate copy of the chromosome (i.e., the grammar tree) and has an internal reading head that reads from the grammar tree; typically, each cell reads from the grammar tree at a different position. The character symbols represent instructions for cell development that act on the cell or on connections that fan-in to the cell. During a step of the development process, a cell executes the instruction referenced by the symbol it reads, and moves its reading head down in the tree. One can draw an analogy between a cell and a Turing machine. The cell reads from a tree instead of a tape and the cell is capable of duplicating itself; but both execute instructions by moving the reading head in a manner consistent with the symbol that is read. The grammar tree thus functions as a "program" and each character as a "program-symbol."

Development starts with a single cell called the *ancestor cell* connected to an input pointer cell and an output pointer cell. Consider the starting network and the encoding depicted in figure 1. The ancestor cell initially possesses a reading head positioned on the root of the tree. Its registers are initialized with default values (e.g., threshold > 0). As this cell repeatedly divides it gives birth to all the other cells that will eventually make up the neural network.

The two input/output pointer cells to which the ancestor is linked (indicated by square boxes in the figure) do not execute any program-symbol. Rather, at the end of the development process, the upper pointer cell is connected to the set of input units, while the lower pointer cell is connected to the set of output units. These input and output units are created during development; they are not added independently at the end. After development is complete the pointer cells can be deleted. In figure 1, the final decoded neural net has two input units labeled 1 and 2, and one output unit labeled 1.

A cell also manages a set of internal registers, some of which are used during development, while others determine the weights and thresholds of the final neural net. The contents of the link register is a pointer to one of possibly several fan-in connections (i.e., links) into a cell. The following are a list of the program-symbols used in the cellular encoding.

- A *division program-symbol* creates two cells from one. In a *sequential division* (denoted by **S**) the first child inherits the input links, the second inherits the output links of the parent cell and the first child connects to the second with weight 1. The link is oriented from the first child to the second child. This is illustrated in steps 1 and 3 of figure 1. In a *parallel division* (denoted by **P**) both child cells inherit the input and output links from the parent cell (in step 2 and step 6). Since there are two child cells, a division program-symbol must label nodes of arity two. The first child moves its reading head to the left subtree and the second child moves its reading head to the right subtree. Finally, when a cell divides, the values of the internal registers of the parent cell are recopied in the child cells.

- A *value program-symbol* modifies the value of an internal register of the cell. The program-symbol **I** increments (and **D** decrements) the value of the link register, which is a *pointer* to a specific fan-in connection or "link." The link register has a default initial value of 1, thus pointing to the leftmost fan-in link. Incrementing the link register moves the pointer to the right, but does not directly modify a connection. Modifications to specific connections are accomplished by first resetting the link register.

- An unary program-symbol **C** modifies the neural network topology by removing (i.e., "cutting") the link which is pointed to by the link register. The program-symbol denoted + sets the weight of the input link pointed by the link register to 1, while - sets the weight to -1 (see step 7).

  The program-symbols **C**, +, and - do not explicitly indicate to which fan-in connection the corresponding instructions are applied. When **C**, + or - is executed it is applied to the link pointed to by the link register.

- The program-symbol **A** increments (and **O** decrements) the threshold.

- The waiting program-symbol denoted **W** makes the cell wait for its next rewriting step. In some cases, the final configuration of the network depends on the order in which cells execute their corresponding instructions. For example, in figure 1, performing step 7 before step 6 would produce a neural net with an output unit having two negative weights instead of one. **W** is necessary for those cases where the development process must be controlled by generating appropriate delays.

- The *ending program-symbol* denoted **E** causes a cell to loose its reading head and become a neuron. Since the cell does not read any subtree of the current node, there need not be any subtree to the node labeled by **E**. Therefore **E** labels the leaves of the grammar tree.

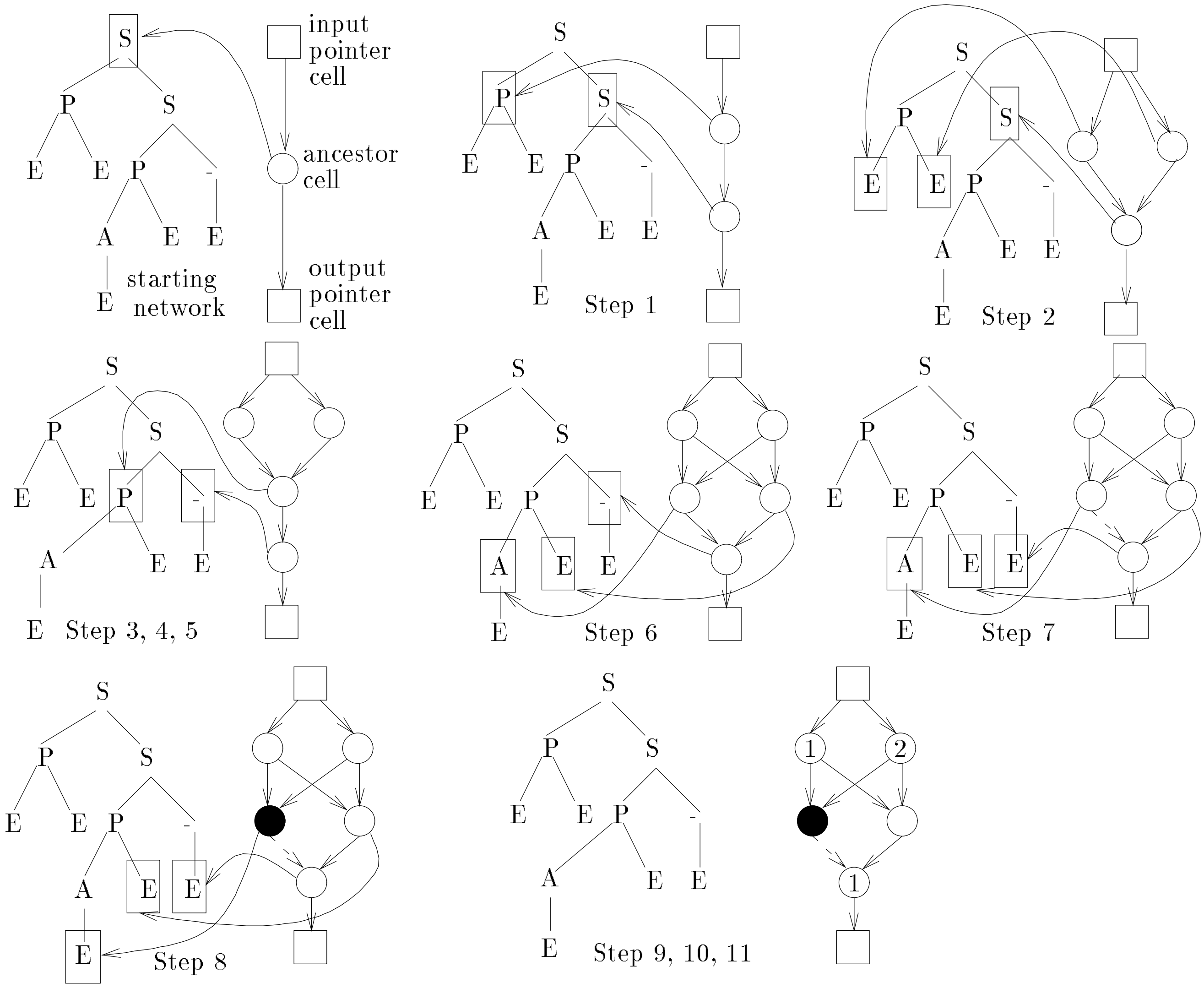

Figure 1: *The development of a XOR network. The left half of each panel represents the grammar tree, while the right half are cells in the developing network. The empty squares at the top and bottom level of the developing neural network represent input/output pointer cells. Circles represent cells or neurons. Cells that are waiting to execute a program-symbol are connected to the grammar tree by long curved arrows. These arrows point to reading heads that are represented by rectangles around program-symbols in the grammar tree. An empty circle is a cell with threshold* > 0*, while a circle filled with black indicates threshold* > 1*. A continuous line in the neural network indicates a weight* 1*, and a dashed line a weight* -1.

The sequence in which cells execute program-symbols is determined as follows: once a cell has executed its program-symbol, it enters a First In First Out (FIFO) queue. The next cell to execute is the head of the FIFO queue. If the cell divides, the child which reads the left subtree enters the FIFO queue first. This order of execution tries to model what would happen if cells were active in parallel. It ensures that a cell cannot be active twice while another cell has not been active at all.

Up to this point in our description the grammar tree does not use recursion. One is able to develop only a single neural network from a grammar tree. But one would like to develop a family of neural networks, which share the same structure. For this purpose, we introduce a recurrent program-symbol denoted **R** which allows a fixed number of loops L. The cell which reads **R** executes the following algorithm:

```
1- life := life - 1
2- If (life > 0) reading-head := root of the tree
3- Else  reading-head := subtree of the current node
```

where *life* is a register of the cell initialized with L in the ancestor cell (see figure 2).

Thus a grammar develops a family of neural networks parametrized by L. This implementation of the recurrence allows precise control of the growth process. The development is not stopped when the network size reach a predetermined limit, but when the code has been read exactly L times through. (Note that recurrence in the grammar does not imply that there is recurrence in the resulting neural network.) The number L parametrizes the structure of the neural network. We also introduce a branching program-symbol denoted **B** which labels nodes of arity 2. This program-symbol allows us to encode a family of networks where the first network has a particular structure that may be different than other networks developed from the same grammar tree. A cell that reads **B** executes the following algorithm:

```
1- If (life > 1) reading-head := left subtree of the current node
2- Else  reading-head := right subtree  of the current node
```

We stress the compactness of the encoding in figure 1 and figure 2. Four bits are enough to encode all of the 13 program-symbols discussed in this paper. The length of the encoding for a network computing the parity of arbitrary large size (including weights and thresholds) is thus only 44 bits. To give a comparison, Kitano needs 120 bits to encode the architecture of the parity of size 2 (i.e., exclusive-or) without weights or thresholds. For networks that process larger parity problems, the enclosing is increasingly larger. The encoding of the grammar as a tree instead of a set of rules is one of the keys to this compactness.

2.2 Genetic programming and cellular development

Koza has shown empirically that the tree data structure is an efficient encoding which can be effectively recombined by genetic operators. *Genetic programming* has been defined by Koza as a way of evolving computer programs with a genetic algorithm (Koza and Rice, 1991; Koza, 1992). In Koza's genetic programming paradigm the individuals in the population

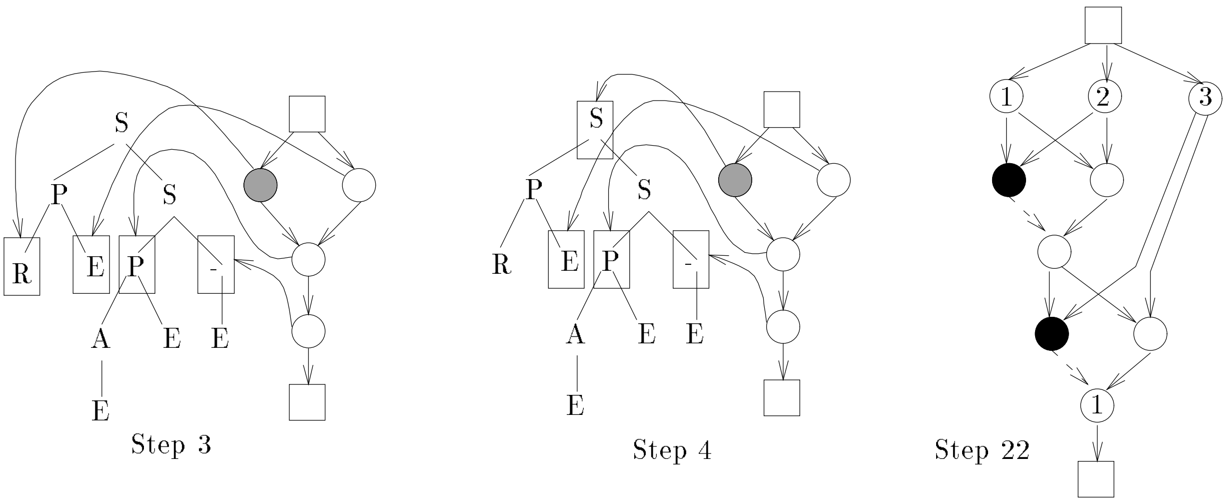

Figure 2: *A recurrent developmental program for a neural network that computes the parity for three inputs. The initial life of the ancestor is 2. The first two steps are the same as those shown in the preceding figure. The total number of steps is* 22 *since the tree is executed 2 times. The cell which executes the recurrent program-symbol is filled with gray.*

are LISP S-expressions which can be depicted graphically as rooted, point-labeled trees with ordered branches. Since our chromosomes (grammar-trees) have exactly the same structure, the same approach is used for generating and recombining grammar trees. The set of alleles is the set of cell program-symbols that label the tree, the search space is the hyperspace of all possible labeled trees. During crossover, a sub-tree is cut from one parent tree and replaces a subtree from the other parent tree; the result is an offspring tree. The subtrees which are exchanged during recombination are randomly selected. Figure 3 gives an example of cross-over.

2.2.1 A description of the genetic algorithm

Gruau (1992a) has designed a sequential genetic algorithm that uses local selection and mating based on a geographical lay out of individuals. Individuals reside on a 2-D grid and mate with the best chromosome found during a random walk in the neighborhood. This implements isolation by distance (Collins and Jefferson, 1991), such that different local regions on the grid may display local trends in terms of evolutionary development. The population size p and the number of generations g are not fixed. Rather, the genetic algorithm deduces p and g using a function that yields the expected number of generations given a population size ($g = p/10$ in our simulation) and the total allocated CPU time. The genetic algorithm is similar to a steady state model in that it uses the following strategy to decide when to create a new individual and when to delete an individual from the current population. The genetic algorithm evaluates the average time taken for the evaluation of one chromosome. It then computes the number of generations that could still be processed, keeping the current

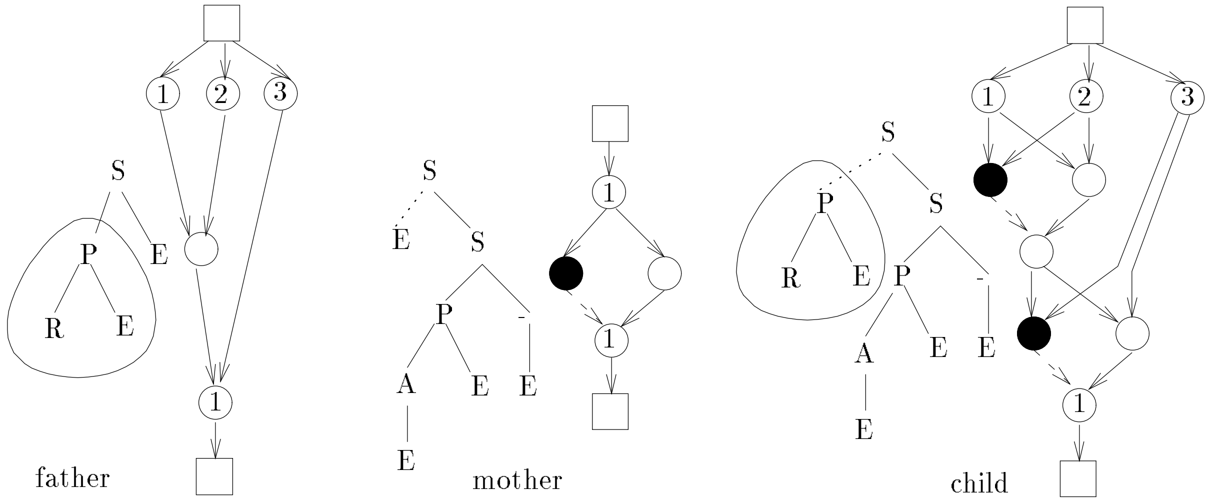

Figure 3: *An illustration of recombination between two trees. The second order neural network is developed from the resulting offspring, where the life parameter of the ancestor cell is initialized to 2. The father tree encode an OR neural net. The selected subtree to be cut is encircled. The mother tree almost encodes a XOR, except the number of input nodes is incorrect. The dotted line indicates the subtree which is to be pruned and replaced. The resulting offspring encodes a general solution for the parity problem.*

population size p. If it is higher than $p/10$ then it creates an individual, else it deletes an individual. Deletion is carried out by randomly selecting 50 individuals from the population, and removing the worse individual in this sample. When the allotted computation time has elapsed the final population size p and number of generation g will be such that $g = p/10$. Each time a new individual is created and evaluated with a time t the average evaluation time is updated using $t_{avg} = (p * t_{avg} + t)/(p+1)$. Since the average evaluation time increases as more generations of individuals are processed, the number of individuals in the population decreases. Thus, the population size can dynamically adapt to the change in the amount of time require for fitness evaluation. In the early generations, the chromosomes are quickly evaluated and a large number of solutions can be explored. In later generations, evaluation can become very expensive and only a few chromosomes stay in competition.

2.2.2 Strategic mutation

Over the course of the experiments, we noticed that one of the most difficult problems faced by the genetic algorithm was to correctly place the recurrent program-symbol **R**. In the case of the parity problem, a correct estimation of where to put a program-symbol **R** is enough to solve the recurrence, as indicated by figure 1 and figure 2. On the other hand, an incorrect placement prevents the genetic algorithm from correctly developing any of the other networks in the family of recurrent networks except the very first one. This "all or nothing" feature causes a lack of smoothness in the fitness landscape which makes placing the **R** program-symbol difficult. In order to overcome this problem we used a strategic form of mutation for the **R** program-symbol. By mutating **R** 10 times more often than other program-symbols, most **R** program-symbols are quickly removed and, in general, allele **R** is 10 times less represented than other program-symbols of arity 1. In order to offset this bias against **R**, each program-symbol of arity 1 has a probability of $4 * t_{mut}$ to mutate into a **R** program-symbol, where t_{mut} is the mutation rate of the other alleles. The combined effect of these mutations strategies is to explore a large number of different placements for **R**.

Gruau (1992a) has shown that this approach reliably generates solutions to the parity and symmetry problems. Gruau also reports that after the genetic algorithm has generated a family of recursively developed networks that handle the lower order cases (3 to 6 inputs), the recursive network encoding represents a general relation and automatically generalizes to handle arbitrarily large problems. This approach can therefore learn to solve special classes of problems (parity or symmetry for 500 inputs nodes, for example) which cannot be learned with traditional gradient based training algorithms. Note that a network with 500 binary input nodes has 2^{500} possible input patterns. It is not possible to iterate over a set of training examples which would be large enough to adequately represent the function spaces associated with these problems (c.f., Mühlenbein, 1990).

3 Adding learning to cellular development

One way to speed up the search for functional networks is to add some form of learning to the developmental process. In general, to be effective the learning process must be less costly than reproducing and testing a new individual or, if the cost of learning approaches or exceeds that of reproducing and testing a new individual, learning must provide a greater chance of resulting in improved behavior.

There are numerous ways in which learning could be added to the developmental process and several problems that must be considered. First, we are working with Boolean networks. There are several reasons why the learning algorithm would ideally result in Boolean weights. One reason to learn only Boolean weights is that the learned information could be coded back into the grammar. Coding the learned weights back into the grammar is useful if some form of Lamarckian evolution is employed. By Lamarckian evolution, we mean that chromosomes (grammar trees) produced by recombination are improved via learning and these improvements are encoded back onto the chromosome. The altered chromosomes are then placed in the genetic population and allowed to compete for reproductive opportunities.

An obvious criticism of Lamarckian evolution is that it is not Darwinian evolution: it is not the way that most scientists believe evolution actually works. Another criticism is that *if* we wish to preserve the schema processing capabilities of the genetic algorithm, then there are reasons to believe that Lamarckian evolution should not be used. Consider the simple case in which chromosomes are bit strings instead of grammar trees. Changing the coding of an offspring's bit string alters the statistical information about hyperplane subpartitions that is implicitly contained in the population (Goldberg, 1989; Whitley, 1994). Theoretically, applying local optimization to improve each offspring undermines the genetic algorithm's ability to search via hyperplane sampling. The objection to local optimization is that changing inherited information in the offspring results in a loss of inherited schemata, and thus a loss of hyperplane information.

Algorithms that employ some form of Lamarckian evolution are often referred to as *hybrid genetic algorithms*, since the genetic algorithm is hybridized some form of local optimization or learning (Davis, 1991). Hybrid algorithms that incorporate local optimizations may result in greater reliance on hill-climbing and less emphasis on hyperplane sampling. This could result in less global exploration of the search space, since it is hyperplane sampling that it the basis for the claim that genetic algorithm globally sample a search space. But hybrid genetic algorithms are not well understood analytically and, despite theoretical concerns, typically do well at optimization tasks.

3.1 The Baldwin effect

There are other ways that evolution and learning can interact that is not Lamarckian in nature. One phenomenon which has recently received new attention is the *Baldwin effect* (Hinton and Nolan, 1987; Ackley and Littman, 1991). From an implementational point of view, exploitation of the Baldwin effect requires that individuals employ some form of learning, but the acquired behavior is not coded back to the genetic encoding as in Lamarckian evolution. Instead, the learned behavior affects the objective function. Therefore fitness is a function of both inherited physical attributes directly encoded in the genes, such as network structure and inherited weights, plus learned behavior. By changing the evaluation of individual strings, learning changes the fitness landscape.

The idea that learned behavior could influence evolution was first proposed by J.M. Baldwin (1896) almost a hundred years ago. If specific learned behaviors become absolutely critical to the survival of individuals then there is selective advantage for genetically determined traits that either generally enhances learning, or which predisposes the learning of specific behaviors. At the very least, Baldwin's hypothesis indicates that learning will impact the direction of evolution. In its most extreme interpretation, the *Baldwin effect* suggests that selective pressure could make it possible for acquired, learned behavior to become genetically predisposed or even genetically determined via Darwinian evolution.

Hinton and Nolan (1987) offer the following simple example to illustrate the *Baldwin effect*. Assume that the fitness landscape for a given minimization problem is flat, with a single well representing a target solution. The Hinton and Nolan example includes the additional complication that finding the well via learning in the form of random local search occurs probabilistically. The closer an individual is to the target solution, the greater the probability that individual will find the well and thus learn the target solution. Learning in this situation builds a funnel shaped basin of attraction around the well that represents the *probability* of finding the well. In this way, the fitness landscape is transformed into a less difficult optimization problem, and genetic search is attracted toward the solution found by learning. Genetic search without learning fails to evolve a solution; this is not surprising, since the target solution is a needle in a haystack. (Hinton and Nolan describe the "well" as a spike, or needle, in a maximization problem.)

Hinton and Nolan's work illustrates the Baldwin effect, but provides little insight about

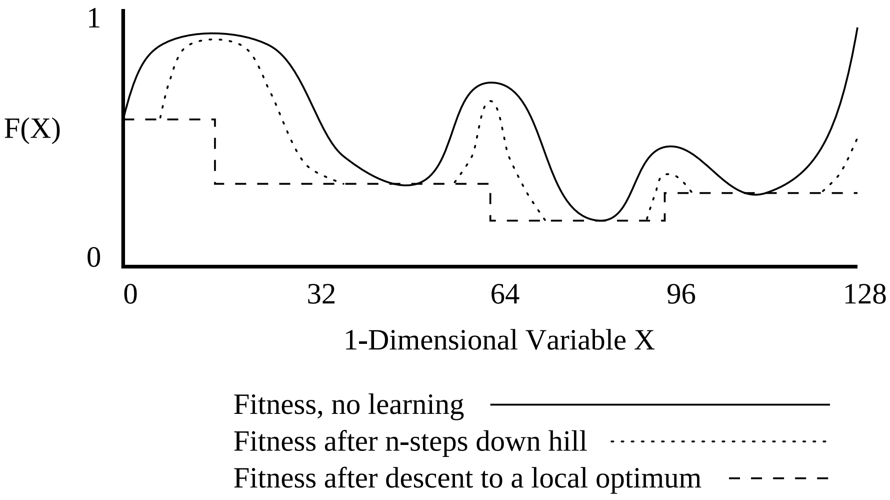

Figure 4: *The effective of learning (or local search) on the fitness landscape of a one dimensional function. When viewed as a minimization problem, improvements move downhill on the fitness landscape. This figure compares N-step local search as well as full descent to a local optimum against the fitness landscape without learning.*

whether the Baldwin effect can be intentionally exploited to accelerate evolutionary learning. Belew (1989) also offers a review and a critique of assumptions underlying the Hinton and Nolan model. In particular, the number of learning trials allocated to the individual must be sufficient to occasionally locate the well; otherwise, it remains a needle in a haystack.

Exploiting the Baldwin effect need not require finding a needle in a haystack and improvements need not be probabilistic. Figure 4 illustrates how n-steps of local optimization (e.g. learning) can in effect change the fitness landscape. Also, note that if each individual always uses learning to fully converge to a local minimum, then all strings in the basin of attraction of that local minimum have the same evaluation. Changing the fitness landscape in this way has strong implications for changing the fitness values associated with hyperplane partitions in the search space.

In the current context we would like to exploit some form of learning that can correct the specification of a network made by the grammar tree corresponding to a particular cellular encoding. We wish to test the hypothesis that learning which changes the value returned by the original evaluation function, but which does not alter the genetic code will result in a genetic search that more quickly finds networks to solve the parity and symmetry problems.

The simplest way to exploit the Baldwin effect in the current study is to use some form of learning to change the weights in the networks so as to reduce errors while processing training data. Note that cellular encoding is used to develop whole families of neural networks. Because learning is costly, in the simplest case we only use learning during iteration $L = 1$. (Recall that L initializes the `life` variable and $L = 1$ results in one iteration of development with no recursive development.) This will give us a simple form of what we will refer to as *Baldwin learning.* We will also contrast this general strategy for combining learning and genetic search with Lamarckian evolution, where the effects of learning are coded back onto the chromosome and passed on to offspring during reproduction.

3.2 Developmental learning

There is still yet another way learning can be used in the current context that is intermediate between Lamarckian evolution and directly learning weights. We will refer to this intermediate form of learning as *developmental Baldwin learning*, or more concisely, *developmental learning*. This form of learning exploits a unique characteristic of the recursive encoding used by the cellular encoding method. Let N_L be the neural network developed with `life` initialized by L. Since the encoding allows for recursive moves in the grammar tree, learning on Network N_1 which occurs for $L = 1$ can be exploited for the development of networks N_L for values of L greater than 1.

Developmental learning as defined is not Lamarckian in the sense that the learning is coded back onto the chromosome for developmental purposes, but subsequent offsprings that are produced by that individual do not carry the acquired behavioral changes in their genotype encodings. What is actually happening is that the genetic code is used to store information which is learned during the "lifetime" of the individual as development of different networks occurs. However, this learned information is not directly passed on to offspring during reproduction.

It should also be stressed that *developmental learning* works, if it works, because it is also exploiting the *Baldwin effect*. Thus, developmental learning is just a more effective way of learning and a more effective way of exploiting behaviors acquired during the lifetime *and development* of the individual.

For both developmental learning and Lamarckian evolution, training can actively take place for iteration $L = 1$ with minimal cost because network N_1 is generally small. Training only modifies a few weights in our experiments; therefore, only a few steps are being taken in the function space. (Consult figure 4 again for the general implications of n-step learning.) The learned information can be coded back into the grammar, so that the modified grammar generates the modified weights. The modified grammar can then be used to develop larger nets N_L where $L > 1$. The only difference in these two forms of search, then, is that in developmental learning offspring *do not* inherit the acquired behavior while in Lamarckian evolution offspring *do* inherit the acquired behavior. The evaluations of the offspring are otherwise the same. Therefore, it is important to compare the form of Lamarckian evolution used in this paper specifically against developmental learning, and not directly against the simpler form of Baldwin learning.

Figure 5 illustrates how the grammar can successfully register a weight modification for $L = 1$, and replicate it for $L = 2$. The network N_2 represented in figure 5(e) has two weights set to -1. In this example, the network N_L would have L weights set to -1. The method used for back-coding is explain in section 3.5.

The exploitation of recursive development works in the case of problems such as parity or symmetry because these problems have a regular recursive solution. Grammar attributes learned for N_1 can be successfully reused in larger networks. At this point it is not clear if many functions have a compact recursive definition where this strategy can be applied. However, in general, exploiting the Baldwin effect does not rely on any features specific to cellular development.

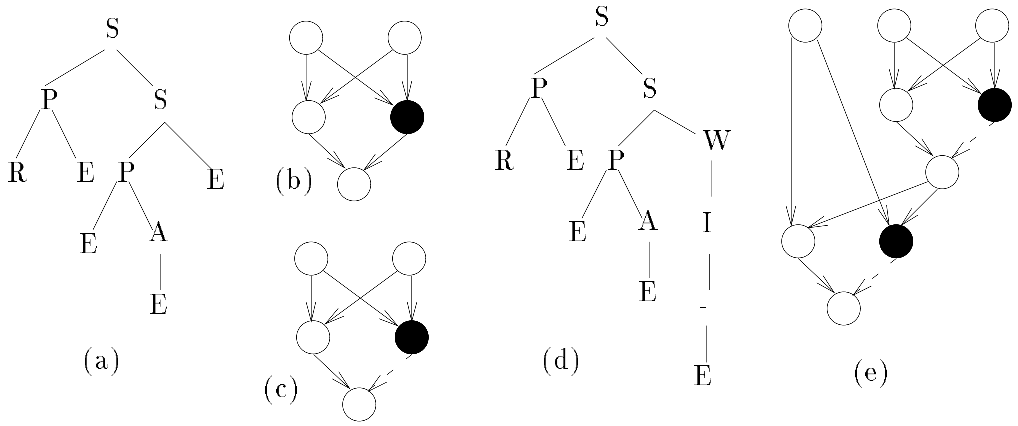

Figure 5: *The method for backcoding the chromosome. (a) Chromosome produced by the genetic algorithm. (b) Development of neural net N_1. (c) After training N_1, a weight feeding the output unit has been modified to -1 (dashed line). (d) The modified chromosome. (e) Development of neural network N_2 using the modified chromosome.*

3.3 Comparative goals

Our goals, then, are to compare genetic search with no learning to simple Baldwin learning where weights are directly learned and used to change the string evaluation for N_1. We also will directly compare genetic search without learning to Lamarckian evolution and to developmental Baldwin learning. We stress that simple *Baldwin learning* should not be directly compared again developmental learning or Lamarckian evolution, since these approaches use a more complex form of learning. However, it is interesting that developmental learning in some sense uses the same weights as those learned by the simple form of Baldwin learning, but it exploits these weight more effectively.

For Lamarckian evolution and developmental learning it is necessary to recode the learned behavior back onto the chromosome. Since cellular encoding specifies a Boolean network, the learning algorithm we use should also result in a Boolean network. In addition, we would like the learning process to be very inexpensive. This suggests the use of a fast, relatively minimal form of learning. More specifically, an ideal learning algorithm for the set of experiments we have outlined would be a one-pass learning algorithm that returns one or two minor changes in the network that potentially improve performance. Given the requirements we have outlined, we choose to use a simple variant of Hebbian learning. Because of the limitations of Hebbian learning, explicit training will be restricted to connections that feed into output units.

3.4 Hebbian learning on output units

Assume that we have a Boolean neural network with one hidden layer where it is known that the first layer of connections have correct weights and hidden units have correct thresholds. Clearly, this implies that the problem has been reduced to learning a linearly separable function with respect to the hidden units (McClelland and Rumelhart, 1988). Perceptron learning might be used at this point, except this algorithm does not result in a Boolean net and it is not a one-pass learning algorithm. If we limit learning to a single pass, however, we can look at the activations of both hidden nodes and output nodes and collect information about whether the connection should be inhibitory or excitatory between any particular hidden unit and output unit. That is, for each training pattern and each pair of hidden-output units, indicate the weight should be positive (i.e., +1) if both the hidden unit and output unit is "on" and indicate the weight should be negative (i.e., -1) if the hidden unit is on and the output unit is off. (We do not consider cases where the hidden unit is off, since these cases do not effect the activation of the output unit in a feed-forward network.)

This mode of updating weights follows the same principle as Hebb's rule: increase the strength of the connection between neurons that are simultaneously active and move toward a negative, inhibitory weight between neurons that are not simultaneously active (McClelland, et al. 1986).

For the parity and symmetry applications the neural nets have a single output unit. We used the following algorithm to learn new weights for the links that feed into the output units of the neural net. Each training pattern in presented to the input units. The activation level of each hidden neuron is computed, until all hidden units are processed. The desired output value is then clamped to the output unit. For each link l feeding the output unit, the following Hebbian information is computed. Each link l has a variable d_l. If the activities of the units linked through l are the same, d_l is incremented; otherwise d_l is decremented. The variable d_l is thus a correlational measure of the activities of the neurons linked through l.

Before learning, the variables for each d_l are initialized to zero. Note that this approach is somewhat different from perceptron learning in as much as we collect statistical information over all of the training patterns, not just those that result in errors. This learning is similar to the local learning used in Boltzman machines (Ackley, et al. 1985); however, in these experiments we are not concerned with escaping local minima and have no need for the simulated annealing component that is used in the Boltzman machine to achieve a global search of weight space.

After processing all the patterns, the variable d_l is used to determine whether to flip the weight w_l or not. We consider the subset of links l for which d_l and w_l have an opposite sign (Recall that $w_l = \pm 1$). We will modify only the weights of the links in this subset. These are the links for which the weight is different from the correlation. For this subset of links, $|d_l|$ is a measure of the strength of the correlation evidence that the weight should change. Hence, we sort the links of the subset according to the quantity $|d_l|$. The first link, that is, the one with the biggest $|d_l|$ is flipped with a probability 1. The second link is flipped with a probability 0.05. In general, the i^{th} link is flipped with a probability $(0.05)^{(i-1)}$. This learning algorithm ensures that at least one link will be flipped if at least one link l has d_l and w_l of opposite sign. Under this condition, the average number of flipped link lies between 1 and

$1/(1-0.05) \approx 1.05$. Less than two links are flipped on average. After the weight changes are made we process all the training data again. We accept a set of weight changes only if the fitness has improved. Only one epoch of training is used. In our experiments, bias terms (i.e., threshold values) are not learned and must be found by genetic search.

Experiments show that this learning, although very simple, is sufficient to flip the correct weight in an exclusive-or (XOR) network. Assume there is a network with two hidden units and all the weights are 1; note that one hidden unit will compute logical AND and the other will computes OR. In order to get an XOR function one must set the weight between the AND unit and the output unit to -1. Thus in the case of XOR, and in fact all parity problems (of which the XOR function is an example), the limited form of Hebbian learning which we use can exploit a fortuitous feature of the problem. However, our results on the symmetry problem indicate that such fortuitous features need not be present for this learning strategy to be effective.

We are investigating more advanced forms of learning, but for the moment we are largely interested in the interaction between evolution and learning. We are trying to obtain solutions as fast as possible and to keep the cost of learning minimal. We have therefore intensionally kept the learning component simple.

3.5 Operationalizing developmental learning

One of the advantages of the recursive encoding scheme is that evaluation can be done in an incremental fashion. For parity or symmetry, a network can be evaluated with respect to its ability to solve a 2 input case. If a solution is not found for the 2 input case, no further evaluation is done. If a solution to the 2 input case is found, the next recursive iteration of development can be carried out and the 3 input case can be evaluated. Evaluation continues until the networks fail to achieve a satisfactory solution for X inputs. This form of evaluation is fast early in the evolutionary process and becomes more expensive as networks become more proficient at handling a larger number of inputs.

Analogously, we use learning only on the neural network during the first iteration of the recursive development process. This neural net is the smallest of a whole family of networks that can be generated by a chromosome; it is smallest both in term of the number of units and the number of weights, so little time is expended on learning. But we would also like the larger neural nets developed by the grammar tree to benefit from the learning process. To do this, we encode the weight changes obtained during the training of the first neural net back to the chromosome using a combination of appropriate program-symbols. We can then develop the entire family of associated networks so as to exploit learning via the modified chromosome. During the development of another network, when a cell of the intermediate network graph executes the new program-symbols added by learning it will set its weights accordingly, thereby reproducing the same subnetwork as the one contained in the first neural network after learning.

One requirement of back-coding learning onto the chromosome is that the new code should allow us to correctly develop the first iteration neural network, including the modified weights. This is not always simple. Normally, when a cell becomes a neuron it executes the

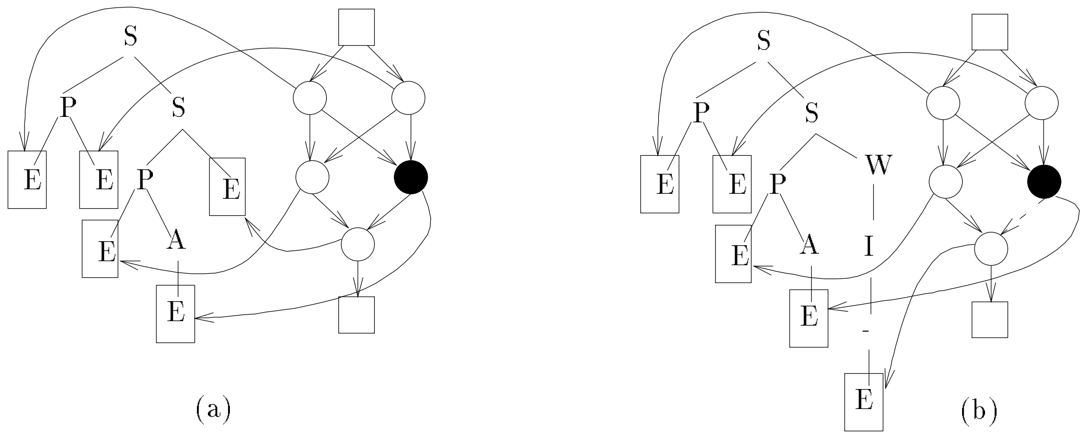

Figure 6: *Back-coding on a grammar-tree that builds an XOR network. Illustration (a) is the net before back-coding. In illustration (b) the grammar tree has been modified and the corresponding weight change is shown. The modified network computes XOR.*

program-symbol **E**, and loses the reading head so that it can no longer rewrite itself. But in order to do the back-coding, we must keep the reading head. This reading head points to the precise node where the particular subtree of program-symbols should be inserted in order to accomplish a change of weight.

Consider the value of the link register in figure 6 at the time the cell became a neuron. If this value was 1 and the learning changes the weight of the second input link into -1, then we generate the following subtree: `I(-(E))`. The program-symbol **I** sets the link register to 2, the program-symbol **-** sets the second input link to -1. Let N be the node (i.e., program-symbol) pointed by the reading head of the output neuron. This node has an arity of 0, and has previously transformed the cell into a neuron. In order to insert the new subtree, we must transform the leaf-node into an interface node of arity one. We replace the program-symbol **E** by the unary program-symbol **W** for Wait, and then insert the subtree that adds in the learned weight. When the developing output cell executes this program-symbol, it positions its head on the next subtree, and waits for the next rewriting step.

We cannot be sure that this type of simple change in the grammar-tree will always correctly code the learned behavior. The new encoding will produce the learned behavior if two conditions are satisfied:

- Learning is done only on the first neural net of the family.
- During the development with the modified code, for any cell C_r of the intermediate neural network that starts to read the inserted subtree, those cells that have fan-out connections that feed into cell C_r must have lost their reading heads. This ensures that the new changes affecting C_r do not interact with the development of those cells whose fan-out connections feed into C_r.

The first neural net of the family has the feature that no more than one neuron reads a given leaf of the grammar tree. This is not the case for higher order neural nets. If the first condition is not met, two distinct neurons could read the same leaf, and both could try to concatenate a distinct subtree corresponding to their own weight modification at this leaf. It is not possible to insert two distinct subtrees at a single leaf. Moreover, these two subtrees could encode contradictory modifications of the weights.

If the second condition is satisfied then the change introduced by the added subtree will remain local. Otherwise it may produce unexpected side effects. For example, in figure 5 assume the character **A** in the grammar tree were replaced by a **P**. The corresponding cell (which is black in figure 5) will now split instead of changing its threshold and there will be three fan-in connections to the output cell. Also assume we still wish to change the second fan-in connection to a negative weight, but not the third connection. If the inserted weight change executes before the parallel division, **P**, then both the second and third fan-in connection will be negative. If the weight change executes after the parallel division, then only the second weight will be negative.

In order to ensure the proper sequence of development, we could add several waiting program-symbols **W** to key positions so that the new inserted section of the tree executes in the proper sequence. However, this could result in an increase in the amount of time needed to develop the higher order neural nets. More critical, however, is the likelihood that it would make it more difficult to do Lamarckian evolution because the chromosome could quickly become very large. This means that the encoding would no longer be quite as compact and recombination might have to process deeper and more complex trees. While this particular coding problem could have serious impact on Lamarckian evolution, it has less significance for developmental learning. Adding long wait delays *does not* impact developmental learning, since the "expanded" chromosomes never enters the genetic population. Thus, there is no problem adding enough "wait" delays to ensure that all other development processes have terminated before effecting learned weight changes.

However, in the current experiments we have settled on the following compromise. By replacing the $0-$ary program-symbol **E** by the unary program-symbol **W** and then attaching the subtree which produces the weight change, we add at least one wait delay program-symbol, and hope that in this way, we sufficiently increase the probability that the second condition is verified. For reasons of consistency, we used this approach for both Lamarckian and developmental learning strategies. This method does not guarantee that our encoding of learned behavior does not unexpectedly interact with the developmental process, but empirically we find that in most cases it does not.

3.6 Development and Evaluation

Before looking at the results of the experiments, some discussion of the evaluation function is helpful. The following rule decides exactly when it is necessary to develop the neural network N_2, and then N_3 and so on, until $N_{l_{max}}$.

> **Developing Rule:** If N_1, N_2, ..., N_L have already been developed, then, continue to develop N_{L+1} if N_L correctly processes more than a fraction r of

the training set correctly, where r is a given threshold. If $L > L_{max}$ stop the development.

The threshold r is a user defined parameter set according to the particular application. Using this rule, only network N_1 is developed in the early generations. In the last generations, up to L_{max} neural networks are developed; the limited evaluation in early generations thus saves considerable computation time. Since the size of N_L is also more than L times bigger than the size of N_1, for $L = 1, 2, \ldots, L_{max}$, the evaluation of the fitness can last $L_{max} * (L_{max} + 1)/2$ times longer. We used $L_{max} = 10$; thus, the computation cost ratio between evaluations for N_1 and $N_{L_{max}}$ can be as much as a factor of 55.

An especially important detail is the evaluation function. We must evaluate a family of neural networks. The fitness of one neural network is based on mutual information, and it lies between 0 and 1. Gruau (1992a) presents a formal definition of mutual information. The fitness of the neural network family is simply the sum of the fitnesses of the networks that have been developed. If, for example, the fitness is 3.5, it means that the first three neural networks have been developed and are approximately correct, and the fourth neural network has also been developed, but is only partially correct.

The basic GA does not typically develop the neural networks N_L for $L > 1$ for several generations. However, when Hebbian learning is used the neural net N_1 is quickly found. Therefore when Hebbian learning is used the genetic algorithm starts to develop network N_2 from the second generation; this has the side-effect of making each evaluation more costly.

4 Experiments

We now describe the result of some experiments which compare the different ways of adding learning. The *basic GA* refers to the genetic algorithm without any learning. We have described the encoding in section 2.1. Recall that this encoding includes not only the architecture but also the weights. Thus, it is possible to run the genetic algorithm without any learning. In the learning experiments, once the neural net has been developed we perform Hebbian learning. If we use only the improved fitness of the trained network N_1 (i.e., when $L = 1$ and there is no recursion) we call it simple *Baldwin learning.* Note that this should speed up the search in finding networks that handle the minimal number of inputs, but doesn't help change the evaluation of higher order networks that process more inputs. If we code the learned information back to the chromosome, and use it to develop higher order networks, that is *developmental learning.* If the improved strings are placed back in the genetic population and allowed to compete for reproductive opportunities, the result is *Lamarckian evolution.* The following experiments will compare these four learning modes on two different tasks of increasing complexity.

4.1 Parity

To solve this problem, the genetic algorithm must find a grammar encoding that develops a family of neural networks (N_L), such that N_L computes the parity function of $L + 1$ inputs.

LEARNING MODE	TIME (sec)	time SD	EVALUATIONS	evaluation SD
basic GA	173	9.051	17808	822.79
Learning only on the output unit				
Baldwin	135	12.162	12511	678.68
Developmental	59	2.687	7160	255.27
Lamarckian	55	2.546	6800	249.33

Table 1: *Experiments for the parity problem which ran on four nodes of the parallel machine Ipsc860.*

The parity function returns the sum of all the inputs, modulo 2. As indicated in section 3.4, the parity problem is special with respect to learning on the output unit. The network N_1 implements the XOR. This XOR network has all weights equal to 1, except one weight that links the AND to the output units, as shown in figure 1. Thus, learning only needs to change weights that feed the output unit to find a solution.

In order to produce a valid comparison, we ran 50 trials and computed the mean and standard deviation of two variables: the time required to find a solution, and the total number of neural nets evaluations. The genetic algorithm ran on four nodes of the parallel machine Ipsc860. The computational power of this machine is 40 MIPS which is roughly comparable to 1.5 SPARC-2 machines. The population size ranges from 3500 to 2000 on each node.

We ran initial experiments until we found a population size and associated run-time limit such that the genetic algorithm was able to solve this problem on each of the 50 attempts. Our parallel genetic algorithm consists of several sequential genetic algorithms, each one running on one node of a MIMD machine. The model of parallel genetic algorithm is intermediate between an "island model" and a massively parallel model; each island is itself a 2-D grid implementing a sequential simulation of the genetic algorithm. The subpopulations are arranged as a grid of islands. The exchange of individuals occurs continuously. When creating a new individual, the subpopulations exchange an individual with a probability 0.01 and perform crossover with a probability 0.99.

These results are reported in terms of computations per processor in table 1. Time is reported in seconds. Since four processors were used in these experiments, computation times and evaluations should be multiplied by a factor of four. Simple Baldwin learning produces a speed up of 1.3 over the basic GA. Note that Lamarckian evolution works slightly better than developmental learning, but the two approaches are very similar. An example of a network found by the genetic algorithm that solves the parity problem is given in figure 7. (During the development of a network, it is possible to develop input neurons that are not used. Inputs are applied to the net in such a way that the extra inputs do not cause a problem during evaluation. For simplicity, we do not illustrate this type of net. To extend the biological metaphor, the neurons which are not used simply die.)

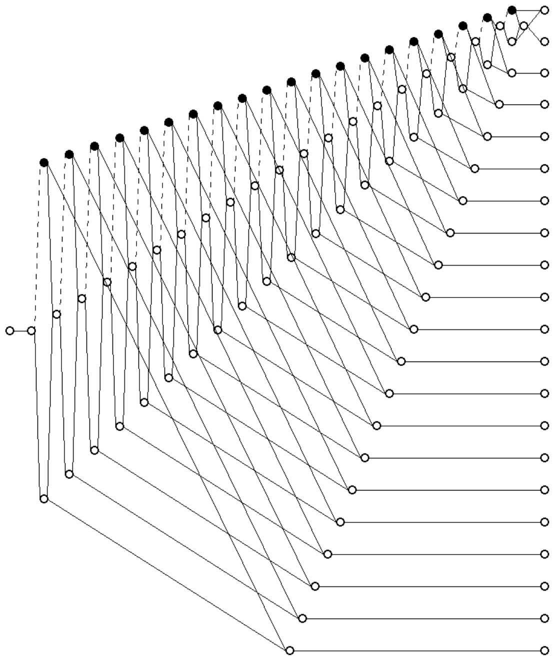

Figure 7: *A neural network for the parity of 21 input units.*

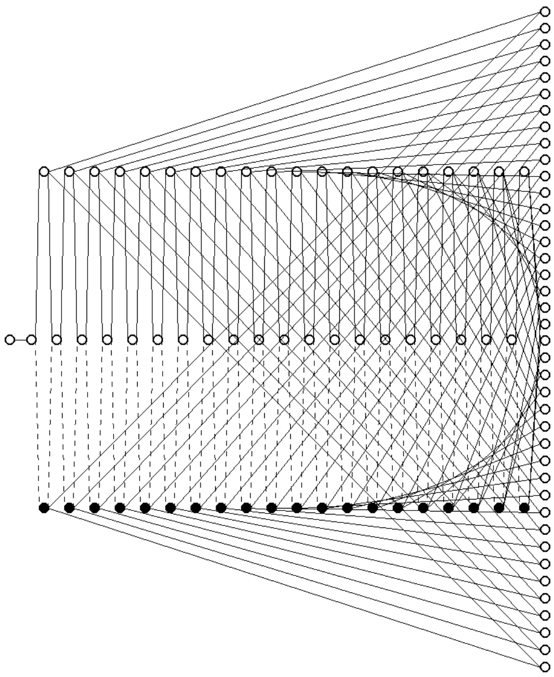

Figure 8: *A neural network for the symmetry of 40 input units.*

LEARNING MODE	TIME (sec)	time SD	EVALUATIONS	evaluation SD
basic GA	91	10.042	7332	706.38
Learning only on the output unit				
Baldwin	74	6.573	5811	418.46
Developmental	48	3.651	4352	335.94
Lamarckian	53	5.295	4381	406.96

Table 2: *Experiments for the symmetry problem which ran on 32 nodes of the parallel machine Ipsc860.*

4.2 Symmetry

The problem is to find an encoding that develops a family of neural networks (N_L), such that N_L computes the symmetry function of $2L+1$ inputs. The symmetry of $2L+1$ binary inputs returns 1 if and only if the input is symmetric and the middle bit is 1. This function has a fairly compact grammatical encoding and is not too difficult to learn with the genetic algorithm. The genetic algorithm ran on 32 nodes of an Ipsc860. The population size ranged from 300 to 200 individuals on each node, and between 9200 and 6400 on the combined set of processors. Table 2 reports an experiment with no learning and the use of the various learning models. The results are based on 30 experiments; every run finds a solution.

We indicate the computation time taken by each processor. Multiply by 32 to find the total computation effort. Baldwin learning does better than the basic GA, while Lamarckian evolution and developmental learning are almost equal. An example of a network found by the genetic algorithm that solves the parity problem is given in figure 8.

5 Conclusions

Several novel ideas are explored in the current study. The mechanics of cellular encoding are described, and we show that genetic search can be used to find cellular encodings which developed neural networks that solve whole families of parity and symmetry problems.

The addition of learning to the search process produced two interesting results. First, the notion of developmental learning was introduced. By allowing learning to affect subsequent development, early learning can be more effectively exploited and the high cost of learning on larger networks developed later in the "life" of the grammar tree can be avoided. In the current experiments, developmental learning which exploits the Baldwin effect also provides a more fair comparison between Lamarckian evolution and Darwinian evolution. Developmental learning allows the entire family of networks which can be developed from a cellular encoding to exploit learning. The same is true of the Lamarckian strategy, since offspring that inherit weights learned for network N_1 use this information to develop all subsequent networks.

The second main finding of this paper is that developmental learning appears to be competitive with Lamarckian evolution for the problem of finding a family of Boolean neural networks. The use of learning (or local optimization) to change the fitness landscape without using Lamarckian evolution could be of general interest to that part of the genetic algorithm community concerned with function optimization. It is unclear, or course, whether our results pertaining to the *Baldwin effect* generalize to other domains, but this possibility is well worth exploration.

Finally, the work reported here moves the research aimed at combining genetic algorithms and neural networks to a foundation that is in many ways more biologically appealing. This work also raises several new questions. The generality and practicality of the cellular encoding approach needs to be explored with respect to solving arbitrary function approximation problems using neural networks. We expect cellular encoding to be efficient only if the mapping to be learned has some regularities that can be captured by recursion. In future work, more complex forms of learning also need to be considered.

6 Acknowledgement

We thank the people who provided us access to the IPSC 8 nodes of the L.I.P., and the IPSC 128 nodes at Oakridge. We thank Professor Cosnard for his comments and corrections. Dr. Whitley acknowledges the contributions that Rik Belew and David Ackley have made to this work through their publications on learning and evolution as well as the ideas they have shared in informal conversations. Dr. Whitley was supported by NSF grant IRI-9010546. Mr. Gruau was supported by the CEA.

References

[1] Ackley, D.H., Hinton, G.E., & Sejnowski, T.J. (1985). A learning algorithm for Boltzman machines. *Cognitive Science, 9,* 147-169.

[2] Ackley, D.H., & Littman, M. (1991). Interactions between learning and evolution. *Proc. of the 2nd Conf. on Artificial Life*, C.G. Langton (Ed.), Addison-Wesley.

[3] Baldwin, J.M. (1896). A new factor in evolution. *American Naturalist, 30,* 441-451.

[4] Belew, R.K. (1989). When both individuals and populations search: Adding simple learning to the Genetic Algorithm. *Proc. of the Third International Conf. on Genetic Algorithms*, D. Schaffer (Ed.), Morgan Kaufmann.

[5] Belew, R.K., McInerney, J., & Schraudolf, N. (1990). Evolving Networks, Using the Genetic Algorithm with Connectionist Learning. Technical Report. CSE-CS-90-174, Univ. Calif. San Diego.

[6] Collins, R., & Jefferson, D. (1991). Selection in massively parallel genetic algorithm. *Proc. of the Fourth International Conf. on Genetic Algorithms*, R. Belew and L. Booker (Eds.), Morgan Kaufmann.

[7] Davis, L. (1991). Hybrid Genetic Algorithms. In L. Davis (Ed.), *Handbook of Genetic Algorithms* (pp. 54-60) Van Nostrand Reinhold.

[8] Goldberg, D.E., (1989). *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley.

[9] Gruau, F. (1992a). Genetic synthesis of Boolean neural networks with a cell rewriting developmental process. In D. Whitley and J.D. Schaffer (Eds.), *Combinations of Genetic Algorithms and Neural Networks,* IEEE Computer Society Press.

[10] Gruau, F. (1992b). Cellular encoding of genetic neural network. Technical Report 92.21, Laboratoire de l'Informatique pour le Parallélisme, Ecole Normale Supérieure de Lyon.

[11] Hinton, G.E., & Nowlan, S.J. (1987). How learning can guide evolution. *Complex Systems, 1*, 495–502.

[12] Kitano, H. (1990). Designing neural network using genetic algorithm with graph generation system. *Complex Systems, 4*, 461–476.

[13] Kitano, H. (1992). Neurogenetic Learning: an integrated method of designing and training neural networks using genetic algorithms. Technical Report CMU-CMT-92-13 Carnegie Mellon University.

[14] Prusinkiewicz, P., & Lindenmayer, A. (1992). *The Algorithmic Beauty of Plants.* Springer-Verlag.

[15] McClelland, J.L., Rumelhart, D.E., & Hinton, G.E. (1986). The appeal of parallel distributed processing, In, *Parallel Distributed Processing, Vol. I*, D.E. Rumelhart and J.L. McClelland, eds. MIT Press.

[16] McClelland, J.L., & Rumelhart, D.E. (1988). *Explorations in Parallel Distributed Processing.* MIT Press.

[17] Miller, G. and Todd, P., & Hedge, S. (1989). Designing neural networks using genetic algorithm, *Proc. of the Third International Conf. on Genetic Algorithms*, D. Schaffer (Ed.), Morgan Kaufmann.

[18] Mjolsness, E., Sharp, D.H. & Alpert, B.K. (1989). Scaling, machine learning, and genetic neural nets. *Advances in Applied Mathematics, 10*, 137-163.

[19] Mühlenbein, H. (1990). Limitations of multi-layer perceptrons networks - steps towards genetic neural networks. *Parallel Computing, 14*, 249–260.

[20] Koza, J.R. & Rice, J.P. (1991). Genetic generation of both the weights and architecture for a neural network. *Intern. Joint Conf. on Neural Networks, Seattle 92.*

[21] Koza, J.R. (1992). *Genetic Programming.* MIT Press/Bradford Books.

[22] Schaffer, J.D., Caruana, R.A. & Eshelman, L.J. (1990). Using genetic search to exploit the emergent behavior of neural networks. In S. Forrest (Ed.), *Emergent Computation* (pp. 244-248). Amsterdam: North Holland.

[23] Schaffer, J.D., Whitley, D. & Eshelman, L.J. (1992). Combinations of genetic algorithms and neural networks: a survey of the state of the art. In D. Whitley and J.D. Schaffer (Eds.), *Combinations of Genetic Algorithms and Neural Networks.* IEEE Computer Society Press.

[24] Whitley, D. (1994). A Genetic Algorithm Tutorial. *Journal of Statistic and Computing*, in press.

[25] Whitley, D., Starkweather, T., & Bogart, C. (1990). Genetic algorithms and neural networks: optimizing connection and connectivity. *Parallel Computing, 14,* 347–361.