Cellular Encoding Applied to Neurocontrol

Darrell Whitley, Frederic Gruau and Larry Pyeatt
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523 USA
whitley,gruau,pyeatt@cs.colostate.edu

Abstract

Neural networks are trained for balancing 1 and 2 poles attached to a cart on a fixed track. For one variant of the single pole system, only pole angle and cart position variables are supplied as inputs; the network must learn to compute velocities. All of the problems are solved using a fixed architecture and using a new version of cellular encoding that evolves an application specific architecture with real-valued weights. The learning times and generalization capabilities are compared for neural networks developed using both methods. After a post processing simplification, topologies produced by cellular encoding were very simple and could be analyzed. Architectures with no hidden units were produced for the single pole and the two pole problem when velocity information is supplied as an input. Moreover, these linear solutions display good generalization. For all the control problems, cellular encoding can automatically generate architectures whose complexity and structure reflect the features of the problem to solve.

1 Introduction

One area where genetic algorithms have been used for training neural networks is reinforcement learning. For these types of applications, a training set of input-output pairs does not exist. Rather, the neural network is applied to a problem and the performance of the network is used to supply a reinforcement signal.

One particularly interesting example of this kind of work is the neurocontrol pole balancing experiments of Alexis Wieland (1990;1991). Using a genetic algorithm Wieland trained fully recurrent neural networks to balance a pole fixed to a cart moving on a finite track using only the pole angle and cart position as inputs. Typically, velocity information is provided for this problem, but in one variant of this problem a fully recurrent network had to learn to estimate velocities. Wieland also trained networks to balance 2 poles and a jointed pole. In this paper we reproduce results similar to Wieland's for 1 and 2 poles and also evolve cellular encodings for neural networks to solve these same problems. Cellular encoding is a language for local graph transformations that controls the division of cells that grow an artificial neural network (ANN) (Gruau and Whitley, 1993; Gruau 1994). Earlier versions of the cellular encoding language described only Boolean neural networks. In the current version of cellular encoding, real-valued weights are evolved. In addition, some constraints are imposed on the size of networks during the development process.

Our results indicate that Wieland's approach to training neural networks using genetic algorithms works best with small populations (i.e., 100 strings) and with higher than usual mutation rates (e.g. 30%). Our results also indicate that during the evolution of cellular encoded neural networks, the most compact network is not found. Rather, larger networks are typically evolved where some subsets of hidden nodes in the network can be replaced by a single connection or by some smaller, more concise subset of neurons. As a result, simple networks were obtained for solving these same neurocontrol applications. In particular, when velocity information is supplied as an input, good linear solutions to both the one-pole and the two-pole problems were produced. The linear solutions represent a somewhat unexpected result for the problem of simultaneously balancing two poles. As expected, when velocity information is not supplied as an input these problems are much more difficult to solve.

1.1 Variants on the Pole Balancing Problem

Wieland (1990) used a genetic algorithm to solve several variants of the classic problem of balancing a pole attached to a cart that moves on a finite track. In the most common version of this problem the pole angle is restricted to ± 12 degrees and the pole angular velocity and cart velocity are supplied as inputs along with the pole angle and cart position. Given the ± 12 degree restriction, this problem is known to have a linear solution that provides a good approximation to an optimal control strategy.

The variants of this problem solved by Wieland include one version where only the pole angle and cart position are given as inputs and a fully recurrent network is used that must learn to estimate the velocity information. In Wieland's experiments, the set of inputs fans into each unit of a fully recurrent network. One of the units of the recurrent network is chosen as an output unit.

One variant of this problem involves balancing a jointed pole and another involves balancing two poles of different length and mass, where the length and mass of the small pole are ten percent of those of the big pole. The big pole length and mass are 1 meter and 1 kilogram. The cart track is 4.8 meters. Cart position is scaled from ± 2.4 meters to an input range of ± 1. Pole angles ranging from ± 36 degrees are scaled to ± 1. Pole angles outside this range produce a failure signal. Cart velocity ranging from ± 1.5 meters per second and pole angular velocity ranging from ± 115 degrees per second are scaled to ± 1. Velocities outside of these ranges are allowed, but the resulting inputs will also be scaled accordingly and produce inputs that are outside the ± 1 range.

We attempted to reproduce Wieland's results for 1 and 2 poles. A population size of 100 was used in all experiments, and the mutation rate was 30%. Instead of using exactly the same genetic algorithm as Wieland, we used the Genitor algorithm; this is a steady state genetic algorithm using linear ranking for selection purposes. We used both a fixed mutation rate as well as an adaptive mutation strategy (Starkweather et al., 1990). . The results reported in this paper used the adaptive mutation strategy. The selection bias was 2.0. For the fixed architecture experiments, each weight was encoded using 8 bits with values distributed between $-1, -\frac{253}{255}, -\frac{251}{255}, \ldots, \frac{253}{255}, 1$ with no representation for zero. This distribution, which was used by Wieland, keeps weight contributions in a range to which the transfer function is sensitive. Also Wieland notes that since there is no representation for zero, it should be difficult for the system to exploit unstable fixed points. (In other words, it would be difficult for the cart to remain motionless with the pole perfectly vertical.)

This raises another critical issue: What evaluation function was used by Wieland? Wieland (1990:95) states, "The basic fitness of any network controller is defined simply as the time that it was able to keep the pole(s) from falling down or the cart from hitting an end of the track." Wieland's statements suggest that the system is tested once from a single start state; one would assume this would be with the cart centered and the pole in a vertical position. Wieland also does not address the question of generalization. We explore the generalization capabilities of the neural networks developed in this paper as well as the use of incremental evaluation functions that utilize multiple start states.

2 Cellular Encoding with real weights

Cellular encoding is a language for local graph transformations that controls the division of cells which grow into an artificial neural network (Gruau and Whitley, 1993; Gruau, 1994). Each cell has an input site and an output site and can be linked to other cells. A cell also possesses a list of internal registers that represent local memory. The registers contain neuron attributes such as weights or the threshold. The graph transformations can be classified into cell divisions, local topology transformations and modifications of cell registers.

A cell division replaces one cell called the parent cell by two cells called child cells. A cell division must specify how the two child cells will be linked. For practical purposes, we give a name to each graph transformation; these names in turn are manipulated by the genetic algorithm. In the *sequential* division `SEQ` the first child cell inherits the input links, the second child cell inherits the output links and the first child cell is connected to the second child cell. In the *parallel* division `PAR` both child cells inherit both the input and output links from the parent cell. Hence, each link is duplicated and the child cells are not connected. Examples of cell division are given by Gruau and Whitley (1993). In general, a particular cell division is specified by listing for each child cell, which link is inherited from the mother cell. In this paper, we used two other divisions. Division `CPI` (resp. `CPO`) is like a sequential division, except that the input links are duplicated (for `CPO` the output links are duplicated) in both child cells.

Local topology transformations remove or add links without changing the number of cells. We used the transformation `CYC` that adds a recurrent link from the

output site to the input site of the cell to which `CYC` is applied. This simple transformation makes it possible to generate any kind of recurrent neural network because the recurrent link can be duplicated many times.

The `WEIGHT` instruction is used to modify cell registers. It has k integer parameters, each one specifying a real number in floating point notation: the real is equal to the integer between -255 and 256 divided by 256. (The resulting weight distributions are similar, but not identical to those used by Wieland.) The first parameter sets the threshold, and the last $k-1$ parameters set the $k-1$ weights of the first input links. If a neuron happens to have more than $k-1$ input links, the weights of the supernumerary input links will be set by default to the value 256 (i.e., $\frac{256}{256} = 1$). All cells eventually will read a `WEIGHT` program symbol; this terminates the development of the cell. Consistent with Wieland's work, the transfer function of the neuron is a clipped linear function between -1 and 1.

The cellular code is a tree called a *grammar-tree*, labeled by names of graph transformations. Each cell carries a duplicate copy of the grammar tree and has an internal register called a reading head that points to a particular position of the grammar tree. At each step of development, each cell executes the graph transformation pointed to by its reading head and then advances the reading head. After cells terminate development they lose their reading-head and become neurons.

The order in which cells execute graph transformation is determined as follows: once a cell has executed its graph transformation, 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.

We use three control program symbols: `PROGN`, `CONT` and `WAIT`. The program symbol `PROGN` has an arbitrary number of subtrees, and all the subtrees are executed one after the other, starting from the subtree number one. A `CONT` program symbol is used to move a reading head reading the subtree number i to the root of subtree number $i+1$. `CONT` has 0 subtrees. The `WAIT` program symbol is used to delay the setting of the weights by a certain number of time steps specified in a second subtree. An example of how a `WAIT` operation can impact cellular developement is given by Gruau and Whitley (1993).

2.1 Syntactic Constraints and Genetic Operators

We used a BNF grammar to specify both a subset of syntactically correct grammar-trees and the underlying data structure. The data structure is a tree by default. When the data structure is not a tree, it can be a **`list`**, **`set`** or **`integer`**. By using syntactic constraints on the trees produced by the BNF grammar, a recursive nonterminal of the type **`tree`** can be associated with a range that specifies a lower and upper bound on the number of recursive rewritings. In our experiments, this is used to set a lower bound m and an upper bound M on the number of neurons of the final neural network. For the **`list`** and **`set`** data structure we set a range for the number of elements. For the **`integer`** data structure we set a lower bound and an upper bound of a random integer value. The **`list`** and **`set`** data structures are described by a set of subtrees called the "**`elements`**." The **`list`** data structure is used to store a vector of subtrees. Each of the subtrees is derived using one of the elements. Two subtrees may be derived using the same element. The **`set`** data structure is like the **`list`** data structure, except that each of the subtrees must be derived using a different element.

Recombination. Recombination must be implemented so that two cellular codes that are syntactically correct produce an offspring that is also syntactically correct (i.e., that can be parsed by the BNF grammar). Each terminal of a grammar tree has a primary label. The primary label of a terminal is the name of the nonterminal that generated it. Crossover with another tree may occur only if the two root symbols of the subtrees being exchanged have the same primary label. This mechanism ensures the closure of the crossover operator with respect to the syntactic constraints.

Recombination between two **`trees`** is the classic crossover used in Genetic Programming (Koza, 1992), where two subtrees are exchanged. Crossover between two **`integers`** is disabled. Crossover between two **`lists`**, two **`sets`** or two **`arrays`** is implemented like crossover between bit strings, since the underlying arrangement of all these data structures is a string.

Mutation. To mutate one node of a tree labeled by a terminal t, we replace the subtree beginning at this node by a single node labeled with the nonterminal parent of t. Then we rewrite the tree using the BNF grammar. To mutate a **`list`**, **`set`** or **`array`** data structure, we randomly add or suppress an element. To mutate an integer, we add a random value uniformly distributed between ± 32.

Each time an offspring is created, all the nodes are mutated with a small probability. For the single pole problem the mutation probability for the `tree` nodes is 0.005, for the `list` node and the `set` node it is 0.05, while for the `integer` node it is 0.25. For the problem of balancing two poles, we increased the mutation probability of the `tree` node to 0.05.

3 The Genetic Algorithm and Evaluation Function

For the cellular encoding experiments we used a parallel Genetic Algorithm (GA) with 32 subpopulations. A more complete description of this algorithm is given by Gruau (1995). The parallel genetic algorithm combines the advantages of the massively parallel model (Collins and Jefferson, 1991) and the island model (Muhlenbien et al., 1991). Individuals are distributed on islands where each island is a grid forming a 2-D torus. Islands are also arranged as a grid and individuals can migrate only to the four neighbor islands. Not all the sites of a 2-D grid are occupied. The density of population is kept around 0.5. During mating, a site s is randomly chosen on the grid. Two successive random walks are performed and the best individuals found on the 2 random walks are mated. The offspring is placed on site s.

The migration rate is 1%. An individual is exchanged between two adjacent processors after 100 individuals have been created with crossover. The MIMD parallel machine used is an IPSC860, with 32 processors. The subpopulation on each processor is 64 individuals, so the total population is 2048.

3.1 The Evaluation Function

The fitness of a given neural network for the problem of balancing one or two poles is the number of time steps the poles can be balanced over different initial conditions of the system.

We tested each neural net using a set of eleven parametrized initial states reported in table 1. We choose the three parameters in a different way for each problem, so as to match parameters to the specific features of the problem. For example, the initial value for the big pole in the two pole problem was chosen very small compared to the single pole. This is because it is impossible to recover from a situation were the big pole is bent more than a few degrees.

3.2 Incremental Evaluation

For each evaluation, we pick an initial start state from among the eleven possible start states; each start state is tested for up to 1000 time steps. Before the neural network is evaluated on the next start state it must control the system for the current start state for at least 500 time steps. When a network fails to successfully control the system for at least 500 time steps for the current start state, evaluation is terminated. The evaluation function is given by the total number of time steps for which the system was successfully controlled over all start states tested. Thus, networks with better performance also receive more evaluation time. When a network successfully controls the system for all 11 initial states for 1000 time steps, the genetic algorithm is terminated. For each genetic algorithm run, we compute a generalization test to make sure that the resulting neural network could balance the pole over 100,000 time steps when starting from an initial setting where all the variables are 0.

The population is divided into 32 subpopulations. Fitness evaluation in each subpopulation is different because the 11 initial start states used for training are sorted into different predefined sequences. As a result, different subpopulations can evolve different behaviors and thus evolve different genetic material. This also has significant implications with respect to the effects of migration. In many migration schemes individuals representing above average solutions are moved from one subpopulation to the next. If a superior individual migrates to a new subpopulation it may quickly spread its genetic material through that subpopulation. For the fitness function used in these control experiments, however, an individual with a good evaluation in one subpopulation may or may not be above average in another subpopulation because it may not have learned to solve problems that appear early in the sequence of start states for that subpopulation. Only those individual's whose behavior generalizes to other situations (i.e., different initial states) will be able to compete in different subpopulations.

3.3 The Single Pole Problem

We experimented on four variants of the classic problem of balancing a single pole attached to a cart on a finite track. The most common variant of this problem uses 4 inputs (pole angle and velocity, cart position and velocity) and a bang-bang control strategy. Following Wieland we also tested a variant with 4 inputs and a continuous control strategy where the force applied to the cart is the output unit's activity multiplied by 10. Again following Wieland, we experimented with

cart position	0	$.9x$	$-.9x$	x	$-x$	0	0	0	0	x	$-x$
cart velocity	0	0	0	0	0	0	0	0	0	0	0
pole position	0	0	0	$-\theta$	θ	$\theta/2$	$-\theta/2$	θ	$-\theta$	0	0
pole velocity	0	0	0	0	0	$\hat{\theta}/2$	$-\hat{\theta}/2$	0	0	$-\hat{\theta}$	$\hat{\theta}$

Table 1: 11 initial settings of the cart and the pole, parametrized by three numbers

Problem Description	x	θ	$\hat{\theta}$
single pole with 4 inputs	2.4	27	65
single pole with 2 inputs	2.16	18	43
two poles: long pole	2.16	1.8	4.3
two poles: short pole		0	0

Table 2: Different values of the three parameters. The cart position (x) is expressed in meters, the pole position (θ) in degrees and the pole velocity ($\hat{\theta}$) in degrees per second.

a variant of the problem where only 2 inputs are given, (pole angle and cart position) and a recurrent network must be used to compute the relevant velocity information. This problem was also solved using both a bang-bang and continuous control strategy.

For the recurrent networks, neuron activities are updated in parallel. The initial activation of all neurons is set to 0. For its first computation, the recurrent network is relaxed 20 times to give the network time to initialize its internal states. For each subsequent computation the network is relaxed four times in parallel before its output is considered.

The following syntactic constraints were used for the single pole problem. The recurrence is limited to a recurrent link that goes from the output of a neuron directly to its input. The weights and threshold are constrained to lie between 1 and -1. The number of neurons in the network varies between 5 and 21. We ran some experiments with no lower bound on the number of neurons in the network and found that the genetic algorithm was generating only networks with one unit. Rapid premature convergence to this linear solution prevented the genetic algorithm from exploring other solutions. Fixing a lower bound on the number of units prevented this problem.

We ran a generalization test based on that used by Whitley et al. (1993) where 625 initial settings of the cart and of the pole are generated. Each of the normalized 4 input variables representing cart position, cart velocity, pole position, and pole velocity take the following 5 values: 0.05, 0.25, 0.5, 0.75, 0.95. (Note that these values actually scale to positions -0.9, -0.5, 0, 0.5 and 0.9 over the ± 1 input range.) This results in $5^4 = 625$ test cases. Generalization is tested by counting the number of test cases for which the neural network can balance the pole for 1000 time steps. Whitley et al. reported an average of 406 successes over 625 test cases using all 4 inputs and a bang bang control strategy.

The results obtained using cellular encoded networks as well as fixed architectures using our implementation of Wieland's methods are reported in table 3. Each result represents an average over 10 experiments. These results suggest that the bang bang control strategy is easier to learn and that learning with a bang bang control strategy produces better generalization. If we allow continuous output of our neural net, then the set of solutions to the problem is the set of continuous values instead of the set of Boolean values, which is much smaller. This suggests that as the set of possible output values increases, generalization is decreased.

Learning time increases and generalization decreases when the neural network must learn to compute the velocity information on its own. This is due to the lack of precision with which the speed is computed by the neural network.

3.4 Two Poles

Table 3 also reports results for the experiments involving two poles. The neural networks have 6 input units which are respectively the cart position, the cart velocity, the big pole position, the big pole velocity, the small pole position and the small pole velocity. The failure angle for the two pole problem is ± 36 degrees. We found that a bang bang control strategy did not allow precise enough control in this case so we considered only continuous control. There are a few differences between the syntactic constraints used for the one pole problem and the two pole problem. We do not use the program symbol `CYC`, and thus the

	Cellular Encoded Nets		Fixed Architecture Nets	
Problem	Learning	Gen.	Learning	Gen
1-pole 4 inputs bb	2,234	430	49,500	308
1-pole 4 inputs c	19,011	386	88,300	306
1-pole 2 inputs bb	75,000	314	191,100	229
1-pole 2 inputs c	270,000	220	214,700	202
2-poles 6 inputs c	570,000	513	434,500	432

Table 3: Note, the experiment with the single pole and cellular encoding were run on a sequential machine. They took on average 1mm for the bang bang control 10 mm for the continuous control.

resulting neural network will not be recurrent. In addition, the number of hidden units and output units in the network is bounded to be between 9 and 21. Sampling over the full ranges of the input variables is impractical for testing generalization because the two pole system can be controlled within only a narrow range of values. We defined new ranges of the variables for training and testing purposes. The actual input ranges did not change; only the values used for training and testing. The intervals for the cart variable are the same as the ones used in the learning set: the position varies between ± 2.16 meters, the velocity varies between ± 1.35 meters per second. The interval for the big pole is two times the interval used in the learning set (See Table 2). Thus, the system is tested with large pole angle settings between ± 3.6 degrees, and the pole velocity between ± 8.6 degrees per second. We then used 625 test cases with settings at 0.05, 0.25, 0.5, 0.75 and 0.95 intervals within these reduced ranges. The small pole was always set to zero, because it moves so quickly. The poles could be balanced on average for 513 of the 625 initial settings using these reduced ranges.

4 Analysis of the Network Structure

By simulating the neural network produced by the genetic algorithm, we noticed that many neurons are either constantly saturated at ± 1, or always use the linear part of the transfer function and compute the identity. In the nonrecurrent case this made it possible to obtain a mathematically equivalent simplified neural network, modulo the effects of rounding errors.

4.1 One Pole With Velocity Information

We took a neural net found by the cellular encoding algorithm with an average generalization performance (430 out of 625), simplified the networks and were able to remove all hidden units. Moreover, the generalization of the linear solution was 453. It is known that this problem has a good linear solution when the pole angle is within ± 12 degrees of vertical, but it was unclear to us whether a linear solution could generalize as well as the networks using hidden nodes, since the pole is allowed to move over a larger range than ± 12 degrees. Our results show that linear solutions do yield good generalization for this problem over large ranges of pole angles.

4.2 One Pole Without Velocity Information

Figure 1 shows two examples of neural networks generated by the genetic algorithm for the one pole problem without velocity information. All these neural networks show the same features which enable them to compute the speed. First, the information flows from the input units to the output units through four layers. The reason for this is that the neural network is relaxed four times before its output is considered. Second, some intermediate neurons are used to retain information about the previous state. These particular neurons are pointed out in figure 1. When the information flows through these neurons, it takes five or six time steps to go from the input layer to the output layer. These neurons are used to register the previous cart position and pole position, and to compute the speed.

4.3 Two Poles With Velocity Information

Wieland suggested that the 2 pole problem was very difficult; Wieland used 10 neurons to solve this problem. Moreover, we had designed a hand coded solution to this problem before running the genetic algorithm that had 16 hidden units.

Figure 2 (a) shows a neural network generated by the genetic algorithm for balancing two poles. Figure 2 (b) shows the ANN obtained after having simplified these neurons. Apart from the input units, the final neural network contains no hidden units.

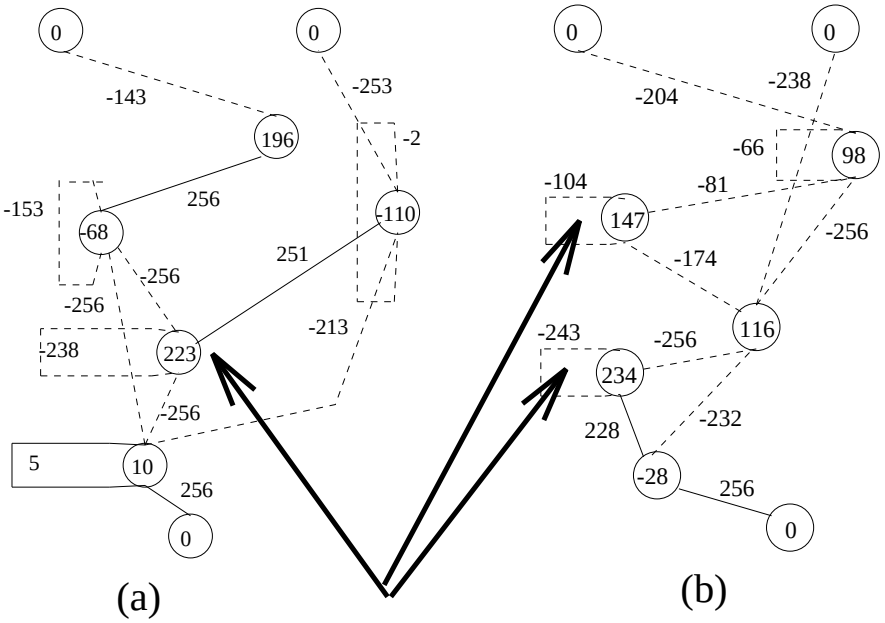

Figure 1: Two neural networks generated for the one pole without velocity information, arrows point to the neurons used to retain information about the previous state. Inputs are shown at the top of the network diagram.

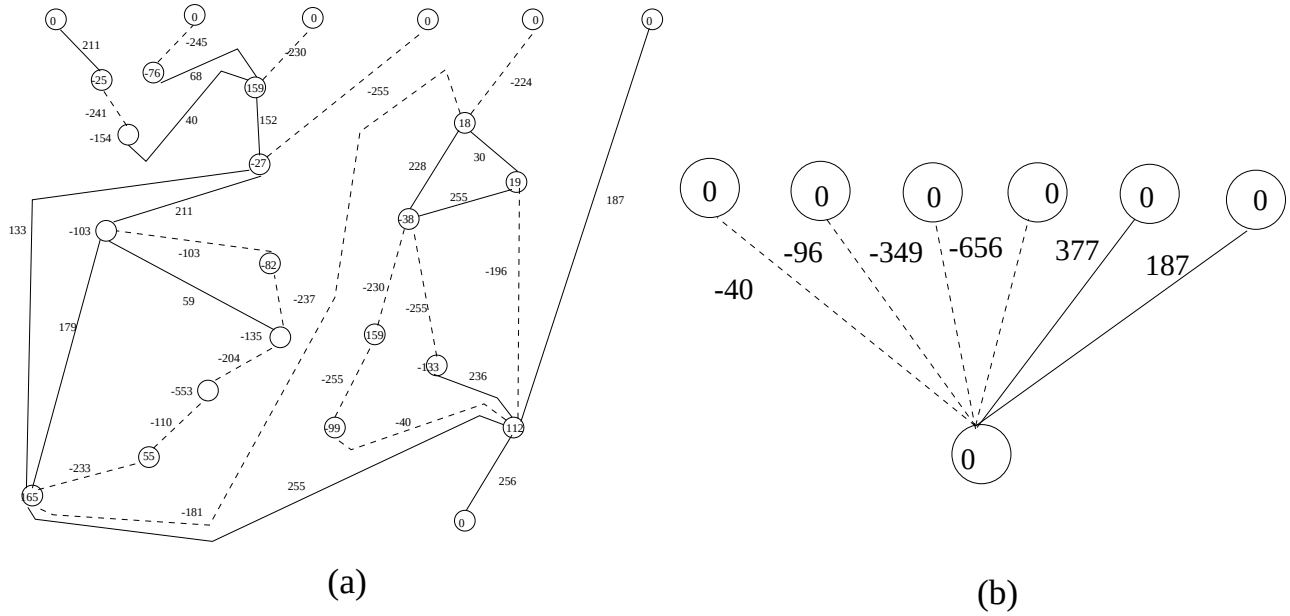

Figure 2: Simplification of the neural network found by the GA for the balancing of two poles. The ANN with 19 units (a) is reduced to a linear solution after simplification (b).

For the linear solution to the two pole problem the threshold of the output unit is 0, and the weights (after scaling by 1/256) are -0.16, -0.23, -1.36, -2.9, 1.47, 0.87. Note that several of the resulting weights are outside the ± 1 ranges; this was achieved by the cellular code by using multiple parallel connections to overcome the normal ± 1 range allowed for weights.

The solution found by the genetic algorithm is not obvious. It begins with four negative weights. The third and fourth negative weights indicate that if the big pole is to the right (resp. left) then push the cart to the left (resp. right). This is counterintuitive, because it makes the big pole move even further from vertical. But then, weight 5 and weight 6 are positive; when the small pole makes an angle bigger than the big pole in absolute value, then the cart is pushed right (resp. left) and the two poles are brought back to the null position. The first two small negative weights tend to push the cart back to the center.

5 Conclusions

This paper applies a new version of cellular encoding to the problem of balancing one or more poles on a cart moving on a fixed track. The new version of cellular encoding allows for data structures encoding the weights of the neural network; it also uses syntactic constraints to bound certain properties of the neural networks, such as the number of hidden units. The use of real-valued weights in conjunction with cellular encoding addresses a criticism of previous work with cellular encoding that was restricted to evolving Boolean networks.

Cellular encoding produced solutions competitive with those obtained using methods and architectures developed by Wieland on these same problems. The advantage of cellular encoding is that it could automatically find small architectures whose structure and complexity fit the specificity of the problem. This provided new insight about the complexity of the problem we are solving. In particular, the results indicate that competitive linear solutions exist not only for the problem of balancing one pole between ± 36 degrees, but also the problem of balancing two poles. The problem of balancing the pole without velocity information represents a more complex control task.

The work reported here also reemphasizes the need to study both generalization and learning time when evaluating systems for adaptive neurocontrol.

Acknowledgements

This work was supported in part by NSF grant IRI-9312748. We thank Oakridge National Laboratories for providing access to their Intel IPSC860.

References

[Collins and Jefferson, 1991] Collins, R. and Jefferson, D. (1991). Selection in Massively Parallel Genetic Algorithms. In Booker, L. and Belew, R., editors, *Proc. of the 4th Int'l. Conf. on GAs*, pages 249–256. Morgan Kauffman.

[Gruau, 1994] Gruau, F. (1994). *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm*. PhD thesis, Ecole Normale Supérieure de Lyon.

[Gruau, 1995] Gruau, F. (1995). Automatic Definition of Modular Neural Networks. *Adaptive Behavior*, In Press.

[Gruau and Whitley, 1993] Gruau, F. and Whitley, D. (1993). Adding Learning to the Cellular Developmental of Neural Networks: Evolution and the Baldwin Effect. *Journal of Evolutionary Computation*, 1(3):213–233.

[Koza, 1992] Koza, J. (1992). *Genetic programming: A paradigm for genetically breeding computer population of computer programs to solve problems*. MIT Press, Cambridge, MA.

[Mühlenbein et al., 1991] Mühlenbein, H., Schomisch, M., and Born, J. (1991). The Parallel Genetic Algorithm as Function Optimizer. *Parallel Computing*, 17:619–632.

[Starkweather et al., 1990] Starkweather, T., Whitley, L. D., and Mathias, K. E. (1990). Optimization Using Distributed Genetic Algorithms. In Schwefel, H. and Männer, R., editors, *Parallel Problem Solving from Nature*, pages 176–185. Springer/Verlag.

[Whitley et al., 1993] Whitley, D., Dominic, S., Das, R., and Anderson, C. (1993). Genetic Reinforcement Learning for Neurocontrol Problems. *Machine Learning*, 13:259–284.

[Wieland, 1990] Wieland, A. (1990). Evolving Controls for Unstable Systems. In *Connectionist Models: Proc. 1990 Summer School*, pages 91–102. Morgan Kaufmann.

[Wieland, 1991] Wieland, A. (1991). Evolving Neural Network Controllers for Unstable Systems. In *International Joint Conference on Neural Networks, Seattle*, pages 667–673.