A Comparison between Cellular Encoding and Direct Encoding for Genetic Neural Networks

Frédéric Gruau	Darrell Whitley	Larry Pyeatt
CWI, Kruislaan 413	Computer Science Dept.	Computer Science Dept.
1098 SJ Amsterdam	Colorado State University	Colorado State University
The Netherlands	Fort Collins, CO 80523 USA	Fort Collins, CO 80523 USA
gruau@cwi.nl	whitley@cs.colostate.edu	pyeatt@cs.colostate.edu

ABSTRACT

This paper compares the efficiency of two encoding schemes for Artificial Neural Networks optimized by evolutionary algorithms. Direct Encoding encodes the weights for an a priori fixed neural network architecture. Cellular Encoding encodes both weights and the architecture of the neural network. In previous studies, Direct Encoding and Cellular Encoding have been used to create neural networks for balancing 1 and 2 poles attached to a cart on a fixed track. The poles are balanced by a controller that pushes the cart to the left or the right. In some cases velocity information about the pole and cart is provided as an input; in other cases the network must learn to balance a single pole without velocity information. A careful study of the behavior of these systems suggests that it is possible to balance a single pole with velocity information as an input and without learning to compute the velocity. A new fitness function is introduced that forces the neural network to compute the velocity. By using this new fitness function and tuning the syntactic constraints used with cellular encoding, we achieve a tenfold speedup over our previous study and solve a more difficult problem: balancing two poles when no information about the velocity is provided as input.

1 Introduction

For reinforcement learning problems a training set of input-output pairs does not exist. Instead, the behavior of a particular instantiation of an artificial neural network (ANN) is used to supply a reinforcement signal that evaluates performance by providing either positive or negative feedback. Particularly difficult reinforcement learning problems are those that require a recurrent network. Genetic algorithms would appear to be particularly well suited for training neural networks for this class of reinforcement learning problems, since they require only feedback based on performance. On the other hand, there are well known problems when attempting to train neural networks using evolutionary algorithms that emphasize recombination.

Wieland [9] introduced several interesting test problems for neurocontrol that involve balancing one or more poles fixed to a cart moving on a finite track. Possible actions on the system include applying a force to the cart to push it to the left or right. This can be a bang-bang controller, such that there is one fixed force that is applied to the cart, or the force may be variable depending on the output of the controller. Note that this corresponds to having a discrete (bang-bang) output action or a continuous (variable force) output action. Traditionally, there are 4 inputs to such a system: pole angle and pole velocity as well as cart position and cart velocity. Wieland offered a solution to the pole balancing problem using only the pole angle and cart position as inputs. Wieland's solution involves the use of a fully recurrent network which can potentially learn to compute velocity information.

For the classic pole balancing problem, velocity information is supplied as an input and the pole angle is restricted to ± 12 degrees; in this case a good linear solution is known to exist. Our recent experimental studies also suggest that good linear solutions exist for some problems that involve balancing two poles when velocity information is provided. We have produced results similar to Wieland's for 1 and 2 poles and have also evolved cellular encodings for neural networks to solve these same problems. While a recurrent network should be required when velocity information is not provided, we have found that neural networks with no recurrent connections are able to balance poles from multiple initial states for 100,000 time steps. These networks are able to do pole balancing by implementing cyclic strategies that keep the system in a state of oscillation. A new evaluation function is introduced that penalizes oscillation and forces the networks to learn to compute velocity information.

Networks were developed in two ways for the current study. First, networks were evolved using cellular encoding. This method generates both the weights and the architectures of the neural networks. Second, neural networks with fixed architectures were trained using a genetic algorithm to find the weights. Following the approach originally used by Wieland, each weight was encoded using 8 bits with values distributed among the values $\{-1, -\frac{253}{255}, -\frac{251}{255}, ..., \frac{253}{255}, 1\}$ with no representation for zero [9]. Several networks of varying sizes were tried and the best network for solving each problem was selected. The fixed architecture networks included a bias input, which was not used by Wieland. Also, the number of neurons used was different than those reported by Wieland. Relatively small populations were used (100 strings) with a high mutation rate (30%).

The goal of the paper is to continue the comparison between direct encoding and cellular encoding started in [8]. The use of the new evaluation function and new syntactic constraints has allowed us to solve the simpler version of the pole balancing problem more than ten times faster with cellular encoding (CE). This speedup is both in terms number of evaluations and wall clock time. We also solve a more difficult version of the pole balancing problem where two poles are balanced and no information about velocity is supplied. We had not previous attempted this problem (e.g. [8]) and this problem was not addressed by Wieland.

In general our results indicate that cellular encoding is able to optimize both the architecture and the weights at the same time. In contrast, direct encodings assumes an a priori architecture which is suboptimal in the general case. Achieving good performance with direct encoding requires testing many architectures individually. Due to the number of architectures and parameters available, this can take considerable effort. As a result, better overall performance can be obtained with cellular encoding.

One benefit of using cellular encoding is that the structure of the resulting neural networks fits the problem at hand and can be analyzed. 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 are not obvious and represent a somewhat unexpected result for the problem of simultaneously balancing two poles. For the case where the velocity is not known solutions with one or two hidden units were produced.

2 Reproducing Wieland's Results

This section describes the experiments done using a direct encoding of the weights, following [9]. The balancing of one pole is a four input problem. These are the pole angle and angular velocity, as well as the cart position and velocity. One variant of the pole balancing problem involves balancing one pole with out using the velocity information; in this case only pole angle and cart position are used as inputs. Another variant involves balancing two poles of different length and mass, where the length and mass of the small pole are ten percent of that of the big pole. This problem has 6 inputs: angle and velocity for each pole, plus the cart position and velocity. The most difficult problem is a combination of the last two: balancing two poles without velocity information. This problem has three inputs: cart position and two inputs for the angles of the large and small pole. The big pole length and mass are 1 meter and 1 kilogram. The length of the cart track is 4.8 meters. The inputs are scaled before being input to the neural network. 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. The simplest measure of performance is given by testing how long a neural controller can balance the pole(s) without allowing the poles angles to go outside the range of ± 36 degrees and without crashing into the ends of the tracks.

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 software package; 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. The results reported in this paper used the adaptive mutation strategy: the rate of mutation is increased for parents that are more similar. Genitor uses a selection bias on the linear ranked gene pool to select genes for crossover. The selection bias was 2.0, indicating that the best individual in the population was twice as likely to be selected for crossover as the mean individual in the population.

The networks used by Wieland had a fixed number of fully connected hidden units and as many inputs as specified by the specific problem. In addition to

Wieland's inputs, one more input unit with a fixed value of 1.0 was added as a bias. One of the hidden units is arbitrarily chosen to be the output unit. Since the network is fully connected, it does not matter which unit is selected to be the output, as long as the same unit is used in each trial. The classic 1-pole 4 input problem was solved using a single hidden unit and 5 input units, including the bias unit. The 1-pole 2 input problem was solved with 3 hidden units and 3 input units, including the bias input. The 2 pole, six input problem was solved using 5 hidden units and 7 input units, including the bias unit. We were unable to find an architecture to solve the 2-pole problem without velocity information. The networks we used are slightly different from those used by Wieland, who did not use a bias input. Wieland used 6 neurons to solve the 1-pole 2 input problem and 10 neurons to solve the 2 pole, six input problem. Wieland does not report a solution to the 2-pole problem without velocity information. The network sizes and genetic algorithm parameters that we used were chosen after many trials and represent the best performance that we could obtain. The results of the experiments are reported in Table 1, and will be compared to those obtained with cellular encoding.

3 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 [3] [2]. Each cell has an input site and an output site and can be linked to other cells with directed and ordered links. A cell also possesses a list of internal registers that represent local memory. The registers are initialized with a default value, and are duplicated when a cell division occurs. The registers contain neuron attributes such as weights a threshold value. The graph transformations can be classified into cell divisions and modifications of cell registers.

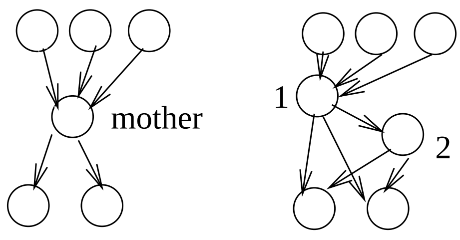

Figure 1: Illustration of the CPO division (a) Before the mother cell divides (b) The two child cells.

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 denoted `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 denoted `PAR` both child cells inherit both the input and output links from the parent cell. Hence, each link is duplicated. The child cells are not connected. Examples of these types of cell division are given by Gruau and Whitley [3]. In general, a particular cell division is specified by indicating for each child cell which link is inherited from the mother cell. In this paper, we used another division called `CPO` illustrated in figure 1. It is similar to a sequential division, except that the output links are duplicated in both child cells.

Before describing the instructions used to modify cell registers it is useful to to describe how a neural network unit performs a computation. The default value of the weights is 1, and the bias is 0. The default transfer function is the identity. Each neuron computes the weighted sum of its inputs, applies the transfer function and then multiplies the result by a coefficient which is 1 by default. See the figures 3 and 4 for examples of neural networks. The instruction `MULT` x sets that coefficient to x. It is worth mentioning that in [8], we did not used a multiplicative coefficient, and imposed weights to scale between -1 and $+1$. As a result, the genetic program could not directly produce ANNs with no hidden units. It produced one network with 19 units; however, this network could be simplified to an ANN with no hidden units, since all the neurons were computing identity function (see section on the analysis of the ANN structure) or they add a constant output. In fact, the encoding generated excess neurons only to obtain weights greater than 1. By using a multiplicative coefficient, the encoding can produce directly ANNs without hidden units (assuming no hidden units are required.) The ANN's computation is performed with integers; the activity is coded using 12 bits so that 4096 corresponds to activity 1. The instruction `SBIAS` x sets the bias to $x/4096$. The instruction `STEP` sets the transfer function to the clipped linear function between -1 and $+1$. 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 parameters are used to set the k weights of the first input links. If a neuron happens to have more than k input links, the weights of the supernumerary input links will be set by default to the value 256 (i.e., $\frac{256}{256} = 1$).

The cellular code is a *grammar-tree* with nodes 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 to the left or to the right subtree. After cells terminate development they lose their reading-heads and become neurons.

The order in which cells execute graph transformations 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 also used the control program symbol `PROGN`. 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.

Consider a control problem where the number of control variables is n and the number of sensors is p. We want to solve this control problem using an ANN with p input units and n output units. At the beginning of the development the initial graph of cells consists of p input units connected to a reading cell which is connected to n output units. The input and output units do not read any code, they are fixed during all the development. In effective these cells are pointers or place-holders for the inputs and outputs. The initial reading cell reads at the root of the grammar tree. It will divide according to what it reads and generate all the cells that will eventually generate the final decoded ANN.

3.1 Syntactic Constraints and Genetic Operators

We used a BNF grammar as a general technique to specify both a subset of syntactically correct grammar-trees and the underlying data structure. The default data structure is a tree. 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 in the final neural network architecture. For the `list` and `set` data

```
    <axiom> ::= <nn>

<nn>[0..8]  ::=  PAR(<nn>)(<nn>)
               | CPO(<nn>)(<nn>)
               | SEQ (<nn>)(<nn>)
               | <attribute>

<attribute> ::=
          (PROGN : set[0..4] of
              (WEIGHT: list[8..8] of
                  (integer[-255..+255]))
              (SETMULT(integer[-8..+8]))
              (SBIAS(integer[-4096..+4096]))
              (STEP) )
```

Figure 2: Syntactic constraints used to restrict the space of possible solutions

structure we set a range for the number of elements in these structures. 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. So for example, the rule

$$< A >::= (list[2..2]of(0)(1))$$

generates the trees ((0)(0)), ((0)(1)), ((1)(0)), ((1)(1). The rule

$$< A >::= (set[2..2]of(0)(1))$$

generates only the trees ((0)(1)) and ((1)(0)).

Figure 3.1 shows the syntactic constraints. The nonterminal <nn> is recursive. It can be rewritten recursively between 0 and 8 times. Each time is it rewritten recursively, it generates a division and adds a new ANN unit. Thus the final number of ANN units will be between 1 and 9. Note that in this particular case, the size of the phenome is proportional to the size of the genome, therefore constraints of the grammar in fig. 3.1 which controls the size of genome result directly in constraints on phenome growth which controls the phenome size.

The nonterminal <attribute> is used to implement a subset of four possible specializations of the ANN units. The first 8 weights can be set to values between -1 and $+1$. The multiplicative coefficient can be set to a value that ranges from -8 to $+8$, and the bias is set to a value between -1 and $+1$. The transfer function can be set to a clipped linear function instead of the default transfer function. Since the lower bound on the `set` range is 0, there can be 0 specializations generated, in which case the ANN unit will compute the sum of its inputs and apply the identity function. Because the upper bound on the `set` is 4, all the 4 specializations can be generated. In this case, the neuron will make a weighted sum, subtract the bias, apply the clipped linear function, and multiply by a coefficient.

Crossover. Crossover must be implemented such 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 simple mechanism ensures the closure of the crossover operator with respect to the syntactic constraints.

Crossover between two `trees` is the classic crossover used in Genetic Programming as advocated by Koza [5], where two subtrees are exchanged. Crossover between two `integers` is disabled. Crossover between two `lists`, or two `sets` 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 `tree`, `list` and `set` nodes the mutation rate is 0.05, while for the `integer` node it is 0.5.

4 Comparative Experiments for Cellular Encoding and Direct Encoding

We used a parallel Genetic Programming Algorithm with 32 subpopulations. A more complete description of this algorithm is given by Gruau [4]. The parallel genetic program combines the advantages of the massively parallel model [1] and the island model [6]. 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 a paragon with 400 processors. Each of the 32 subpopulations is 128 individuals and resides on one processor, so the total population is 4096.

4.1 The Evaluation Function

Defining the fitness function can have a critical impact on the quality of results since it is mainly the fitness which shapes the solution. Our initial experiments used a fitness based on the number of time steps the pole could stay balanced [8]. Our early experiments indicated that learning to control the system for a single random start state was easy to achieve, but that the resulting generalization was poor. To alleviate this problem the neural network had to balance the pole starting from 11 initial positions, for 1000 time steps in each of those positions. Those initial positions were chosen at the frontier of the domain where it seemed impossible to recover. This enabled us to generate recurrent ANNs to balance one pole on a cart, using only the cart position and the pole angle as input. Our assumption was that the recurrent connections makes it possible to remember the previous position of the cart and the pole, and to compute the speed. More recently, we discovered that this hypothesis does not appear to be true. Using the same setting as the one described in [8] we evolved nonrecurrent networks to see whether the recurrent links were important. The genetic program could generate an ANN without hidden units (no hidden units) that could balance the pole for the 11 initial positions and for over 100,000 time steps. Obviously this network does not compute the speed! It is able to balance the pole and the cart by keeping them swinging between left and right. The amplitude of the oscillation is subjected to very small increase or decrease over the time. This is due to round off errors. Starting from the resting position, 100,000 time step is not sufficient for the amplitude of the oscillation to reach the fatal value where the system crashes, although the system will crash sooner or later. To force the ANN to compute the speed, we included in the fitness a term which penalizes oscillations.

We now present the new fitness function. We used a single initial position where all the variables are zero except the big pole angle whose value is θ. Since the fitness is more complicated, a single fitness case is sufficient to provide enough precision to allow for good generalization. We used $\theta = 18$ degrees for the one pole problem and $\theta = 4.5$ degrees for the two pole problem. The cart and poles where simulated for 1000 time steps. The fitness is the weighted sum of two fitnesses. The first fitness is the number of time steps the system could stay balanced, divided by 1000. The second fitness is 0 if the pole was balanced less than 100 time steps, else it is a constant K multiplied by the inverse of the sum of the absolute values of all the four normalized variables { cart position, cart velocity, big pole angle, big pole velocity }, over the last hundred time steps. The constant K was chosen to be 0.75. Obviously, if the pole is swinging, then the angle and velocities are going to be high in absolute value, and the inverse of the sum is going to be very low. Thus this second fitness directly penalizes swinging. By penalizing oscillation the fitness function is also biased toward finding neural networks that can return the pole to a centered upright position. If the second fitness is greater that one, then it is set to one. The total fitness is 10% of the first fitness plus 90% of the second fitness. On all the runs we did two generalization tests:

- We balanced the pole for 100,000 time steps.
- We checked to make sure that the ANN was able to bring the cart and the pole back to a centered upright position when started from the initial position in the training set.

Note that the result on the fixed architecture are reported with the old fitness function; this weakens the comparison.

4.2 The Single Pole Problem

We experimented on two 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 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. Thus, the magnitude of the force that is applied smoothly varies between -1 and +1, as opposed to the bang-bang control. Again following Wieland, we experimented with a variant of the problem where only 2 inputs are given, (pole angle and cart position) and an ANN with a parallel dynamic must be used to compute the relevant velocity information. This problem was also solved using a continuous control strategy.

For the parallel dynamic case, the initial activation of all neurons is set to 0. For each computation the network is relaxed three times in parallel before its output is considered.

	CE code		Direct code	
Problem	Learning	Gen.	Learning	Gen
1-pole 4 inputs	1400	250	142	466
1-pole 2 inputs	21,000	225	1276	206
2-poles 6 inputs	34,000	569	410	510
2-poles 3 inputs	840,000	300	—	—

Table 1: Results of the experiments. The learning column reports the average number of evaluations needed to find the solution, the Gen. column report the performance in generalization

We ran a generalization test based on that used by Whitley et al.[7] 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 ANN 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 (Direct Encoding) using our implementation of Wieland's methods are reported in Table 1. Each result represents an average over 30 experiments.

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.

Cellular Encoding needs more evaluations than the fixed architecture coding to solve these problems. However, it should be noted that the numbers reported for the fixed architecture are the best that we could obtain after many trials with different architectures. Overall, it took a great deal of human intervention to get the fixed architecture to solve the problems rapidly and reliably. At the beginning, fixed encoding needed more than 200,000 evaluations to solve the one pole problem with 2 inputs and no velocity, or the 2 pole problem with 6 inputs (including velocity). The time spent finding parameters for quick and reliable convergence is not shown in the table. Also, we were unable to find a fixed architecture solution to the 2 pole, 3 input problem without velocity information. Cellular Encoding did find a solution to this problem.

4.3 Two Poles

Table 1 also reports results for the experiments involving two poles. The ANNs 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.

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 the same as the one used in [8]. The angle varies between ± 3.6 degrees, and the pole velocity between ± 8.6 degrees per second. We then used 625 test cases using 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 569 of the 625 initial settings using these reduced ranges.

Finally, we ran an experiment with two poles, but without using information about pole or cart velocities. The ANN had three inputs: the cart position, the big pole angle, and the small pole angle. We run a single experiment using a larger number of processors (64 processors) and a larger population per processor (256) for a total population of 16,384. A solution was found in 51 generations. For this solution, if the big pole is set to 4.8 degrees, the ANN is able to bring the two poles and the cart back to the null position after 200 time steps. We were not able to solve this problem with the fixed architecture encoding.

5 Analysis of the ANN Structures

One Pole For the four input problem where the velocity is given the genetic program found solutions with no input units. It thus automatically generated minimal networks.

Figure 3 (a) shows an example of neural networks generated by the genetic program for the one pole problem without information about velocity. All these ANNs show the same features which enable them to compute the speed. First, the minimum path between the input and the output unit is smaller than three. The reason for this is that the ANN is relaxed three times before its output is considered. If the minimum path was greater than three, the current state of the cart and the pole could not affect the control decision taken by the ANN until the next time step. Second, some intermediate neurons are used to retain information about the previous state. As a result there are other possible paths between the input and the output units with length strictly greater than three. When the information flows through these neurons, it takes four or more 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. In the ANN represented in figure 3, there are 6 hidden units; three of them use the clipped linear transfer function. Their transfer functions are piecewise linear, and are equal to the identity around the origin. Those three neurons are never saturated. That is, the net input always stays in the linear part of the transfer function. Therefore we can replace the transfer function by the identity function. The ANN was simplified by hand by merging together the ANN units that compute the identity. The network is mathematically equivalent to the ANN shown in figure 3 (b), which has to be relaxed only 2 times and has only two hidden units. We tested this solution to verify that it was working in the same way. This ANN is simple enough that its behavior can be fully understood. If the pole is far from equilibrium, the variables (cart position and pole angle) are high, the two hidden units are saturated, and no information can be used about velocities. The resulting control is not smooth and one can see the pole swinging very violently. When the variables are smaller, then the two hidden units are not saturated, they can compute the velocities, and the pole is very quickly brought to rest. Thus there are two modes of computation which can clearly be distinguished. For a test case where the pole angle is set at 18 degrees, this is far from equilibrium and the pole swings violently. After hundred time steps it changes its mode and suddenly the pole is smoothly brought back to rest.

Two Poles with velocity. About 80% of the neural networks generated by the genetic program to control the two poles had no hidden units. The remaining 20% had one or two hidden units that could be simplified. 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 GP that had 16 hidden units. The threshold of the output unit is 0, and the weights of one solution (after scaling by 1/256) are -0.16, -0.23, -1.36, -2.9, 1.47, 0.87. 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 car to the left (resp.

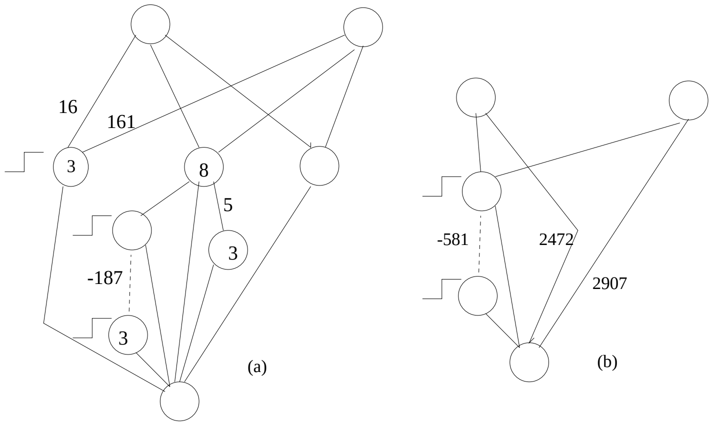

Figure 3: (a) An ANN generated for the one pole without velocity information. Inputs are shown at the top of the network diagram. We specify the ANN unit clipped linear transfer function using a schematization of the graph of this function. Inside the circle we represent the multiplicative coefficient when it is different from 1. The bias are not represented, because they are all 0. We also represent the weights when they are different from 256. (b) This is the equivalent ANN obtained after simplification. It has now only 3 hidden units.

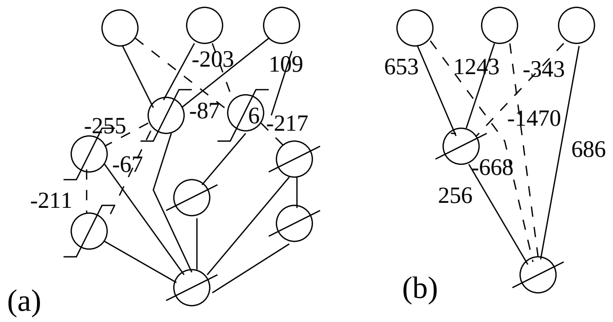

Figure 4: (a) ANN found by the GP for the balancing of two poles. It has seven hidden units. (b) The ANN (b) is mathematically equivalent to the ANN (a) it has only one hidden units.

right). This at first seems counterintuitive, because it is going to make the big pole bend down even worse. But then, weight 5 and weight 6 are positive; when the small pole makes an angle bigger that 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.

Two Poles without velocity. Figure 4 shows a mathematical equivalent ANN that has only one hidden unit, and must be relaxed only one time instead of three. We picked a solution found by the genetic algorithm that has seven hidden units, which are all computing the identity. Therefore they can be all simplified. We need to keep one hidden unit however to maintain the delay that makes it possible to compute the speed.

6 Conclusions

This paper compares a new version of Cellular Encoding with direct encoding on 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. We compared Cellular Encoding with a Direct Encoding method that produces architectures with a fixed size.

Cellular Encoding required more evaluations to solve the problems than the Direct Encoding method. However, the large amount of human effort, intervention, and experience required to find the proper architecture for Direct Encoding makes it less attractive than Cellular Encoding. The solutions found by Cellular Encoding are smaller than 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. In the cases studies here these are often architectures without hidden units or architectures that can simplify to one or two hidden units. This provides 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 two poles without velocity information represents a more complex control task.

This problem could be solved using cellular encoding, which evolved networks using only one or two hidden units. This problem could not be solved however with the fixed architecture coding.

Acknowledgements

This work was supported in part by NSF grant IRI-9312748 and by a TMR european fellowship to Frédéric Gruau. We thank San Diego Super Computer Center for providing an access to their Paragon.

Bibliography

[1] Collins, R. and Jefferson, D. (1991) Selection in Massively Parallel Genetic Algorithms. *Proc. 4th International Conf. on Genetic Algorithms*, Morgan-Kaufmann, pp 249-256.

[2] Gruau, F. (1994) Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm PhD Thesis, Ecole Normale Supérieure de Lyon. anonymous ftp: lip.ens-lyon.fr (140.77.1.11) pub/Rapports/PhD PhD94-01-E.ps.Z (English) PhD94-01-F.ps.Z (French)

[3] Gruau, F. and Whitley, D. (1993) Adding learning to the the cellular developmental process: a comparative study *Evolutionary Computation* 1(3):213–233.

[4] Gruau, F. (1994 Automatic Definition of Modular Neural Networks, *Adaptive Behavior* 3(2) 151–184

[5] Koza, J. (1992) Genetic Programming: On the Programming of Computers by Means of Natural Selection, Cambrige, MA *the MIT Press.*

[6] Muehlenbein, H., Schomish, M. and Born, J. (1991) The parallel genetic algorithm as function optimizer, *Parallel Computing* 17:619–632.

[7] Whitley, D., Dominic, S., Das, R. and Anderson, C. (1993) Genetic reinforcement learning for neurocontrol problems, *Maching Learning* 13:259–284.

[8] Whitley, D., Gruau, F., and Pyeatt, L. (1995) Cellular Encoding Applied to Neurocontrol *Proc. 6th International Conf. on Genetic Algorithms*, Morgan-Kaufmann, pp 460–467.

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