Multiple Interacting Programs: A Representation for Evolving Complex Behaviors

Peter J. Angeline
Natural Selection, Inc.
Vestal, New York
angeline@natural-selection.com

ABSTRACT

This paper defines a representation for expressing complex behaviors, called *multiple interacting programs* (MIPs*)*, and describes an evolutionary method for evolving solutions to difficult problems expressed as MIPs structures. The MIPs representation is a generalization of neural network architectures that can model any type of dynamic system. The evolutionary training method described is based on an evolutionary program originally used to evolve the architecture and weights of recurrent neural networks. Example experiments demonstrate the training method's ability to evolve appropriate MIPs solutions for difficult problems. An analysis of the evolved solutions shows their dynamics to be interesting and non-trivial.

1.0 Introduction

Artificial neural networks are used extensively for learning and representing complex behaviors. The typical neural network is composed of a set of simple computational units that interact in a specific manner to produce a global behavior. The chief appeal of this representation is that it can be trained to perform many tasks that would be too difficult or time consuming to program by hand.

The popularity of neural networks increased rapidly with the introduction of a particular supervised learning technique: *back propagation of error* or *backprop* for short [1][2]. Backprop implements a gradient descent search on the representation of a neural network to minimize the error between a network's computed outputs and the desired outputs [1]. Figure 1 shows an example of a feedforward neural network and the form of computation performed at each of its units.

As shown in the figure, each non-input unit applies the same *activation function* to its summed input to determine its output value. The activation function selected determines representational properties of the network [2].

In order to implement a gradient search, conditions must be placed on the activation function to ensure that a gradient is defined for the search space. In particular, for backprop to be applicable, the activation function must be non-linear, non-decreasing, and differentiable everywhere ([1] p. 324). Similar limitations are required for all conventional neural network training methods that use local search space information to determine updates for weights in the architecture.

Given the general applicability of neural networks and backprop's ease of implementation, the computational constraints placed on individual units via the activation function have become assumed into the standard network definition. These constraints translate into modeling limitations for the architecture. Exactly what limitations are imposed by the constraints of gradient method are not well understood. While it has been shown that feedforward neural networks are capable of representing any computable function given a sufficient number of hidden units [3], in cases where the activation function does not correspond to the desired behavior, a neural network will require an inordinate number of units to model the behavior and will not generalize appropriately for the task. It would be preferable to fit the form of each unit's computation to the role it performs in desired behavior. Such an architecture would be more concise and would generalize more consistently with the task at hand.

Evolutionary computations have been extensively applied to the training of neural networks ([3] - [11] and many others). [10] describes GNARL, an evolutionary program ([12][13]) that evolves both the architecture and weights of recurrent neural networks. In [10] it is argued that since GNARL does not rely on gradient information, the computational constraints typically placed on individual units through the activation function can be removed, although no demonstration of this is provided. One of the objectives of the present paper is to demonstrate a form of that claim.

This article introduces a representation for modeling complex behaviors and an evolutionary training algorithm for constructing instances of this representation that implement desired complex behaviors. Multiple Interacting Programs (MIPs) represent behaviors using a set of independent “programs” that interact to realize a global behavior. Each program in a given set is unique and stored in the form of a parse tree (see Figure 2), a structure extensively evolved in genetic programs ([14][15]).

The following article is structured as follows. The following section describes the MIPs representation and the evolutionary program used to train it. Section 3 describes several experiments that demonstrate the training methods ability to create appropriate behaviors using this representation. Section 4 discusses the results shown in Section 3 while Section 5 gives some conclusions and suggests future work.

2.0 Multiple Interacting Programs

Multiple Interacting Programs (MIPs) combine the flexibility of a genetic program with the organization of a neural network. In short, a MIPs individual is virtually equivalent to a neural network where the computation performed at each unit is replaced with an independent evolved equation (c.f. Figure 1). Each unit’s equation is represented as a parse tree (see Figure 2) that is evaluated to determine the output of the unit. Consequently, the activation function for each unit is uniquely evolved to suit the role of the unit in producing the desired behavior.

While some researchers have investigated combinations of neural networks with genetic programming ([16] - [18]), and still others have investigated individuals containing multiple parse trees ([19] - [21]), MIPs combines these two ideas into a distinct representation that more closely follows the spirit and organization of neural network architectures. Accordingly, in this paper, an individual’s evolved set of parse trees is referred to as a *MIPs net*.

2.1 Formal Definition of MIPs

Formally, a MIPs net, M, is defined by a 4-tuple $M = (\Sigma, \alpha, \theta, \kappa)$ where the symbols denote the following sets:

- Σ is the set of functions associated with the problem. This corresponds to the *function set* in genetic programming (Koza 1992) and may include "protected" versions of functions such as "%" which is division that returns 0.0 if the denominator is equal to 0.0.
- α is the set of input parameters given by the problem. This corresponds to the set of input nodes for a neural network and the terminal set in genetic programming. It may include special terminals like $\Re$, called the *random numeric terminal,* which when selected returns a randomly generated real value drawn uniformly from a user-defined range.
- θ is the set of output units associated with the problem. This is equivalent to the set of output nodes in a neural network. Note that there can be several output units in a MIPs net. This is distinct from the single output typically returned from a genetic program.
- κ is the set of intermediate units to be used in constructing solutions for the problem. These units are equivalent to the set of hidden nodes in a neural network.

In a MIPs net, distinct parse trees are associated with each intermediate and output unit defined for the individual, with the number of parse trees in an individual given by $|\theta \cup \kappa|$. Each parse tree, called the unit's *update expression,* stipulates the value that unit takes on when next evaluated. Update expressions are expressed in terms of Σ, α, θ, and κ.

2.2 Evaluation of MIPs Nets

Execution of a MIPs net proceeds as follows. Without loss of generality, assume a MIPs net M with output units identified as $T_0 \ldots T_n$, where $|\theta| = n + 1$. Also assume the intermediate units in M are identified as $T_{n+1} \ldots T_m$ and $|\kappa| = m - n$. An execution of M evaluates each update expression in turn, starting with the update expression for T_m and ending with T_0. The value returned from the update expression defines the output value for the unit.

Exactly when a unit's value is replaced depends on the type of evaluation performed. When using *asynchronous updates,* a unit's value is updated immediately after evaluation of its update expression. This is a popular form of evaluation used with neural network architectures. S*ynchronous* *updates* replace each unit's values simultaneously after evaluation of all update expressions. Updated unit values are then available on the following evaluation of the MIPs net. This update scheme is more amenable to modeling dynamical systems expressed as difference or differential equations. To date, neither of these update methods appears to be preferable to the other.

Architecturally, there are two broad classes of MIPs nets that can be defined in a manner similar to neural networks. Define a *feed-forward MIPs net* such that the following holds:

$$\Gamma(T_i) = \Sigma \cup \alpha \cup \{T_{i+1} \ldots T_m\} \quad \text{(EQ 1)}$$

where $\Gamma(T_i)$ is the language used to construct the update expression associated with unit T_i. Equation 1 limits a given unit's parse tree to reference only units with a higher ordinal number. Given the order of execution, this excludes units that have not been evaluated, which prevents any circular references and holds true to the name "feed-forward." This definition is consistent with that of feed-forward neural networks.

Define a *recurrent MIPs net* such that a unit's update expression can refer to any unit in the individual. Formally:

$$\Gamma(T_i) = \Sigma \cup \alpha \cup \theta \cup \kappa \quad \text{(EQ 2)}$$

which permits a unit to reference the value of both units evaluated before and after the itself thus allowing circular references. Such organizations are necessary when state information is required to accurately model a complex behavior. Again, this definition is consistent with the definition of a recurrent neural network. In the experiments below, only recurrent MIPs nets are investigated since they are computationally more interesting.

A MIPs net is most clearly displayed as a system of equations rather than a set of parse trees. For instance, an instance of a recurrent MIPs net of the form $\Sigma = \{+, /, -, *\}$, $\alpha = \{\mathrm{In}, \Re\}$, $\theta = \{T_0\}$, and $\kappa = \{T_1, T_2\}$ would be:

$$
\begin{aligned}
T_2 &= T_2 + 0.3745\mathrm{In}(T_0 + T_1) \\
T_1 &= \frac{\mathrm{In}}{T_2 + 0.97831}(T_0 - 0.11345) \\
T_0 &= (\mathrm{In} + T_1)\mathrm{T}_0
\end{aligned}
\qquad \text{(EQ 3)}
$$

where $\Re$ is the random numeric terminal described above. Equation 3 shows the update expressions for each of the intermediate and output units in the MIPs net. As already described, this MIPs net is evaluated starting with T_2 and ending with T_0, which is the designated output unit. The reference in the update expression for T_2 to the current values of T_1 and T_0 denote recurrent links. Equation 3 also demonstrates a form of self-reference in the update expression for T_0. When recurrent links are permitted, each unit must be initialized to a user-defined value prior to the first evaluation.

Often when evaluating a system of equations, several iterations are performed prior to examining the output values. This implicit iteration permits a refinement of the output values before they are applied and permits the expression of more complex behaviors with a smaller number of equations. Multiple iterations increase the length of the recurrent paths that contribute to the output values to the number of iterations performed rather than just a single update.

2.3 Training MIPs Nets

A MIPs net is trained using an evolutionary program [12][13] adapted from a previous evolutionary program, called GNARL, that evolved recurrent neural networks [10]. The purpose of GNARL was to investigate the evolution of both the architecture and weights of conventional recurrent neural networks. The difference between GNARL and the training method described here is that rather than manipulating weights, nodes, and links between nodes, the current mutation operations manipulate a set of parse trees forming a system of equations.

There are currently ten mutations used in the MIPs training algorithm. They can be grouped by what size structure they manipulate in an individual. At the finest grain are the mutations that modify individual elements in a single expression of an individual:

- *insert element* - select an element from one expression in the MIPs net and insert a new language element that is not a terminal above it making the current element its child. Select terminals for any additional required parameters.
- *delete element* - Select a language element that is not a terminal and remove it from the expression promoting its child to its position. If the element has more than one parameter then select one parameter to promote at random and delete the rest.
- *cycle element* - select an element and change it to another element with the identical number of arguments.

At a more intermediate level are mutations for entire subtrees of an expression:

- *swap subtrees* - select a subtree in an expression and exchange its position with one of its siblings.
- *replace subtree* - select a subtree in an expression and replace it by a randomly generated subtree.
- *delete subtree* - select a subtree in an expression and replace it with a terminal.

And finally, at the coarsest level are the mutations that operate at the tree level:

- *swap trees* - select two equations in the MIPs net and swap their order of evaluation. Repair any reference in the other update expression to refer to the same expressions as before.
- *add tree* - insert a new intermediate unit into the individual at a randomly selected position and generate a random parse tree for its update expression. Repair any references in the other update expressions so they refer to the same units as before the addition.
- *delete tree* - select an intermediate unit in the individual and remove it and its associated update expression. Replace all references to the deleted unit in the individual with a randomly selected terminal or unit reference. Repair all other unit references so they refer to the same units as before the deletion.
- *numeric terminal mutation* - Apply Gaussian noise with a user-defined variance to all numeric terminals within a selected tree. In the experiments below, the variance was set to 0.001

A number of the above mutation operations have been previously employed as complements to subtree crossover in certain genetic programs ([22] - [24]). Note that the *add tree* and *delete tree* mutation are responsible for increasing and decreasing the number of intermediate units in the MIPs net.

When creating an offspring from a parent, the number of mutations applied using a Poisson random variable with a user-defined mean, set to 4 for the experiments below. Given a value of *k*, a copy of the parent is made and *k* mutation operations are selected with replacement from the above set and applied in turn. The relative probability of selection for each mutation can be individually defined, however, in the studies below each mutation was equally likely except for *add tree* and *delete tree* which were set to zero.

Similar to genetic programming, a user-defined limit is placed on the number of language elements allowed in an individual. In MIPs, a maximum number of elements is defined for the entire system of equations only; the MIPs net is free to distribute its language elements among the expressions as needed. If a created child violates the maximum element limit then the child is discarded and the reproduction is nullified. A minimum number of language elements per update expression is also enforced to encourage that each unit to perform some computation.

Parents are selected using *probabilistic selection* ([13]). For each individual, *k* other population members are selected uniformly and a point is awarded to the individual for each of the *k* with worse fitness. For the experiments described below the number of comparisons was set to five. The population is then sorted using this derived score and the top *n%* are reserved as parents for the next generation. During reproduction, each parent creates a single offspring in turn, starting with the first parent, and iterating through all parents until once again filling the population.

2.4 Comparison to Automatically Defined Functions and Indexed Memory

[25] describes a variant to genetic programming, called *automatically defined functions* (ADFs), that on the surface may appear to be similar to MIPs nets. An ADF is a function, also represented as a parse tree, that is co-evolved with the main body of a genetic program. The main program can refer to the ADF multiple times within its computation. ADFs can be chained such that the main

body can call *ADF1* which can call *ADF2* and so on, with the hierarchical calling structure strictly enforced. When a call to an ADF is encountered during evaluation of a program, the ADF is executed much as a function call in a standard procedural language. The ADF's result is returned to the calling program as the value of the ADF's reference. Note that each time a given ADF is called its associated expression is evaluated anew.

The dissimilarities between ADFs and MIPs nets are significant. The chief distinction lies in the differences between the execution of a parse tree with ADFs and the execution of a MIPs net. As mentioned above, each time a reference to an ADF is encountered during evaluation, the ADF is reevaluated, potentially returning a distinct value each time it is called. In each update expression of a MIPs net, every reference to another unit's value returns the same value. Each update expression is evaluated only once per evaluation of a MIPs net. As stated above, a MIPs net is modeled after the evaluation of a dynamical system, in particular a neural network, while ADFs implement more of a modular programming approach.

MIPs nets are actually more similar to *indexed memory*, a genetic programming add-on that provides modifiable memory to an evolved program [26]. When using indexed memory, a bank of memory cells is associated with each individual and referenced using specially provided functions that read from and write to individual cells. Each memory reference must be explicitly constructed in the evolved program allowing evolution to develop memory organizations that assist in solving the task. The units in a MIPs net can also be viewed as a set of memory cells which are accessed through explicit references, although the similarity between MIPs nets and indexed memory ends there [28].

3.0 Experiments

In the following section, three experiments that construct MIPs nets to solve non-trivial problems are described. The first investigates the sunspot prediction problem and demonstrate that MIPs may find solutions to some problems more quickly than other methods for inducing behaviors. The second experiment demonstrates MIPs on a classic problem, the artificial ant, and shows that MIPs can find interesting novel solutions to established problems. The final experiment shows

MIPs solving a new difficult problem that involves the construction of a task specific memory. All of the following experiments assume a fixed number of intermediate units. For experiments involving a variable number of intermediate units in a MIPs net see [29].

3.1 Sunspot Time Series Modeling

Since the year 1700, the average number of sunspots observed each month has been recorded. This data is available on-line.[1] Figure 3 shows a plot of the average number of sunspots per year for the time period from 1700 to 1993. This data is a naturally occurring chaotic time series which makes it more appealing for experimentation than artificial chaotic time series, such as the Mackey-Glass equation, since the underlying generating model is unknown.

The objective of this experiment is to predict the number of sunspots that will be observed in the following year given a minimal number of data points from previous years. This problem was investigated previously using a genetic program ([22][23]). [23] gives results for this problem using a genetic program with some evolutionary programming modifications. In that study the population size was 250, the maximal size of the evolved trees was 50 nodes, the number of generations was 1000 and a total of 30 trials were performed.

Following [23], the MIPs net was defined as follows:

$$\begin{aligned} \Sigma &= \{+, -, *, \%, \sin, \cos\} \\ \alpha &= \{d_1, d_2, d_4, d_8, \Re\} \\ \theta &= \{Out\} \\ \kappa &= \{T_1, T_2\} \end{aligned} \qquad \text{(EQ 4)}$$

where *d1* is the number of sunspots observed in the year before the predicted year, *d2* is the number of sunspots observed two years before the predicted year and so on. It should be noted that the data was not separated into training and test sets making this a time series modeling task rather than a time series prediction task. Given that the objective is to minimize the prediction error of the evolved function, the fitness function is the sum of the squared prediction error for each data

1. The World Wide Web address is http://www.ngdc.noaa.gov/stp/SOLAR/SSN/ssn.html.

point (years 1708 to 1993). The actual fitness function negates the error to make this a maximization problem.

This problem was solved with MIPs using the same parameters as in [23]. The maximum number of nodes in an individual was set to be 50. The networks were updated synchronously and recurrent connections were allowed. Once the values for *d1*, *d2*, *d4* and *d8* were set, a MIPs net was executed three times before the value of the output unit was read. Thus, recurrent paths of length three or less could contribute to the computation if present in the network. The initial values of all units were set to 0.0 for each new set of input values.

In order to compare MIPs to the method investigated in [23], 30 trials were run. Figure 4 shows a comparison of the mean best-of-generation fitness per generation averaged over the 30 runs for both methods. Early in the run the genetic program appears to progress more quickly, however, the MIPs runs eventually equal and surpass the genetic program's performance. A t-test performed on the fitness of the best individuals found in each of the trials showed a significant statistical difference between the respective means (p<<0.01) in favor of MIPs.

The following network was evolved in one of the MIPs runs and achieved a fitness of -53362 for the training set:

$$
\begin{aligned}
T_2 &= \cos\left(\frac{d_4}{\mathrm{Out}}\right) \times d_1 \times 0.180054 \\
T_1 &= \cos\left((-0.922186 - \sin(\mathrm{Out} - d_2)) \times d_2\right) \\
\mathrm{Out} &= \frac{\left(T_2 - \left(-\left(-\frac{(-0.363858)}{d_8}\right)\right) + (-0.692829) \times -d_1\right)}{\left(\cos\left(-\frac{d_8}{(d_2 + \cos(\sin(\cos(0.56601))) \times d_8)}\right)\right)}
\end{aligned}
\qquad \text{(EQ 5)}
$$

where $a \times b$ denotes multiplication for clarity of presentation. The equations are shown without simplification in order to accurately portray the style of the solution found. Note that of the two intermediate units in this network only one is used in the computation of the output unit. Consequently, T_1 can be removed from the network entirely. The equation for T_2 refers to the result of Out and consequently implements a recurrent link.

3.2 Artificial Ant Problem

The goal of the artificial ant problem is to create a controller for a mechanical ant such that the ant eats all of the food available in the environment within a predetermined time limit. The environment used in this experiment is the 32x32 toroidal grid shown in Figure 5 with the dark gray squares denoting positions that contain food and light gray squares denoting positions along the shortest path through the environment. The ant has a single sensor in front that can detect when food is ahead. The controller must execute one of four actions on each time step: *move forward*, *turn left*, *turn right* or *noop*. Each of the four actions requires a single time step to accomplish.

[27] first investigated a version of this problem, using a slightly different trail of food than shown in Figure 7, when comparing the evolvability of recurrent neural networks to finite state machines both encoded as bit strings. The neural networks evolved in [27] used two inputs, five hidden units and four output units. The two input units were designated as "Food" and "No Food" meaning that when food was in front of the ant the "Food" unit would be set to 1.0 and the "No Food" unit would be set to 0.0. The four outputs of the network denoted the four possible commands for the controller. The input nodes were connected to all other nodes except the input nodes while all other nodes were connected to all all but the input nodes.

[10] also investigated this problem using the same unit assignments as in [27] but allowed the number of hidden units and the interconnectivity to be evolved. The resulting network had a complex interconnectivity between nine hidden units and the four output units and incorporated interesting dynamics to solve the problem. [15] investigated a version of this problem using the path of Figure 7 and a high-level procedural language for representing solutions.

For the present experiment, the task language was chosen to be similar to the computations found in the neural networks investigated in [27] and [10] in order to demonstrate MIPs solving a problem involving multiple outputs. The form of the MIPs nets evolved for this problem were as follows:

$$\begin{aligned} \Sigma &= \{+, -, *, \%\} \\ \iota &= \{\mathrm{Food}, \mathrm{NoFood}, \Re\} \\ \theta &= \{\mathrm{Move}, \mathrm{Left}, \mathrm{Right}, \mathrm{Noop}\} \\ \kappa &= \varnothing \end{aligned} \tag{EQ 6}$$

where Food and NoFood are the input parameters to the network following [23]. Note that only output nodes are included in the MIPs net and no intermediate nodes. This forces a solution to reuse the state of the various output units must also encode any state information required to traverse the path of food within the time limit.

As in [27], the command executed at a time step is the one associated with the unit having the largest value. A total of 400 time steps were allowed per individual with the fitness calculated as the amount of food eaten within the time limit. The initial values of the units were initialized to 1.0 before the first input and then not reinitialized for the length of an ant's run. This allowed the state information encoded in the output units to contribute to subsequent computations. Each evaluation used a single execution cycle and synchronous updates. Other parameters were as follows. The population size was 1000 individuals with 100 parents saved each generation and used to create nine additional offspring for the next generation. The maximum number of nodes allowed in an individual was set to 100.

In one run, the following network scored the maximum of 89 in generation 92:

$$\begin{aligned} \mathrm{Noop} &= \mathrm{Left} \times \mathrm{NoFood} + \mathrm{Left} \\ \mathrm{Right} &= \frac{\mathrm{Left}}{\left(\frac{\mathrm{Right}}{\mathrm{Right}}\right)} - \frac{-1.434651}{\mathrm{NoFood}} \\ \mathrm{Left} &= \frac{0.852099}{\mathrm{NoFood}} \\ \mathrm{Move} &= \frac{\mathrm{Left}}{-0.198859 + (\mathrm{Left} - -0.200719)} \times \frac{-0.696450(\mathrm{Right} - \mathrm{Move})}{\mathrm{Move}} \end{aligned} \tag{EQ 7}$$

where $a \times b$ denotes multiplication for clarity of presentation. Again the equations have not been simplified. Note that only the NoFood parameter is used in the solution.

To analyze the dynamics of this solution, the activations of the four output nodes were plotted for an environment that contained all food and a second environment that contained no food. Figure 6 shows the activations of the four output units over 50 time steps when food is always present. Note that the units quickly settle down to constant values such that the Move node is always selected. Given that food is always present, continuously moving forward is a logical strategy. The dynamics of this network are more interesting for the case when no food is ever present in the environment and the controller must search the space. In this case, the Noop, Left and Right nodes again take on constant values, with Right being greatest, while the Move unit displays a complex self-similar variation through time, as shown in Figure 7. When the activation of the Move unit falls below 2.286750, the Right unit's activation is largest and a right turn is executed by the ant. The graph shows that when searching for food, the limit behavior of this controller twice executes a Move followed by four consecutive Rights and then a Move followed by six consecutive Rights. This self-similar patter is evident through 2000 iterations of this controller.

3.3 5-Bit Reverser

The 5-Bit Reverser problem was designed to test the ability of MIPs to create a task-specific memory. Here, the problem is to memorize then reconstruct a 5-digit binary string in reverse order. Consequently, a MIPs net must store the values of all five binary digits one at a time and then reproduce them one at a time in reverse order on its output unit. To do this, it must consecutively encode the sequence into a set of unit activations and then decode those activations into the reversed sequence.

After the presentation of the last digit, null symbols are placed on the input unit to signal that the reverse string should be returned. Thus the sequence of inputs (*1, 0, 0, 1, 0, null, null, null, null, null*) should produce the sequence (**, *, *, *, *, 0, 1, 0, 0, 1*) where "*" is a "don't care" symbol.

The single real-valued input parameter takes on a value of 1.0 when the next bit in the string is a "1", a value of -1.0 if the next bit is a "0" and a value of 0.0 when the *null* symbol is input. Positive activation for the output unit is interpreted as a bit value of "1" while negative values are interpreted as a bit value of "0".

The definition of the MIPs nets evolved for this problem were as follows:

$$\begin{aligned} \Sigma &= \{+, -, *, \%, \sin, \cos, \text{iflte}\} \\ \alpha &= \{I, \Re\} \\ \theta &= \{Out\} \\ \kappa &= \{T_1, T_2, T_3\} \end{aligned} \qquad \text{(EQ 8)}$$

where *iflte* is the conditional function "If-Less-Than-Or-Equal" a four argument function that returns the third argument if the first is less than or equal to the second and returns the fourth argument otherwise. This function is commonly used in genetic programming ([15]).

A network was allowed no more than 100 language elements distributed over the four units. A population of 2000 individuals was evolved with 200 parents saved and used to create nine additional offspring in the next generation. A network was evaluated once and the units were updated synchronously. Fitness was set to be the number of bits correctly returned when tested over all 32 bit strings of length five. Synchronous updates were used and a MIPs net was evaluated as described above.

The following network achieved a maximum of 160 points at generation 275:

$$\begin{aligned} T_3 &= \text{iflte}(\text{Out}, -0.089242, I, \sin(I)) \times 0.710937 + \text{Out} - \sin\left(T_1 \times \left(\sin\left(\frac{T_1}{\text{Out}}\right) - I\right)\right) \\ T_2 &= T_1 \times T_1 \\ T_1 &= \sin(\sin(\cos(0.457164))) \\ \text{Out} &= T_1 \times T_3 \end{aligned} \qquad \text{(EQ 9)}$$

First note that the T_1 and T_2 units have a constant activation and that T_2 is not referenced by any other unit in the network. Consequently, unit T_2 can be removed the without altering the computation. Since T_1 is a constant always equal to 0.70445193, the state of the network can be described completely by the activations of units T_3 and Out. Consequently, the activations of these two nodes after the fifth bit is processed, serves as an encoding of that binary sequence that this MIPs net can decode into the reverse sequence.

Figures 8 and 9 show comparisons of the encoding and decoding for two similar five bit sequences using the MIPs net in Equation 9. Figure 8 shows the encoding and decoding of the binary strings 00000 and 10000 which differ in their first bit as presented. Interestingly, as Figure 8 shows, the Out and T_1 units take on very similar values just prior to decoding both strings, however, the slight difference between the encodings is enough so that their final output values are on opposite sides of the Out = 0.0 axis. In Figure 9, the encoding and decoding of the sequences 00000 and 00001 are compared. Here the two sequences are identical for the first four bits and then differ in their final bit. Given only a difference in their final bits, their respective encodings end up in very different positions in the space. Subsequent decoding shows very different paths traversed by the network when reconstructing the reversed strings.

Figure 10a shows the encoded positions for all 32 five bit binary sequences. The clustering of the encoded positions is based on the content of the binary sequences using the first bit presented as the lease significant bit. Figure 10b shows the space filling curve formed when the encoded points are connected in numerical order, assuming the first bit is the least significant. This elegant solution employs an efficient recursive packing of the encoded strings into the two dimensional space and suggests that similar MIPs networks, using the same functions and terminals, could represent longer binary strings and decode them accurately.

4.0 Discussion

The experiments above demonstrate several important points. First, the results prove that MIPs nets are evolvable and can be trained to represent complex behaviors. Next, the evolutionary program proposed is able to manipulate populations of MIPs nets to induce specific complex behaviors. In addition, the experiments validate the claim made in [10] that the computational constraints a gradient descent training algorithms places on a neural network can be removed when an evolutionary computation is used for training. This fact permits the generalization of neural network architectures into systems of arbitrary interacting equations and frees the training method to construct task appropriate computations for each node in the network.

The sunspot experiment shows that in some cases evolving a MIPs net will produce statistically significantly better results than evolving a single genetic program under similar conditions. While the experiment does not show that MIPs will be superior in all cases when compared to genetic programming, it does clearly indicate that evolving multiple interacting programs does not necessarily take longer nor produce worse results as when evolving a single parse tree. This result is reminiscent of similar results obtained in [25] showing genetic programs using ADFs perform better and require less computation that when evolving non-ADF parse trees. Future work should compare ADFs and MIPs on various problems to determine which of their respective representational distinctions help or impede evolution and under what circumstances.

The artificial ant experiment demonstrates that MIPs can evolve neural network-like structures under significant restrictions and can also evolve solutions with multiple output units. In this experiment, MIPs solved a well known problem previously solved with recurrent neural networks but used no intermediate units in its solution. Instead, the MIPs net was forced to use the state of the output units to encode its problem solving state. In general, MIPs nets should be able to represent complex behaviors for problems more efficiently than standard neural networks since the individual computations in a MIPs net and the language used to express these computations can be more task specific than a single common activation function.

In the 5-bit reverser problem, MIPs solved a difficult temporal problem that required the development of a task-specific memory. The resulting network used only three of the provided four units to solve the task with one of the three units evaluating to a constant. The two active units first encoded a five digit bit string into their real-valued outputs and decoded the sequence in reverse order when signaled to do so. Note that the first binary digit given to the MIPs net was not decoded for ten cycles. The elegant self-similar dynamics of the evolved MIPs net shows the described technique’s ability to harness interesting task appropriate dynamics within a MIPs net.

The MIPs representation is general enough to model a wide variety of dynamic systems. The experiments described in this paper employed synchronous updates which is equivalent to interpreting the MIPs net as a set of difference equations that model a discrete time dynamic system. This may be the genesis of the self-similar characteristics of the solutions evolved for both the ant and 5-bit reverser problems. MIPs nets can also represent a system of differential equations simply by modifying the synchronous update algorithm to use an appropriate method of integration. This alteration would allow continuous time dynamic systems to be evolved. The question of if continuous time dynamic systems are evolvable using the techniques described here is also a subject for future work.

The MIPs methodology also holds promise for evolving collections of programs that are required to operate or interact in distributed or coarse-grained parallel computing environments. Since the model of interaction is based only on result reference, the specific model of communication between computing elements is flexible. Further, by restricting the accessibility of unit referencing, a MIPs network can be designed with a particular connectivity. For instance, an interconnection network can be enforced in the referencing scheme of the MIPs net by allowing units to refer to the values of other units that would be adjacent. For a hypercube architecture, this would involve assigning gray codes to each unit in the MIPs net and permitting reference to only those units whose gray code differs by at most a single bit. Other parallel and distributed interconnection methods are equally easy to enforce.

5.0 Conclusions

This report demonstrates that the proposed evolutionary training algorithm is able to induce systems of equations that solve non-trivial problems and, consequently, that a MIPs net is a viable representation for the evolutionary induction of complex behaviors. The differences between MIPs nets and neural networks emphasize the limitations of neural network architectures with more conventional training methods while their similarities highlight systems of equations as a general representation for complex behaviors.

The basic philosophical impetus for the neural network representations, namely using a collection of simple computational units interacting to produce a globally complex behavior, is a powerful model that has proven to be effective for a wide variety of problems. To date, this concept has been explored only for a very limited class of simple computations. Expanding the types of simple computations that can be performed by each unit in a network allows the creation of interesting neural networks variants that may ultimately be more efficient, easier to train and be more task specific.

6.0 References

[1] D. Rumelhart, D., G. H. Hinton and R. Williams (1986). “Learning Internal Representations Through Error Propagation,” in [2].

[2] D. Rumelhart, and J. McClelland (Eds.) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*. Cambridge MA: MIT Press. 1986.

[3] K. Hornik, M. Stichcombe, and H. White (1989) “Multi-layer feed-forward networks are universal approximators,” *Neural Networks*, **3**, pp. 359-366.

[4] D. Montana and L. D. Davis, “Training feedforward neural networks using genetic algorithms,” in *Proceedings of the Eleventh International Conference on Artificial Intelligence*, San Mateo, CA: Morgan Kaufmann, 1989, pp. 762-767.

[5] D. B. Fogel, L. J. Fogel and V. W. Porto, “Evolving neural networks,” *Biological Cybernetics*, **63**, pp. 487-493, 1990.

[6] H. Kitano, “Designing neural networks using genetic algorithms with graph generating system,” Complex Systems, 4, pp. 461-476, 1990.

[7] D. Whitley, T. Starkweather, and C. Bogart, “Genetic algorithms and neural networks: Optimizing connections and connectivity,” *Parallel Computing*, **14**, pp. 347-361, 1990.

[8] R. K. Belew, J. McInerney, and N. N. Schraudolf, “Evolving networks: Using the genetic algorithm with connectionist learning,” in *Artificial Life II*, C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, Eds. Reading MA: Addison-Wesley, pp. 511-547, 1992.

[9] R. D. Beer and J. C. Gallagher, “Evolving dynamical neural networks for adaptive behavior,” *Adaptive Behavior*, **1** (1), pp. 91-122, 1992.

[10] P. J. Angeline, G. M. Saunders, and J. B. Pollack, “An evolutionary algorithm that constructs recurrent neural networks,” *IEEE Transactions on Neural Networks*, **5** (1), pp. 54-65, 1994.

[11] F. Gruau, “Genetic Micro Programming of Neural Networks,” In *Advances in Genetic Programming*, K. Kinnear, Ed., Cambridge, MA:MIT Press, pp. 495-518, 1994.

[12] L.J. Fogel, A. J. Owens, and M. J. Walsh, *Artificial Intelligence through Simulated Evolution*, New York: John Wiley, 1966.

[13] D. B. Fogel, *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, Piscataway, NJ: IEEE Press, 1995.

[14] N. L. Cramer, “A Representation for the adaptive generation of simple sequential programs,” In *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, J. Grefensttete, Ed., Hillsdale, NJ: Lawrence Erlbaum, pp. 183-187, 1985

[15] J. R. Koza, J.R. *Genetic Programming: On the programming of computers by means of natural selection*. Cambridge, MA: MIT Press, 1992.

[16] B.-T. Zhang, P. Ohm, H. Muhlenbein (1997) “Evolutionary Induction of Sparse Neural Programs”, *Evolutionary Computation*, **5** (2), in press.

[17] A. Teller and M. Veloso (1996) “Neural Programming and an Internal Reinforcement Policy,” In *Late Breaking Papers at the Genetic Programming 1996 Conference*, J. Koza (ed.), Stanford University Bookstore, pp. 186-192.

[18] R. Poli (1996) “Evolution of Graph-like Programs with Parallel Distributed Genetic Programming,” In *Proceedings of the Seventh International Conference on Genetic Algorithms*, Thomas Back (ed.), San Francisco, CA: Morgan Kaufmann, pp. 346-353.

[19] S. Luke and L. Spector (1996) “Evolving Teamwork and Coordination with Genetic Programming,” in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. Koza, D. Goldberg, D. Fogel, and R. Riolo (eds.), San Francisco, CA:Morgan Kaufmann, pp. 150-156.

[20] T. Haynes and S. Sen (1997) “Crossover Operators for Evolving a Team,” in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. Koza, K. Deb, M. Dorigo, D. Fogel, M. Garzon, H. Iba and R. Riolo (eds.), San Francisco, CA:Morgan Kaufmann, pp. 162-167.

[21] H. Iba, (1997) "Multiple Agent Learning for a Robot Navigation Task by Genetic Programming," in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. Koza, K. Deb, M. Dorigo, D. Fogel, M. Garzon, H. Iba and R. Riolo (eds.), San Francisco, CA:Morgan Kaufmann, pp. 195-200.

[22] P. J. Angeline, "Two Self-Adaptive Crossover Operations for Genetic Programming," in *Advances in Genetic Programming: Volume II*, P. Angeline and K. Kinnear (eds.), Cambridge MA: MIT Press, pp. 89-110, 1996.

[23] P. J. Angeline, "An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel and R. L. Riolo, Eds., Cambridge MA: MIT Press, pp. 21-29, 1996.

[24] U.-M. O'Reilly and F. Oppacher, "A Comparative Analysis of Genetic Programming," in *Advances in Genetic Programming Volume 2*, P. J. Angeline and K. E. Kinnear Jr., Eds., Cambridge MA: MIT Press, pp. 23-44, 1996.

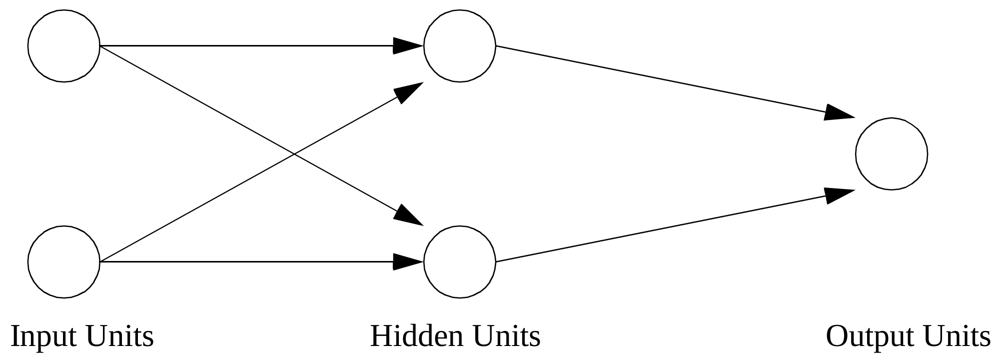
[25] J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Subprograms*, Cambridge, MA: MIT Press, 1994.

[26] A. Teller (1994). "The Evolution of Mental Models", in *Advances in Genetic Programming*, K. Kinnear (ed.), Cambridge, MA: MIT Press, pp. 199-219.

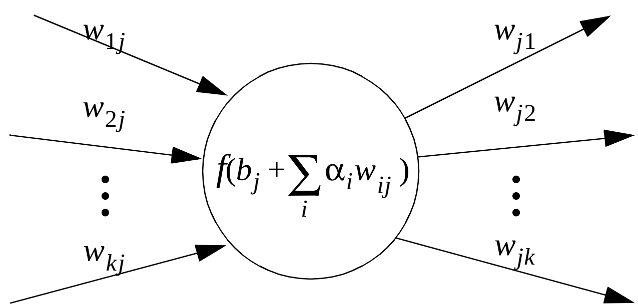
[27] D. Jefferson, R. Collins, C. Cooper, M. Dyer, M. Flowers, R. Korf, C. Taylor, and A. Wang, "Evolution as a theme in artificial life: The Genesys/Tracker system," in *Artificial Life II*, C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, Eds. Reading MA: Addison-Wesley, pp. 549- 578, 1992.

[28] P. Angeline (1997) "An alternative to indexed memory for evolving programs with explicit state representation," in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. Koza, K. Deb, M. Dorigo, D. Fogel, M. Garzon, H. Iba and R. Riolo (eds.), San Francisco, CA:Morgan Kaufmann, pp. 423-430.

[29] P. Angeline (1997) "The benefits of distributed solutions when evolving symbolic equations," SPIE AeroSense Conference, San Diego, In Press.



(a)



(b)

FIGURE 1. Representation for traditional neural networks. (a) Example of a feed-forward neural network with three layers. First, values are placed on the input nodes and then the activation of the hidden units are computed followed by the activation of the output units. (b) Form of the computation performed at each node (except input nodes). Input weights are associated with links to the node and away from the node. The node's activation is computed using a function, *f*, that takes the sum of the inputs and a bias term as an argument. The same function is computed at all nodes in the network. The resulting activation for this node is passed along the output connections to other nodes. Node number *j* is shown.

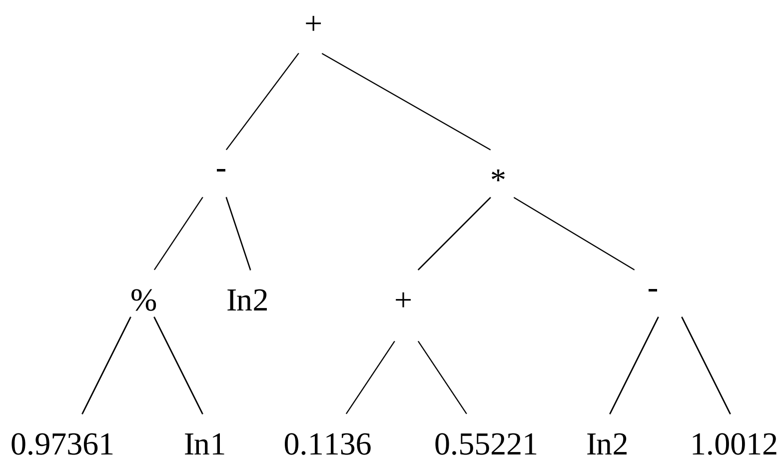


FIGURE 2. Example of a parse tree representation for an evolved program.

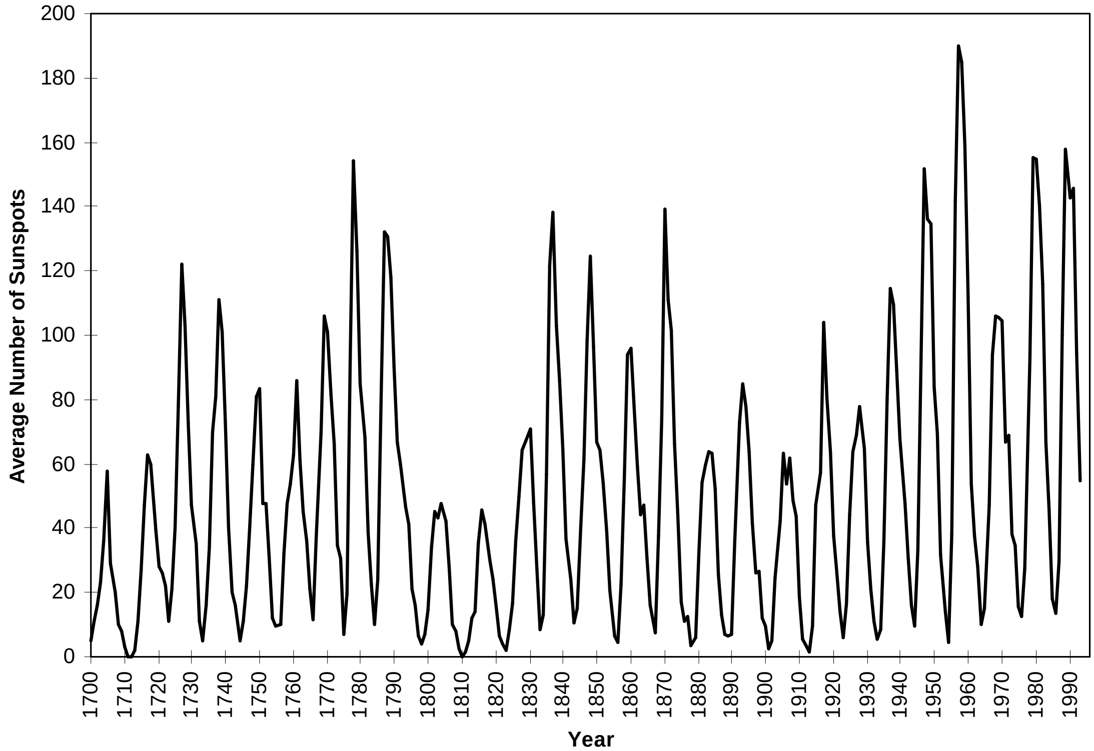


FIGURE 3. Plot showing the average number of sunspots observed for the years 1700 to 1993 shown on x-axis. This data is a naturally occurring chaotic time series with an approximate period of 11.4 years.

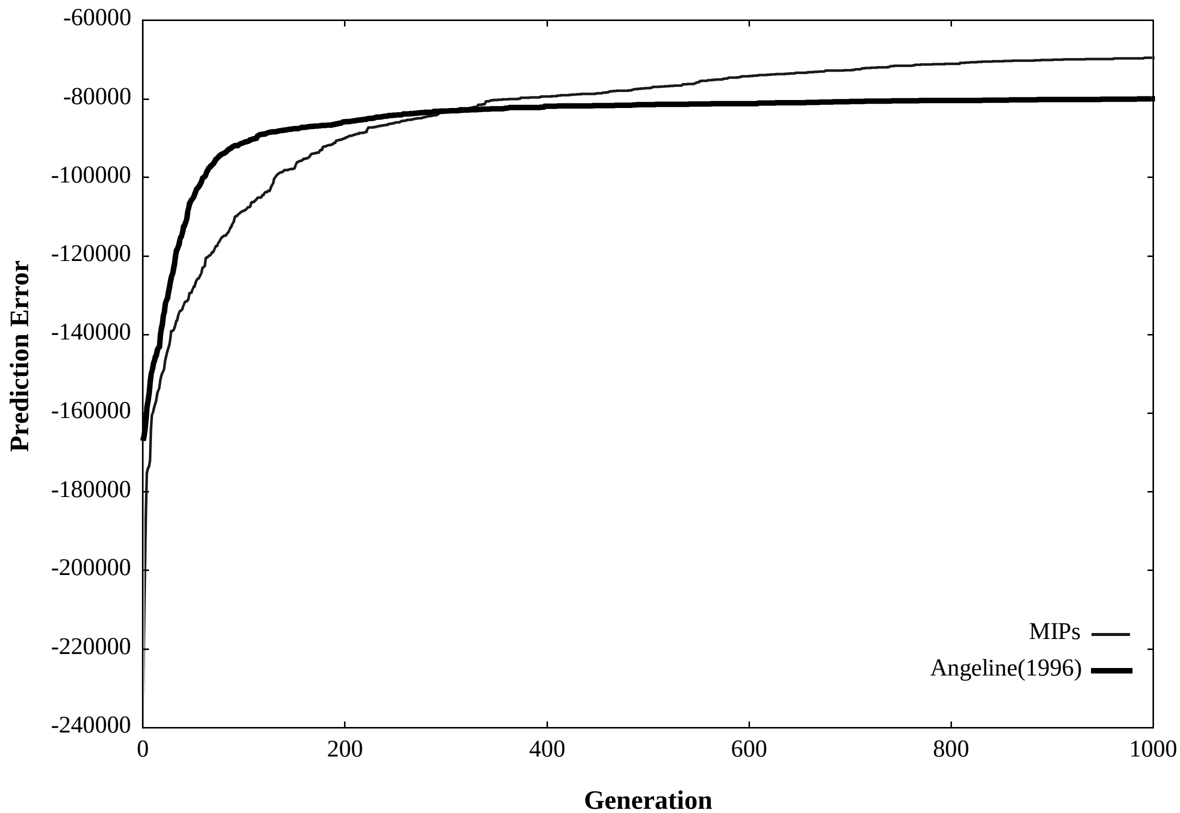


FIGURE 4. Prediction error versus Generation for the evolutionary methods solving the sunspot prediction problem.

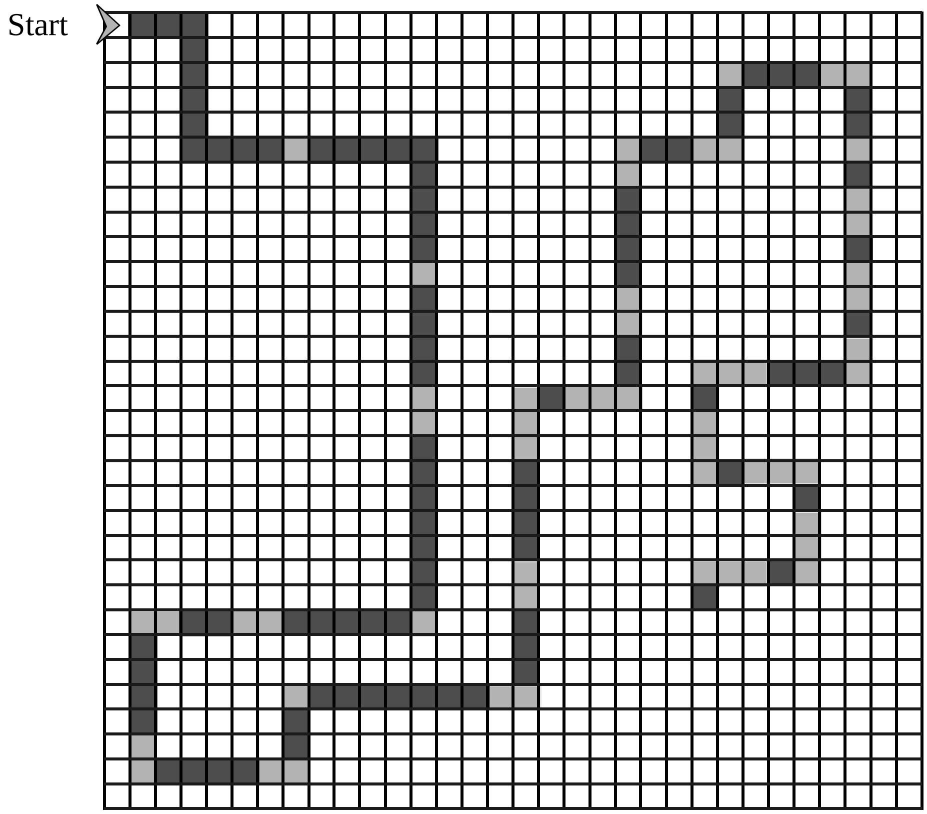


FIGURE 5. Trail of food used in the Artificial Ant problem. Dark gray squares contain food while the light gray squares show the shortest path through all 89 spaces containing food. At the start of each trial the ant begins at the position indicated facing East.

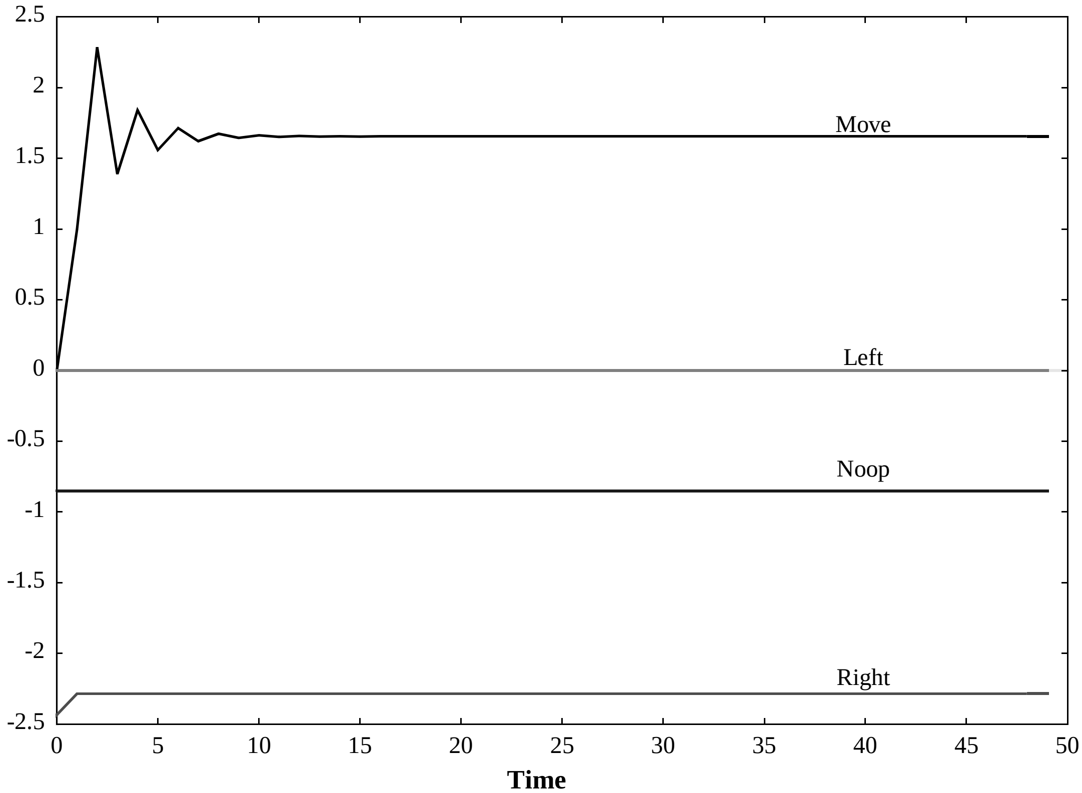


FIGURE 6. Output activations of the four nodes in the ant network over time when food is always present in the environment.

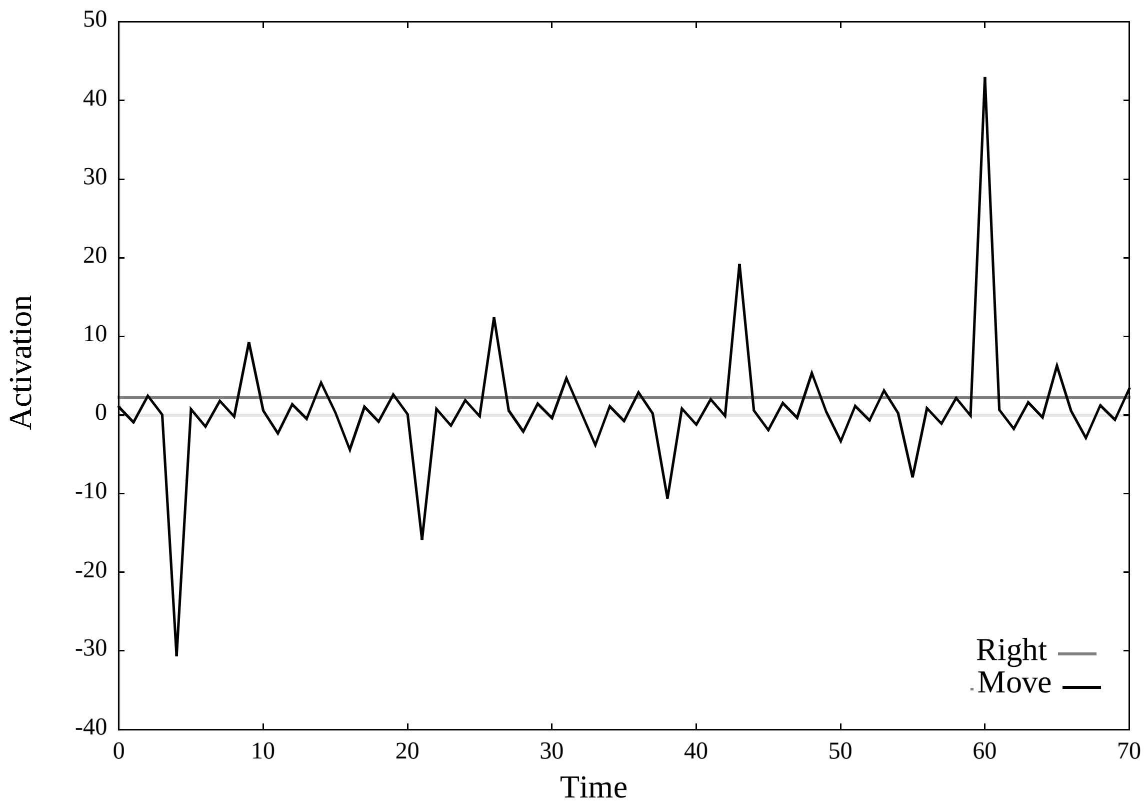


FIGURE 7. Plot of the activations of the Move (solid line) and Right (dashed line) units over the first 70 time steps in an environment that contains no food. Right unit holds a constant 2.286750 activation throughout. The Move unit's activation displays a self-similar chaotic pattern over the time period shown. This self-similar pattern is evident at least through time step 2000. Each time that Move is greater than Right the ant executes a Move command. Otherwise it executes a Right command.

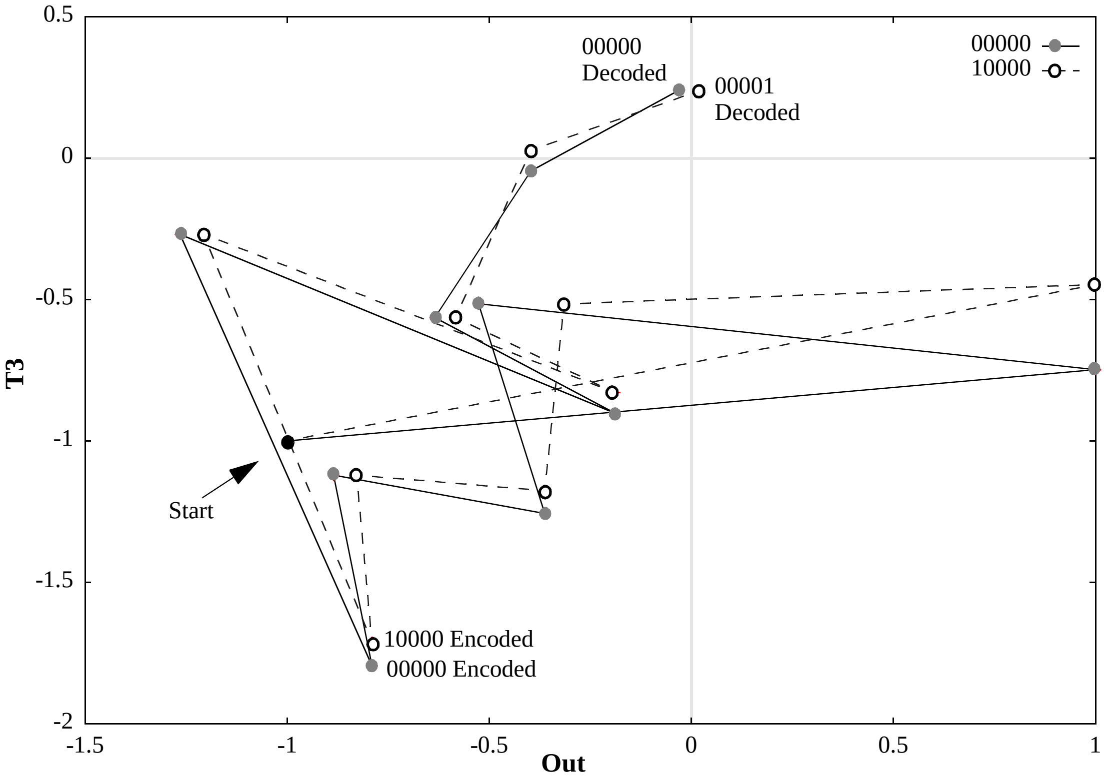


FIGURE 8. A plot of the changes in activation values for the *Out* and *T3* units during encoding and decoding of the similar strings 00000 and 10000 by the MIPs net of Equation 9. Dashed line shows the path for 10000 while the solid lines shows the path for 00000. Both examples begin with the units values initialized to (-1,-1) and then take similar paths through both the encoding and decoding stages. Note that the difference between the two encoded positions is not large but is sufficient to place the last decoded digits on opposite sides of the *Out*=0 axis.

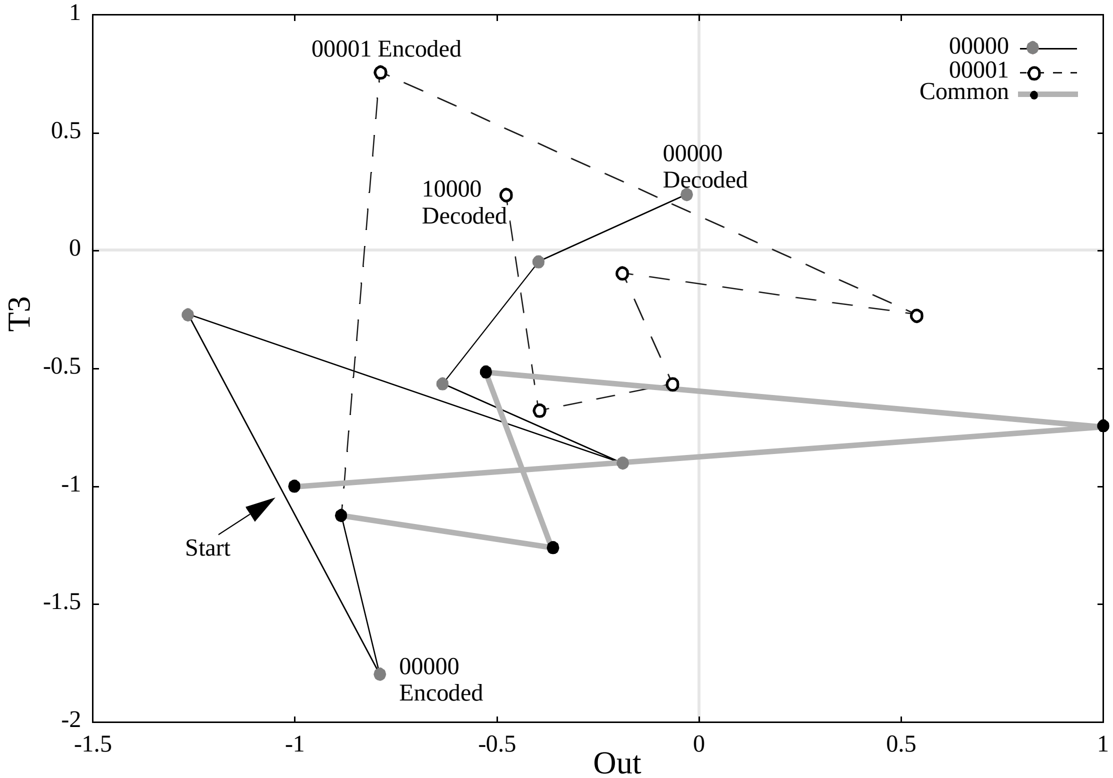


FIGURE 9. A plot of the changes in activation values for the *Out* and *T3* units during encoding and decoding of the similar strings 00000 and 00001 by the MIPs net of Equation 9. Dashed line shows the path for 00001 while the light solid lines shows the path for 00000 and the gray solid line shows the common pathway. The two paths do not diverge until the encoding of the last bit since they are exactly the same until that input step Once they diverge they take very different routes to reconstruct their respective five bit sequences.

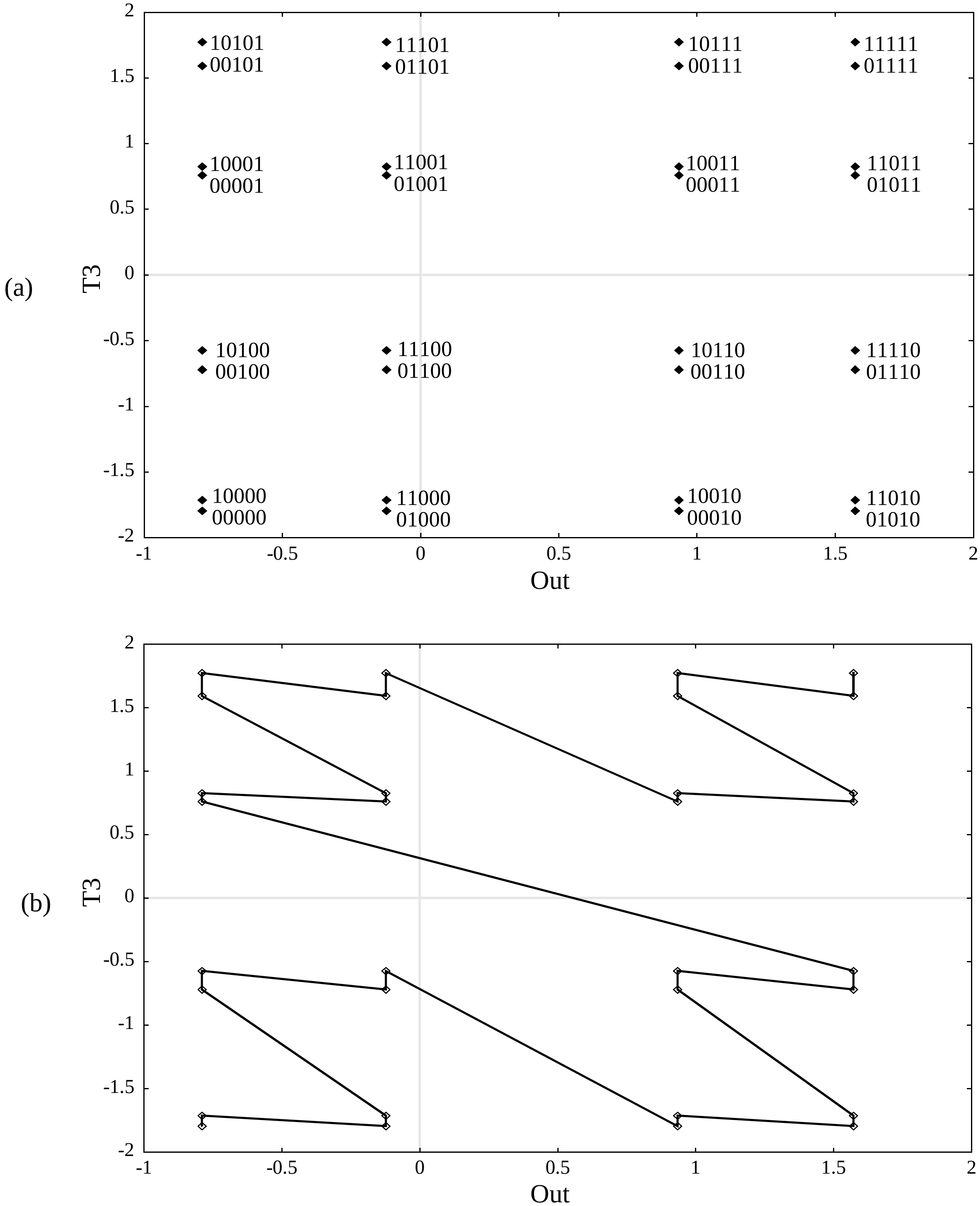


FIGURE 10. (a) The activations of the units *Out* and *T3* for all possible five digit binary sequences after encoding. The space is divided into an interesting organization based on the content of the sequence. (b) The space filling curve that results from connecting the various encoded positions in binary sequence order with the first position as the lease significant bit. The curve demonstrates an efficient packing of the encoded sequences into the two dimensional space and suggests that similar MIPs nets could accurately encode longer strings.