Internal Reinforcement in a Connectionist Genetic Programming Approach

Astro Teller astro@cs.cmu.edu (412) 268-7123

Manuela Veloso mmv@cs.cmu.edu (412) 268-8464

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Genetic programming (GP) is a successful machine learning technique that provides powerful parameterized program primitive constructs using evolution as its search mechanism. However, unlike some machine learning techniques, such as Artificial Neural Networks (ANNs), GP does not have a principled procedure for changing parts of a learned structure based on that structure's past performance. GP is missing a clear, locally optimal update procedure, the equivalent of gradient-descent backpropagation for ANNs. In this article, we introduce a new mechanism for defining and using performance feedback on program evolution. This "internal reinforcement" principled algorithm is implemented within a new connectionist representation for evolving parameterized programs, namely "neural programming." We present the algorithms for the generation of credit and blame assignment in the process of learning programs using neural programming and internal reinforcement. The article includes a brief overview of genetic programming and empirical experiments that demonstrate the increased learning rate obtained by using our principled program evolution approach.

Contents

1 Introduction

This article introduces a new machine learning method for learning complex programs through evolution. The method has been developed with the goal of incorporating a principled updating procedure in program evolution, until now unaccomplished in genetic programming. A new program representation, *neural programming*, is also introduced as a connectionist programming language that supports our principled *internal reinforcement* in program evolution.

Machine learning has successful representatives of the practice of *explicit credit assignment*. In explicit credit-assignment machine learning techniques, the models to be learned are constructed so that *why* a particular model is imperfect, *what part* of that model needs to be changed, and *how* to change the model can all be described analytically with at least locally optimal (i.e. greedy) results. Feedforward artificial neural networks (ANNs) with the backpropagation algorithm are an example of such an explicit credit-assignment machine learning approach. Genetic programming (GP) is a successful representative of the machine learning practice of *empirical credit assignment* [5]. Empirical credit assignment allows the dynamics of a learning system to implicitly determine credit and blame of a hypothesis target description during its search. Evolution does empirical credit assignment as it searches its hypotheses space by an evaluation of the fitness of different target descriptions [1].

The goal of this work is to bridge this credit-assignment gap by finding ways in which explicit and empirical credit assignment can find mutual benefit *in a single machine learning technique*. In short, it would be desirable, in GP, to have reinforcement of programs be more specific (directed towards particular parts or aspects of a program) and more appropriate (telling the system *how* to change those specific parts).

This article presents a new method to accumulate explicit credit assignment information. We introduce a new connectionist program representation, neural programming, which organizes GP programs are into a network of nodes replacing program *flow of control* with *flow of data*. In neural programming, the acquired feedback values are collectively referred to as the *credit-blame map*, representing the propagation of punishment and reward through each evolving program.

The credit-blame map is then used by an explicit *internal reinforcement* method that provides a reasoned method to guide search in the field of program induction. When hill-climbing in any space, it is always possible to sample a few points and then choose the best of those to continue from. When the gradient is available, however, it is always better (locally at least) to move in the direction of the gradient. Program evolution can work with random samplings of nearby point in program space, but we show that it can converge more effectively with our internal reinforcement procedure. We introduce internal reinforcement as an approximation to the gradient function for program evolution.

In summary, this article has three main contributions. First, we contribute the neural programming representation as a new, connectionist, representation for evolving programs. Second, we describe how this representation can be used to deliver explicit, useful, internal reinforcement to the evolving programs to help guide the learning search process. And third, we demonstrate the effectiveness of both the representation and its associated internal reinforcement strategy through empirical machine learning experiments applied in signal

classification domains.

2 Genetic Programming: Basic Concepts

This article addresses a weak point in the field of Genetic Programming Computation (GP). Before we can describe how GP can be improved, it will help to first give an overview of GP's general process.

Evolutionary Computation (EC) is a field that includes such areas of study as Genetic Algorithms, Evolutionary Strategies, Evolutionary Programming, and Genetic Programming. Figure 1 shows the general process by which *individuals* (e.g., Lisp programs in the case of GP) can be learned to perform well at specified tasks (e.g., curve fitting, classification, or control systems in the case of GP).

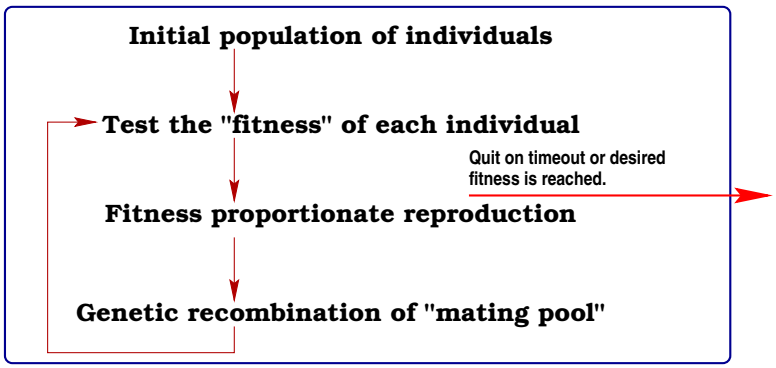


Figure 1: This is a simple outline of how evolutionary computation works.

Because this article addresses a problem that can most clearly be improved in genetic programming, this section will focus on the basic mechanism of GP. However, this section has intentionally been left general enough to emphasize aspects of EC that are relevant to the central contribution of this article, Neural Programming (NP) and Internal Reinforcement in Neural Programming (IRNP).

Determining the *fitness* of each program is the important first step in the evolutionary loop. This can be done in a large number of ways. The single most common mechanism for determining this "how good is it?" value for each program (and the mechanism used by NP) is shown in table 1. This process of training with a set of examples in which the correct output is known is called "supervised learning" in machine learning.

The next step in GP, *fitness proportionate reproduction*, is the exploitation phase of the search in which attention is focused on the pieces of the search (programs in this case) that look most promising. Given that exploitation means focusing the search towards the high fitness programs, and given that each program in an evolving population represents a unit of search, the way to focus search is to throw out some of the low fitness programs and replace them with copies of the high fitness programs. There are a number of popular schemes for doing this, primarily *tournament selection*, *rank selection*, and *roulette selection*. Table 2 outlines *tournament selection* as the reproduction strategy used in NP. The resulting mating

For each program p in the evolving population
    For each training input S_i $(1 \leq i \leq |S|)$
        Call L_i the correct/desired program response (the signal label)
        Run program p on input S_i and get its response, R_p^i
        $G_p = \sum_{i=1}^{|S|} \frac{\mathcal{F}(L_i, R_p^i)}{|S|}$

Function $\mathcal{F}$ differs with each task for which EC is used to create a solution.
G_p is the fitness of program p

Table 1: Common calculation of program fitness in the supervised machine learning form of GP.

pool is of the same size as its parent population and in it high fitness programs have higher representation and low fitness programs have lower representation.

For a population of M programs, **Do M times**
    Pick K programs from the population using a uniform random probability.
    Of these K programs, copy the program with highest fitness into the mating pool.

Table 2: Outline of tournament selection in GP.

Genetic programming must, by definition, include an aspect of exploration as well as an aspect of exploitation. The search process, in which selected individuals are changed in an attempt to find even better parts of the search space, is called *genetic recombination* of the mating pool. The two most popular forms of genetic recombination are crossover and mutation.

In crossover, two or more programs are chosen and some "genetic material" is exchanged between them. The hope is that high fitness programs are made up of "building blocks" and that these building blocks can be reshuffled among high fitness programs with positive effect at least some of the time. Crossover is often referred to in GP as *sexual recombination* to indicate that the inspiration for crossover is the apparent usefulness of genetic material sharing that takes place in the sexual reproductive process of most animals.

As an example of crossover, let us take the example of evolving an arithmetic formula using only the operators $+$, $-$, $*$, / and operands x and constants to approximate e^x. Crossover is explained here in the context and representation of standard genetic programming because that paradigm and representation will be referenced most heavily later in this article. Figure 2 shows two example functions, written in a tree structured format to make them easier to understand (as is traditional in Genetic Programming).

The details of the crossover mechanism vary widely through EC because the representations in which this crossover takes place vary widely. The common thread that identifies a search operator in EC as "crossover" is the exchange of material between two or more

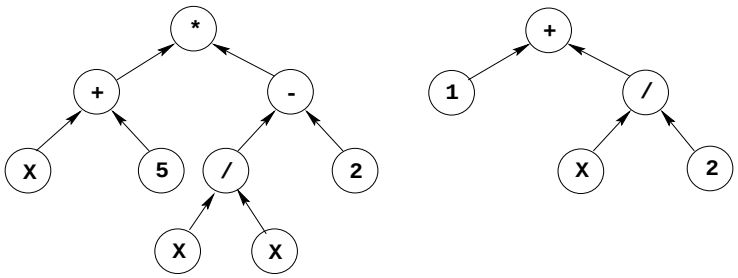


Figure 2: Two example functions in a hypothesized evolution example, **before** crossover.

population members. Though the following is not a defining characteristic of crossover, it is almost always the case in GP, as it is currently practiced, that when material is selected for this genetic exchange, the material is chosen **at random** (or nearly at random). The traditional GP crossover procedure is "choose one subtree from each program and exchange the subtrees. An example of such a subtree exchange is shown in figure 3.

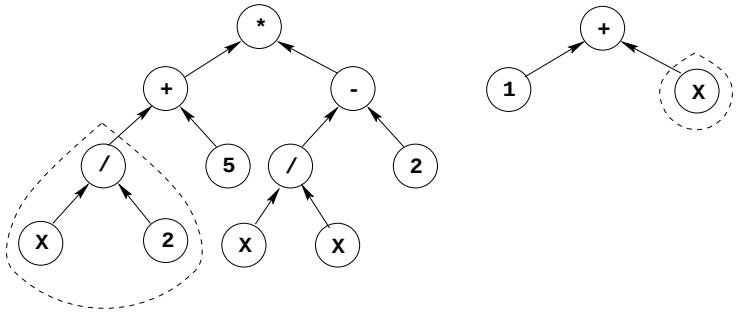


Figure 3: The two example functions from Figure 2 **after** crossover. The dashed lines indicated the exchanged genetic material (sub-functions in this case).

In the recombination search operator called *mutation*, a single individual is taken and changed in some way that is independent of the other members of the population. This style of simulated evolution search operator is inspired by the biological genetic recombination process of mutation. Mutation in chromosomes can happen either because a gene is changed through the proverbial "cosmic ray" or through a mis-copying that takes place as a cell is being duplicated.

As an example of mutation, let us continue with the example of evolving an arithmetic formula using only the operators $+, -, *, /$ and the operands x and constants to approximate (fit) a set of data points. This time the two functions shown in Figure 2 will be changed through mutation. This means that some part of the individual (function) will be selected and changed to a new value (a new subtree in this particular genotype representation). How these subtrees to be changed are selected and how the subtrees to replace them are created will be addressed in detail later in this article. Figure 4 illustrates the mutation (subtree replacement) process in GP.

The details of the mutation mechanism vary widely through EC because the representations in which mutation takes place vary widely. The common thread that identifies a search operator in EC as "mutation" is the creation of new genetic material and the replacement

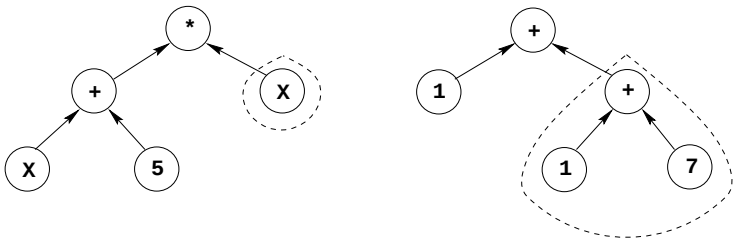


Figure 4: The two example functions from Figure 2 **after** mutation.

of existing genetic material in an individual with this new genetic material. Though the following is not a defining characteristic of mutation, it is almost always the case in GP, as it is currently practiced, that when material is selected for this genetic replacement, the material is chosen **at random** (or nearly at random).

Also notice that while mutation does not seem to be recombination in the biological sense, mutation is often equivalent to crossover with a randomly generated individual (e.g, in the GP framework) and so is identified as a recombination strategy in EC.

In summary, there are three main phases of the GP learning loop and this article contributes novel aspects to one of them:

- *Evaluation of the fitnesses of each individual*
  It is **not** with this aspect of GP that this article takes issue.
- *Fitness-proportionate reproduction into a mating pool*
  It is **not** with this aspect of GP that this article takes issue.
- *Genetic recombination of the mating pool*
  It **is** with this aspect of GP that this article takes issue. There is no evidence that the aimless recombination that is so common in GP is better than focused recombination. On the other hand, this article provides specific evidence that there is something useful about carefully and purposefully choosing pieces of material to change or exchange during program transformations.

3 Neural Programming

The essence of a programming language is one or more basic constructs and one or more legal ways of combining those constructs. A measure of the extensibility of a language is the ease with which new constructs or new construct combinations can be incorporated into the language. It is the high degree of extensibility in GP that we want to wed to the focused update policies possible in other machine learning techniques.

3.1 The NP Representation

The Neural Programming representation consists of a graph of nodes and arcs that perform a **flow of data**, rather than the flow of control in typical programming languages. The nodes in a neural program can compute arbitrary functions of the inputs. So a node can still be the

sum of inputs to the node and a sigmoid threshold. But it can also use other functions such as MULT, READ, WRITE, IF-THEN-ELSE, and, most importantly, potentially complex user defined functions for examining the input data. Examples of neural programs are given later in this section. Table 3 enumerates the important characteristics of the NP representation.

- An NP program is a general graph of nodes and arcs.
- Each NP node executes one of a set of functions (e.g., Read-Memory, Write-Memory, Multiply, Parameterized-Signal-Primitive-3, etc.) or zero-arity functions (e.g., constants, *Clock*, etc.).
- An arc from node x to node y (notated (x, y)) indicates that the output of x flows to y as an available input.
- On **each** time step t $(0 < t < T)$, **every** node takes some of its inputs, according to the arity of its function, computes that function, and outputs that value on *all* of its output arcs. *Data flow, not control flow.*
- One type of node function is "Output." Output nodes collect their inputs and create the program response through a function OUT of those values. In this article OUT is a simple weighted average. Each value is weighted by the timestep it appears on.

Table 3: The critical characteristics of the NP representation.

Throughout this article, the characteristics of NP will be used and discussed in greater detail. Let us here highlight a few aspects of NP. A parameterized signal primitive (PSP) is a piece of code, written by a user of NP that expresses a way of extracting information of the input signal in a parameterized form. Example PSPs might return the AVERAGE or VARIANCE of values in a range of the signal being examined as specified by the inputs to that node. This kind of embedding of complex (often co-evolved) components as primitives in the evolving GP system has repeatedly been shown to be effective (e.g., [21]). Furthermore, these powerful parameterized-signal-primitives, as part of the learning process, can be used in place of brittle preprocessing.

The topology of NP programs is not rigidly fixed to the semantics of the function each node happens to execute. That means that, for example, a node that only "needs" two inputs (e.g., a node executing the function DIVIDE) may have more input arcs than it can use (e.g., it has four input arcs and simply ignores two of them) or have fewer than it can use (e.g., it has only one input and so returns some default value).

Each NP node may have many output arcs. See Figure 6 for a simple example. The multiple forked distribution of good values from any point in the program is a valuable aspect of the NP representation. The idea is that once a "valuable" piece of information has been created, it can be sent to different parts of the NP program to be used further in a variety of different ways. This fan-out is an advantage of connectionist representations from which GP program representations can profit by incorporating. This same advantage

can be achieved through the use of memory (variables). The advantage here is that the topology makes explicit this distribution (flow) of information. This explicitness turns out to be exactly what is required to make internal reinforcement tractable.

In a data flow machine in which function evaluation at the nodes is instantaneous, there are two distinct options for computing the output of a node from its inputs. The first is that all nodes act simultaneously on the outputs generated on the previous timestep. In other words, there is no order to the evaluations of nodes in a program: they evaluate in parallel. In the second case, the nodes are evaluated in a particular order, so that if $D_{x,t}$ indicates the evaluation of node x on timestep t, the evaluation order is $D_{0,0}$, $D_{1,0}$, ..., $D_{N_p,0}$, $D_{0,1}$, $D_{1,1}$, ... $D_{N_p,T}$ where N_p is the number of nodes in the hypothetical program p. For illustrative purposes, in the examples below we assume that the NP programs are evaluated according to the first rule: all nodes evaluate in parallel.

Given that there is a timestep threshold imposed on the evolving programs, a reasonable question to ask is, "How much of a burden is this threshold?" or alternately "Can the evolving programs take advantage of additional time in which to examine an input signal?" The answer to this second question is "yes" and experimental evidence for this can be found in [39].

There are two dominant forms of change that evolving programs typically undergo: crossover and mutation. Mutation is the change of one (usually atomic) part of the program to another aspect of the same type. Crossover is the sexual reproduction of two programs; two programs "mate" by exchanging program material between them.

> *While NP programs look more like recurrent ANNs than traditional tree-structured GP programs, NP programs are changed,* **not** *by adjusting arc weights (NP arcs have no weights), but by changing both what is* **inside** *each node as well as the* **topology** *and* **size** *of the program.*

3.2 Illustrative Examples

NP programs are evolved and explanations using evolved examples are not practical because the evolved examples are not concise. Instead we illustrate the NP representation through a set of constructed examples. Of course, any of the following example programs and program fragments could have been the result of evolution.

3.2.1 Example 1: The Fibonacci Series

Figure 5 shows an extremely simple NP program. This program computes the Fibonacci series and sends each successive element out of the program fragment on Arc4.

There is only one initialization necessary for the correct operation of NP programs: "what input values should all nodes use on their very first computation?" Since NP programs are data flow machines, each arc is a potential memory value and so there must be some initial state to the program. For this example, let us initialize each program so that all arcs have the value 1 when a program starts up. Table 4 shows how the values of the arcs change over time.

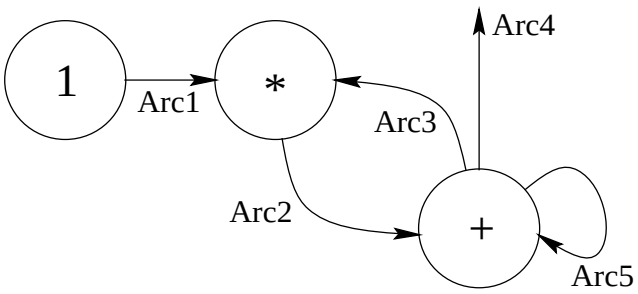


Figure 5: A simple NP program that computes the successive elements of the Fibonacci series. *All input/arc values are 1 on the first time step.*

Step	Arc 1	Arc 2	Arc 3	Arc 4	Arc 5
0	1	1	1	1	1
1	1	1	2	2	2
2	1	2	3	3	3
3	1	3	5	5	5
4	1	5	8	8	8
...					

Table 4: Progression of arc values over time for the simple NP program shown in Figure 5.

3.2.2 Example 2: The Golden Mean

Let us now change slightly the computation of the simple NP program from example 1. Instead of outputing a list of exponentially increasing values (as in the program shown in Figure 5) let us design an NP program that approximates the "Golden Mean"[1] through its OUTPUT node. To do this, all we need to do is to add an extra node that does Division (DIV) and pass it as its two parameters (i.e., its two input arcs) $\text{fib}(i)$ and $\text{fib}(i-1)$ as they are computed (shown in Figure 6).

Table 5 shows how this computation plays out through the arcs as the timesteps pass.

3.2.3 Example 3: Foveation

Foveation is the process changing focus of attention in response to previous perceptions. For example, this iterative process of foveation is what gives us the illusion of seeing with high-resolution across our field of vision when, in fact, our fovea (the high resolution area of the retina) only fills about 5% of our field of view.

The Fibonacci examples illustrate how the flow of data works and how the fan out of values can significantly reduce the size of a solution expression. In this example we illuminate another important feature of NP programs: the ability to foveate. NP programs have the ability to use the results of an examination of the input signal to *guide* the next part of that

[1]The golden mean is $\frac{1\pm\sqrt{5}}{2}$. This example program approximates $\frac{1+\sqrt{5}}{2}$. Note that $\lim_{n\to\infty} \frac{\text{fib}(i)}{\text{fib}(i-1)} = \frac{1+\sqrt{5}}{2}$.

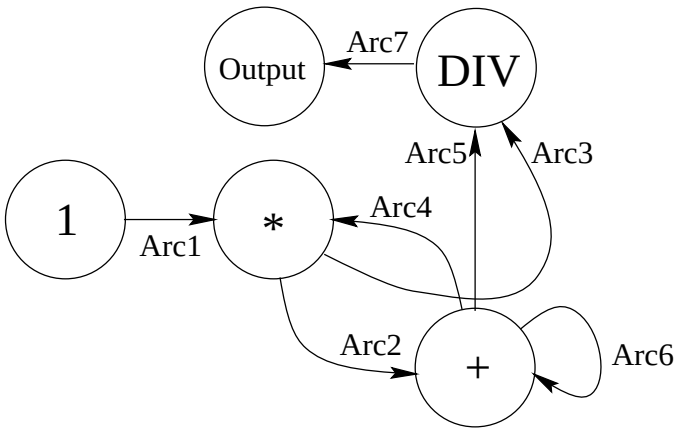


Figure 6: A simple Neural Program that iteratively improves an approximation to the golden mean. *This program assumes that all input values are* **1** *on the first time step.*

Step	Arc 1	Arc 2	Arc 3	Arc 4	Arc 5	Arc 6	Arc 7	OUTPUT	Golden Mean
0	1	1	1	1	1	1	1	NA	1.618034
1	1	1	1	2	2	2	1	1	1.618034
2	1	2	2	3	3	3	2.0	1.5	1.618034
3	1	3	3	5	5	5	1.5	1.5	1.618034
4	1	5	5	8	8	8	1.6667	1.5417	1.618034
5	1	8	8	13	13	13	1.6000	1.5533	1.618034
...									

Table 5: Progression of arc values over time for the simple NP program shown in Figure 6.

examination.

> NP programs view their inputs (called *signals* when appropriate to avoid confusion with "inputs" to a node) through *Parameterized Signal Primitives* (PSP), variable argument functions defined by the NP user.

Let us assume that this NP program is examining signals that are video images. PSP-Variance is a user-defined PSP that takes four arguments, a_0 through a_3, (interpreted as the rectangular region with upper-left corner (a_0, a_1) and lower-right corner (a_2, a_3)) as input and returns the variance of the pixel intensity in that region. Figure 7 shows what could be part of a larger NP program. The node indicated with a double circle computes the function PSP-Variance.

To simplify the explanation, this particular NP program fragment delivers static values for three of those four inputs. The fourth input indicated by a dashed circle, changes as the program proceeds. That means that PSP-Variance, at each time step, computes its function over the region (50,17,104,a_3). The simplest way to explain this mechanism is to give the pseudo-code to which it is equivalent (see Table 6). Note that IF-T-E (If-Then-Else) is the function "if $(a_0 < a_1)$ then return a_2 else return a_3." Assuming again that all arcs are initialized to 1, this program finds a one-sided local minimum of PSP-Variance with respect

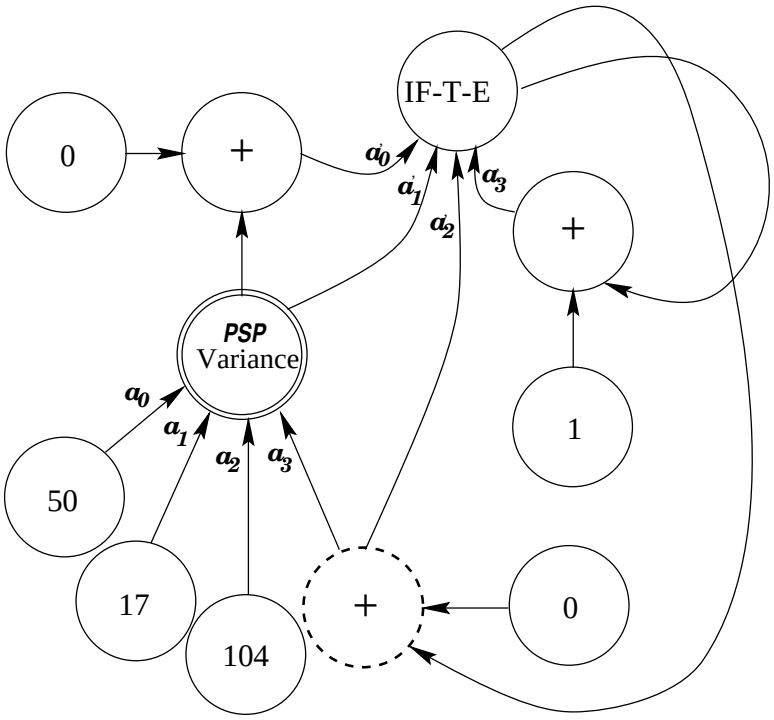


Figure 7: A simple NP program fragment. The output value from the dashed circle node is being iteratively refined to minimize the value returned by the PSP-Variance node.

to its fourth parameter. In general, the program fragment increments the fourth parameter only if (PSP-Variance(50,17,104,$a_{3,t}$) < PSP-Variance(50,17,104,$a_{3,t-1}$)) (where $a_{3,t}$ is a_3 on timestep t). This is a concise example of NP foveating: using the values it perceives to focus further investigation of the input in question.

$\text{VAR}_0 = 1$
$\text{VAR}_t = \textbf{PSP-Variance}(50,17,104,a_{3,t})$
IF $(\text{VAR}_{t-1} < \text{VAR}_t)$
    THEN $a_{3,t+1} = a_{3,t}$
    ELSE $a_{3,t+1} = a_{3,t} + 1$

Table 6: Pseudo-code for the behavior of the NP program fragment in Figure 7. In this figure, VAR_t denotes the value of VAR at time t and $a_{3,t}$ denotes the value of argument a_3 at time t.

3.3 Evolving Subroutines

Each node may have multiple outputs. Since one of the advantages of substructure to evolving programs is the availability of regularity [21], this opportunity to spread computed values to multiple locations simultaneously is an alternative to explicit subroutines in evolving programs.

Here is another way to understand the value of this kind of programmatic fan-out. In biology, the process of *branch duplication* takes place, in which a gene (or at least a region of DNA) is duplicated within a chromosome. This redundancy is almost never harmful because these regions of DNA encode for proteins and encoding for them more than once does not reduce the abundance of these proteins. Now, however, biological mutation has the opportunity to change one of these regions, encoding for a new, "experimental" protein, without disrupting the supply of the original, useful protein. In the same way, if there is a particularly good "idea" present in an NP program, this "idea" (i.e., series of output values from a node) can be distributed, through the multiple output arcs, to a number of different parts of the program.

Further, an additional output arc can be added to a node that has been identified as the source of one of these "good ideas." This node can lead into an unused, or under-used part of the program. This area of the program now has the opportunity to create some added value using this input. If it does, learning has succeeded; the performance has improved. If it does not succeed, then no harm has been done to the program, because the "good idea" is still being used correctly in other parts of the program. Notice that this "parallelization" of evolution (i.e., independence of sub-parts of the program) is accomplished because NP programs are data-flow, rather than control-flow programs and because there can be multiple OUTPUT nodes in a single NP program.

Our work as reported in this article contributes a specific technique to identify these "good ideas" (Section 5.1). Section 6 makes explicit how unused or under-used areas of a program can be identified and exactly how the "good idea" node and the unused/under-used program areas can be connected. For completeness, however, it is worth mentioning at least one method for adding explicit hierarchy into the NP paradigm. The only insight required is that *each node computes a function and that that function can be co-evolved.* Figure 8 below pictures this embedding process.The general process of co-evolution of subroutines is dealt with in detail in a number places (notably [21]). These same pieces of wisdom are equally applicable to evolving hierarchy within a single NP program.

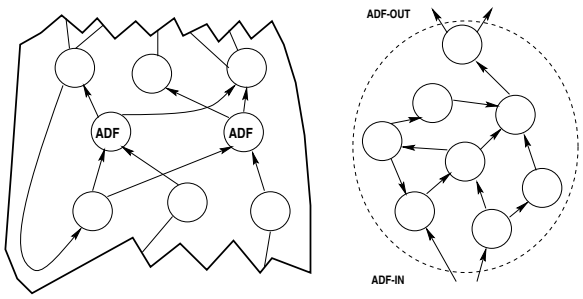


Figure 8: How ADFs (automatically defined functions) could be implemented in NP.

4 Introduction to Internal Reinforcement

Evolution is a learning process. In NP (or GP for that matter) programs are tested for fitness, preferred according to those fitness tests, and then *changed.* Programs need to

become new programs. These program transformations have a specific goal. That goal is to produce programs that are better, which is to say score higher on the fitness evaluations, than their ancestors. Much of the time this will not happen, but the success of evolution as a learning process is directly linked to *how often* a novel program is really more valuable than the parent it came from. Currently, program transformations are usually random in EC. Even when they are not random, they do not transform the programs *based on how those programs have behaved in the past.* If we could only look into a program and see which parts of it are "good" and which parts "bad," we could write transformation rules that were much more effective, which is to say, we could dramatically improve the action of evolution. That is the motivation for the principled update procedure at the heart of this article: *internal reinforcement.*

Now that we have introduced the neural programming representation, we can describe a mechanism to accomplish *internal reinforcement.* In Internal Reinforcement of Neural Programs (IRNP), there are two main stages. The first stage is to classify each node and arc of a program with its perceived contribution to the program's output. This set of labels is collectively referred to as the *Credit-Blame map* for that program. The second stage is to use this Credit-Blame map to change that program in ways that are likely to improve its performance.

Our ongoing research includes investigation into which methods to use to best accomplish the goals of internal reinforcement. We have identified several methods for accomplishing each of the two stages. This article focuses on one technique for each of the two stages.

Table 7 shows the evolutionary learning process for NP and how IRNP fits into that picture. One Credit-Blame map is created *for each program in the population* and when the time comes to perform genetic recombination (the search method in EC) on a particular program, the Credit-Blame map *for that particular program* is used.

5 Creating a Credit-Blame Map

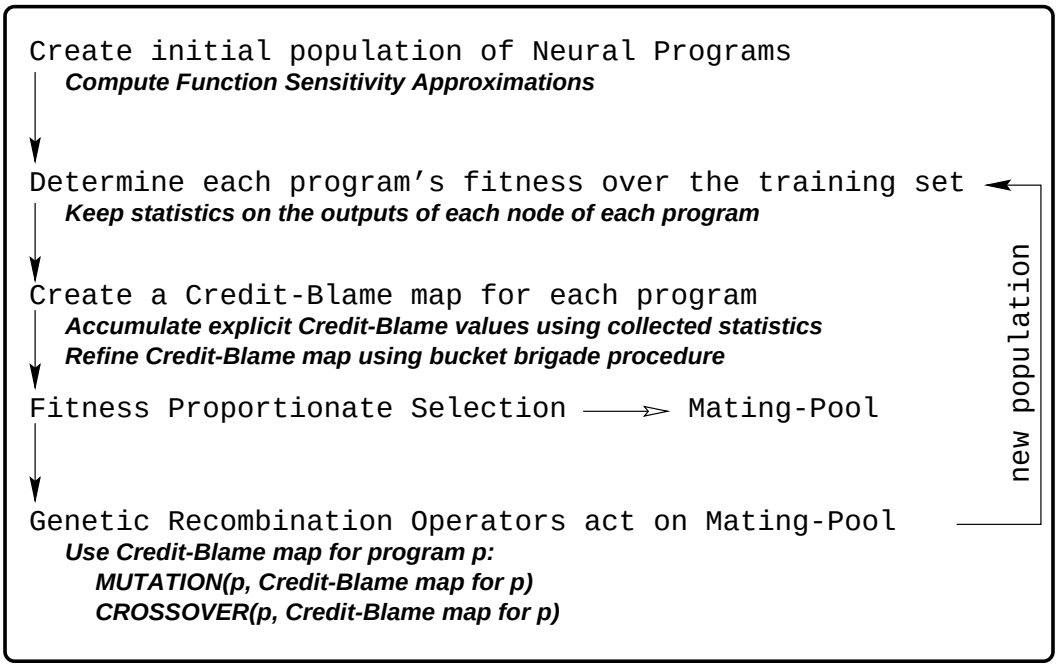
Without loss of generality, we can assume that the evolving NP programs are trying to solve a target value prediction problem. This is so because classification problems (a non-ordered set of output symbols to be learned) can be decomposed into target value prediction problems (an ordered set of output symbols to be learned). Therefore, let us consider an abstract input to output mapping to be learned by the neural programs.

5.1 Accumulation of Explicit Credit Scores

For each program p, for each node x in p, over all time steps on a particular training example S_i, we compress[2] all the values node x outputs into a single value H_x^i. Let the correct answer (the correct target value) for training instance S_i be L_i. In other words, L_i is the desired output for program p on training instance S_i.

We now have two vectors for all $|S|$ training instances: $\vec{L} = [L_1..L_i..L_{|S|}]$ and $\vec{H}_x = [H_x^1..H_x^i..H_x^{|S|}]$. We can compute the statistical correlation between them. We call the abso-

[2]The compression function used in this article is *mean.*



IRNP additions to EC

Table 7: The high level flow of NP learning.

lute value of this correlation the explicitly computed **Credit Score** for node x, notated as $\mathbf{CS}_x$. This computation is shown in Equation 1.

$$\mathrm{CS}_x = \left| \frac{E(\vec{H}_x - \mu_{\vec{H}_x}) * E(\vec{L} - \mu_{\vec{L}})}{\sigma_{\vec{H}_x} * \sigma_{\vec{L}}} \right| \tag{1}$$

This credit score for each node is an indication of how valuable that node is to the program. It is certainly the case that nodes with low credit scores at this stage may still be critical to the success of the program in question, but it is also certainly the case that nodes with high credit scores *could be* very valuable to the program, even if they are currently unutilized. Note that an NP program is, *by definition*, 100% correct if it has a node with a credit score of 1.0 and that node is connected to an output node[3]. This *explicit* credit score can also be thought of as the *individual* credit score for the node. That is, the explicit credit score takes into account only how the node acts as an individual, not how it acts as part of a large group of tightly coupled nodes (i.e., the program it is a part of).

This measured correlation (Equation 1) is not the only or necessarily best measure for the individual usefulness of a program node. For example, this measured correlation is a linear measure that can miss more subtle non-linear relations between the series of values passing through the node and the correct program output value. The claim here is not that this method for measuring a node's individual value to a program is perfect, only that is a good and useful approximation.

[3]This is true under the condition that there are no other arcs in the program that terminate in an OUTPUT node.

The set of explicit credit scores for all nodes provides a Credit-Blame map for the program: a value associated with each node in the program that indicates its individual contribution to the program. However, we want the Credit-Blame map to capture not only a node's immediate (individual) usefulness, but also it's usefulness *in the context of the program topology*. The following example highlights why the explicit credit scores do not, by themselves, capture this information.

In this example, nodes x and y produce values and node z computes an XOR of these two values. In this case, even if z has a high credit score, x and y may not (e.g. $\mathrm{CS}_z = 0.97, \mathrm{CS}_x = 0.14, \mathrm{CS}_y = 0.07$). There is nothing provably wrong with this situation but clearly, the topological notion of usefulness has not been captured in these explicit credit scores. This can be seen because the nodes x and y in this example are partly[4] responsible for node z's success (and are therefore useful) but still have low credit scores.

The Credit-Blame map can be refined to attend to this type of indebtedness relationships by passing credit and blame back through the NP programs along the arcs. The statistical correlation between $\vec{L}$ and $\vec{H}_x$ constitutes a first approximation to the credit score for node x. Because nodes are connected to each other and only a few are directly connected to the OUTPUT node, and because each node performs a specific function, the Credit-Blame map needs to be further refined. This process of refining the Credit-Blame map to take advantage of the topology of the program is described in Section 5.3.

5.2 Function Sensitivity Approximation

To pass back credit and blame through the neural program topology, we must first answer an important question: "How does each node act as a function of its inputs?" In other words, "What is the responsibility of each input parameter for the output value produced by each function?"

This problem is very difficult for arbitrary functions, which is one of the main reasons why ANN backpropagation requires differentiable functions (e.g., the sigmoid or the Gaussian). Unfortunately, we can not always differentiate the functions used in NP programs as they may not always be differentiable (e.g. If-Then-Else).

In our work, we introduce *Function Sensitivity Approximation*, a method for "differentiating" an arbitrary function that can be treated as a black box. The two questions that function sensitivity approximation can automatically answer about a black box function's relation to its inputs are "How many and how few parameters can it take (min and max arity)?" and "How sensitive is the output value to changes in its inputs?" This discovered sensitivity is a substitute to the derivative of the function in question.

Let us say that the sensitivity of some function f with respect to one of its arguments (call that argument a_i) is the likelihood that the output will change *at all* when the value of a_i is changed to a new *random value* selected uniformly from the legal range of values (e.g.,

[4] We can say that nodes x and y are partly responsible for the credit score node z receives because, by definition, the output of the function XOR is dependent on its inputs and CS_z is, by definition, dependent on the output of XOR. The reason we do not say that nodes x and y are entirely responsible for CS_z is that the function at node z is also an important factor (over and above the inputs node z receives from nodes x and y) in determining the value of CS_z.

[-1000,1000]).

Before describing this discovery of a function's input sensitivity, the issue of nondeterminism must be visited. Does the proposed technique for investigating "black box" functions still work if some of the functions may be nondeterministic? The good news is "yes." PSP-Variance is such an example (see Section 3.2.3). PSP-Variance takes four input parameters and returns the variance of pixel intensity in the rectangular region described by the four parameters. PSP-Variance(10,15,101,219) is an image region that contains 18,564 pixels. A fairly accurate value for the pixel intensity variance in that region can be achieved by sampling a small fraction of those 18,564 pixels. Sampling has, by nature, an element of non-determinism.

Table 8 shows the simple process for finding the sensitivity of each parameter of a general, possibly nondeterministic function f. The procedure in Table 8 is performed for all values of A between 1 and the maximum arity of the function. We do not have to determine this maximum arity ourselves. The key insight is that, finding the sensitivity gives us the minimum and maximum arities for each function since, for example, the maximum arity is, by definition, that parameter number above which further parameters have a sensitivity of zero. Nondeterminism can also be handled by this process and is adjusted for through the calculation of "Noise" as shown in Figure 8.

Noise := 0; Sensitive := 0;
DO Q_s **times**
    Let A be the arity of function f
    Let $\vec{a}$ be the input vector for function f
    Pick uniform random values $a_1, a_2, ..., a_A$ for $\vec{a}$
    $\text{Result}_0 := f(\vec{a})$
    $\text{Result}_1 := f(\vec{a})$
    Change $\vec{a}$: parameter $a_i \leftarrow$ random value
    $\text{Result}_2 := f(\vec{a})$
    If ($\text{Result}_0 \neq \text{Result}_1$) Noise := Noise + 1
    If ($\text{Result}_0 \neq \text{Result}_2$) Sensitive := Sensitive + 1
$\mathcal{S}_{f,A,i} := \frac{\text{Sensitive–Noise}}{Q_s}$

Table 8: Function Sensitivity Approximation: the process for finding $\mathcal{S}_{f,A,i}$, the sensitivity of a particular parameter a_i for some function f that is given a parameter vector with A elements.

As an example, consider the function "ADD" for which the user introduced a ceiling so that any set of numbers that sums to a number greater than that ceiling effectively sums to exactly that ceiling[5]. Not having giving it a lot of thought, we would have simply described all the parameters of ADD as equally important (which is true within the sampling error) and all 100% sensitive to changes in any of those parameters. However, when running our function sensitivity approximation procedure as we introduced, we find that this is not true.

[5]Implementation detail: specifically the legal range of values used in this article was [0..256]. This naturally caused the ADD ceiling enforced to be the value 256.

# Params	arg 1 sensitivity	arg 2 sensitivity	arg 3 sensitivity	arg 4 sensitivity
1	0.996312			
2	0.655752	0.659795		
3	0.246964	0.259000	0.240082	
4	0.068905	0.068309	0.064403	0.076065

Table 9: **ADD** (min arity is **1** and max arity is **4**)

# Params	arg 1 sensitivity	arg 2 sensitivity	arg 3 sensitivity	arg 4 sensitivity
4	0.322822	0.326114	0.503329	0.490406

Table 10: **IFTE** (IF ($X < Y$) Then U Else V) (min arity is **4** and max arity is **4**)

Table 9 shows the values returned by the procedure. It makes sense (after looking at line four of Table 9) that with four parameters that vary randomly in the legal range of values, a random change in one of those parameters has only a 6.9% chance of affecting the value returned by this application of the function "ADD."

The benefits of Function Sensitivity Approximation are particularly clear in the context of a function such as "if-then-else." IF-Then-Else is the function "if ($a_0 < a_1$) then return a_2 else return a_3". Left to figure it out for ourselves, we originally assigned the four sensitivities as (1.0,1.0,0.5,0.5). a_2 and a_3 are certainly equally important and each has a sensitivity of 0.5. The first two parameters, however, only matter *with respect to each other*. So for two random values a_0^0 and a_1^0, changing a_0^0 to some new random value a_0^1 has only a 33% chance of changing the value of the relevant test: ($a_0 < a_1$). The procedure outlined in Table 8 discovered this counterintuitive result automatically as shown in Table 10.

Function sensitivity approximation is useful exactly because it works without prior information about the function to be analyzed. This means that function sensitivity approximation also works on user defined functions. The Parameterized Signal Primitive *PSP-Variance*, an example user defined function used by NP, also produces informative sensitivity values.

As just described, all functions are evaluated within the context of the training examples on which the NP programs will operate. This does not affect many functions (e.g., "ADD") but certainly has an effect on an input-sensitive function like PSP-Variance. PSP-Variance requires four parameters and this is easily detected. More interesting, Table 11 shows that the second and fourth parameters to PSP-Variance are slightly more sensitive to changes than are the first and third parameters. This does not "mean" anything to NP and IRNP, it is taken for the process of evolution as a fact about the world. But we can step back and see that, because PSP-Variance interprets its four inputs (x_1, y_1, x_2, y_2) as a rectangle in a video image with upper left corner (x_1,y_1) and lower right corner (x_2,y_2), this 3% additional sensitivity for the second and fourth parameters tells us that the particular images in this domain tend to be very slightly more variable along the Y-axis. In itself this does nothing to help solve the pattern recognition problem, but it is an interesting side-effect of this process of automatic discovery of function argument sensitivity.

# Params	arg 1 sensitivity	arg 2 sensitivity	arg 3 sensitivity	arg 4 sensitivity
4	0.853429	0.887229	0.846371	0.875599

Table 11: **PSP-VARIANCE** (min arity is **4** and max arity is **4**)

5.3 Refining the Credit-Blame Map

We can now combine the topology of the NP program, the explicit credit score for each node, and the sensitivity values of each primitive function in a bucket-brigade style backward propagation. This bucket-brigade refines the credit scores at each node following the procedure presented in Table 12. The credit scores are refined according to the network topology and sensitivity of the node functions.

Until no further changes
  For each node x **in the program**
    For each output arc (x, y) **of that node**
      y is, by definition, the destination node of (x, y)
      Let f_y be y's node function
      Let A_y be the number of inputs y has
      Let i be such that (x, y) provides a_i to y
      Let $\mathcal{S}_{f_y,A_y,i}$ = Sensitivity of f_y (relative to A_y and i)
      $\mathbf{CS}_x = \mathbf{MAX}(\mathbf{CS}_x, \mathcal{S}_{f_y,A_y,i} * \mathbf{CS}_y)$

Table 12: The process of bucket brigading the Credit Scores (CS) throughout an NP program.

The high level structure of the procedure presented in Table 12 is as follows. For each node, for each output arc from that node, the node's credit-score will be updated to be the maximum of the credit-score it already has and the credit-score of the node pointed to by that output arc *times* the sensitivity of that destination node to that particular output arc. We explain this process in detail through a series of questions and answers designed to identify the important elements of this procedure.

A good first question for this particular method of spreading credit and blame out more appropriately over each neural program is, "does this process always converge?" The answer is that as long as the definition of "no further changes" is more specifically "no node changed its CS value by more than ϵ" ($\epsilon > 0$) then the process always[6] halts and typically in only a few passes.

Note that because of the way $\mathcal{S}_{f,A,i}$ is defined (see Table 8), parameter a_i is very occasionally replaced *by itself* and so SENSITIVITY is usually less than 1.0, contributing to the small

[6]Proof: If the halt criteria isn't satisfied after a pass, then at least one node credit score has increased by at least ϵ and no credit score has decreased in value (by construction, see Table 12). The total value in the Credit-Blame map for program p can be at most N_p (the number of nodes in p), so the total number of loops can be no more than $\frac{N_p}{\epsilon}$.

number of passes required for the Credit-Blame map to reach quiescence. This answer to the convergence question is also the answer to the question, "why do not you use a discount factor (γ)? Is not that usual in various forms of bucket brigade?" Using a discount factor is a common way to insure convergence, but as just noted, it is empirically unnecessary.

In this context, in which we make clear the use of a sensitivity value for each function, we can now ask "why define sensitivity in that way?" Remember that we said that the sensitivity of function f_y with arity A to input a_i is the likelihood that the output will change *at all* when the value of a_i is changed to a new *random value* selected uniformly from the legal range of values.

There is no reason to believe that, in a complex system such as an evolving NP program, a node that outputs O_1 will always have a similar effect to a node that outputs O_2, no matter how close O_1 and O_2 are on the number line. For example, consider the function READ-MEMORY(O_1) that returns the value stored in the program's memory array index O_1. Out of context of a particular program, READ-MEMORY(5) and READ-MEMORY(6) have as much semantic similarity as READ-MEMORY(5) and READ-MEMORY(77). For this reason, sensitivity in NP is a percentage of how often the output value of a function is changed **at all**, not *by how much* that output changes.

There is also no reason to believe that in a complex system such as an evolving NP program, any particular set of numbers is more or less likely than any other to occur as inputs to a node.[7] The sensitivity discovery process described in Table 8 could, for example, change a_i to $(a_i \pm \Delta)$. Then $\mathcal{S}_{f,A,i}$ would measure the likelihood that the output will change when small changes are made to the input a_i. But since, unlike explicit credit-blame assignment systems (e.g., ANNs), NP cannot enforce these small changes throughout the program, it is better to have a measure of sensitivity that matches how the inputs are likely to change: to first approximation, *uniform randomness*.

Finally, consider the equation for refining the credit scores: $\mathrm{CS}_x = \mathbf{MAX}(\mathrm{CS}_x, \mathcal{S}_{f_y,A_y,i} * \mathrm{CS}_y)$. "Why should CS_x be set to the maximum of itself and $\mathcal{S}_{f_y,A_y,i} * \mathrm{CS}_y$?" We first address the function MAX as an appropriate operator and then examine the appropriateness of the second operand. In an NP program it is the norm for a single node's output to be used in a number of different contexts. We would not want to penalize a node for creating an output that is very useful in one part of the program, but is not taken advantage of in another part of the program. If the outputs of a node could be "taken advantage of" (in the sense defined by the explicit credit score measure), then it is clear that the blame for not taking advantage of that output elsewhere in the program is a problem with that other part of the program, not the node in question. This means that a node's credit score should be a maximum of *some function* of its individual credit score and the credit scores of the nodes to which it outputs.

Further, consider the case in which node x has an explicitly computed credit score of CS_x. Even if none of x's children (i.e., nodes that take x's output as input) has a credit score as high as CS_x, if we believe that the explicit credit score measure is a good first approximation to the usefulness of a node in a program, then we should insure that CS_x

[7] This is of course, not entirely true. There are always some more "popular" numbers in a system, but this regularity is often caused by the *interactions* between the various functions and this is sufficiently complicated that the uniform approximation was adopted.

is never less than its original value. Thus, we introduce $\mathrm{CS}_x = \mathrm{MAX}(\mathrm{CS}_x, F_r(\mathrm{CS}_y))$ where F_r is some function to be determined. Now we need to pick some reasonable function F_r to apply to the credit scores of the children of node x.

The introduced sensitivity analysis of Section 5.2. can now be used. We already have a value that expresses the sensitivity of a node y to an input a_i as a function of how many inputs y has and the particular function that y happens to compute. But that's exactly what we want! The amount of reward (think CS_x) a node x that points to a node y deserves for that "reference," is exactly how good node y is, CS_y, scaled by (i.e., times) how responsive (i.e., sensitive) y is to changes in the values that x is passing it. So we have our function $F_r(\mathrm{CS}_y)$; it is $\mathcal{S}_{f_y,A_y,i} * \mathrm{CS}_y$.

This discussion highlighted the characteristics of our reinforcement procedure. So in summary, the refinement of credit scores in the Credit-Blame map is derived from the initial credit scores, the program's topology, and the discovered sensitivity of each possible node function.

5.4 Credit Scoring the NP arcs

NP program transformations operators (e.g., crossover and mutation) also affect NP program arcs. So far, the discussion of the Credit-Blame map has entirely focused on assigning credit and blame to the nodes. The topology of the NP programs, that is the program nodes *and* arcs, is used heavily in making this map, but the resulting map assigns one floating point number to each node and no number to the arcs.

The explanation for this discrepancy is that arcs are even more context dependent than the nodes that define them. For example, when considering whether to delete a particular arc (x, y), CS_y is a relevant value, but the value of CS_x is much less so. When, on the other hand, considering whether to reroute arc (x, y) to some other node z (i.e., $arc(x, y) \rightarrow arc(x, z)$) the current values CS_x, CS_y, and CS_z are all relevant. As is detailed in the next section, the Credit-Blame map has a great impact on the arcs during the IRNP process, but only indirectly through the credit scores of the nodes in the program to be recombined.

5.5 Exploration vs. Exploitation Within a Program

We have already touched on the issue of exploration vs. exploitation within the search process. A similar tension exists within the recombination of a single program. On the one hand, it seems clear that IRNP should leave alone the "best" parts of the program and focus its changes on the "worse" program aspects. There are, however, two problems with this view. The first is that a "bad" part of the program must be more carefully defined. There are program nodes that have very low scores in the program's Credit-Blame map and **do** affect the values flowing into the OUTPUT nodes and there are low score nodes that **do not** affect the values flowing into the program OUTPUT nodes. This is the *node participation* problem. To be most effective, IRNP should change the first type of low score nodes, but not the second. This is so because, for example, changing what function a particular node computes is a piece of wasted search if that node's old function had no effect on any of

```
For each node x in the program
    Participation_x ← 0
For each node x in the program
    if (node x is an OUTPUT node)
      Participation_x ← 1
While (flags still changing)
    For each node x in the program
      if  (arc (x, y) exists and creates a_i for node y) and
          (node y has S_{f_y,A_y,i} > 0) and
          (Participation_y = 1)
        Participation_x ← 1
```

Table 13: The procedure for assigning the *participation flags* to nodes in each program's Credit-Blame map.

the program's OUTPUT nodes (under the assumption that none of the functions have side-effects).

The second problem with seeing IRNP's job as simply focusing on the "bad" parts of a program is that, occasionally, the best way to improve a program is to make the right change to an aspect of the program that is already working well. It is easy to imagine a program in which node y computes $a_0 + a_1$ is *almost* right, but the program would work even better if that node computed $a_0 * a_1$ instead.

IRNP does address both of these issues. With regards to the second problem, IRNP *does* occasionally change high credit-score aspects of a program. It is partly for this very reason that the mutation operators only look at a fraction of the nodes in a program before picking one to change. This means that with low probability, the "worst" program aspect seen by a particular mutation operator, will still be one of the high credit-score nodes for that program. An interesting piece of future work for IRNP is the following. Instead simply restricting how often the recombination operators change high credit-score aspects of a program, *how* these aspects are changed could be different. In other words, for example, mutation could be further refined so that it did "less damaging" mutations when a high credit-score node was chosen to be changed (e.g., ADD $\rightarrow$ MULT is "less damaging" than ADD $\rightarrow$ If-Then-Else).

IRNP also addresses the node participation problem. If the Credit-Blame map had a *participation* flag for each program node, IRNP could take advantage of these flags. This is exactly the case. We now describe how these flags are set. These participation flags are set using the process shown in Table 13. Now that the Credit-Blame map includes these participation flags, the mutation and crossover operators can be adjusted to take advantage of them.

6 Using a Credit-Blame Map

The second phase of the internal reinforcement is the use of the created Credit-Blame map to increase the probability that the program updates lead either to a better solution or to a similar solution in less time. There are two basic ways that the Credit-Blame map can be used to do this enhancement: through improvement of either the mutation or crossover operators. In brief, mutation is the process of recombination of a single genotype and crossover is the process of recombination of two or more genotypes through genetic material exchange. Considerable information is available on these methods through sources such as [20].

The possibility of using internal reinforcement (explicit credit-blame assignment) not only for mutation (which has analogies to the world of ANNs) but for crossover as well is important. Traditional GP uses random crossover and relies entirely on the mechanism of empirical credit-blame assignment. Work has been done to boot-strap this mechanism by using the evolutionary process itself to evolve improved crossover procedures (e.g. [6, 35]). This work has reaped some success, but because of the co-evolutionary nature of the work, it has not yielded much insight into the basic mechanism of crossover. IRNP has the potential not only to improve on the existing GP mechanism, but also to help *explain* the central mystery of GP, namely crossover. That is, crossover often works better than mutation in GP, but why is still poorly understood. IRNP, or some improved descendant of IRNP may be able to provide explicit information on crossed-over programs and learn what if any real value crossover adds to GP. This one of the interesting directions for future work.

6.1 Mutation: Applying a Credit-Blame Map

Mutation can take a variety of forms in NP. These various mutations are: add an arc, delete an arc, swap two arcs, change a node function, add a node, delete a node. Notice that "change a node" function and "swap two arcs" are not atomic, but have been included as examples of non-atomic, but basic mutation types. In the experiments shown in the next section, each of these mutations took place with equal likelihood in both the random and internal reinforcement recombination cases. For example, to add an arc under random mutation to an NP program, we simply pick a source and destination node at random from the program to be mutated and add the arc between the nodes.

Internal reinforcement can have a positive effect on this recombination procedure. For each recombination type, we pick a node or arc (depending on the mutation type) that has maximal or minimal credit score as appropriate. For example, when deleting a program node, we can delete the node with the lowest credit score instead of just deleting a randomly selected node.

Below are the IRNP procedures for each of the six mutation types mentioned above. Notice that when the terms "large" and "low" are used (as opposed to the unambiguous terms "highest" and "lowest"), this indicates that the largest or least credit score is selected from among a *sampled subset* of nodes or arcs, depending on the context.

- **Add an Arc**

  1. Pick a node x with a large credit score.

2. Pick a node y with a low credit score **and Participation**$_y = 1$ and A inputs such that y would still be sensitive to input a_{A+1}
3. Add an arc (x, y)

- **Delete an Arc**

1. Pick a node y with a low credit score such that y would still be sensitive to its inputs if one were removed **and Participation**$_y = 1$.
2. Pick a node x with a low credit score such that there exists an arc (x, y).
3. Delete arc (x, y).

- **Swap Two Arcs** (see Figure 9)

1. Let x be the node with highest CS_x.
2. Let (x, y) be the output arc of x to a node y that minimizes CS_y.
3. For all arcs (u, v) such at v is an OUTPUT node, pick the arc (u, v) that minimizes CS_u.
4. Delete arcs (x, y) and (u, v) and create arcs (x, v) and (u, y).

- **Change a Node Function**

1. Pick a node x that has a low credit score and such that (x, y) exists and creates input a_i for node y and $\mathcal{S}_{f_y, A_y, i} > 0$ **and Participation**$_y = 1$.
2. Change the function that x computes to another function of similar or lower arity.

- **Add a Node**

1. Create a new node z with f_z, a randomly selected function.
2. Let A be the arity of f_z and let O_z be the number of output arcs from z.
3. Find high credit score nodes x_1, ...x_A and create the arcs (x_1, z) ... (x_A, z).
4. Find low credit score nodes y_1, ... y_{O_z} such that $\mathcal{S}_{f_{y_i}, A_{y_i}+1, A_{y_i}+1} > 0$ **and Participation**$_{y_i} = 1$ for all i in $[1..O_z]$.
5. Create the arcs (z, y_1) ... (z, y_{O_z})

- **Delete a Node**

1. Pick a low credit score node x **with Participation**$_x = 1$.
2. Remove x and arcs (x, y) and (z, x) for all nodes y and all nodes z in the program.

For each of the procedures, the alternative to IRNP is the equivalent of the traditional recombination strategy in GP. This "vanilla" strategy in NP is simply to chose randomly among all syntactically legal options (i.e., no program-behavior based bias in the recombination). Equivalently, this "vanilla" method for recombination can be thought of as IRNP with random values in the Credit-Blame map.

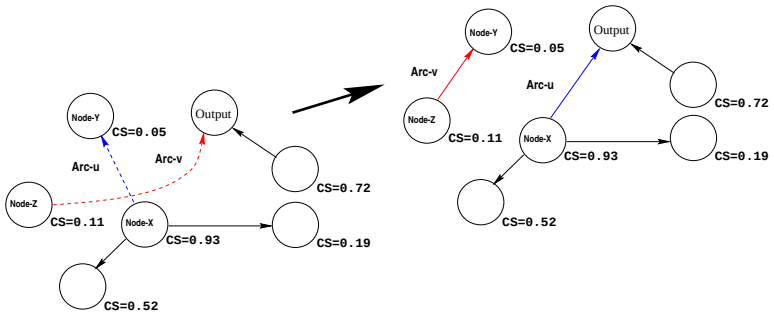


Figure 9: This demonstrates the *Swap Two Arcs* mutation procedure.

6.2 Crossover: Applying a Credit-Blame Map

In the random version of crossover, one simply picks a "cut" from each graph (i.e., a subset of the program nodes) at random and then exchanges and reconnects them. Figure 10 pictures this division of a program into two pieces. Details on how this fragment exchange can be accomplished so as to minimize the disruption to the two programs can be seen in [35].

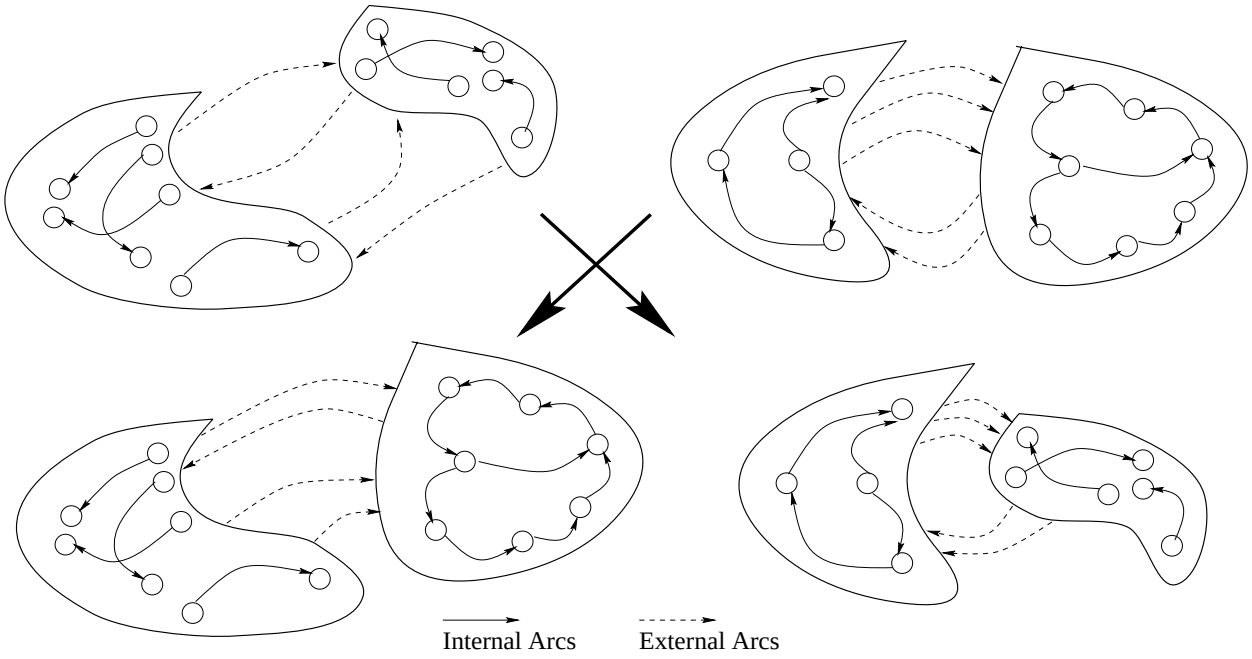


Figure 10: Crossover in NP: A single graph of nodes and arcs is fragmented with a cut into two fragments such that every node in the original graph is now either in Fragment_1 or Fragment_2 and every arc is either an *internal* or *external* arc.

We keep this underlying mechanism and present an IRNP procedure that selects "good" program fragments to exchange. This means that IRNP has, as it's only job to *choose the fragments* to be exchanged, but the way in which program fragments are exchanged and reconnected is unaffected by IRNP. There is much to be gained by taking advantage of the Credit-Blame map during this fragment exchange and reconstitution phase, but to focus this work and its contributions, this aspect of the use of credit-blame assignment has been left

as future work.

Given that we separate a program into two fragments before crossover, let us define *CutCost* to be the sum of all credit scores of *inter*-fragment arcs, and *InternalCost* to be the sum of all credit scores of *intra*-fragment arcs in the program to be crossed-over.

NP program arcs have a shifting meaning and so their credit score must be interpreted within the context of the search operator being used. *For the context of crossover we take the credit score of an arc to be the credit score of its destination node.*

Now we say that the cost of a particular fragmentation of a program is equal to *CutCost/InternalCost*. If we try to minimize this value for both of the program fragments we choose, we are much less likely to disrupt a crucial part of either program during crossover. Table 14 outlines this IRNP crossover procedure.

Pick k random cuts of prog p (Fragment$_1$, Fragment$_2$)
For candidate cut i
  For each arc(x, y) in p
    Let CS$_{arc(x,y)}$ = CS$_y$
    if (x and y are in the same Fragment$_j$) (j$\in${1,2})
      InternalCost = InternalCost + $CS_{arc(x,y)}$
    else
      CutCost = CutCost + CS_y
  $CutRanking_i$ = CutCost / InternalCost
Choose the cut that produced the LOWEST *CutRanking* **with at least one participating node on each side of the cut**

Table 14: The IRNP process for choosing a "good" fragment of a program to exchange through crossover.

7 Experimental Results: With and Without IRNP

Neural programming is a representation for the evolution of programs. This representation is part of the PADO project on signal understanding [36, 38, 37, 35].

This is not an article about the machine learning system called PADO, but because the following experiments were run in the context of PADO a few words must be given as explanation. Most briefly, PADO is a learning environment that decomposes classification problems into discrimination problems, evolves sub-solutions to these discrimination problems, and then orchestrates these sub-solutions into an overall solution to the original classification problem. The evolution that takes place inside PADO can be of any type and in the context of this article that evolution is the evolution of NP programs with and without IRNP for the purposes of comparison. Significant detail about PADO can be found in [37, 38, 39].

Put slightly differently, at the highest level, PADO divides the signal classification problems (i.e. "which of C classes is this signal?") into many different binary discrimination problems (i.e. "is this signal from class i or not?"). After each generation, PADO attempts to orchestrate one or more of the programs evolved in each of these discrimination populations in order to solve the original signal classification problem.

Our research (e.g., [40]) has demonstrated that PADO and NP are highly effective across a wide range of signal types and sizes, both real world and manufactured. The aim of this section is to demonstrate, on two distinct real-world signal classification problems, that NP can learn a difficult problem and that IRNP is a significantly more effective way to perform recombination on the population than is random recombination.

7.1 Natural Images

7.1.1 Description of the Domain and Problem

There are seven classes in the domain used in the following experiments. Figure 11 shows one randomly selected video image from each of the seven classes in both the training and testing sets. This particular domain was created as a domain for machine learning and computer vision [41]. Each element is a 150x124 video image with 256 level of grey. Originally, these images were color images, but the color was removed from the images to make the problem more challenging [38].

The seven classes in this domain are: Book, Bottle, Cap, Coke Can, Glasses, Hammer, and Shoe. The lighting, position and rotation of the objects varies widely. The floor and the wall behind and underneath the objects are constant. Nothing else except the object is in the image. However, the distance from the object to the camera ranges from 1.5 to 4 feet and there is often severe foreshortening and even deformation of the objects in the image.

7.1.2 Setting NP up to Solve the Problem

In the experiment in this section, the total population size was 1750 (i.e., $250 * 7$). Each point on each graph is an average of at least 60 independent runs. A total of 350 (50 from each of 7 classes) images were used for training and a separate set of 350 (50 from each of 7 classes) images were withheld for testing afterwards.

The PSPs used in this experiment were as follows:

PSP-Point(a_0, a_1) returns the pixel intensity at the pixel/point (a_0, a_1).

PSP-Average(a_0, a_1, a_2, a_3) returns the *average* pixel intensity in the image region specified by the rectangle with upper left corner (a_0, a_1) and lower right corner (a_2, a_3).

PSP-Variance(a_0, a_1, a_2, a_3) returns the *variance* of the of the pixel intensities in image region specified by the rectangle with upper left corner (a_0, a_1) and lower right corner (a_2, a_3).

PSP-Min(a_0, a_1, a_2, a_3) returns the *lowest* pixel intensity value in the image region specified by the rectangle with upper left corner (a_0, a_1) and lower right corner (a_2, a_3).

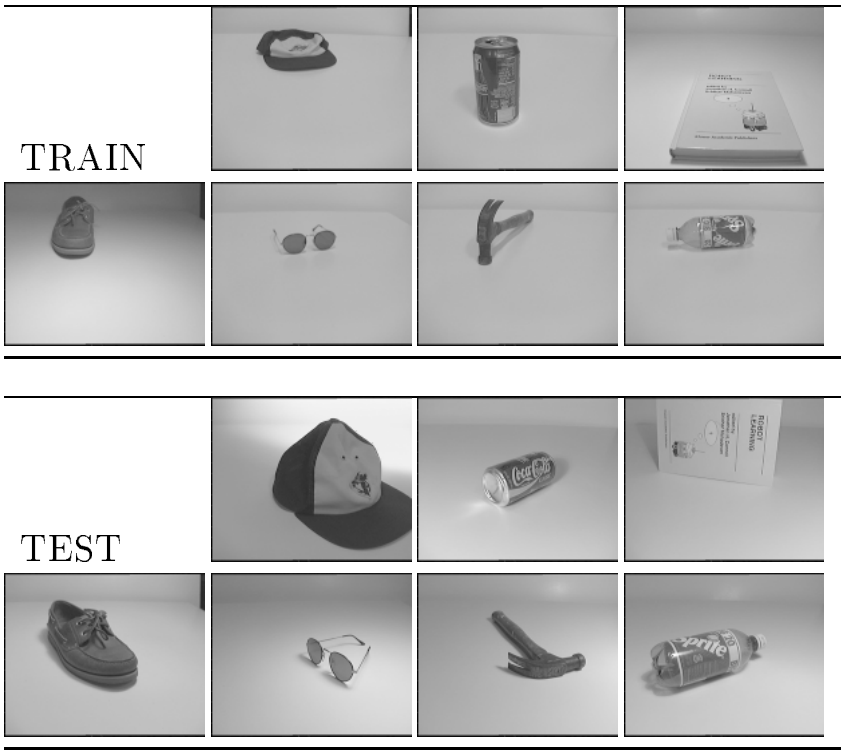


Figure 11: A random training and testing signal from each of the 7 classes in this classification problem.

PSP-Max(a_0, a_1, a_2, a_3) returns the *largest* pixel intensity value in the image region specified by the rectangle with upper left corner (a_0, a_1) and lower right corner (a_2, a_3).

PSP-Diff(a_0, a_1, a_2, a_3) returns the absolute different between the *average* pixel intensity above and below the diagonal line (a_0, a_1) to (a_2, a_3) inside the bounding rectangle with opposite corners (a_0, a_1) and (a_2, a_3).

7.1.3 The Results

During each run, the generalization performance on a separate set of testing images was recorded and Figure 12 plots the *mean* of each of these values. Figure 12 shows the computational effort in generations required to reach a particular level of test-set generalization performance for NP learning with and without IRNP.

The most important feature of Figure 12 is that NP learns more than *twice as fast* when IRNP is applied to the recombination during evolution. Also notice that NP learns quite well on this difficult image classification problem. Random guessing in this domain would achieve only about 14.28% correct generalization performance.

It is worth noting that the performance that achieved on any domain is related to the particular orchestration strategy chosen. NP has, on this particular domain, achieved generalization performance rates as high as **86%**.

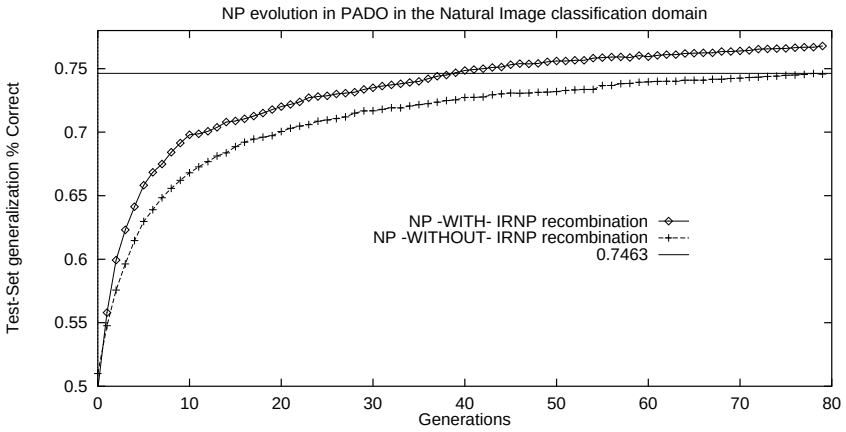


Figure 12: NP learning **with and without IRNP**.

7.2 Acoustic Signals

7.2.1 Description of the Domain and Problem

The database used in this experiment contains 525 three second sound samples. These are the raw wave forms at 20K Hertz with 8 bits per sample for about 500,000 bits per training or testing sound. These sounds were taken from the SPIB ftp site at Rice University (anonymous ftp to spib.rice.edu). This database has an appealing seven way clustering (70 from each class): *the sound of a Buccanneer jet engine*, *the sound of a firing machine gun*, *the sound of an M109 tank engine*, *the sound on the floor of a car factory*, *the sound in a car production hall*, *the sound of a Volvo engine*, and *the sound of babble in an army mess hall*. There are many possible ways of subdividing this sound database; the classes chosen for these experiments are typical of the sort of distinctions that might be of use in real applications.

7.2.2 Setting NP up to Solve the Problem

In the experiment in this section, the total population size was 1750 (i.e., $250 * 7$). Each point on each graph is an average of at least 55 independent runs. A total of 245 (35 from each of 7 classes) images were used for training and a separate set of 245 (35 from each of 7 classes) images were withheld for testing afterwards.

The PSPs used in these experiments were as follows:

PSP-Point(a_0, a_1) returns the wave height at the moment in time specified by $(a_0 * 256 + a_1)$.

PSP-Average(a_0, a_1, a_2, a_3) returns the *average* wave height in the sound starting at time $(a_0 * 256 + a_1)$ and ending at time $(a_2 * 256 + a_3)$. This PSP is useless for long time periods since it's return value will be, by definition, the waveform's midline.

PSP-Variance(a_0, a_1, a_2, a_3) returns the *variance* of the wave height in the sound starting at time $(a_0 * 256 + a_1)$ and ending at time $(a_2 * 256 + a_3)$.

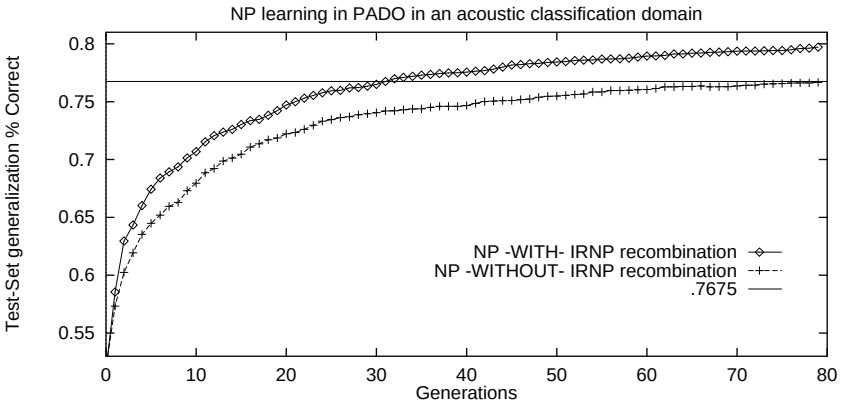


Figure 13: NP learning **with and without** IRNP.

PSP-Min(a_0, a_1, a_2, a_3) returns the *lowest* wave height in the sound starting at time $(a_0 * 256 + a_1)$ and ending at time $(a_2 * 256 + a_3)$.

PSP-Max(a_0, a_1, a_2, a_3) returns the *largest* wave height in the sound starting at time $(a_0 * 256 + a_1)$ and ending at time $(a_2 * 256 + a_3)$.

PSP-Diff(a_0, a_1, a_2, a_3) is equivalent to ABS(PSP-Average(a_0, a_1, $a_{0'}$, $a_{1'}$) − PSP-Average($a_{0'}$, $a_{1'}$, a_2, a_3)) where ($a_{0'}$,$a_{1'}$) is the time midpoint between (a_0, a_1) and (a_2, a_3).

Notice that, other than minor adjustments necessary to reflect the change in signal type, these parameterized signal primitives are exactly the same as the PSPs used in the visual classification experiment discussed in Section 7.1. This was not done to demonstrate the generality of these PSPs. Quite the contrary, this similarity in the experimental procedure was done to highlight how little was done to tune NP in order to achieve the reported results. NP, evolving with or without IRNP, is able to make good use of these very simple PSPs that are not well focused to solving either of the domains in which they were applied.

The fitness used for evolutionary learning (training of the NP programs) was based upon distance from returned confidence to the correct confidence for each training example. Given this model of one class chosen per sound, if NP just guessed, it could achieve an generalization performance of 1/7 (14.28%) correct[8].

7.2.3 The Results

Figure 13 shows the generalization percent correct NP reaches on average on each generation, with and without IRNP.

Notice that in these experiments, for both orchestration strategies, IRNP learning is almost *three times as efficient* as learning without it.

[8]Given that the training and test sets have an even number of signals from each class.

8 Related Work

We now situate our work on Neural Programming and Internal Reinforcement within the larger context of machine learning in AI. While it is nearly impossible to read everything that has been written on a field the size of algorithm evolution, we believe that NP and IRNP represent important original contributions to the field. Therefore, this article focuses on the similarity between our work and other research programs rather than their differences.

8.1 Algorithm Evolution

Neural Programming is an extension of the genetic programming paradigm. Genetic programming (GP) is a term for the automatic generation of programs by means of natural selection. Genetic programming as a term began with [20], but the origins of algorithm evolution are much older. As long back as the 1960's, work like [16] was laying the ground work for what genetic programming has become today. [11] was the work that formed the bridge from traditional GA to the field that has since become genetic programming.

The NP and IRNP approaches are used, in the context of our work, for signal classification problems. Genetic programming has been applied to a number of visual and acoustic classification problems. In most of these cases, this work has looked at small images (e.g., font bitmaps [3, 21]). In the cases where larger signals are examined, GP is almost always used as an aid, not the actual program that examines the signal directly (e.g., [24, 32, 13]). Time series prediction has also been the topic of some GP research, but again, only successful in specific cases (e.g., [25]). The use of arbitrary memory was introduced into genetic programming in [33] and later papers have examined aspects of memory and data structures in evolved programs (e.g., [22, 4, 2, 9, 23]). [34] demonstrated that GP, with a few paradigmatic additions, was Turing complete. Some research has been done on recursion and looping in GP (e.g., [19, 10, 22]), but how to tractably evolve complex programs with iteration and/or recursion and extensive, effective memory use is still very much an open question.

GP is by no means the only machine learning method that employs evolution as a learning model. Of the most immediate relation to this research, Evolutionary Programming (EP) was first described as a method for learning finite state machines (FSMs) [16]. More recently EP has, as a field, moved to evolving vectors of real numbers. Some work in EP still is related to aspects of the NP representation and IRNP. [15] is a good example of this work. In [15], finite state machines are evolved and used to predict time series data. FSMs are topologically identical to NP programs. What happens at each node is entirely different though. FSMs are flow of control and the "computation" done at each node is simply to decide which node (state) to pass control to next based on the next input symbol. FSMs can also be described as data-flow machines, but the effect is the same. [15] is notable in that a self-adaptation strategy is used to try to improve the mutation operator acting on these FSMs which has a similar goal to that of IRNP, though the method for solving the problem is very different.

Both because of its representational similarities and because of its computational class equivalence (i.e., both are Turing complete representations), recurrent ANNs (e.g., [29]) are

also of relevance to the NP and IRNP research.

8.2 Focused Search Policies in Program Space

Both GP (e.g., [20, 8]) and ANNs (e.g., [29]) continue to be well investigated. In ANNs, the focus on improving the power of the technique has not been on changing what is inside an "artificial neuron." Works like [14, 30] have, however, investigated the possible additional benefit of complicating and un-homogenizing artificial neurons. To the best of our knowledge, in the context of ANNs and principled update policies like backpropagation, these investigations have not yet been extended to arbitrary, potentially non-differentiable functions like those typically used by human programmers and by GP programs.

Though its similarity to NP is in representation, not in use or objectives (i.e., IRNP), [27] is worthy of note. Poli describes Parallel Distributed Genetic Programming (PDGP), a type of genetic programming which can be used for the development of parallel programs in which symbolic and neural processing elements can interact. PDGP is based on a graph-like representation of the parallel programs. These programs, can be changed through crossover and mutation operators in the style of the traditional GP paradigm. [7] is another recent example of GP moving from the traditional tree structured representation out into the more general world of general graph representation for program evolution.

In GP, the focus of investigation for increased power of the technique has generally not been on changing the GP representation or on finding principled (non-random) update policies, as in our work. Some work, has, however, been done in these areas. [28] describes a process for trying to find sub-functions in an evolving GP function that are more likely than randomly selected ones to contribute positively to fitness when crossed-over into other programs. [6] and [15] describe possible approaches for allowing the mechanism of evolution to provide self-adaptation all the way down to the single node level.

The ADATE work creates a very different representation for algorithm evolution with the goal of improving the efficiency of the search [26]. In ADATE, programs are evolved in pure functional ML (i.e., no loops, only recursion). ADATE does not use mutation and crossover, but instead defines a set of *transformations* that are always syntactically legal, type legal, and, with some of the transformations, guaranteed to produce a new program that performs at least as well on the training set (though of course that guarantee does not extend to a random sampling from the same distribution of inputs). These program transformations do not, however, react to the *behavior* of the programs, nor target specific aspects of a program for change because of those observed behaviors.

As was described in Section 4, issues of credit-blame assignment are central to the NP representation and the IRNP procedure. The IRNP procedure uses a form of bucket-brigade to deal with part of the credit assignment problem. The contribution of this work is, however, not the bucket brigade solution to the credit assignment problem. The contributions of NP and IRNP do include the identification of a credit assignment problem in EC and the application of the bucket-brigade algorithm to help tackle that issue.

When looking at related work in the credit assignment area, even restricting our attention to the field of AI, the problem of credit assignment has been discussed in a wide variety of contexts. The bucket-brigade algorithm is one of the oldest versions discussed as an explicit

mechanism by Holland [17] or as an implicit mechanism in works such as [44]. The variant of a *profit-sharing plan* was introduced in [18].

The bucket-brigade algorithm is just a special case of the general temporal difference methods (TDM) [31] like Q-learning (though that is not the historical order of the two ideas) [42]. Back propagation is another form of TDM and so the connection can also be made to the bucket-brigade algorithm. This connection is brought out in works such as [12]. For an excellent short introduction to some of these issues, see [43].

9 Conclusions

This article has contributed a new representation for learning complex programs and a procedure to achieve explicit credit blame in program evolution. The new connectionist program language, *neural programming,* has been developed with the goal of enabling a new principled update framework for algorithm evolution. This work introduces *internal reinforcement* as the first such principled update mechanism created for the field of genetic programming. Neural programming enables the construction of a *credit-blame map* for each evolving program. We further introduce a *sensitivity function approximation* algorithm to compute the principled feedback analysis for the general function primitive constructs of a program. A sensitivity-based bucket-brigade leads to the construction of a *credit-blame map* for each program with sufficient detail to allow internal reinforcement to perform focused, beneficial search operations during program evolution.

We illustrated these techniques with empirical experiments that show that internal reinforcement improves the speed and accuracy of neural programming learning. The experiments also demonstrated that neural programming can successfully learn to correctly classify large signals from different classes in real world domains.

The goal of this article has been to communicate the exciting result that, through the exploration of new program representations, we have captured the explanation and principled update power of explicit credit-assignment with the flexibility and generality of classical genetic programming.

Acknowledgements

The first author is supported through the generosity of the Fannie and John Hertz Foundation.

This research is sponsored in part by the Department of the Navy, Office of Naval Research under contract number N00014-95-1-0591. Views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of the Department of the Navy, Office of Naval Research or the United States Government.

References

[1] L. Altenberg. The evolution of evolvability in genetic programming. In Jr. K. Kinnear, editor, *Advances In Genetic Programming*, pages 47–74. MIT Press, 1994.

[2] D. Andre and A. Teller. A study in program response and the negative effects of introns in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, page 12, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[3] David Andre. Automatically defined features: The simultaneous evolution of 2-dimensional feature detectors and an algorithm for using them. In K. Kinnear, editor, *Advances In Genetic Programming*, pages 477–494. MIT Press, 1994.

[4] David Andre. The evolution of agents that build mental models and create simple plans using genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 248–255, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[5] P. Angeline. *Evolutionary Algorithms and Emergent Intelligence.* PhD thesis, Ohio State University, Computer Science Department, 1993.

[6] P. Angeline. Two self-adaptive crossover operators for genetic programming. In P. Angeline and K. Kinnear, editors, *Advances in Genetic Programming 2.* MIT Press, 1996.

[7] Peter J. Angeline. An alternative to indexed memory for evolving programs with explicit state representations. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 423–430, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[8] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, dpunkt.verlag, November 1998.

[9] Scott Brave. The evolution of memory and mental models using genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 261–266, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[10] Scott Brave. Using genetic programming to evolve recursive programs for tree search. In P. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 10. MIT Press, Cambridge, MA, USA, 1996.

[11] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In John J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA, 24-26 July 1985.

[12] H. Brown Cribs and Robert E. Smith. Classifier systems renaissance: New analogies, new directions. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 547–552, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[13] Jason Daida. Algorithm discovery using the genetic programming paradigm: Extracting low-contrast curvilinear features from SAR images of arctic ice. In P. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 21. MIT Press, Cambridge, MA, USA, 1996.

[14] F. Dellaert and R.D. Beer. Co-evolving body and brain in autonomous agents using a developmental model. In *Technical Report CES-94-16, Department of Computer Engineering and Science.* Case Western Reserve University, Cleveland, OH 44106, 1994.

[15] L. Fogel, P. Angeline, and D. Fogel. An evolutionary programming approach to self-adaptation on finite state machines. In J. McDonnell, R. Reynolds, and D. Fogel, editors, *Proceedings of the 4th Annual Conference on Evolutionary Programming*. MIT Press, 1995.

[16] L. Fogel, A. Owens, and M. Walsh. *Artifical Intelligence through Simulated Evolution*. New York: Wiley, 1966.

[17] J. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[18] J.H. Holland and J. S. Reitman. Cognitive systems based on adaptive algorithms. In *Pattern Directed Inference Systems*. Academic Press, 1978.

[19] Kenneth E. Kinnear, Jr. Generality and difficulty in genetic programming: Evolving a sort. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 287–294, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[20] J. Koza. *Genetic Programming*. MIT Press, 1992.

[21] J. Koza. *Genetic Programming 2*. MIT Press, 1994.

[22] William Langdon. Evolving data structures with genetic programming. In Stephanie Forrest, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*. Morgan Kauffman, 1995.

[23] William Langdon. Data structures and genetic programming. In P. Angeline and K. Kinnear, editors, *Advances in Genetic Programming 2*. MIT Press, 1996.

[24] T. Nguyen and T. Huang. Evolvable 3d modeling for model-based object recognition systems. In K. Kinnear, editor, *Advances In Genetic Programming*. MIT Press, 1994.

[25] E. H. N. Oakley. The application of genetic programming to the investigation of short, noisy, chatoic data series. In T. C. Fogarty, editor, *Lecture Notes in Computer Science: Evolutionary Computation*. Springer-Verlag, 1994.

[26] Roland Olsson. *Inductive Functional Programming using Incremental Program Transformation*. PhD thesis, University of Oslo, 1995.

[27] Riccarod Poli. Discovery of symbolic, neuro-symbolic and neural networks with parallel distributed genetic programming. In *Technical Report CSRP-96-14, School of Computer Science, University of Birmingham*. University of Birmingham, 1996.

[28] Justinian P. Rosca and Dana H. Ballard. Discovery of subroutines in genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 9, pages 177–202. MIT Press, Cambridge, MA, USA, 1996.

[29] D.E. Rumelhart, G.E. Hinton, and R.J Williams. Learning internal represenations by error propogation. In *Parallel Distributed Processing*. MIT Press, Cambridge, MA, USA, 1986.

[30] K. Sharman, A. Alcazar, and Y. Li. Evolving signal processing algorithms by genetic programming. In *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, 1995.

[31] R.S. Sutton. Learning to predict by the methods of temporal differences. In *Proceedings of the International Conference on Machine Learning*. AAAI Press, 1988.

[32] Walter A. Tackett. Genetic programming for feature discovery and image discrimination. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kauffman, 1993.

[33] A. Teller. The evolution of mental models. In Kenneth E. Kinnear, editor, *Advances In Genetic Programming*, pages 199–220. MIT Press, 1994.

[34] A. Teller. Turing completeness in the language of genetic programming with indexed memory. In *Proceedings of the First IEEE World Congress on Computational Intelligence*, pages 136–146. IEEE Press, 1994.

[35] A. Teller. Evolving programmers: The co-evolution of intelligent recombination operators. In K. Kinnear and P. Angeline, editors, *Advances in Genetic Programming 2*. MIT, 1996.

[36] A. Teller and M. Veloso. PADO: Learning tree structured algorithms for orchestration into an object recognition system. Technical Report CMU-CS-95-101, Computer Science Department, Carnegie Mellon University, 1995.

[37] A. Teller and M. Veloso. Program evolution for data mining. In Sushil Louis, editor, *The International Journal of Expert Systems. Third Quarter. Special Issue on Genetic Algorithms and Knowledge Bases.*, pages 216–236. JAI Press, 1995.

[38] A. Teller and M. Veloso. PADO: A new learning architecture for object recognition. In K. Ikeuchi and M. Veloso, editors, *Symbolic Visual Learning*. Oxford University Press, 1997.

[39] Astro Teller. *Algorithm Evolution with Internal Reinforcement for Signal Understanding*. PhD thesis, Computer Science Department, Carnegie Mellon University, 1998.

[40] Astro Teller. *Algorithm Evolution for Signal Understanding*. PhD thesis, Computer Science Department, Carnegie Mellon University, 1998, forthcoming.

[41] S. Thrun and T.M Mitchell. Learning one more thing. Technical Report CMU-CS-94-184, Computer Science Department, Carnegie Mellon Unversity, 1994.

[42] Christopher J.C.H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, 1989.

[43] S. W. Wilson and D. E. Goldberg. A critical review of classifier systems. In *Proceedings of the Third International Conf. On Genetic Alogirithms*. Morgan Kauffman, 1989.

[44] S.W. Wilson. Hierarchical credit allocation in a classifier system. In *Genetic Algorithms and Simulated Annealing*. Morgan Kaufman Publishers, 1987.