Efficient Learning Through Evolution: Neural Programming and Internal Reinforcement

Astro Teller ASTRO+@CS.CMU.EDU
BodyMedia, Inc., 4 Smithfield St. Suite 1200,‘ Pittsburgh, PA 15222, USA

Manuela Veloso MMV+@CS.CMU.EDU
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA

Abstract

Genetic programming (GP) can learn complex concepts by searching for the target concept through evolution of population of candidate hypothesis programs. However, unlike some 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 performance on the training data. GP is missing a clear, locally optimal update procedure, an equivalent of gradient-descent backpropagation for ANNs. This article introduces a new mechanism, "internal reinforcement," for defining and using performance feedback on program evolution. A new connectionist representation for evolving parameterized programs, "neural programming" is also introduced. 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 some of our extensive experiments that demonstrate the increased learning rate obtained by using our principled program evolution approach.

1. Introduction

There are many mechanisms that provide efficient supervised learning algorithms that vary along several dimensions, in particular in the way that the search progresses by updating the model being learned.

Supervised learning has successful representatives of the practice of *explicit credit assignment*. 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 results. Feed-forward artificial neural networks (ANNs) with the back-propagation algorithm are an example of an explicit credit-assignment learning approach.

In this paper, we use genetic programming (GP) as a machine learning mechanism. Concepts to be learned are defined as parameterized programs with powerful primitive constructs. GP searches for the target concept by evolving a population of candidate hypothesis programs. As opposed to ANNs, GP is a representative of the machine learning practice of *empirical credit assignment* (Angeline, 1993). The learner implicitly determines credit and blame of a hypothesis during its search without an explicit computation of which parts of the description need to be changed. Evolution does empirical credit assignment as it searches based on the evaluation of the fitness of different target descriptions (Altenberg, 1994), but it progresses by randomly changing the most promising descriptions.

There is therefore a gap in the way that machine learning algorithms perform credit assignment. The goal of this work is to bridge this gap, finding ways in which explicit and empirical credit assignment can find mutual benefit in a single machine learning technique.

This paper introduces a new GP approach to learn complex programs through evolution with explicit credit assignment. The method has been developed with the goal of incorporating a principled updating procedure in program evolution, until now unaccomplished in genetic programming. The GP learner identifies and updates specific parts or aspects of a program as a function of the program's performance. A new program representation, *neural programming*, is introduced as a connectionist programming language that supports a principled *internal reinforcement* in program evolution.

Neural programming organizes genetic programs into a network of nodes replacing program *flow of control* with *flow of data*. We introduce a method to accumulate explicit credit assignment directly attached to the nodes of the program network. The acquired feedback values are collectively referred to as the *credit-blame map*, representing the propagation of punishment and reward through each evolving program. We then introduce an *internal reinforcement* algorithm that uses the credit-blame map to provide a reasoned method to

guide search in the field of program induction.

When hill-climbing in a space of hypotheses, it is possible to sample a few points and then choose the best of those to continue from. When the gradient is available, however, it is more efficient, locally at least, to move in the direction of the gradient. Program evolution can work with fitness-guided random samplings of nearby points in program space. However in this paper, we show that program evolution converges more effectively if guided by our internal reinforcement procedure. Internal reinforcement acts as an approximation to the gradient function for program evolution.

In summary, this paper 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. The paper also includes a brief overview of genetic programming.

2. Neural Programming

The program representation used in GP can impact the performance of the algorithm. We first briefly overview GP. We then introduce our neural programming representation that supports the internal reinforcement learning procedure.

2.1 Brief Overview of Genetic Programming

Genetic programming is a subfield of general evolutionary computation in which programs in a population evolve to be *well fit* to represent a given dataset.

Determining the *fitness* of each program is hence the important first step in the evolutionary loop. This can be done in a large number of ways. Within a supervised learning task, Table 1 shows the mechanism we use for determining this "goodness" of each program.

The next step in GP, *fitness proportionate reproduction*, is the exploitation phase of the search in which attention is focused on the highest fit programs. There are a number of popular schemes for propagating the influence of the best fit individuals in the population, primarily *tournament selection*, *rank selection*, and *roulette selection*.

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

For each program p in the evolving population
For each training input S_i $\quad (1 \leq i \leq \lvert S \rvert)$
Call L_i the correct program label
Run program p on input S_i and get response R_p^i
$G_p = \sum_{i=1}^{\lvert S \rvert} \frac{\mathcal{F}(L_i, R_p^i)}{\lvert S \rvert}$,
where G_p is the fitness of program p and $\mathcal{F}$ is an arbitrary error function specific to the learning task.

Table 2 outlines *tournament selection* as the reproduction strategy used in our work. The resulting mating pool is of the same size as its parent population and programs have a representation in the new population proportional to their fitness.

Table 2. Outline of tournament selection in GP.

For a population of M programs, *Do M times*
Randomly pick K programs from the population.
Copy highest fit of these K programs for mating.

Genetic programming includes exploration and 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 programs are chosen and some "genetic material" is exchanged. In *mutation*, a single individual is taken and changed in some way that is independent of the other members of the population.

The details of the crossover and mutation mechanisms vary widely because the representations of the individuals also vary. The traditional GP crossover procedure is to choose one subtree from each program and exchange the subtrees, and mutation affects elements of a program. In general the material for genetic recombination is chosen *at random*.

There is no evidence that the random recombination common in GP is in general better than an eventual focused recombination. This paper provides a mechanism for an informed recombination, as well as specific evidence that it is effective to carefully and purposefully choose pieces of material to change or exchange during program transformations.

2.2 The NP Representation

We introduce the Neural Programming representation as 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, including arithmetic (e.g., multiplication, addition), memory-access (read, write), branching (e.g., if-then-else), and, most importantly, any potentially complex user-defined functions for examining the input data.

The main characteristics of the NP representation are:

- A neural program is a graph of nodes and arcs.

- Each NP node executes a function of some arity.
- An arc (x, y) from node x to node y indicates that the output of x is an input of y.
- On *each* time step t $(0 < t < T)$, *every* node takes the inputs corresponding to the arity of its function, computes the value of the function, and outputs it to *all* of its output arcs.
- One type of node function is "Output." Output nodes collect their inputs and create the program response through a function OUT of those values. We use OUT as an average weighted by its time-step.

We illustrate the NP representation through a set of constructed examples.

Fibonacci numbers Figure 1 shows a very simple neural program. This program computes the Fibonacci series and sends each successive element out of the program fragment on Arc4.

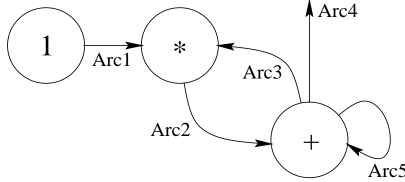

Figure 1. A simple neural program that computes the Fibonacci series. All arc values are initialized to 1.

Since neural 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, the initial value for all arcs is the value 1 and table 3 shows how the values of the arcs change over time.

Table 3. Progression of arc values over time for the neural program in Figure 1.

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

Golden mean The program in Figure 1 can be easily extended to a neural program that approximates the "Golden Mean."[1]. Figure 2 shows such program where we explicitly represent its OUTPUT node.

Table 4 shows the initial values of some of the arcs, where the OUTPUT node, computing the OUT function mentioned above, approximates the value 1.618034 of the golden mean.

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

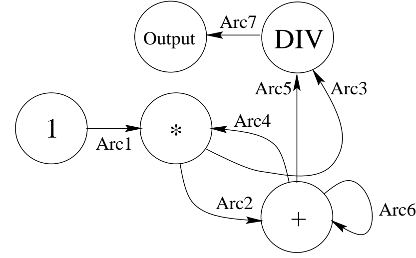

Figure 2. A neural program that iteratively approximates the golden mean. All arc values are initialized to 1.

Table 4. Progression of arc values over time for the simple neural program shown in Figure 2.

Step	Arc 3	Arc 5	Arc 7	OUTPUT
0	1	1	1	NA
1	1	2	1	1
2	2	3	2.0	1.5
3	3	5	1.5	1.5
4	5	8	1.6667	1.5417
5	8	13	1.6000	1.5533

Foveation The Fibonacci series and the golden mean 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 neural programs: the ability *to foveate*. Foveation is the process of changing the focus of attention in response to previous perceptions. Neural programs have the ability to use the results of an examination of the input signal to *guide* the next part of that examination.

Neural 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 an neural 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 3 shows what could be part of a larger neural program. The node indicated with a double circle computes the function PSP-Variance. The IF-T-E (If-Then-Else) node computes the function "if $(a_0 < a_1)$ then return a_2 else return a_3."

This particular neural 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 $(50,17,104,a_3)$. Table 5 gives the pseudo-code equivalent to the neural program fragment of Figure 3.

Assuming again that all arcs are initialized to

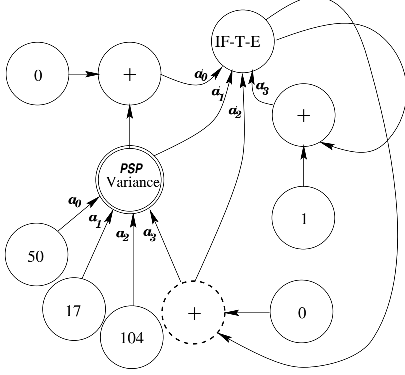

Figure 3. A neural program fragment where the output value from the dashed circle node is being iteratively refined to minimize the value returned by the PSP-Variance node.

Table 5. Pseudo-code for the NP fragment in Figure 3.

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

1, this program 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.

3. 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, namely to produce programs that are better, which is to say score on the fitness evaluations higher than their ancestors.

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.

3.1 Creating a Credit-Blame Map

Without loss of generality, we assume that the evolving neural programs are trying to solve a target value prediction problem. In fact, 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).

3.1.1 Explicit Credit Scores

For each program p and 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 absolute value of this correlation the explicitly computed *Credit Score* for node x, notated as CS_x. This computation is shown in Equation 1.

$$\text{CS}_x = \left| \frac{E(\vec{H_x} - \mu_{\vec{H_x}}) * E(\vec{L} - \mu_{\vec{L}})}{\sigma_{\vec{H_x}} * \sigma_{\vec{L}}} \right| \quad (1)$$

This credit score for a node indicates how valuable that node is to the program. This measured correlation provides an empirically-shown good linear approximation.

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 its usefulness *in the context of the program topology.*

As an example, let nodes x and y produce values and node z compute the XOR of those values. Even even if z has a high credit score, x and y may not. There is nothing provably wrong with this situation but the topological notion of usefulness has not been captured in these explicit credit scores. 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.

The Credit-Blame map is refined by passing credit and blame back through the topology of the neural pro-

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

grams. The statistical correlation between $\vec{L}$ and $\vec{H_x}$ presented above constitutes only the first explicit approximation to the credit score of node x.

3.1.2 Function Sensitivity Approximation

To pass back credit and blame through the neural program topology, we must first answer an important question: "What is the responsibility of each input value for the output value produced by a 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 neural programs as they may not 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 sensitivity is a substitute to the derivative of the function in question.

Table 6 shows the process for finding the sensitivity of each parameter of a general, possibly nondeterministic function f. The procedure is performed for all values of A between 1 and the maximum arity of the function. Nondeterminism can also be handled by this process and is adjusted for through the calculation of "Noise" as shown in Table 6.

The benefits of Function Sensitivity Approximation are particularly clear in the context of a non-differentiable functions, 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 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 6 discovered this counter-intuitive result automatically as shown in Table 7.

Table 6. 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.

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
$\mathcal{S}_{f,A,i} := \frac{\text{Sensitive} - \text{Noise}}{Q_s}$

Table 7. Input sensitivity for the function *IFTE*.

# Params	arg 1	arg 2	arg 3	arg 4
4	0.32	0.33	0.50	0.49

Function sensitivity approximation is useful exactly because it works without prior information about the function to be analyzed. The algorithm successfully applied it to all the user defined functions. Parameterized Signal Primitives used in the experiments.

3.1.3 Refining the Credit-Blame Map

We can now combine the topology of the neural 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 8. The credit scores are refined according to the network topology and sensitivity of the node functions.

Table 8. The process of bucket brigading the Credit Scores (CS) throughout a neural program.

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 (x, y) provide the ith input to y
   Let $\mathcal{S}_{f_y,A_y,i}$ = Sensitivity of f_y to A_y and i
   $CS_x = MAX(CS_x,\ \mathcal{S}_{f_y,A_y,i} * CS_y)$

The node's credit score is now updated to be the maximum of its explicit credit-score and the product of the credit score of its input node by the sensitivity of that destination node to that particular output arc.

This discussion highlighted the characteristics of our reinforcement procedure. 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.

3.2 Credit Scoring the NP Arcs

Neural program transformations operators (e.g., crossover and mutation) also affect neural 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 neural 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.

3.3 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 (Koza, 1992).

Internal reinforcement can have a positive effect on the recombination procedure. For each recombination type, we pick a node, arc, or part of the program as a function of its credit score. 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.

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 the "change a node" and "swap two arcs" functions are not atomic, but have been included as examples of non-atomic 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. And Figure 4 illustrates the mutation of swapping two arcs.

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. This exchange can be accomplished so as to minimize the disruption to the two programs.

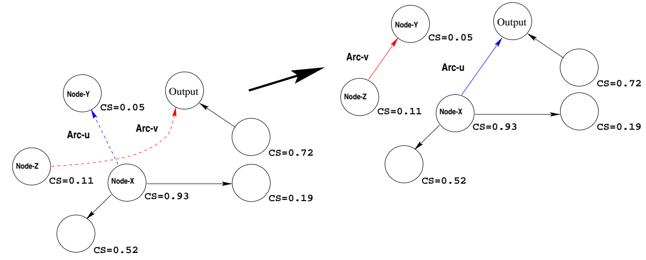

Figure 4. The *Swap Two Arcs* mutation procedure.

We keep this underlying mechanism and present an IRNP procedure that selects "good" program fragments to exchange. This means that IRNP has, as its only job, to *choose the fragments* to be exchanged, but the way in which program fragments are exchanged and reconnected is unaffected by IRNP.

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. Table 9 shows the procedure to apply IRNP to crossover.

Table 9. The IRNP process for choosing a "good" fragment of a program to exchange through crossover.

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

Neural 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. 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.

4. Experimental Results

IRNP is developed within our research in machine for signal understanding (Teller, 1998). Most briefly, our 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. Our research (e.g., (Teller, 1998)) has demonstrated that IRNP is highly effective across a wide range of signal types and sizes, both real world and manufactured. We show here a few results on two distinct real-world signal classification problems which demonstrate that IRNP can learn a difficult problem and that IR is a significantly more effective way to perform recombination on the population than is random recombination.

4.1 Natural Images

There are seven classes in the domain used in the following experiments. Figure 5 shows one randomly selected image from each of the seven classes in both the training and testing sets. Each element is a 150x124 image with 256 level of grey. This particular color images was created by other researchers for learning and vision (Thrun & Mitchell, 1994). We removed the color from the images to make the problem more challenging.

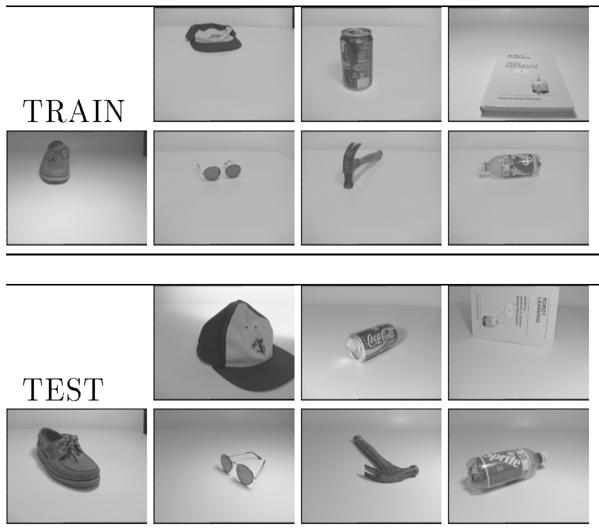

Figure 5. A random training and testing signal from each of the 7 classes in this classification problem.

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 ft and there is often severe foreshortening and even deformation of the objects in the image.

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 seven classes) images were used for training and a separate set of 350 (50 from each of seven classes) images were withheld for testing afterwards.

We use several parameterized signal primitives (PSP) functions. The simplest one, *PSP-Point*(a_0, a_1), returns the pixel intensity at the pixel (a_0, a_1). Others with four arguments (a_0, a_1, a_2, a_3), *PSP-Average, PSP-Variance, PSP-Min, PSP-Max* return the corresponding pixel intensity values over the image region specified by the rectangle with upper left corner (a_0, a_1) and lower right corner (a_2, a_3). And *PSP-Diff*(a_0, a_1, a_2, a_3) returns the absolute difference between the average pixel intensity above and below the diagonal line (a_0, a_1) to (a_2, a_3).

During each run, the generalization performance on a separate set of testing images was recorded and Figure 6 plots the *mean* of each of these values.

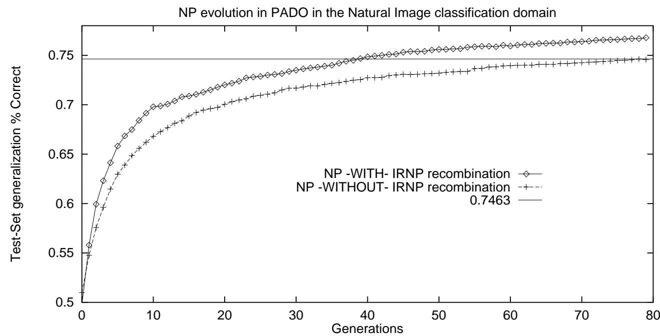

Figure 6. NP learning *with and without IRNP.*

The main result of the experiment is that NP learns more than twice as fast when IRNP is applied to the recombination during evolution. Also notice that NP significantly learns well this difficult image classification problem, as random guessing is only 14.3% correct generalization performance. It is worth also noting that with a more domain-tuned orchestration strategy, IRNP has, on this particular domain, achieved generalization performance rates as high as *86%*.

4.2 Acoustic Signals

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 examples from each class), with sounds of a Buccan-

neer jet engine, a firing machine gun, an M109 tank engine, the floor of a car factory, a car production hall, a Volvo engine, and the babble in an army mess hall.

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 seven classes) images were used for training and a separate set of 245 (35 from each of seven classes) images were withheld for testing afterwards.

We used equivalent signal primitives to the ones used with the natural images. The arguments in this case refer to time periods instead of rectangular regions of the image. For example, *PSP-Average*(a_0, a_1, a_2, a_3) returns the average wave amplitude in the sound starting at time ($a_0 * 256 + a_1$) and ending at time ($a_2 * 256 + a_3$).

The fitness used during the evolution of the neural programs was based upon the difference from the returned and correct confidences for each training example. Again, with equally represented seven classes, random guessing is a 14.3% correct generalization.

Figure 7 shows the generalization percent correct NP reaches on average on each generation, with and without IRNP. This learning is mainly three times as efficient as learning without it.

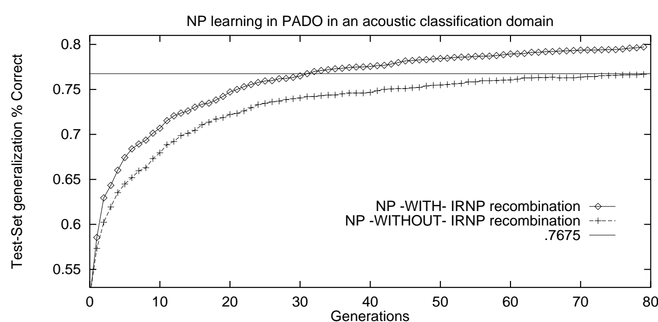

Figure 7. NP learning *with and without* IRNP.

5. Conclusion

This paper has contributed a new representation for learning complex programs and a procedure to achieve explicit credit blame in program evolution. A new program representation language, *neural programming,* is introduced with the goal of enabling a 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 domains.

The goal of this paper 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

This work was done while the first author was in the Computer Science Department at Carnegie Mellon University.

The first author was supported through the generosity of the Fannie and John Hertz Foundation. This research was 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

Altenberg, L. (1994). The evolution of evolvability in genetic programming. In Kinnear, Jr., K. E. (Ed.), *Advances In Genetic Programming*, pp. 47–74. MIT Press.

Angeline, P. J. (1993). *Evolutionary Algorithms and Emergent Intelligence.* Ph.D. thesis, Ohio State University, Computer Science Department.

Koza, J. (1992). *Genetic Programming.* MIT Press.

Teller, A. (1998). *Algorithm Evolution with Internal Reinforcement for Signal Understanding.* Ph.D. thesis, Computer Science Department, Carnegie Mellon University.

Thrun, S., & Mitchell, T. (1994). Learning one more thing. Tech. rep. CMU-CS-94-184, Computer Science Department, Carnegie Mellon Unversity.