1 PADO: A New Learning Architecture for Object Recognition

Astro Teller and *Manuela Veloso*
Carnegie Mellon University

Abstract

Most artificial intelligence systems today work on simple problems and artificial domains because they rely on the accurate sensing of the task world. Object recognition is a crucial part of the sensing challenge and machine learning stands in a position to catapult object recognition into real world domains. Given that, to date, machine learning has not delivered general object recognition, we propose a different point of attack: the learning architectures themselves. We have developed a method for directly learning and combining algorithms in a new way that imposes little burden on or bias from the humans involved. This learning architecture, PADO, and the new results it brings to the problem of natural image object recognition is the focus of this chapter.

1. Introduction

In general, AI systems use symbols to represent knowledge and to reason about tasks. Most of these systems today still work on simple problems and artificial domains. One of the main reasons for this is the common assumption that sensing is not only perfect, but also that sensors return specific symbols, not raw data. The signal-to-symbol problem is the task of converting raw sensor data into a set of symbols that the data can be seen as representing.

One of the main goals of computer vision is to provide a solution to the signal-to-symbol problem. In particular, this goal involves object recognition, i.e., the ability to recognize what (and where) objects are shown in an image. Machine learning can do induction on a set of examples to learn to discriminate among classes. These two fields, machine learning and computer vision, are natural mates and are particularly suited to cooperate on object recognition tasks.

Several proven machine learning architectures, such as neural networks, have been integrated with computer vision and the results in the recognition of "everyday" objects have been modest at best. It is possible that better parameter values, more training data, or faster computers will allow one of these architectures to make some significant advance in the field of object recognition. This chapter proposes a different view: that given the experienced difficulties with the current architectures, a more profitable path is to investigate new architectures.

Clearly, any solution to the object recognition problem needs to be grounded in an *algorithm* that processes intensity values from an image signal. Consider the particular task of differentiating between many different images of natural objects in natural settings. This task can be solved by learning a separate algorithm for discriminating image signals of each object class.

Our new architecture, PADO, is a technique for learning these algorithms directly, so that there is no built-in commitment to the manner in which the algorithm investigates the image and arrives at a decision. No features are chosen for PADO and no attention focusing strategy is built in. PADO (**P***arallel* **A***lgorithm* **D***iscovery and* **O***rchestration*) uses an evolutionary strategy to accomplish these feats of self-generation. The motivation for PADO, the details of PADO, and results on a challenging vision problem are the topic of this chapter.

A challenging vision problem in object recognition can be found in high resolution, noisy images of real world objects in natural settings. In the course of this chapter, such an image set will be introduced as an example domain in which PADO can learn and perform. PADO's impressive performance on this difficult recognition problem will then be repeated in a second vision domain with different characteristics.

This chapter will provide a solid basis for understanding this unique architecture and why it works. The experimental results will show PADO's promise as a new, practical solution to the general object recognition problem. The PADO architecture is entirely independent of the signal type to be classified, and this construction promises similar results on signal types as varied as sonar, speech, and text. This chapter is both a dissection of a tested approach and the introduction of a new way of doing signal understanding.

2. Motivation

The Signal-to-Symbol Problem is a major problem in AI, both in its own right, and because many other, more symbolic parts of AI need to be given symbols, rather than raw data.

As introduced above, PADO is designed to attack the general signal-to-symbol problem. Why pick vision as PADO's first test domain? The real reason is that vision is AI's weakest link. The two most obvious examples of agents performing the signal-to-symbol translation are in hearing and seeing. Humans and other animals seem to effortlessly take in information which is not unfairly represented as 1 or 2-dimensional waves and very quickly and without conscious effort determine some correspondence between these raw perceptions and notions about the real world. In humans, we call these notions *symbols*.

Extracting symbols from vision and hearing are both hard problems. But relative to the ultimate goal of human level performance, it is clear that AI has had more success engineering solutions to the sound problem than to the vision problem. So because it is important to AI and because it is still a mostly unsolved field, we chose vision for PADO's first problem domain.

As will be detailed in the next section, learning is relatively new to the field of computer vision and has brought with it only modest improvements in most areas. Given that learning in vision has had this modest success rate in unconstrained problems, we suggest that too much effort is being concentrated on the effort to scale up these architectures so that they work better or with less human intervention (i.e., preprocessing).

After all, Neural Networks (NN) for example, were not designed to model our visual cortex or our brains. They were inspired by very small pieces of our brain. There is no good reason to think that the large amount of processing that has to go on in order to do general object recognition can be done by a NN with a few hundred inputs or one with at most a few tens of hidden units. Even ignoring this, it is not obvious that NNs will be easy to train on a function as complex as the mapping from images to classes. Instead, we designed an architecture that directly addresses some of these issues.

This chapter is organized as follows. Section 3 discusses related work in the areas of object recognition and Genetic Programming (GP). Section 4 provides a short introduction to the learning power of evolution. Section 5 details the PADO architecture and the language choices made for our experiments. Section 6 gives the experimental background, set up, and example pictures. Section 7 shows results on a series of experiments. In Section 8 we discuss some of the most puzzling points that this chapter brings up. Section 9 looks forward to the work in progress and the near future goals we have for this work. Section 10 concludes by bringing together the highlights of the chapter.

3. Related Work

Object Recognition has been a heavily researched area for almost 30 years. Until recently however, the objects to be recognized were usually highly geometrical in shape. In the majority of cases, the recognizer already had some model, either explicit (e.g. CAD)[2] or implicit (e.g. hand-coded features)[3], of what the objects were like; the task was really to try to find objects in the picture and correctly identify their pose.

The demand for more robust systems with few constraints, coupled with the rising cost of programmer time relative to computer cycles has pushed object recognition and machine learning together. The field of computer vision is far too big to even sketch here. While we focus on vision and learning, we will only claim that to obtain similar results to those presented in this chapter, a hand-coded system, if at all feasible, would require a significant amount of programmer time.

Learning has been used for a variety of purposes with respect to object recognition. Researchers have tried to learn which hand-coded features to use on a particular problem [8]. Researchers have tried to learn models of the objects based on a number of well constrained images of these objects and then use these models as mentioned above [20].

Learning has been applied on a larger level, particularly in the form of Neural Networks. The major problem with using Neural Networks is that with today's technology, NNs cannot take full video images as input. Imagine even a small image with 256x256 resolution. A NN with just 16 hidden units fully connected to the 1/16 million inputs would have over 1 million weights to fix. Back propagation would take a **long** time to train such a net.

When preprocessing can be done to significantly reduce the images' resolution while preserving the relevant information, NNs can be effective. Of course this most often occurs when either the problem is not difficult or where the preprocessing is very clever. When the preprocessor has been made very clever, the problem has not really been solved, but simply been moved to the problem of creating a nice preprocessor.

Though it is not directly concerned with object recognition, Pomerleau's work on driving the ALVINN using a NN is an example of a very successful application of NN's to a computer vision problem [10]. Here the preprocessor was a little clever and the problem itself was less difficult than had been previous assumed. This work is important largely because Pomerleau was one of the first to apply learning to this kind of real-time, reactive vision problem.

Thrun's and Mitchell's work using NNs to do visual object recognition is a good example of work with goals similar to our own [19]. They take video images and preprocess them to get low resolution images which are then given to a NN for object recognition training. Their work focuses on studying the effect of life-long learning on the ability to find general object type invariants. This work is mentioned here not only because it is has some experimental similarities to our work, but because we have used their data for some of the experiments discussed in this chapter.

Because part of the PADO system is a type of genetic programming, there should be some mention here of the sort of vision related work that has been done within that paradigm.

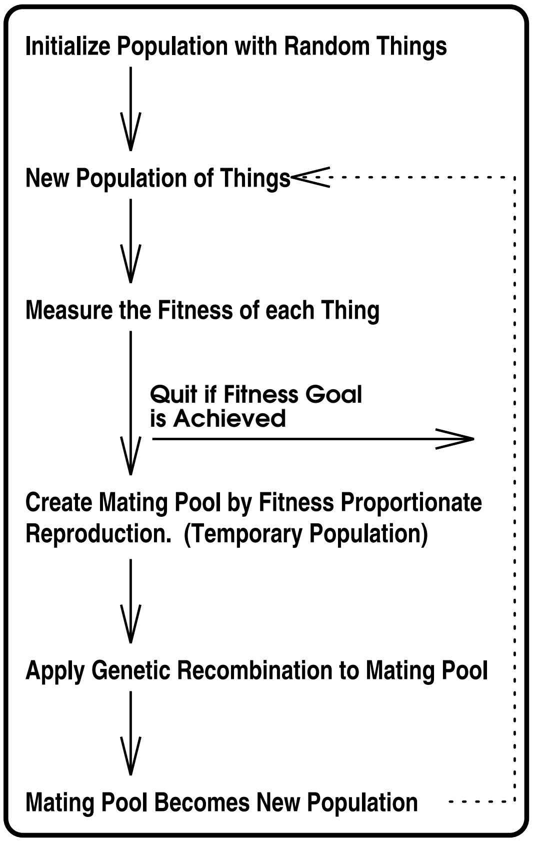
As far as we know there are no published results of the sort discussed in this chapter: that is, none that apply genetic programming or genetic algorithms directly to full video images and do object recognition on the basis of that input. The work that has been done seems to fall into two major categories: bitmap recognition and learned aids for vision problems (including object recognition). There have been some examples of genetic programming applied to bitmaps (usually font bitmaps) in order to do classification [7], [1]. In between, there are works like [4] that applied GP to a restricted subset of a black and white silhouettes of a person and tried to learn where one of the hands was. Learned aids to object recognition can be seen in works like [11] and [9]. For example, in [11], GP is used to improve the performance of an army system for locating tanks by learning to choose from among existing system components.

There is a significant amount of work that is related to PADO, but no single piece of work or combination of several covers the details of the PADO architecture. The method of orchestration and parallel execution of learned algorithms for signal classification has, until now, been unexplored.

4. Evolution for Inductive Generalization

Evolutionary computation is biologically motivated. In nature, we see that the combination of survival of the fittest, fitness proportionate reproduction, and genetic recombination is an extremely powerful tool for finding solutions to biological problems. In this section we introduce the basic nature of such genetic evolutionary processes.

Suppose that we have a large group of *things*, and some measure of how good a *thing* is. We can apply this measure to each of these *things* and get an approximate or exact fitness for each *thing*. Now suppose that we allow each *thing* to be represented in a new group of *things* in proportion to its fitness relative to the other *things* in the group. The best *things* in the old group, are likely to have greater representation in the new group and the worst *things* in the old group are likely to have no representation in the new group. When the new group is fixed to be the same size as the old group this scheme accomplishes both survival of the fittest and fitness proportionate reproduction. If nothing else changed between each successive group, the current group would soon be filled with many instances of the most fit *thing* in the group.


Initialize Population with Random Things
New Population of Things
Measure the Fitness of each Thing
Quit if Fitness Goal is Achieved
Create Mating Pool by Fitness Proportionate Reproduction. (Temporary Population)
Apply Genetic Recombination to Mating Pool
Mating Pool Becomes New Population

Suppose that before measuring the fitness of each *thing* in the new group, we change some of them in a random or semi-random way. This is the most general form of "genetic recombination." These changes introduce some chance that one of the new *things* will have higher fitness than any of the old *things*. After many new groups have come and gone we can expect that the best *thing* in the current group will be much better, according to our measure, than any of the *things* that were in the original group. This is the concept of evolution. (See Chart above).

Evolutionary computation is a form of best-first search. Exponentially increasing representation is given to those *things* that have highest fitness and so those points in the space are exponentially more likely to be examined next, relative to the other points under consideration (i.e. the other *things* in the group) [12, 13].

In the vocabulary of evolutionary computation, a group of *things* is called a **population**. To distinguish successive populations from one another they are referred to as **generations**. The initial population is traditionally called "Generation 0" and each successive generation is numbered in increasing integer order.

The exact structure of a *thing* varies from field to field in evolutionary computation. In genetic algorithms *things* are called **allele strings** and usually take the form of bit strings [5]. In genetic programming *things* are called **functions** and take the form of Lisp-like nested primitive function calls [6]. In PADO *things* are **programs**.

> Functions and Programs are **not** the same [16]. A function is a simple finite mapping from inputs to outputs. In a word, functions are *reactive.* Programs (algorithms) are procedures that incorporate current and past inputs into an iterative or recursive process that may eventually produce an output. Algorithms are *deliberative.* The basis for this distinction will not be expanded further in this chapter. However, this difference between functions and programs is one of the most important distinctions between traditional GP and PADO. See Appendix 1 for an example PADO program.

Genetic recombinations come in many different varieties. The two most common and the two which are directly relevant to this chapter are *crossover* and *mutation.* In crossover two *things* are chosen and one subpart from each is selected. Then these two subparts are exchanged and these two new *things* are placed back in the population. In mutation, one *thing* is chosen and one subpart is selected. This subpart is changed in some random way and this new *thing* is placed back in the population. In both cases the syntax of the *things* is usually constrained so that these changes always produce legal new *things.*

The most crucial aspect of evolutionary computation is that it is **not** a random search of the space of *things.* To the extent to which there is a correlation between syntactic and functional similarity, these recombinations explore *things* which are likely to be similar in their fitness. This correlation is the essence of hill-climbing. Combine this version of hill climbing with the aspects of best-first search already mentioned and we have a powerful tool for searching almost any space whose decomposable elements have some fitness variation.

5. PADO

Part of the PADO architecture falls under the general heading of evolutionary computation. This section will discuss the way PADO works and how its central component is an instance of the general scheme that was described in the previous section. During the first part of this section the inputs will be considered arbitrary signals. Later in the section and then for the rest of the chapter, the specific signal type of a still video image will be used as an example. But because the PADO architecture was designed to apply to any signal type, that is how it will be introduced.

5.1. The Architecture

The goal of the PADO architecture is to learn to take signals as input and output correct labels. **When there are $\mathcal{C}$ classes to choose from, PADO starts by learning $\mathcal{C}$ different** *systems*. System $\mathcal{I}$ is responsible for taking a signal as input and returning a confidence that class $\mathcal{I}$ is the correct label. Clearly, if all $\mathcal{C}$ systems worked perfectly, labeling each signal correctly would be as simple as picking the unique non-zero confidence value. If, for example, system $\mathcal{J}$ returned a non-zero confidence value, then the correct label would be $\mathcal{J}$. In the real world, none of the $\mathcal{C}$ systems will work perfectly. This leads us to the recurring two questions of the PADO architecture:

1. "How does PADO learn good components (systems or programs)?"
2. "How does PADO orchestrate them for maximum effect?"

We will explain how PADO orchestrates these systems in Section 7.3. Now, let's delve into how one of these systems is built.

System $\mathcal{I}$ is built out of several programs. Each of these programs does exactly what the system as a whole does: it takes a signal as input and returns a confidence value that label $\mathcal{I}$ is the correct label. The reason for this seeming redundancy will be justified and discussed in Section 10. PADO's orchestration of these programs into a single system will be discussed in Section 7.3.

PADO evolves these programs along the general lines described in the previous section. Programs learned by PADO are written in an algorithmic language that is PADO-specific. During the training phase of learning, these programs are interpreted, not compiled. So like Lisp, the programs can be compiled or interpreted, but during the "construction" phase they are simply interpreted.

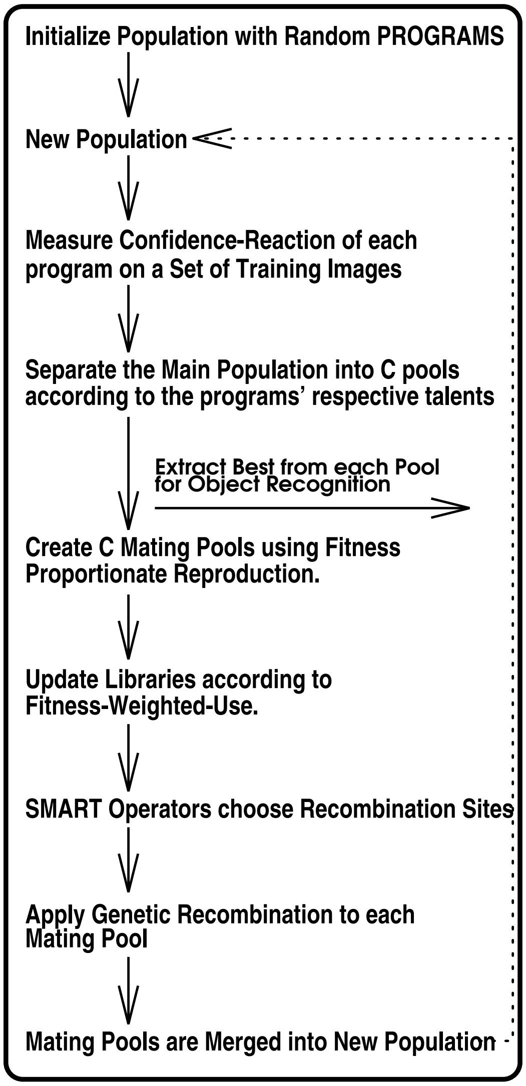
At the beginning of a learning session, the main population is filled with $\mathcal{P}$ programs that have been randomly generated using a grammar for the legal syntax of the language. All programs in this language are constrained by the syntax to return a number that is interpreted as a confidence value between some minimum confidence (MinConf) and some maximum confidence (MaxConf).

At the beginning of a new generation, each program in the population is presented with $\mathcal{T}$ training signals and the $\mathcal{T}$ confidences it returns are recorded. Then the population is divided into $\mathcal{C}$ distinct groups of size $\mathcal{P}/\mathcal{C}$. The programs in group $\mathcal{I}$ are the $\mathcal{P}/\mathcal{C}$ programs that recognized class $\mathcal{I}$ better than any other class in the sense that they maximized a reward function **Reward** when $K = \mathcal{I}$ (K is the class to which PADO is considering assigning program U).

```
int Reward(program U, class $K$, int Guess[ ])        Guess[U][j] is the
    $R = 0$;                                          confidence program U
    Loop $j = 1$ to MaxResponses                      returned for image j
        If $(K = ObjectClass[j])$ Then
            $R = R + ((\mathcal{C} - 1) * Guess[U][j])$;   R is the reward value.
        Else                                          C is the number of classes.
            $R = R - Guess[U][j]$;                    ObjectClass[j] is the object type
return $R$;                                           that appears in image j.
```

On images that the program should return MaxConf for, the reward is multiplied by $\mathcal{C} - 1$ so that, even though this only happens once in $\mathcal{C}$ times, these images will account for half the reward. This seems reasonable since it should be as important to say **YES** when appropriate as to say **NO** when appropriate since these two cases are respectively coverage and accuracy. This normalization provides, on average, zero reward for a purely random classification strategy.

Each group is then sorted by increasing fitness and each program is ranked accordingly. C "mating pools" (temporary groups) are created by putting a copy of $Program_{\mathcal{J}}$ from Group$_I$ into MatingPool$_I$ with probability $2 * rank(\mathcal{J})/(P/C)$. The expected number of copies of the best program in Group$_I$ is 2, the expected number of copies of the median program is 1, and the expected number of copies of the worst program in Group$_I$ is $2/(P/C)$. This method is called rank proportionate reproduction.


Initialize Population with Random PROGRAMS
New Population
Measure Confidence-Reaction of each program on a Set of Training Images
Separate the Main Population into C pools according to the programs' respective talents
Extract Best from each Pool for Object Recognition
Create C Mating Pools using Fitness Proportionate Reproduction.
Update Libraries according to Fitness-Weighted-Use.
SMART Operators choose Recombination Sites
Apply Genetic Recombination to each Mating Pool
Mating Pools are Merged into New Population

The **Libraries** are programs reference-able from all programs. After the division of the population, the libraries are updated according to how widely and how often they were used. These statistics are weighted by the fitnesses of the programs that called them.

Finally, 85 percent of the programs within each mating pool are subjected to crossover and another 5 percent are subjected to mutation. All crossovers take place between two programs in the **same** mating pool. That means they are both recognizers of the same class. Crossover in PADO is more complicated than its standard form in genetic algorithms or genetic programming. In PADO two programs are chosen and given to a "SMART crossover" algorithm.

This algorithm examines the two programs and chooses two sub-graphs in each. Then one subpart from each programs is exchanged. The new pairs of sub-graphs are reconnected, creating two new programs. These two new programs replace the two old programs in the population. The "SMART Mutation" in PADO is also more complicated than the general case described in the previous section. One program is chosen and an "intelligently" chosen subpart is replaced with an "intelligently" generated new element. This changed program then replaces the old program in the new total population.

At this point we merge the C mating pools back into a new total population. Now the process of evaluation, reproduction, and recombination repeats. After many generations we find that the best programs in the population are much better than any that were created (randomly) at the start of the process.

To extract programs to use in the systems, we can pause the process after the evaluation step of a generation and copy out those programs that scored best or near best in each group $\mathcal{I}$. So this architecture is an *anytime* learning system: at any time we can generate a system for signal classification using what we have learned so far.

5.2. The Language of a PADO Program

Figure 1 sketches the structure of a PADO program. Each program is constructed as an arbitrary directed graph of nodes. As an arbitrary directed graph of N nodes, each node can have as many as N *arcs* outgoing.[1] These arcs indicate possible flows of control in the program. In a PADO program each node has two parts: an *action* and a *branch-decision.* Each program has a private stack and an indexed memory. All *actions* pop their inputs from this stack and push their result back onto the stack. These actions are the equivalent of GP's terminals and non-terminals.

The Indexed Memory is an array of integers indexed by the integers. As will be seen below, each program has the ability to access any element of its memory, either to read from it or to write to it [14]. This memory scheme, in conjunction with the main loop described above has been shown to be Turing complete [16]. In practice this memory has a finite range of integers over which it is indexed, and each element can hold integers in a finite range. However, indexed memory can been seen as the simplest memory structure that can practically support all other memory structures. Indeed, indexed memory has been successfully used to build up complex mental models of local geography [14].

After the action at node i is executed, an arc will be taken to a new node. The branch-decision function at the current node will make this decision. Each

[1] For this chapter, two outgoing arcs per node will be used for simplicity.

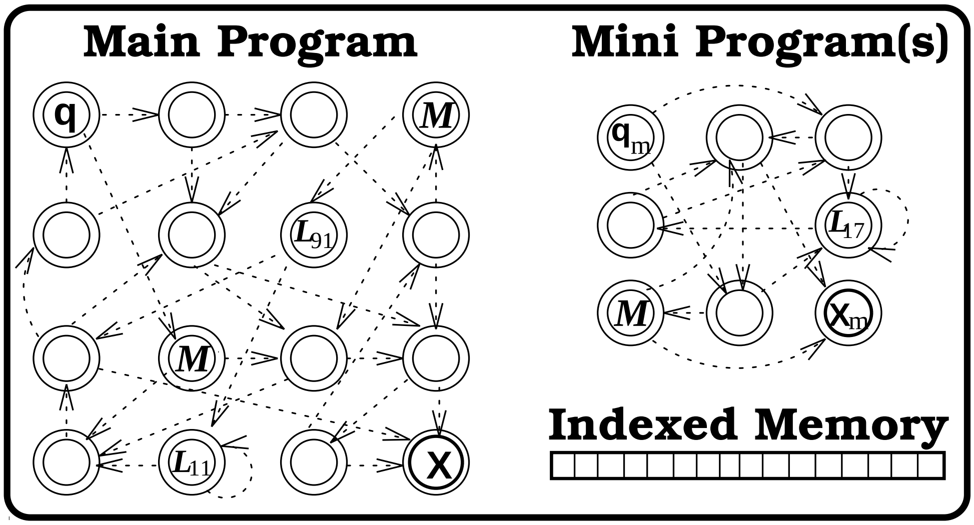


Figure 1: This is the basic structure of a PADO program. There can be one or more Mini programs for each PADO program. Each Mini program may be referenced from the Main program, another local Mini program, or a *Library* program.

node has its own branch-decision function that may use the top of the stack, the previous state number, the memory, and constants to pick an arc.

There are several special nodes shown in Figure 1. Node q is the start node. It is special in no other way than it is always the first node to be executed when a program begins. Node X is the stop node. When this node is reached, its action is executed and then the program halts. When a program halts, its response is considered to be the current value residing in some particular memory location (e.g. response = Memory[0]). If a program halts sooner than a pre-set time threshold, it is started again at its start node (without erasing its memory or stack) to give it a chance to revise its confidence value. A weighted average of the responses the program gives on a particular execution is computed and interpreted as the answer. The weight of a response at time $t_\mathbf{i}$ is **i**. Later responses count more towards the total response of the program. Because of the time threshold, PADO's programs are guaranteed to *halt* and respond in a fixed amount of time.

Node M executes the private *Mini* program as its action. It then executes its branch-decision function as normal. The *Mini* program associated with each *Main* program bears similarity to the concept of ADF's (automatically defined functions) [7]. It may be called at any point in the *Main* program and it evolves along with the *Main* program. *Mini* programs are in every way normal PADO programs; their size is not constrained to be smaller than the *Main* programs. The name *Mini* denotes only that it is owned by the *Main* program. The *Mini* programs may recursively call themselves or the globally available *Library* programs, just like a *Main* program may.

The *Library* programs (e.g., L_{91} in Figure 1) are globally available programs that can be executed at any time and from anywhere just like the Mini programs.

But unlike the Mini programs, where each Mini may be run only during the execution of the PADO program of which it is a part, the *Library* programs are publicly available to the entire population. The method by which these *Library* programs change will be shown in some detail in the discussion section.

Here is a brief summary of the language *actions* and their effects:[2]

Algebraic Primitives: {ADD SUB MULT DIV NOT MAX MIN}
These functions allow basic manipulation of the integers. All values are constrained to be in the range 0 to 255. For example, DIV(X,0) results in 255 and NOT(X) maps the set {1..255} to 0 and {0} to 1.

Memory Primitives: {READ WRITE}
These two functions access the memory of the program. Each program has a memory which is organized as an array of 256 integers that can take on values between 0 and 255. READ(X) returns the integer stored in position X of the memory array. WRITE(X,Y) takes the value X and writes it into position Y of the indexed memory. WRITE returns the OLD value of position Y (i.e. a WRITE is a READ with a side-effect). The memory is cleared (all positions set to zero) at the beginning of a program execution.

Branching Primitives: {IF-THEN-ELSE EITHER}
Calling these "Branching" primitives may be misleading. In both cases the primitive pops X,Y, and Z off the stack and then replaces either Y or Z (not both) depending on the value of X. For IF-THEN-ELSE the test is (X less than 0). For EITHER the test is (X less than RandomNumber) where RandomNumber varies between 0 and 255. These primitives can be used as an action or a branch-decision functions. In the former case, they have no effect on the flow of control.

Signal Primitives: {PIXEL LEAST MOST AVERAGE VARIANCE DIFFERENCE}
These are the language functions that can access the signals. In order to demonstrate PADO's power and flexibility, these same primitives were used for both image and sound data [18]! PIXEL returns the intensity value at that point in the image (or sound). The other five "signal functions" each pop the top four values off the stack. These four numbers are interpreted as (X1,Y1) and (X2,Y2) specifying a rectangle in the image. If the points specified a negative area then the opposite interpretation was taken and the function was applied to the positive area rectangle. LEAST, MOST, AVERAGE, VARIANCE, and DIFFERENCE return the respective functions applied to that region in the image. DIFFERENCE is the difference between the average values along the first and second half of the line (X1,Y1,X2,Y2).

Routine Primitives: {MINI LIBRARY[i]}
These are programs that can be called from the Main program. In addition, they may be called from each other. Because they are programs, and not simple functions, the effect they will have on the stack and memory before completion (if they ever stop) is unknown.

[2] A portion of the discussion section will be devoted to a justification of this language design and its ramifications

- **MINI :** Each MINI has 4 extra legal terminals: X, Y, U, and V. These extra terminals are local variables that take on the values of the four top values popped off the stack when the MINI is called. This MINI program is private to the MAIN program that uses it and can be called as many times as desired from this MAIN program. For this paper, each program was given exactly one MINI which evolves along with its MAIN. In general, a MAIN program could have many MINI programs for its private use. The only distinctions between MAIN and MINI programs, other than the mechanism of referring to (i.e., calling) a MINI, is that MINI programs may become Library programs. MAIN programs currently may not.
- **(LIBRARY[i] X Y U V) :** There are 150 library programs. The **i** is not really a parameter. Instead an *Action* calling a Library program from some program's MAIN, MINI, or from another Library program might look like **Library57**. Like the MINI programs, the Library programs pop four values off the stack and stores these values as four local read-only variables that are legal terminals for Library programs. All 150 Library programs are available to all programs in the population. How these library programs are created and changed will be discussed in Section 8..

6. The Experiments

The discussion of results will focus on two sets of data. One set was taken by us. The other set was taken by Sebastian Thrun who also works in learning and vision [19]. For reference purposes, we will call the first set A and the second set T. Elements in both sets are 256x256 video images with 256 level of grey.

Set T has seven classes: Book, Bottle, Cap, Coke Can, Glasses, Hammer, and Shoe. The lighting, position and rotation of the objects varies widely. The floor and 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 of the objects in the image. See columns one and two of Figure 2 for sample images.

Set A has seven classes: Other, Bear, Long flute, Pan flute, Thermos, Book, and Racket. The class Other is a collection of images which are empty or show a hand holding some object (like a cup or a ball) that is not one of the other six classes. All pictures are taken against a variety of solid colored backgrounds and contain part of a hand and arm. The hand holds one of the objects, often partially occluding it. The location and rotation of the object is only constrained so that the object is completely in the image. The lighting varies dramatically in intensity and position. The distance from the object to the camera ranges from 2.5 to 3.5 feet. The objects are never severely foreshortened. See columns three and four of Figure 2 for sample images.

Set A was created with several criteria in mind. We wanted a set of images that could be easily distinguished by people, but were not trivially different in some way that the computer could "notice". We wanted some noise in the images but did not want to have unconstrained backgrounds. We wanted a sufficient number of classes so that the results could give some indication of PADO's practical value. However, it was important that the number of classes be small enough that the learning and science was not lost in the engineering.

As a result, we chose grey scale images of the seven classes listed above for set A. The backgrounds varied in color but were always solid. The noise was always there in the form of a hand and arm. In general the hand and arm take up about half as much area of the image as the object itself so that the signal to noise ratio is between 1.0 and 2.0. And we chose to test the system on seven classes. Initial tests proved PADO's effectiveness in a three class object recognition problem, so slightly more than doubling the number of classes seemed like the next qualitative step up in difficulty.

Set T was created by Sebastian Thrun for his own work. The pictures he took were originally in color, but allowing PADO access to the color images turned out to be too easy (PADO classification accuracy of 95%), so we removed the color and saturation information and kept only the brightness information. We thought it was important, perhaps even more important, to include data on images taken by someone else for a different purpose as this is the real goal of the PADO project. That is, the ability to distinguish between classes of signals that were in no way designed, taken, or preprocessed with PADO in mind.

Because these images all have at most one object, it could be argued that the results shown in the following sections are not object recognition but rather classification of images based on the object shown in the image. This is a murky distinction. What makes object recognition different from image classification (since in theory an image that contains both a shoe and hammer would be correctly classified both as a SHOE image and as a HAMMER image)? The only reasonable distinction might be that an object recognizer must do more than just know of the object's existence in the image. It must be able to locate it in the image, perhaps even segment it from the rest of the image. In short, the machine learning community would probably label this work as "object recognition" while the vision community would probably label this work as "image classification." Because the distinction is loose we have chosen to label these results as object recognition. More will be said in the discussion section about finding the object in the image.

The following sections show three different experiments. In each of the three experiments the general methodology was the same. The training set consisted of 14 images from each class for a total of 98 images. The testing set consisted of 14 **different** images from each class for a total of 98 images. In order to prevent overfitting, training images were subjected to a variety of transformations (e.g., mirror image, contrast up, etc.) to add "content preserving" noise. The environment was initialized with 2800 random programs, 150 random library programs, and 300 random SMART operators. The environment was then run for 80 generations, examining the fitness of each program relative to the 98 training images. For object recognition, the best seven programs, as determined by their performance

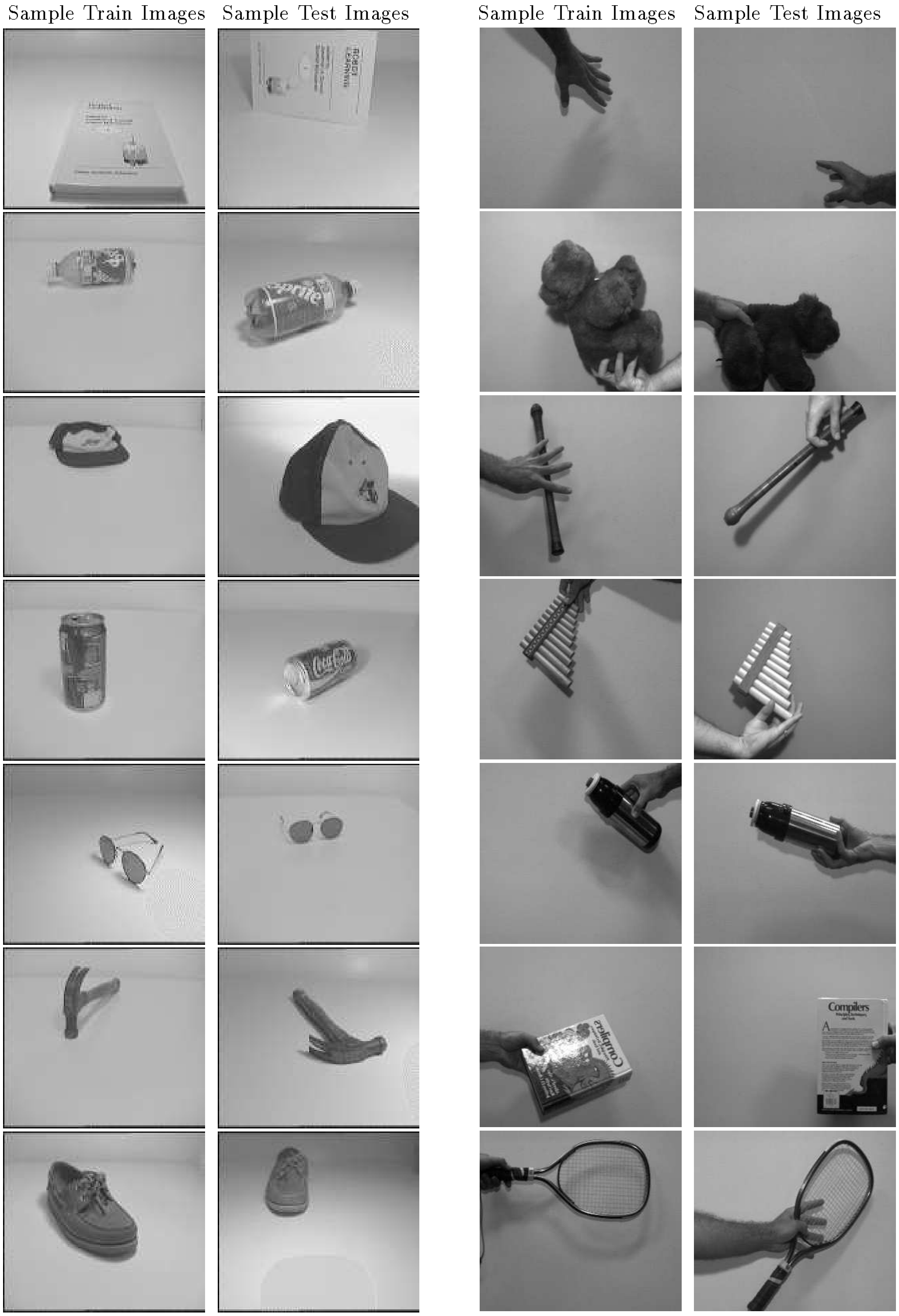


Figure 2: 28 randomly selected images from Sets T and A.

on the training set, were extracted and orchestrated for testing as a complete PADO recognizer. To obtain reliable results, each experiment was done five times.

7. Experimental Results and Analysis

Through this section we show how PADO actually accomplishes difficult tasks in visual object recognition. The results shown here are not the only hurdles that PADO has cleared. In fact, these results are not PADO's most flattering. But they give an accurate, easy to understand picture of the sort of performance that PADO can currently deliver on real tasks. The section is organized in three parts. Each subsection will explain an experiment, detail the results, and then give some basic analysis and implications.

7.1. The Standard PADO Technique

At the end of each generation, each program is placed in the group K that maximizes its reward. See page Section 5.1 for reward function.

7.1.1. Results

Figures 3 and 4 show the average reward of the top $\mathcal{S}$ programs in each class for each generation between 1 and 80 based on 98 test images. In other words, the $\mathcal{S}$ programs from class $\mathcal{I}$ that had the highest reward for the training images are selected. Then each of these $\mathcal{S}$ programs at each generation is reevaluated on 98 test images and is given a reward based on how it performed on these 98 test images. Then these $\mathcal{S}$ rewards are averaged to give the score for class $\mathcal{I}$ that is plotted on the graph. $\mathcal{S}$ is a PADO parameter which for this chapter was set to be seven.

This data was taken on five separate runs and the graph is the average over these five runs. The range of possible rewards is from -100% to 100%. Random guessing would result (on average) in a reward of 0. So a reward of positive 50% (0.50) is actually 75% of the distance from the lowest possible reward to the highest possible reward.

Learning (i.e., improvement) continues after generation 80, but the learning rate continues to diminish. In order to do the number of experiments necessary for reliable data, most of the experimental runs were not allowed to continue past generation 80.

7.1.2. Analysis

Figures 3 and 4 show the increasing ability of the best programs in the population to distinguish *their* class from the others. The most obvious "feature" of the graphs is that some of the curves in both graphs never get much above 40% positive reward. Does this mean that even at generation 80 the most fit programs are still having a hard time distinguishing their class from the others? Basically, the answer is no.

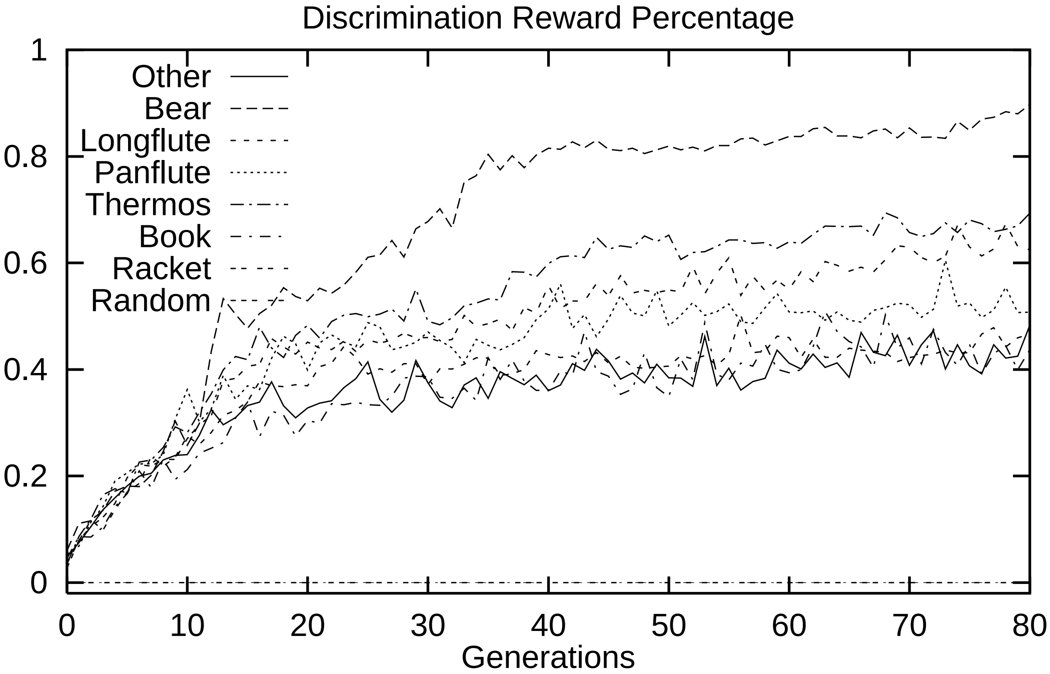


Figure 3: PADO discrimination reward percentage on **test** images of Set A.

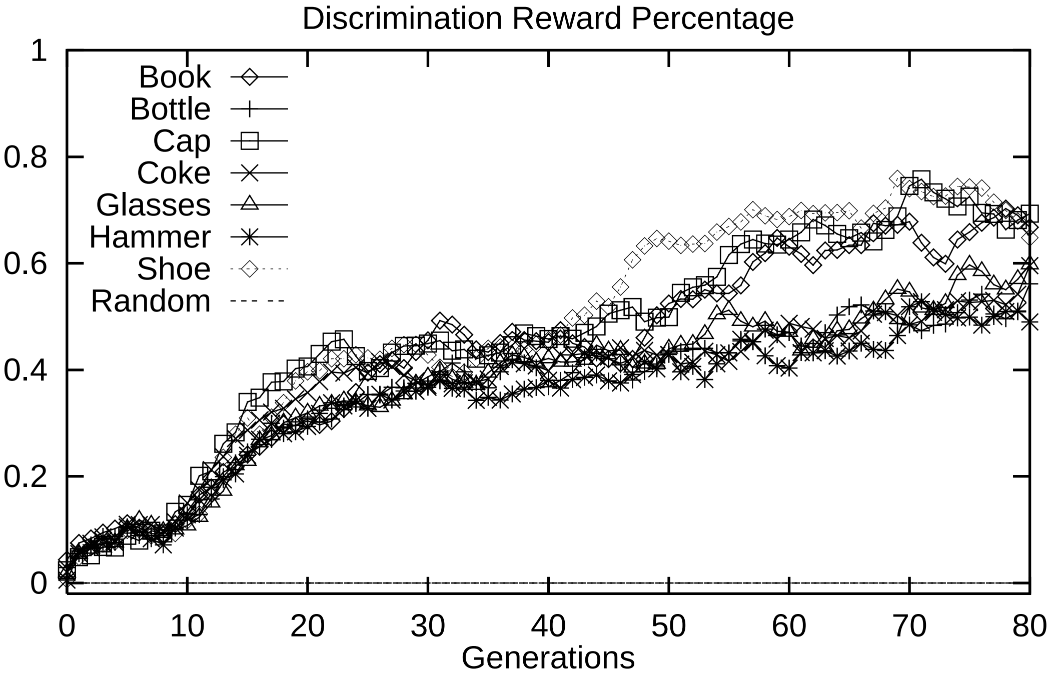


Figure 4: PADO discrimination reward percentage on **test** images of Set T.

Remember first that random choice would average a reward of 0. Because the numbers that these programs return are confidences, it is possible to be “right” about a picture without getting the maximum possible reward. For example, if a program which is later designated as a *Thermos* program returns a confidence of 0 upon seeing an image in which there is no thermos, its reward will be maximal (100%). If the program returns a confidence of 255 then its reward is minimal (-100%). If, however, the program returns a confidence of 10, for example, then its reward will be 92% which is close to the maximum reward it could obtain on this picture. So the fact that these curves do not climb to near 100% reward has two reasons. The first is that the problem is very hard and, though the training set size is small, noise was added to prevent over-fitting. This noise makes it nearly impossible for any program to perfectly fit the data. The second reason is more interesting. There are some pictures which are more clearly from class $\mathcal{I}$ than others. So it makes sense that many of the programs trained to discriminate images of objects from class $\mathcal{I}$ from other images should learn to express real levels of confidence based on how likely they think it is that the picture is what they believe it to be. For example, from some angles the thermos may look like the bear. In such cases, it would be appropriate for thermos-recognizing programs to return non-maximal confidences.

A second important facet of the graphs in Figures 3 and 4 is that there is some variation between the curves in each graph. This is not surprising, as some classes are easier to distinguish than others. For example, the Other class in Set A is very hard to learn because there is no set of features that identifies that set. So learning to distinguish that class from the others requires finding features from many of the other classes. This helps to explain why the Other curve is one of the lowest in Figure 3. Also, the relatively slowly rate of performance increase per generation is a good indication that the difficulty level of these image classes is non-trivial.

7.2. Incremental PADO

This subsection details a similar set of experiments. In these experiments only two classes were trained in the early generations. The other classes were added incrementally in periods of five generations, starting at generation fifteen.

7.2.1. Results

Figures 5 and 6 show the results obtained in reward average. Notice that these newly added class discriminators rise very quickly to the point that they would have been had they been trained from the beginning.

At generation 40, the incremental learning technique has used about half the computation time that the standard PADO learning took for all seven classes. At generation 80, the incremental learning technique has used about 75% of the computer cycles that the standard method uses. Also notice that on average the incrementally learned classes do slightly better than the standard technique.

We tried different training class orderings and obtained similar results. This

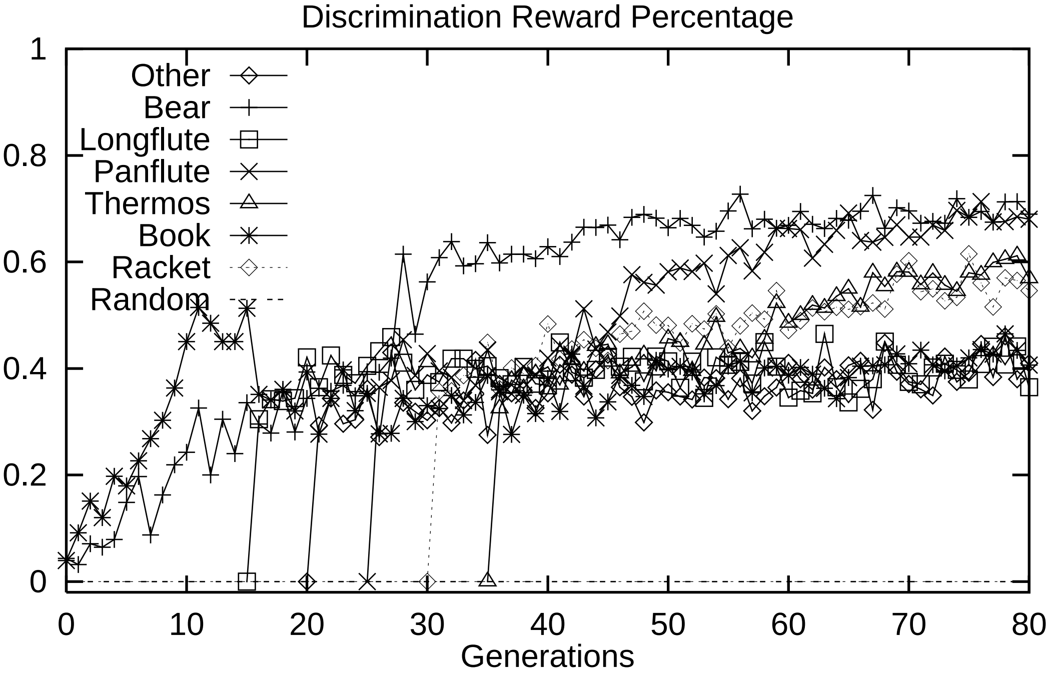


Figure 5: Incremental PADO reward percentage on **test** images of Set A.

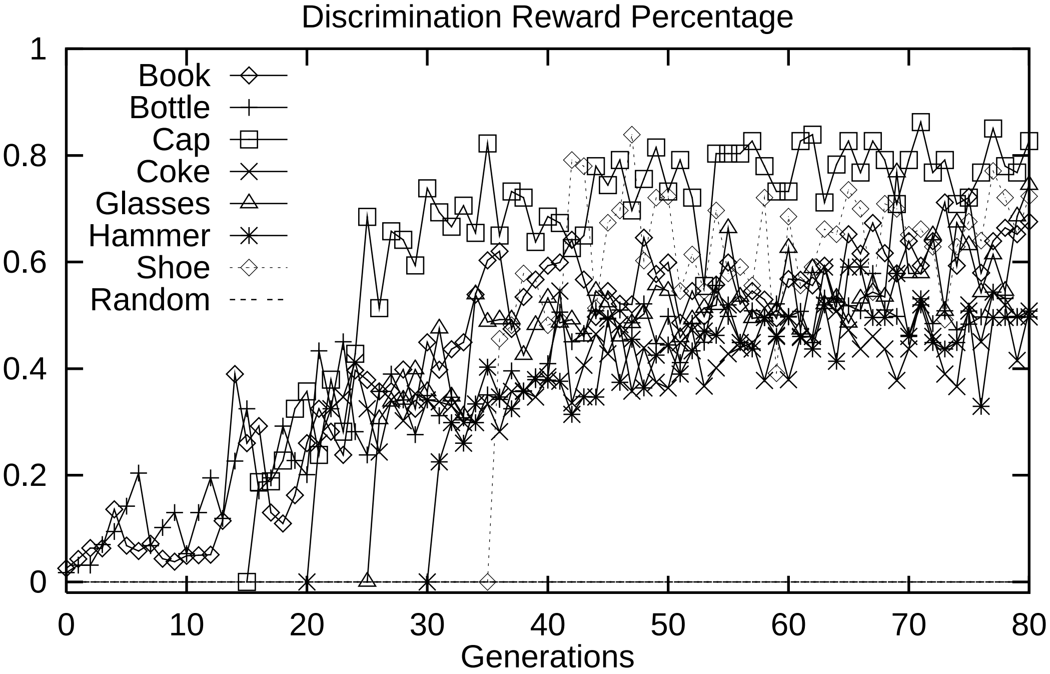


Figure 6: Incremental PADO reward percentage on **test** images of Set T.

suggests that there is nothing special about the order in which the incremental classes are introduced or the period between introductions. The only exception is that, if the rate of introduction of new classes is faster than one every four generations, the learning curve is not quite as steep. This is probably because there is a period of a few generations during which two classes are both trying to get up to speed. Waiting longer between introductions produces a very similar rise in performance. In short, any graph showing an incremental learning process must use a particular introduction rate and a particular sequence for class introduction. These graphs are, however, representative of graphs with various orderings on the classes and various speeds of introduction.

7.2.2. Analysis

The incremental learning graphs in Figures 5 and 6 show similar properties to those discussed in Figures 3 and 4 and the arguments continue to be valid. The difference in these graphs is that two of the classes are trained from generation 0. Then starting at generation 15, a new class and new training images are added at every fifth generation. Unlike the graphs from the standard PADO technique, these graphs do show dramatic jumps in performance over a small number of generations. These jumps are not constant, but increase in steepness as the number of its introduction increases. So it seems that a class added later learns up to the level of its fellow classes "faster" than previously added classes did.

So why is it that these curves of the added classes seem to jump up so quickly to meet the rest of the curves and then continue on as though it had been trained from the beginning? It would be almost impossible to give a definitive answer to this question. Instead, consider the following as a plausible explanation. At some generation $\mathcal{G}$ we choose to add in another class. This means that the number of groups the population will be divided into at the end of generation $\mathcal{G}$ will increase by one and the training set size will increase so that there is an equal number of each image class, including now the new image class. This new group will be formed from the individuals in the population that performed best as discriminators of this new class. Even at the end of generation $\mathcal{G}$ we can expect some non-negligible performance. This new object must be "most like" one of the other objects. So one of the programs from this old class that is most similar will serve better as a discriminator than one we would pick at random. For a few generations afterwards this effect is still important, but does not seem sufficient to explain these sizable jumps in reward. The second half of this explanation lies in the mechanism of the libraries. Rather than delve into them here we will finish this discussion in the natural course of the discussion below about the libraries. What the results of the incremental experiments tell us is that we can quickly get a new class up to comparable performance with the current classes. This may mean that larger numbers of classes can still be learned tractably or even cheaply by leveraging off existing knowledge through this incremental learning technique.

7.3. Object Recognition with PADO

As was outlined in Section 5, object recognition for $\mathcal{C}$ classes is accomplished in PADO by the orchestration of $\mathcal{C}$ different systems. Each of these systems is composed of the $\mathcal{S}$ most fit programs from the corresponding group of the current generation. In order to show object recognition from generation 0 it is necessary to learn all $\mathcal{C}$ program classes from the beginning. The incremental technique actually seems (on average) to produce slightly better programs, but until generation 36 respectively some classes have no programs and so the object recognition could only proceed on a set of test images containing objects from the classes already learned. This would make the graphs so hard to decipher that instead we simply picked programs from the standard PADO strategy.

System$_{\mathcal{I}}$ is built from the $\mathcal{S}$ programs that best[3] learned to recognize an object from class $\mathcal{I}$. The $\mathcal{S}$ responses that the $\mathcal{S}$ programs return on seeing a particular image are all weighted, and their weighted average of responses is interpreted as the confidence that System$_{\mathcal{I}}$ has that the image in question contains an object from class $\mathcal{I}$. PADO does object recognition by orchestrating the responses of the $\mathcal{C}$ systems. The confidence response of each system is initially weighted equally. Then, on a particular test case, the function F (e.g., MAX) takes the weighted confidences from each System$_{\mathcal{I}}$ and selects one class as the image object class. Figure 7 pictures this orchestration learning process.

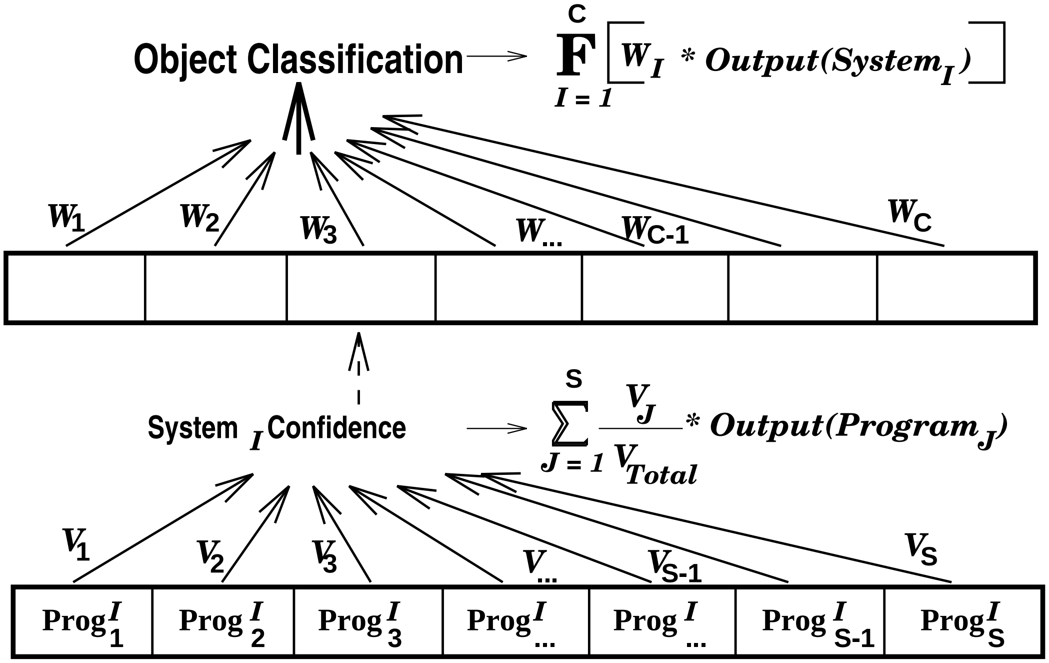


Figure 7: The weights **W** and **V** that are trained early in testing.

[3] Based on the training results from that generation.

7.3.1. Results

In these experiments (Figure 8) we followed an orchestration method where there was some "learning" before the test phase. During this orchestration phase a few "orchestration" images (7 from each class) were shown to the constructed PADO recognizer. The orchestration weights are adjusted by telling PADO after its guess whether it was right or wrong. Specifically, each program $\mathcal{J}$ in a particular System$_{\mathcal{I}}$ has its weight $V_{\mathcal{J}}$ adjusted after each of these initial tests so that its weight increases when it returns a confidence near the correct confidence and decreases if its returned confidence is far from the correct confidence. Similarly, System$_{\mathcal{I}}$ has its weight $W_{\mathcal{I}}$ adjusted in the same manner. This strategy (shown in Figure 7) is only one of many ways that the orchestration could be accomplished. Several other orchestration strategies were also tried with similar success. This orchestration strategy was chosen to obtain these results because it works well and is simple to explain. This extra learning (orchestration) adds only a few milliseconds to the total testing time.

After this pre-testing phase (orchestration phase), the actual testing phase was done. During this phase, 98 images (14 from each group) were used to test PADO's performance. All 98 of these test images were images that had PADO had never seen before, either during learning or during orchestration. The data for Figure 8 was averaged over five runs. On each run, on each generation, a PADO program was compiled from $\mathcal{C}$ systems that were each built from the best programs available during that generation. This PADO program's ability to correctly recognize which object was in each of the 98 test images was then tested.

7.3.2. Analysis

Figure 8 shows the ability of PADO to do object recognition on two sets of image data. The crux of this chapter is "Does PADO succeed at the task of object recognition? Is PADO worth all this trouble?" The most important thing to point out is that if we constructed a simple system that simply guessed at the class of the image by choosing a class at random, it would be right about 14% of the time (shown as a dotted line in Figure 8). At generation 80 the percent of the time that PADO correctly identifies the image class is about 4 to 5 times random performance. On images as unconstrained as these images are, of objects as unfriendly as these objects are, this is a real difference. Issues of scalability and potential application for PADO will be discussed in the next sections.

There are two other items of note about these two object recognition graphs. The first is that they are both graphs of the single class chosen by the orchestration. Though it is not shown on the graphs, data was also taken about the percent of the time that the correct class was ranked second in the orchestration's ranking of the $\mathcal{C}$ confidences. For both data sets this percentage was in the middle 20's. This means that between 80% and 90% of the time the correct class was in the top two in the orchestration's ranking. This is interesting as a symptom of how PADO fails when it does fail, and highlights how, as the number of classes

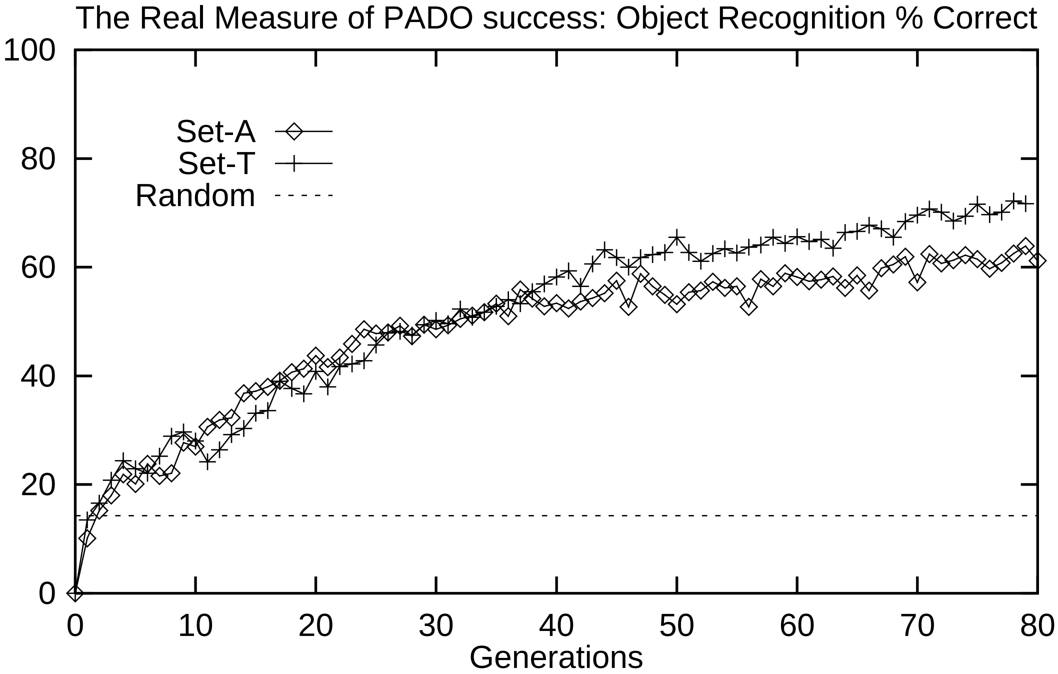


Figure 8: PADO object recognition percentage correct on Sets A and T **test** images.

to choose from increases, the easier it is to get one "bad" vote that disrupts the orchestration for that image.

The second item is the relative heights of the graphs in Figure 8. On average the image set T (without the hand in it) is easier to generalize to than is set A (with the hand in it). The two data sets are difficult for different reasons. Set A includes full rotation and translation and includes the hand and arm that possibly occlude part of the object in question. However, all the objects in set A are more or less parallel to the image plane. For Set T this is not the case. The hand is absent but the objects are rotated and translated **and** foreshortened. Because the hand and arm introduce a significant amount of noise, these results are not surprising. The whole issue of assessing the "difficulty" level in dividing some set of signals into classes is a very open question.

8. Discussion

The previous section gave enough information for the reader to read each graph and extract the data that those graphs represent. Along with each set of results, parts of the previous section were dedicated to the interpretation of that data and its ramifications. To really understand these results and their ramifications, however, a more complete explanation of the process is required. This section, in a question and answer format, will try to explore some of the puzzling issues of PADO and the task of object recognition.

8.1. The language, The Architecture, and The Procedure

Why not use more "helpful" image-related language primitives?

In the language described in this chapter, there are six ways to get basic intensity information from the input image(see Section 5.2). The PADO mechanism is independent of the details of the language used. For example, DIV(X,0) (i.e. the action DIV with X and 0 being the top two elements of the stack) could be redefined to be 0 (instead of 255) and if we re-did all these tests we would get very similar results. There are certainly other language primitives that could have been put into the language which might have improved the results shown here. An obvious example is (SPATIAL-DIFFERENTIAL X Y U V) which would return the spatial differential along the line defined by points X,Y and U,V. A more extreme example would be to make the results of an edge segmenter and edge joiner available through some language primitives. There are two reasons why we did not do this. The first is that in the spirit of trying a non traditional approach to computer vision, it seemed worthwhile to see what level of success we could achieve without borrowing any notions or structures from traditional computer vision. This was originally motivated by the lack of success that computer vision has had with natural objects in natural scenes. In fact, because we believe these results to be non-trivial, that reason seems to have been justified. The second reason is that we are trying to create a learning architecture that requires minimal input or help from the user. The more time a programmer must spend to customize a language in order to get good results, the less "autonomous" the system is. We have shown in this chapter that with the bare minimum of input from the user (i.e. these extremely basic functions for getting image intensity) good results are still possible. And, of course, if still better results are sought, this system can be improved, among other ways, by trying other language primitives.

Why use Indexed Memory as the memory structure?

Indexed Memory has shown itself to be a highly successful memory structure in the field of Genetic Programming [14]. The successful programs whose performance is plotted in Figures 3 through 8 typically use from 5 to 30 of their memory spots very heavily and ignore or largely ignore the rest of the 256 memory elements. The size of the memory was chosen to be 256 elements simply because then every legal value would be a legal pointer into memory and because every memory element could be accessed through a pointer from a legal value. It would have been possible to use other memory sizes, but the results shown above were not found to be sensitive to changes as long as there were at least approximately 50 memory slots. It is possible that a different memory size would be needed for a different problem, but 256 elements of 8 bits each provide for approximately $3.2 * 10^{616}$ different states already.

What part do the Libraries play in PADO's functionality?

Initially all 150 Library programs are initialized to be random legal program with the same characteristics as MINIs. At the end of every generation, the K worst Library programs are removed from the Library and replaced with the MINIs of the K most successful programs of the generation. For this chapter, K was set to seven. The “goodness” of a Library program is the sum of the adjusted fitnesses of all programs that called it, multiplied by how often they called it (with a ceiling of once per fitness case). The adjusted fitness of each program is $Rank[Program(\mathcal{I})] - (MaxRank - MinRank)/2$. Notice that, unlike the MAIN and MINI programs, the individual Library programs do not evolve. Rather they are a storage place for some of the best “ideas” in the population and bad ideas are moved out in favor of other ideas that have a good chance of being good. In this sense, the Library population *evolves* (in fact co-evolves with the main population and the SMART operator population) even though the individual library programs receive neither fitness proportionate reproduction or genetic recombination. How bad library programs are determined, how many are removed, and how new ones are found or created, is still an active area of our research.

What do the SMART Operators do that traditional crossover can not?

When evolving functions or programs (See Section 4 for distinction), there are several issues related to evolvability. The three most important are: language representation, genetic change paradigm, and genetic change operator(s). In GP, the language representation is usually a Lisp-like structure. Because PADO has a radically different language representation, the genetic change paradigm and genetic change operators of GP are no longer appropriate. In GP the main genetic change paradigm is crossover. In standard crossover, two nodes are picked, one from each expression-tree. Then these nodes (and the subtrees under them) are exchanged. PADO can not even maintain this general paradigm, because programs under evolution are arbitrary graphs, not trees or DAGs. Even if we knew some equivalent crossover strategy for general graphs, picking two subgraphs to switch at random would not work. The space of algorithms is much more difficult to negotiate than is the space of functions [15]. In PADO we have developed SMART operators to help us choose which subgraphs to exchange and even how to exchange them. A SMART operator is a program that takes two program graphs as input (e.g. two MAIN or two MINI) and, after some deliberation, indicates two subgraphs, one from each program, that are to be exchanged. These SMART operators are themselves written in the PADO language! Our SMART operators evolve in a separate pool, but at the same time as the main population. This means that how they act changes with the changing needs of the main population. This new approach to genetic operators has been very successful and to it we attribute much of PADO's success. The details of the SMART operators might fill a chapter by itself and are the subject of another publication [17].

Why does PADO evolve programs instead of functions?

As was just mentioned, evolution becomes more difficult (or at least requires smarter recombination) when programs replace functions as the type of *thing* (see Section 4) in the population. If there was no benefit to using programs over using functions, we and PADO could avoid a lot of work. However, it turns out that the results shown in this chapter could not be obtained in a similar fashion when functions were used instead of programs. In order to achieve 80% of this chapter's results using functions, PADO requires 10 times that space (memory) and almost 5 times as many hours of computation. And in the experiments we did, the results never climbed to the near 75% object recognition rate shown in Section 7.3.

How does PADO ensure that every program stops after a fixed time?

As was mentioned in Section 5, the PADO language is Turing complete. If PADO waited until every program was "finished" where finished was defined by some returned value or state of its memory, it is likely that PADO would never get past generation 0; many of the programs generated randomly would run on forever. PADO avoids this problem by constructing each program as an "anytime algorithm." This means that the program can run for as long as it wants, but at any time PADO may interpret what it has done so far as its "answer" (e.g. answer = Memory[0]). Most programs execute their MAIN program (i.e. reach their *stop* node and are restarted) between 5 and 50 times during the first 30 milliseconds of program execution. The PADO environment stops each program after 30 milliseconds and interprets this series of values that the program has returned, via a memory location, (one for each MAIN program *stop*) as its answer. Since the program must return a single value, this series of values is averaged. And because it is likely that the values near the end of its execution are more "informed" about the correct answer (having had more time to "think") the series of values is linearly weighted so that the value returned at time T is weighted by T. This is just one way around the problem of waiting for all programs to halt. There are other ways to constrain the construction of programs so that they all halt in a bounded amount of time. Few of these techniques were investigated. We finish by remarking only that PADO's architecture only requires that it get a fitness for each program in a bounded amount of time. The pre-set time threshold for these experiments was 30 milliseconds. However, how we enforce program completion here is an implementation detail, not a feature of the general PADO architecture.

Why was each program only given 30 milliseconds to run?

If we had given them 1 second each instead of 1/33 of a second, learning would have taken 33 times as long. As a result, we do not have good information on how much better these programs could do if given such a long stretch to think about a single image. Some tests were done using time thresholds as large as 250 milliseconds. There was some increase in the peak performance for each generation, but taking into account the longer time to run each generation, the rate of increase, in computer cycles, of the peak performances for each generation was higher with rates closer to 30 milliseconds. Since machines change and as the environment improves, this 1/33 of second may become more like 1/100 of a second. It is more relevant, then, to talk about how much work gets done in the time allotted. At 1/33 of a second each program is able to evaluate approximately 2000 to 8000 nodes. These numbers are, of course, machine and implementation dependent. Remember, some nodes, like the terminals (e.g. pushing 137 onto the stack) and the simple non-terminals (e.g. ADD), evaluate quickly. But some of the nodes take a very long time to evaluate (e.g. (VARIANCE 0 0 255 255) takes about 0.12 milliseconds). Since each program is given a small, fixed amount of time to run, it must decide how to design its code to balance this difference in the time cost of evaluating various nodes. So the answer is that 1/33 of a second was chosen as a number which balanced well the criteria just mentioned. This choice means that the object recognizer will take $\mathcal{C} * \mathcal{S} * 1/33$ seconds to do the object recognition. For the results described in Section 7 this is about 2 seconds. Another point worth mentioning is that these programs are being interpreted each time they run. If you had $\mathcal{C} * \mathcal{S}$ programs which made up some PADO object recognizer, they could be compiled into LISP, C, Pascal, etc. and then from there compiled into assembly and run. This would probably yield a speed increase of between 5 and 20 fold.

Where is the "Parallel" in PADO?

Above, we mentioned that PADO (on $\mathcal{C} = 7$ classes) took about 2 seconds to make a prediction. Again, this number is machine and implementation dependent. This speed is a little slow for a reactive agent, but for computer vision in general this speed is reasonable. If we were to scale up to $\mathcal{C} = 100$ classes and kept $\mathcal{S} = 7$ programs per system, it would now take about $700 * (1/33) = 21$ seconds to recognize one image. If we compiled the programs as mentioned above, this would probably drop to 2 or 3 seconds, but this speed is still a little slow. Which brings us to the P in PADO: Parallel. Unlike most learned systems, the complex job that PADO does can be easily parallelized. If there are 100 classes and 7 programs have been selected from each of the 100 groups, there are 700 programs to run and if even 50 processors were available, the time to find an answer could be cut by a factor of 50. Because all the programs take exactly the same amount of time to run the speed up from the parallelism will be exactly linear. So a robot that had even 4 processors on board could recognize 100 classes in less than half a second, using the current compiled PADO technology.

Why was the Orchestration a weight vector tuning?

Orchestration is an important part of the PADO architecture. The implementation of this orchestration is less important partly because there are several good solutions. For example, an alternative method for orchestration using PADO itself was tested. That is, each orchestration procedure was a program developed in a separate PADO run in which the inputs were the outputs of $\mathcal{S}$ different object discrimination programs and the output was a confidence. Because this is a program it is easy to see that this could learn to return $\mathcal{I}$ where program $\mathcal{I}$ gave the largest input to this orchestration program. This orchestration program could, in fact, implement the weight vector tuning in its memory or do something even more complicated. Because this implementation of orchestration required a second, smaller training set and took more time, it was not used as the example for this chapter. Its results on object recognition were comparable to those shown in Section 7.3. An additional piece of appeal for the weight vector tuning is that because it is so fast, there is really no reason not to leave it on all the time during its "testing" phase. During its life span of usefulness, conditions may change or even be periodic. This continual orchestration tuning would allow the learned system to quickly turn over the decision making to those programs most suited for the current conditions. Paradigmatically, this continual adjustment also seems like a reasonable way to test the performance of a system. Off-line PADO can do a large amount of computation for learning. During testing, however, it may be the case that the system gets feedback about how it is doing. Any additional learning it can do in real time should not only be allowed, but encouraged. Orchestration can take place as part of the training phase, but we use the word orchestration rather than learning partly because it is the activity of orchestrating the parts that is important, and not that it counts as "learning" or "testing".

How much time did it take to get these results?

As has been discussed, the orchestration takes milliseconds, so the entire cost to develop a PADO recognizer is the time it takes to evolve the programs to be used in that PADO recognizer. It takes approximately 48 hours of CPU time on a DECstation5000/20 to train 7 classes up to 80 generations. Incremental learning up to generation 80 takes about 37 hours of CPU time. Both approaches used about twenty megabytes of memory. The environment is written in C. Further improvements and exploration of alternative techniques are being explored in our current research.

8.2. Related Issues

Why use more than one program to decide on a confidence for class $\mathcal{I}$?

It seems, at first, that if the "best" program is the best, then trusting the response of that program is the best PADO will be able to do. The first reason this approach would not be optimal is that the top few programs are picked as determined by the training phase. There is no guarantee that the individual that best fit the training data will also generalize the best. So that is a good reason for taking a few of the top programs from each group $\mathcal{I}$. If the information extracted by each program from the image were very similar (i.e., if their responses on specific pictures were highly correlated), then that would be the only good reason. However, this correlation is relatively low. It turns out the errors that the best program in group $\mathcal{I}$ makes on test images is often largely independent of the errors the second best program in group $\mathcal{I}$ makes on test images. And similarly for the second best relative to the third, etc. This means (approximately) that we can reduce our error by polling several of these programs. The chance that the majority of them are wrong goes down as the number of them increases. Because the top few programs are much more fit (able to discriminate correctly) than, for example, the average program from that group, there would be a disadvantage to taking too many to use in the PADO object recognition. This is why the PADO object recognizer does not use $\mathcal{C}^2$ programs to do predictions. PADO takes $\mathcal{S}$ programs from each group, where $\mathcal{S}$ is a small constant. So PADO grows linearly in time and space with increasing group size.

How do we know that these images aren't easy to distinguish?

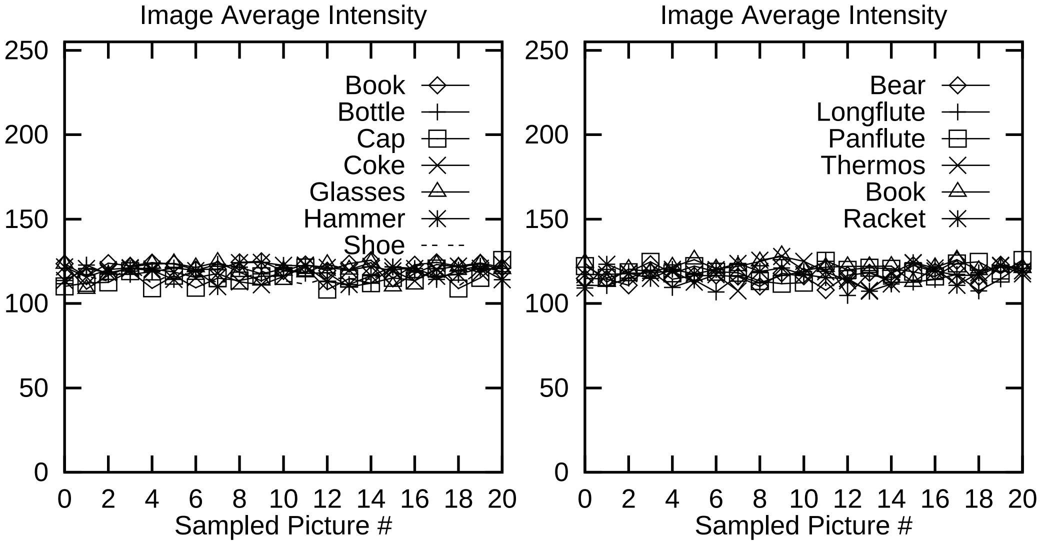


Figure 9: Average image intensity for random images from each class.

Figure 9 shows the average intensity for each of 21 randomly selected images from each class from each set (273 images total). Clearly, this piece of global information is not enough to partition the images into the correct classes. That fact, of course, does not rule out the existence of some global property of the image on which partitioning can be done successfully. Since the five non-local image primitives these programs had were AVERAGE, MOST, LEAST, VARIANCE, and DIFFERENCE all five of these were tested as global properties. The graphs of the other four primitives for both data types look very much like the two shown above. The large amount of variance between pictures in the same class for all four of these tests suggests that there is no real predictive power in these values. So any successful technique for recognizing the objects in these images must be based in part on the ability to focus attention. Beyond the foviation that we have just concluded the PADO recognizer is doing, it becomes very difficult to say how hard two images are to distinguish from each other. Also, because of the complexity and density of these programs, it is very hard to ascertain whether, for example a particular successful program is looking for light-dark boundaries, or whether it is looking for particular shapes, or textures, or any other type of visual clue.

One detail of note is that PADO may give us object position for free in images with a single object each. As was mentioned in Section 6, some people may feel that object recognition requires at least object position location in the image and maybe even pose determination. Each image primitive that PADO executes (PIXEL, AVERAGE, MOST, LEAST,VARIANCE, DIFFERENCE) takes place in a particular part of the image. A simple computation is to find, for each image that some particular "best" program from class $\mathcal{I}$ looks at, what the center of mass of all these image primitive locations is. It turns out that with high probability this center of mass will be near the center of the object in the image. Since the object typically takes about 10% to 25% of the image and moves widely between images, and this happens with regularity, PADO demonstrates its ability to focus attention and returns a free piece of information: object location in the image!

What reasons exist for believing that PADO will scale up to 100 classes?

PADO ranks the population along $\mathcal{C}$ different dimensions. Sometimes it turns out that one of the best in class $\mathcal{J}$ is also a good program in class $\mathcal{T}$, where $\mathcal{T} \neq \mathcal{J}$. In other words, it might turn out that at the end of some generation, program $\mathcal{X}$ is the best at distinguishing bottles from other objects, but also happens to be pretty good at distinguishing coke cans from other objects. This must be because the coke can and the sprite bottle have some visual similarities (e.g. shape or size). To scale up the number of classes, we will introduce a hierarchical orchestration. The difficult part of PADO will become the construction of this hierarchical structure for classes. We can use indications of similarity like the one just mentioned to find correlations between the image classes and build the hierarchy. More will be said on this in the future work section.

Will PADO work with more similar classes?

The results shown in this chapter are not the only tests PADO has passed. For example, set T originally had three dimensions: color, saturation, and brightness. When PADO was given the color information, its performance on object recognition jumped to about 95%. The reason is undoubtably because the images were "too" different along some dimension. That dimension was color. The objects were of such different colors that this information is, by itself, a very good indicator of the class. In an effort to make the problem harder, we gave PADO only the brightness data (the greyscale images shown in Figure 2). PADO didn't do as well, but given that the problem got much harder, still did very well. This will be mentioned again in the future work section. So while PADO's performance on 7 different shoe classes is in question, we could certainly have chosen objects, or signals that represent those objects, that were much less similar.

9. Future Work

Scalability

The issue of scalability is critical to the success of PADO. We are trying to design a learning architecture that can build up useful systems to be applied to natural environments. Because PADO was designed with an eye for success as well as scientific advancement, it must be able to scale up to hard problems.

The most obvious area of scalability is in the number of classes. This chapter dealt with a system that performed well in an environment with 7 classes. While there are real applications for recognizing 5 to 10 classes, most applications need to be able to recognize hundreds of classes. In the discussion section we mentioned that new types of orchestration are an active part of our current and future work. By doing a hierarchical structure for the orchestration, we hope the number of classes PADO can handle will at least move into the hundreds.

A second important area of scalability is performance. PADO's results on set A and set T were much better than random, but they were not perfect. Many applications can be useful even if there is some error, but most applications require error rates of 5% to 10% and some cannot tolerate even that. So PADO's ability to improve its performance is also critical for PADO's future. This issue becomes even more important when we remember that, as PADO begins to tackle hundreds of objects instead of tens of objects, the difficulty of the problem rises very quickly and it would be impressive for PADO to maintain its current performance on seven classes as the number of classes rises.

Though less pressing, time factor is also a scalability issue for PADO. As we require PADO to perform much better on many more classes, the number of generations necessary to achieve this state will skyrocket. So even though most learning can be done "off line" in a non time-critical way, it will become increasingly important to find faster ways to implement PADO and better architectural choices that require less space or time. One point here in PADO's favor is that evolutionary computation lends itself easily to massive parallelization. So by using 100 processors, we could divide PADO's learning time 100 fold. An additional avenue that we have already started to explore is incremental learning. As discussed in the Section 7.2 it is possible to train several classes, and then later add a new class. As our ability to get new classes quickly up to speed improves, we may be able to conquer the time problem this way.

Partly because they are so difficult to understand, it is hard to know exactly what role the Libraries and the SMART operators play in PADO's performance. Clearly the future performance of PADO will depend heavily on how well we can improve these features of the PADO architecture.

Investigation for Greater Understanding

There are several parts of the PADO architecture that are not yet well understood. Two examples just mentioned are the Libraries and the SMART operators. In both cases we have good indications that they are vital to the functioning of the system, but we don't completely understand why. The work we are doing to better understand different aspects of PADO will help us to change these things to achieve some of the goals of scalability mentioned above. Another example of a PADO aspect that should be investigated is the programs themselves. By understanding better how they accomplish their tasks we may be able to learn about how, in general, object recognition and signal understanding is done.

Innovative Applications of PADO

PADO was not designed specifically for images. It was designed to be able to do classification on any set of signals. One of the most important next steps in PADO's growth will be a series of classification tests on a variety of signal types, all of which were not designed with PADO in mind. These signal types will include speech, sonar, radar, text, and several others. The good news is that PADO has already been applied to noises recorded in real world environments and has achieved classification rates [18] as high on these sounds as on the images reported in this chapter!

In summary, the open questions of this research effort are:

- Can PADO scale up to a large number of classes?
- Can PADO learn the same amount much faster?
- Can PADO improve its performance even as the number of classes increase?
- Will PADO prove to be a general signal classification learning architecture?

The investigation of these questions is part of our immediate research agenda.

10. Conclusions

This chapter began with the motivation of the signal-to-symbol problem. AI systems need to reason about high level information, but the world provides a huge amount of noisy raw data instead. Any bridge between these two realms is a significant tool for AI. In humans, we depend most heavily on the signal-to-symbol translation in our visual cortex. Taking this as our cue, we decided to tackle the problem of object recognition. Object recognition's major flaw has been that it does not address unconstrained or "natural" environments very well. Machine learning has, aside from some small pockets of success, not yet delivered in this area. This level of success may be because the learning architectures that have been applied were not designed for the task of understanding real world signals. Out of this belief that new architectures must be found, PADO was born.

This chapter has shown an application of PADO on three related tasks with two different sets of image data. Both sets of image data fall outside the constraint boundaries that object recognition tasks usually require. That is, translation, rotation, lighting, and even foreshortening were allowed for the object classes. In addition, the objects that made up the classes were not simple, uni-colored, geometric, or even rigid in some cases.

On these difficult problems, PADO achieved an object recognition rate of almost 70%. Given that there were seven classes, this is about 5 times better than random guessing would accomplish. As was mentioned in the discussion section, PADO's performance on images from Set T jumps to about 95% when the original color images are used. So the performance described in this chapter is on image sets that have been made deliberately difficult.

PADO achieved this performance with **no** help from users or domain specific information of any kind. To prove this point, PADO was given as its primitives for accessing images the simplest possible functions: Pixel, Least, Most, Average, Variance, and Difference. The fact that these primitives were coded for PADO in half an hour and were found to be sufficient for a different image set (Set T) supports the hypothesis that PADO can achieve a significant level of performance with little or no outside intervention.

The design of PADO's architecture provides for several exciting features.

- At any point during the learning process, a program, as a group of "signal understanding" systems, can be extracted and used immediately for object recognition.
- PADO incorporates evolution into its design, thereby providing the chance to exploit a myriad of different solutions through orchestration.
- The orchestration of independent solutions makes it possible (simple even) to run PADO on a parallel machine for a linear speed increase.
- The architecture's independence from the particular examined signal makes it viable to use PADO for any signal type.

The future paths of PADO research are diverse. In the near future we hope to see PADO performing better, with fewer examples, of more classes, on harder images. In addition, PADO has been designed to perform signal understanding on *any* signal type: from text, to sonar, to spectrum, to speech. Our goal is to have performance at the current highest levels of learned understanding in any signal type we try. As was mentioned in the previous section, PADO has already been applied to noises recorded in real world environments. PADO achieved classification rates [18] as high on these sounds, for the same number of classes, as on the images reported in this chapter.

Researchers today spend a significant amount of their time finding the right learning algorithm for their task, and then tweaking that algorithm until it performs. There are simply too many domains for this to lead to real progress. We believe that PADO, as a domain independent learning architecture, can have a significant impact in solving the general signal-to-symbol problem.

Acknowledgements

We gratefully acknowledge Sebastian Thrun for the permission to use his images and Tom Mitchell for providing hardware so we could take our own images. Also, Peter Stone, Alicia Perez, Eric Siegel, and Joseph O'Sullivan, Matt Glickman, and Mark Wheeler were invaluable as readers.

Appendix

A. Sample Program

This program was the best at recognizing hand-held Rackets in Generation 57.

Legend

Sign	Meaning	Sign	Meaning	Sign	Meaning
#	Node Number	A1	ARC$_1$	00	Stop Node
Action	Action at that node	A2	ARC$_2$	01	Start Node
Branch	Branch-Decision at that node	B-C	Branch-Constant		

MAIN

#	ACTION	BRANCH	A1	A2	B-C	#	ACTION	BRANCH	A1	A2	B-C
01	114	NOT	4	4	-206	02	DIV	MULT	23	41	52
03	174	DIV	21	4	0	04	227	READ	21	21	-69
05	86	MAX	12	37	11	06	VARIANCE	MIN	19	13	-199
07	245	MAX	38	9	99	08	214	MAX	7	36	92
09	WRITE	SUB	42	20	0	10	208	MULT	30	13	-189
11	59	SUB	18	38	201	12	85	MULT	3	26	12
13	184	READ	35	17	0	14	96	MIN	41	19	0
15	11	ADD	5	45	0	16	WRITE	MAX	40	19	-7
17	READ	NOT-EQ	20	34	104	18	243	LESS	40	40	-71
19	40	EQ	44	42	-252	20	32	DIV	30	27	0
21	MOST	MORE	11	6	184	22	12	READ	28	12	0
23	125	NOT-EQ	16	44	0	24	29	EQ	2	17	0
25	147	NOT	33	14	78	26	READ	READ	3	22	-53
27	AVERAGE	MULT	26	13	233	28	57	ADD	4	10	-174
29	199	SUB	31	47	-28	30	LIBRARY[99]	ADD	47	29	0
31	58	DIV	30	6	220	32	WRITE	READ	6	13	-252
33	204	DIV	42	27	-178	34	224	DIV	6	8	58
35	114	MORE	12	30	0	36	121	NOT	48	35	0
37	ADD	ADD	24	25	0	38	80	SUB	29	32	201
39	71	MULT	44	10	209	40	WRITE	SUB	12	8	92
41	80	ADD	3	4	0	42	LIBRARY[39]	MORE	12	14	-228
43	207	NOT-EQ	39	20	103	44	234	MULT	15	16	90
45	91	MIN	34	34	-159	46	AVERAGE	NOT-EQ	25	43	0
47	174	DIV	32	26	0	48	96	MULT	46	31	0

Library[39]

#	ACTION	BRANCH	A1	A2	B-C	#	ACTION	BRANCH	A1	A2	B-C
00	96	NOT	9	8	-156	01	1	NOT-EQ	9	6	0
02	DIFF	ADD	6	5	201	04	57	ADD	1	7	0
05	23	ADD	8	4	-243	06	LIBRARY[76]	SUB	10	7	-83
07	33	MULT	7	7	94	08	29	READ	2	0	-79
09	39	MORE	4	4	0	10	207	DIV	5	10	-142

Library[76]

#	ACTION	BRANCH	A1	A2	B-C	#	ACTION	BRANCH	A1	A2	B-C
00	213	READ	14	4	0	01	LIBRARY[99]	MAX	12	20	-219
02	201	ADD	0	5	218	03	44	READ	15	19	76
04	68	LESS	0	11	0	05	158	DIV	18	2	0
07	88	DIV	7	21	-76	08	VARIANCE	SUB	3	7	-246
10	78	SUB	0	11	0	11	PIXEL	ADD	5	17	0
12	PIXEL	MORE	13	13	0	13	244	NOT-EQ	8	5	0
14	145	LESS	13	17	-32	15	119	NOT	3	17	-232
16	175	EQ	10	10	-223	17	EQ	MORE	5	21	-235
18	EQ	MULT	17	0	0	19	46	LESS	3	15	153
20	118	ADD	2	0	0	21	WRITE	LESS	15	16	240

Library[99]

#	ACTION	BRANCH	A1	A2	B-C	#	ACTION	BRANCH	A1	A2	B-C
00	WRITE	MAX	20	10	0	01	181	MORE	3	7	-180
02	233	DIV	13	5	149	03	NOT	EQ	14	8	-145
04	99	LESS	13	12	42	05	WRITE	MORE	9	4	9
06	54	MORE	9	15	-88	07	EQ	MIN	10	4	-228
08	154	SUB	5	6	0	09	WRITE	SUB	3	17	73
10	101	ADD	0	16	0	11	NOT	MAX	12	11	
12	MINI*	EQ	11	15	-8	13	LEAST	SUB	6	2	-153
14	242	ADD	9	10	154	15	16	ADD	5	15	-111
16	144	MIN	4	16	46	17	248	SUB	10	4	62

*: MINI is never actually called by this program.

References

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

[2] Farshid Arman and J. K. Aggarwal. Cad-based vision: object recognition in cluttered range images using recognition strategies. In *Image Understanding*, pages 33–49. Ablex, 1993.

[3] David J. Braunegg. Marvel: a system that recognizes world location with stereo vision. In *IEEE transactions on Robotics and Automation*, pages 303–310. IEEE, 1993.

[4] Michael Patrick Johnson et al. Evolving visual routines. In Rodney Brooks and Pattie Maes, editors, *Artificial Life IV*, pages 198–209. MIT Press, 1994.

[5] David Goldberg. *Genetic Algorithms: In search, optimization, and machine learning.* Addison-Wesley Press, 1989.

[6] John Koza. *Genetic Programming.* MIT Press, 1992.

[7] John Koza. *Genetic Programming II.* MIT Press, 1994.

[8] S.Z. Li. Toward 3d vision from range images: an optimization framework. In *Image Understanding*, pages 231–261. Ablex, 1992.

[9] Thang Nguyen and Thomas Huang. Evolvable 3d modeling for model-based object recognition systems. In Jr. Kenneth E. Kinnear, editor, *Advances In Genetic Programming*, pages 459–476. MIT Press, 1994.

[10] Dean Pomerleau. *Neural Network Perception for Mobile Robot Guidance.* PhD thesis, Carnegie Mellon University School of Computer Science, 1992.

[11] 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.

[12] Walter A. Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programs.* PhD thesis, University of Southern California, 1994. Available as: Technical Report CENG 94-13. Dept. of Electrical Engineering Systems.

[13] Walter A. Tackett. Greedy recombination and genetic search on the space of computer programs. In L.D. Whitley and M.D. Vose, editors, *Proceedings of the Third International Conference on Foundations of Genetic Algorithms*, pages 118–130. Morgan Kauffman, 1995.

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

[15] Astro Teller. Genetic programming, indexed memory, the halting problem, and other curiosities. In *Proceedings of the 7th annual FLAIRS*, pages 270–274. IEEE Press, 1994.

[16] Astro 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.

[17] Astro Teller. Evolving programmers. In *Proceedings of the Sixth International Conference on Genetic Algorithms.* Morgan Kauffman, 1995. Submitted for review.

[18] Astro Teller and Manuela Veloso. Program evolution for data mining. In *The International Expert Systems Journal.* IEEE, 1995. Submitted for review.

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

[20] Mark D. Wheeler and Katsushi Ikeuchi. Sensor modeling, probabilistic hypothesis generation, and robust localization for object recognition. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 17, March 1995.