Benchmarking The Generalization Capabilities Of A Compiling Genetic Programming System Using Sparse Data Sets

Frank D. Francone
Law Office of Frank D. Francone
4806 Fountain Ave. #77
LA, California 90027
ytns65a@prodigy.com
213-953-8584

Peter Nordin
University of Dortmund,
Dept. of Computer Science
LS11 D-44221 Dortmund Germany
nordin@ease.informatik.uni-dortmund.de. +49 231 9700-956

Wolfgang Banzhaf
University of Dortmund,
Dept. of Computer Science
LS11 D-44221 Dortmund Germany
banzhaf@LS11.informatik.uni-dortmund.de. +49 231 9700-953

ABSTRACT

Compiling Genetic Programming Systems ('CPGS') are advanced evolutionary algorithms that directly evolve RISC machine code. In this paper we compare the ability of CGPS to generalize with that of other machine learning ('ML') paradigms.

This study presents our results on three classification problems. Our study involved 720 complete CGPS runs of population 3000 each, over 500 billion fitness evaluations and 480 neural network runs as benchmarks. Our results were as follows:

1. When CGPS was trained on data sets that were not too sparse, CGPS performed very well, equaling the generalization capability of other ML systems quickly and consistently.

2. When CGPS was trained on very sparse data sets, CGPS produced individuals that generalized almost as well other ML systems trained on much larger data sets.

3. As between CGPS and multilayer feedforward neural networks trained on the same sparse data sets, CGPS generalized as well (and often better) than the neural network.

1. Introduction

Researchers in Machine Learning ('ML') frequently encounter problems of high dimensionality and too little data. An ideal ML system would, of course, be able to generalize well, even though it was trained on sparse data. This paper compares the effect of using sparse data sets on the generalization capability of our Compiling Genetic Programming System ('CGPS') with the generalization ability of a K-Nearest Neighbor classifier systems ('KNN') and Multilayer Feedforward Neural Networks ('MFLN').

We used three classification problems from the ELENA database—two real world problems and one difficult but artificially generated data set. The data sets that we used were the IRIS, the PHONEME and the GAUSSIAN 3D data sets [ELENA 1995].

ELENA provides a set of KNN benchmarks for each of these problems. The ELENA partners derived the KNN benchmarks using the entire ELENA data sets—usually in excess of 5000 training points—and optimized the KNN performance by varying the value of *K* until the best performance was had on the testing set [*Id*].

Our purpose in this paper was to test generalization performance where the number of training instances was severely limited. So the ELENA KNN benchmark figures are useful primarily to set the boundaries of what performance might be expected where we have the luxury of training on an abundance of data. Thus, the real question we address is in this paper is, where the training set is sparse, how close can CGPS and MLFN come to the KNN benchmarks that were derived on much larger data sets?

We performed 720 CGPS runs (240 on each problem) and 480 neural network runs (240 on each of two of the problems). Both the CGPS and MLFN runs were performed using a variety of parameters.

The results were gratifying. CGPS generated high quality solutions without using any experimenter knowledge of the problem domain. CGPS often produced solutions that generalized almost as well as ELENA benchmarks that had been trained on *much* larger data sets. Finally, CGPS' generalization performance was, by several different measures, superior to the performance of our 480 neural network benchmark runs that were trained on the same sparse data sets as were the CGPS runs.

2. An Introduction to CGPS

CGPS is the direct evolution of binary machine code through the use of evolutionary operators such as crossover and mutation. It would be impossible to describe the full CGPS system in the space allotted. CGPS has, however,

been described in detail elsewhere [Nordin 1994, Nordin & Banzhaf 1995, and Nordin, Francone & Banzhaf, 1996].

In our description of CGPS below and in the works referenced, the reader will note that although CGPS is, in a very general sense, a 'Genetic Programming' system, it is also very different from 'Genetic Programming' as described in [Koza 1992] or as that term is used in the literature. Among other things, CGPS does not evolve trees, has no local memory, the CGPS genome is linear, crossover looks more like genetic algorithm string based crossover than like GP crossover, CGPS operates directly on the CPU's registers and CGPS uses only the CPU's instruction set. Thus, despite superficial similarities, CGPS should be regarded as an evolutionary algorithm that is quite distinct from canonical Genetic Programming.

2.1 Description of a CGPS Program

An evolved CGPS program is a sequence of binary machine instructions. Thus, an evolved CGPS program might be comprised of a sequence of three, 32 bit machine instructions. When executed, those three instruction would cause the CPU to perform three operations on the CPU's hardware registers. Here is an example of a simple, three instruction CGPS program that uses three hardware registers:

register 2 = register 1 + register 2 (1)
register 3 = register 1 * 128 (2)
register 3 = register 2 Div register 3. (3)

One of the three hardware registers in this sample CGPS program is selected as the output register. Once the output register is selected, a fitness evaluation for this sample CGPS program would consist of the following steps:

1. Initialize the hardware registers with the input values for the fitness instance;
2. Execute the above three instruction program {(1)-(3)} on the hardware registers as initialized; and
3. Evaluate the value in the selected output register for fitness against the fitness function.

While CGPS programs are apparently very simple, it is actually possible to evolve functions of great complexity using only simple arithmetic functions on a register machine [Nordin & Banzhaf 1995a].

2.2 Protocol for Reporting CPU Time

While there are many advantages to CGPS, the most pronounced advantage is probably speed [Nordin 1994]. Simply put, evolving machine code directly with no compilation is *much* faster than interpreted languages like LISP or compiled languages like C.

We conducted 720 runs with large populations (3000) on each run. We completed the runs in well under 3 weeks on a single Sun 20 Workstation. But this number greatly understates the actual speed of CGPS. On this project, at least 98% of the CPU time was occupied with a research measurement we call 'intron counting'. At the end of each generation, each individual in the population is evaluated for introns by replacing each instruction, one-by-one, with a No-OP instruction and then evaluating the fitness of the individual with the No-OP instruction included. If this replacement of an instruction has no effect on the individual's fitness, we count the instruction as an intron [Nordin, Francone, & Banzhaf 1996].

Intron counting has a *huge* effect on the time it takes to evolve solutions. For example, an individual that is 100 instructions long requires 100 separate fitness calculations that are devoted solely to intron counting. Where the average size of an individual is 100 instructions, intron counting increases the amount of time to evolve a population by a factor of 100. Applying this analysis to our 720 runs, we have calculated that intron checking in the first 20 generations of our runs occupied approximately 98% of the total CPU time in the run. Because the average size of individuals grew after generation 20, the impact of intron counting grew after generation 20—so a 98% figure actually understates the effect of intron counting substantially.

While intron counting yields useful research data [*id*], it also has nothing to do with the immediate job of evolving high quality individuals—intron counting is purely a research tool. Had we used a 'production' version of CGPS stripped of intron counting, this entire project—all 720 runs of 3000 individuals each—would have been completed in about eight hours on our Sun 20 workstation.

Henceforth, we will follow the practice of reporting CPU time figures for the production portion of the CGPS system, without including the 'research' overhead. This approach has two advantages. To begin with, it gives consistent performance figures from problem to problem and from research project to research project—research overhead varies on different projects. Further, although this approach to CPU time reporting is approximate, it is more representative of the time needed to evolve individuals than the gross figures that include research overhead.

A brief note here on the CPU time occupied by the MLFN runs is appropriate. On the GAUSSIAN 3D and the PHONEME problems, the MLFN runs took about 320 hours of CPU time on a 66 Megahertz 486 IBM compatible PC with 20 megabytes of RAM running in 32 bit DOS extended mode. Those same two problems occupied about 4.5 hours of CPU time in the CGPS runs. Of course, the Sun 20 Workstation is a much more powerful machine than the 486, even running in DOS extended mode. Any further comparisons of time are effectively meaningless because the platforms are so radically different.

3. Generalization Issues

3.1 Measuring Generalization

We evaluated the generalization of our CGPS and neural network runs as follows. To begin with, we separated our data into training and testing sets.

For CGPS runs, the best individual of each generation on the training set was evaluated on the testing set. The results of that test were preserved. After completing the CGPS run, we used the data preserved during the run, as described above, to select the best generalizing individual from the run. (We selected the best generalizing individual by selecting the individual with the highest number of 'hits'

on the test data.) We then used that best generalizing individual to represent the generalization of the entire run.

Because the outputs available in the neural network package used were different than the outputs on our CGPS program, we had to choose the best generalizing network in a slightly different manner. For each neural network run, the network with the highest R^2 on the testing set was considered to be the best generalizer of the run. For comparison purposes with the CGPS system, we then calculated the number of 'hits' of this best generalizing network as a percentage of the total testing set.

The foregoing is a decent (though not optimal) way to test generalization [Masters 1995, Katz 1996]. We did not implement more optimal techniques such as the Jacknife and Bootstrap methods [Masters 1995] because the CPU time to perform them on the neural network runs made such techniques utterly impractical.

3.2 Improving The Measure of Generalization With a Third and Off Training Sample Data Set.

We acknowledge one respect in which the above methodology could have been improved. We choose which individual (or network) from a run is the 'best' generalizer by assessing its performance on the testing data. For example, if the best CGPS individual correctly classified 85% of the test cases in generation 10 but in generation 11, the best individual correctly classified only 80% of the test cases, we used the 85% figure to represent the generalization performance of that run.

But once we have chosen the best generalizer in the above manner, the testing set is, strictly speaking, no longer a testing set—it has become part of the training set. The reason? The testing set was used to select among different individuals.

The method that we used does provide useful information about the generalization capabilities of an ML system. In our prior experience, we have found that there is a decent correlation between generalization on the testing set and generalization on entirely new data. But to make our assessment of generalization more rigorous, a *third* data set ought to be added. So while our results here are meaningful, the ability of the best generalizer on the testing set to generalize on yet a third data set would measure generalization more accurately. We plan to implement this capability in our CGPS system in the near future.

3.3 No Free Lunches?

We acknowledge the ongoing controversy over the 'No Free Lunches' theorem and whether it is possible to find *any* algorithm that has *any* meaningful generalization capabilities [*Compare* Wolpert 1994 *with* Rao 1995]. It is not our purpose to add to that discussion here. We simply note that, at this point, the theoretical guidance offered by papers on this subject about how to judge problem 'domains' is sufficiently vague that ML yet remains an experimental science. It is in that spirit that we present these findings. Should it turn out that learning is effectively impossible in this universe, our work here will be for naught and our research group shall have to learn to occupy its time with pursuits other than experimental machine learning.

We do, however, agree with Rao's emphasis on the importance of off-training-set error, as we discussed above. And we repeat here Rao's admonition that "[R]esearchers should be careful not to say that Algorithm A is better than Algorithm B without mentioning that this holds with respect to the particular problem distribution." [*Id*]. That was good advice both before the 'No Free Lunches' Theorem and remains so today.

4. Description of the CGPS Portion of the Experiment

Our CGPS experimental setup stayed mostly constant from problem to problem. Table 1, contains the general experimental specification [Koza 1992].

Table 1. Experimental Specification for CGPS Runs

Objective	Classification of data sets from ELENA Database
Parsimony Factor	0, 0.1, or 1
Explicitly Defined Introns	Enabled or Not Enabled
Crossover/Mutation Mix	95/5, 80/20, 50/50, or 20/80
Terminal Set	Integer constants initialized from 0 - 10. May be mutated to the range, 0 - 255.
Function Set	Addition, Multiplication, Subtraction, Division
Number of Hardware Registers	One more than the number of classes in the data.
Number of Fitness Cases	75-100. Varies with problem.
Number of Testing Cases	75-100. Varies with problem.
Fitness Function	Sum of Absolute Deviations.
Hits	See discussion above.
Wrapper	None
Population Size	3000
Selection for Genetic Operators	Tournament. 4/2. Children replace the losing members of the tournament.
Termination Criteria.	200 Generations or Destructive Crossover falls to less than 10% of the total crossover events, whichever comes first.
Maximum Individual Size	256 instructions.
Total Number of Runs	720

Some of the parameters mentioned in Table 1 merit separate discussion.

4.1 Parameters That We Varied From Run To Run

We varied three of the parameters from run to run. These parameters were: Parsimony Factor, Explicitly Defined Introns, and Crossover/Mutation Mix.

There were twenty-four different combinations of values used for the above parameters (Table 1). Each of those twenty-four combinations of parameters was run for each of our three problem sets on ten different random seeds. As a result, we conducted a total of 240 runs per problem set. On the three problems reported here, we conducted a total of 720 runs.

4.2 Fitness Function

We had to define an appropriate fitness function for classification problems where the number of classes might vary from problem to problem. We did that as follows. We selected one hardware register as the output register. The value in the output register was then evaluated for fitness on a particular fitness case as follows:

If a particular fitness case should have been classified as being a member of *Class 0*, then the fitness of the individual for that fitness case is the absolute value of the difference between the value in the output register and *100*;

If a particular fitness case should have been classified as being a member of *Class 1*, then the fitness of the individual for that fitness case is the absolute value of the difference between the value in the output register and *200*; and so forth where there are more classes.

4.3 Measuring Testing Set Hits

Table 2 is the lookup table that we used to determine the meaning of a value contained in the Output Register of an individual for a particular fitness case.

If the Predicted Class Membership determined by Table 2 for a particular fitness case is correct, then that prediction is counted as a '*Hit*' for that fitness case. The total number of hits for an individual is then divided by the total number of test cases. Where the testing set is 100 in size, and where an individual gets 75 hits on the test set, that individual's Hit score would, therefore, be 75%.

Table 2. Class Membership Corresponding to Output Register Values

Output Register Value of . . .	Predicted Class Membership
50 - 149	Class 0
150 - 249	Class 1
250 - 350	Class 2

Table 2 was obviously designed arbitrarily. The effect of that design is that researcher knowledge of the domain space could not bias the results.

4.4 Function Set

We deliberately used a very simple function set—the arithmetic operators. The purpose of using the same function set that we used for previous symbolic regression work was to maintain as much independence as possible from experimenter knowledge of the problem domain.

4.5 Parsimony Factor

We used three different values for the parsimony factor—0, 0.1, and 1. A value of *n* for the parsimony factor means that the *n* times the length of the individual (measured by the number of instructions that comprise the individual) is subtracted, in the fitness function, from the fitness of the individual. A value of 0.1 for the parsimony factor, therefore, means that 1/10 of the length of an individual is subtracted from the fitness of the individual.

4.6 Explicitly Defined Introns

Explicitly Defined Introns ('EDI's') were developed in our previous work where we found that using EDI's to allow GP to vary the probability of crossover between instructions without effecting the fitness calculation of the individual improved fitness, generalization and speed. [Nordin, Banzhaf & Francone 1996.] We enabled EDI's in the manner described in the aforementioned work for one half of all runs.

4.7 Termination Criterion.

The maximum number of generations that we allowed the system on any CGPS run was 200 generations. However, we have previously reported a strong correlation between the exponential growth of the size of programs and a decline in the portion of total crossover events that can be characterized as destructive crossover [Nordin, Francone & Banzhaf 1996]. In fact, our earlier findings suggest that, at the point where such exponential growth occurs, all effective training is over [*Id*]. Accordingly, we monitored the rate of destructive crossover during training. When destructive crossover fell to 10% of the total crossover events in any given generation, we terminated the run even if it had not reached generation 200.

This early run termination approach saved a substantial amount of CPU time. Over 52% of our 720 runs were terminated by generation 80 because of the early termination criterion. Had these runs not terminated early, they would have continued the full 200 generations. Had we not used our early termination criterion, the total number of generations that our system would have needed to complete all 720 runs would have been 144,000 generations (720 runs times 200 generations per run). Instead, our system needed only 75,134 generations to complete the entire 720 runs. This savings was entirely due to our early termination criterion based on monitoring the destructive crossover rate relative to the total crossover rate.

5. Description of the Multilayer Feedforward Neural Network Portion of the Experiment

5.1 Selection of the MLFN

We decided to run the same sparse data that we used in our CGPS runs on at least one other established machine learning system. We spent some time deciding whether to use a Multilayer Feedforward Neural Network or a Prob-

abilistic Neural Network ('PNN'). Generally speaking, a PNN is better for classification problems and is much faster than a MLFN. We had access to MLFN and PNN software from the same vendor so we discussed with the vendor which of the modules would work better on our sparse data set problem. It was suggested that the MLFN module would be more effective where the number of training samples were limited [Katz 1996]. As a result, we ran the same sparse data used for our CGPS runs on the N-TRAIN MLFN module from version 1.02 of the N-TRAIN Neural Network Development System [McCormick and Katz 1992]. N-TRAIN uses the backpropogation algorithm to train multilayer feedforward neural networks. N-TRAIN runs in 32 bit DOS extended protected mode. Thus, it is very fast for a PC based system.

5.2 Neural Network Architectures

We used a fully connected three layer MLFN with linear neurons in the input layer and sigmoid neurons in the other layers. We varied the number of hidden neurons according to the following protocol, which we set in advance of training and testing. For a network that has *n* independent variables as inputs, we did 30 training runs using 30 different random seeds for each network architecture that could be constructed consistent with the following rules:

1. The network has an input layer consisting of *n* linear neurons,
2. The network has one hidden layer consisting of *k* sigmoid neurons where $n <= k <= n*2$; and
3. The network has an output layer consisting of one sigmoid neuron.

So where there were three inputs for a problem, we did a set of 30 runs using 30 random seeds each for three layer architecture consisting of 3, 4, 5 and 6 hidden neurons, respectively.

Based on our past experience, this range of hidden neurons usually gives a good coverage of the better network structures. When the above runs were finished, we examined whether generalization performance appeared to be improving at either the high or low end of the hidden neuron architectures. If so, we performed runs for the next two numbers of hidden neurons in the direction that the network performance generalization was improving. By way of example, if the generalization performance in the above example appeared to be improving between 5 and 6 hidden neurons, we would do 30 runs using 30 random seeds for 7 and 8 hidden neurons architectures. If the performance continued to improve, would do two more hidden neurons and so forth. {We note, in retrospect, that this procedure of expanding the number of hidden neurons based on existing results probably biased the experiment in favor of the MLFN architecture. No such interative procedure was followed on the CGPS portion of the experiment. Were we to do this again, we would eliminate that portion of the experiment.}

After we finished the above procedure, we performed enough extra runs on that problem using new random seeds distributed evenly over all of the hidden neuron configurations used in the above procedure so that the total number of runs on that problem with the MLFN equaled 240 runs. Completing 240 runs with the MLFN—the same number of runs that we performed with the CGPS system—greatly improves the usefulness of the difference in means statistical test between the MLFN and the CGPS, which test we will use later [Crow, Davis & Maxfield 1960].

Based on this protocol, we performed 60 runs each with 3-6 hidden neurons on the GAUSSIAN 3D problem, which had 3 inputs. We also performed 30 runs each with 3-10 hidden neurons for the PHONEME problem set, which had 5 inputs. Therefore, the total number of runs we performed on the GAUSSIAN 3D problem set was 240 and the number on the PHONEME problem set was 240.

5.3 Data Preprocessing

Our CGPS runs required integer inputs. Neural networks require that inputs be scaled in the range 0 to 1 or -1 to 1 range. We scaled the inputs to the 0 to 1 range.

For reasons that are discussed below, we ran the MLFN only on the GAUSSIAN 3D and PHONEME problem sets. Each of those sets had only two output classes. Therefore, we gave the output data to the network as follows: All instances that were Class 0 were assigned an output value of 0—all those of Class 1 were assigned an output value of 1.

5.4 MLFN Parameter Settings

For the most part, we used the default parameter settings that come with the program—having found them in our previous work to be an effective parameter set. There were, however, three parameters that we changed from the default setting, each of which is detailed below.

The reader will note that we made our changes in the MLFN parameters using the suggestions of the proprietor of the MLFN software. His suggestions were excellent and greatly improved the performance of the MLFN system.

5.5 Learning Rate and Neuron Type Parameters

In our initial MLFN runs, we used the N-TRAIN default learning rate and neuron type settings. The results from those runs were unexpectedly poor. The vendor suggested that a much lower learning rate of 0.25 would be closer to optimum for our problems. In addition, he suggested that we use linear instead of sigmoid neurons on the input layer [Katz 1996]. We made those two adjustments to the parameters with greatly improved training results.

5.6 Error Tolerance Parameter Used to Determine Testing Set Hits

In order to compare the CGPS hits with MLFN hits, we had to measure the hit rate of the MLFN. In order to measure the MLFN hit rate, it is necessary to set a parameter called 'error tolerance' [McCormick and Katz 1992]. This proved to be the most difficult parameter to set in a principled manner.

Simply put, the error tolerance parameter determines how close to 0 or 1 the network output must be in order to be counted as a 'hit.' The program default setting of 0.10 meant that the network must output a value of less than

0.10 or greater than 0.90 to be counted as a 'hit.' All outputs between 0.10 and 0.90 were counted as wrong.

This default setting caused very poor performance by the MLFN because, the networks that had the best R^2 on the testing set were generating a lot of 'correct' answers that were between 0.10 and 0.90. (By 'correct', we mean that the network output was closer to the correct answer than it was to the wrong answer.)

The solution on this might initially seem to be to set the error tolerance to 0.4999. This, however, forces the MLFN to 'guess'. That is, where there are only two classes, the MLFN cannot do worse than 50% hits with an error tolerance of 0.4999. But the reader will note that the CGPS system was set up so that it was not forced to guess. So this solution was also not acceptable.

We finally elected to use an error tolerance of 0.25. This seemed to deliver good performance but to prevent the MLFN from using a range of outputs (between 0.25 and 0.75) where there was little or no relation between the network output and the correct answers. This is not an entirely satisfactory solution but it was the best we could devise given the differences between the paradigms.

5.7 Run Termination and Testing Granulation

Run termination on the MLFN problem was determined by number of passes of the backpropogtion algorithm through the training set. We started each run with 200 runs through the training set, tested and stored the results. Thereafter, we tested every 50 runs through the training set. Each run comprised 10,200 runs through the training set and was then ended.

We examined the course of training under this protocol on about 50 of the runs and satisfied ourselves in each case that: (1) the granularity of testing was sufficient to catch the best or very close to the best network in generalizing capabilities; and (2) that we had performed enough runs through the training set to assure that the run had indeed located the best maximum generalizer for that random seed and network configuration.

6. The IRIS Dataset.

6.1 The Full Dataset.

The IRIS dataset contains three classes and the total number of data instances is 150. Each of the classes is a type of IRIS plant [ELENA 1995, page 35]. This is a simple problem domain and, given a large enough training set, good results are to be expected [*Id*].

6.2 The KNN Benchmark.

The ELENA benchmark classifier for this problem is a KNN classifier run using the Leave-One-Out method with $K = 7$. A KNN (K-Nearest-Neighbors) classifier determines the class of a fitness case by examining the class membership of its K nearest neighbors. Performance is optimized by varying K for any particular problem domain Because of the use of the leave-one-out method, the dataset for deriving the KNN benchmark was 149 instances. The resulting error rate for the KNN classifier was between 0% and 7.3% [ELENA 1995 at 35-6]. We assume that the wide range of the error rate was due to a wide 95% confidence interval on this relatively small sample size (149). In short, a hit rate for CGPS of between 92.7% and 100% on the IRIS data would mean the CGPS result is statistically equivalent to the KNN benchmark.

7. IRIS Generalization Results

7.1 Sparse IRIS CGPS Training Set

Our training set size was 75, which is about half the size of the training set used in deriving the KNN benchmark. Despite the relatively sparse training set, CGPS did very well.

7.2 IRIS CGPS Results

Every IRIS CGPS run yielded at least one individual that generalized at a hit rate that was statistically indistinguishable from the ELENA KNN benchmark.

92% of the IRIS runs resulted in individuals that generalized better than or equal to 95% hits and 17% of the IRIS runs resulted in individuals that generalized better than 96% hits. Furthermore, CGPS found its benchmark-quality solutions quickly. In 78% of IRIS runs, our *GP system had evolved an individual that generalized as well as the KNN benchmark by the end of generation one of training*. In fact, our CGPS system generated at least one new benchmark quality solution approximately once every one half second.

7.3 IRIS MLFN Results.

We did not run the MLFN system on the IRIS data because, in discussions with the MLFN software vendor, he indicated that he had run the IRIS data and that his results were in the range that made them likely to be statistically indistinguishable from our CGPS runs. Given our time constraints and what turned out to be the simplicity of the IRIS problem, we believe it safe to conclude that the MLFN generalization would have been statistically indistinguishable from CGPS generalization on this problem. Accordingly, we did not run the IRIS problem on MLFN.

8. PHONEME Recognition Dataset.

8.1 The Dataset.

The PHONEME recognition database contains two classes of data—nasal vowels (Class 0) and oral vowels (Class 1) from isolated syllables spoken by different speakers. "This database is composed of two classes in 5 dimensions. There are 5404 patterns; 3919 for class zero and 1586 for class one. . . . [T]his number of samples is just sufficient for the database dimension." [ELENA 1995 at 30]. Note that, while 150 data points was sufficient for the IRIS problem, 5404 data points is regarded as 'just sufficient' for the PHONEME Recognition set because of its higher dimensionality and greater difficulty [*Id*].

8.2 The KNN Benchmark

The benchmark on this problem is a KNN classifier with *K* set to 20. The benchmark was derived from 5404 training instances. The mean misclassification rate calculated by the KNN Benchmark is 14.2% [ELENA 1995. Page 31]. Therefore, any CGPS or MLFN run that has Hits that equal or exceed 85.8% has generalized as well as the Benchmark.

9. Generalization Results on the PHONEME Dataset.

9.1 Sparse Training Set for CGPS and MLFN on the PHONEME Dataset.

We gave our system only 100 data points for training and another 100 for testing rather than the 5404 data points used to derive the KNN Benchmark. Thus, we withheld from the CGPS, over 98% of the training instances that were available to the Benchmark. And we did so on a data set where ordinary statistical analysis suggests that our system ought to need those extra training samples to do a good job [ELENA 1995. Page 30-1].

9.2 Relative Generalization Results of CGPS and MLFN on Sparse PHONEME Recognition Dataset.

Both CGPS and the MLFN did respectably at finding individuals that generalized well—many runs produced individuals that were of near benchmark quality despite the sparse training set. Table 3 contain the mean hit rates results for all, the best 25% and the best 10% of PHONEME Recognition runs for both MLFN and CGPS.

Table 3. PHONEME Recognition Results. Mean Hits by ELENA KNN Benchmark, by CGPS and by MLFN systems. All Runs, Best 25% Of Runs, Best 10% Of Runs, and Best Run.

System	All Runs	Best 25%	Best 10%	Best Run
KNN	85.8%	----	----	---
CGPS	77.6%	80.5%	81.2%	85%
MLFN	71.6%	79%	80.9%	82%

By any measure on this problem, CGPS outperformed the MLFN in generalization. The differences in performance are, for the most part statistically significant. To test statistical significance, we performed the test for a hypothesized difference between two means assuming unequal variances. Table 4 sets forth our results. P is the probability that the actual difference between the CGPS and MLFN means is less than the hypothesized difference.

Table 4. PHONEME Recognition Results. Statistical Significance Level of Hypothesized Difference Between the CGPS Mean Hits and The MLFN Mean Hits.

Differences	All Runs	Best 25%	Best 10%
Actual Diff.	6%	1.5%	0.3%
Hypothesized Diff.	5%	0.9%	0.05%
P	0.0141	0.0341	0.0884

Measured in CPU time, CGPS generated a solution that generalized at least 90% as well as the Benchmark figure about once every 1 minute. And, finally, CGPS generated a solution that was almost identical to the Benchmark figure about once every 55 minutes.

10. The GAUSSIAN 3D Dataset

10.1 The Full Dataset.

GAUSSIAN 3D is a three input, two class database that is generated artificially. Class 0 is a the set of points with a normal distribution across the three input axes with zero mean and standard deviation of 1. Class 1 is a similar series of points except that the standard deviation is 2. [ELENA 1995. Page 14.] This is an difficult classification problem because the classes are linearly non-separable and because of overlap between the two classes [Figure 1].

10.2 The Benchmarks.

There are several benchmarks available on this problem:

1. The theoretical Bayes confusion boundary is 21.4%. It is not possible to generalize better than this on this problem [ELENA 1995. Page 16]. Therefore a Hit rate of 78.6% should be the best possible performance of any classifier on these data.
2. ELENA's KNN classifier (K = 35) using the leave-one-out method yields an error on 5000 training samples of 22.2%. Therefore, a hit rate of 77.8% would be competitive with the KNN benchmark [ELENA 1995]
3. A Binary Boltzman Machine and A Learning Vector Quantization system attained results virtually the same as the Bayes minimum but had to use 4,500,000 and 100,000 training samples respectively to attain that result [ELENA 1995].

10.3 Generalization Results On The GAUSSIAN 3D Dataset.

The task we assigned to CGPS and the MFLN on the GAUSSIAN Data set seemed daunting. We gave them only 100 training samples. In short, we withheld fully 98% of the 5000 training samples that were available to the KNN benchmark measure. Figure 1 is a projection of 100 points from the GAUSSIAN 3D data set into two dimensions. CGPS' and the MLFN's jobs were to distinguish the darker diamonds (class 0) from the lighter dashes (class 1).

Again, both CGPS and the MLFN did respectably at finding individuals that generalized well. Table 5 contains the mean hit rates results for all, the best 25% and the best 10% of the GAUSSIAN 3D runs for MLFN and CGPS.

Table 5. GAUSSIAN 3D Results. Mean Hits by ELENA KNN Benchmark, by CGPS and by MLFN systems. All Runs, Best 25% of Runs, Best 10% of Runs and Best Run

System	All Runs	Best 25%	Best 10%	Best Run
KNN	77.8%	----	----	----
CGPS	57.4%	63.9%	67%	72%
MLFN	56.8%	62.4%	64.9%	68%

Once again CGPS outperformed the MLFN in the overall mean and on the upper tail of the generalization distribution. Again, the differences in are statistically significant. Table 6 presents the same types of statistical results for the GAUSSIAN 3D data set as did Table 4 for the PHONEME recognition data.

Table 6. GAUSSIAN 3D Results. Statistical Significance Level of Hypothesized Difference Between the CGPS Mean Hits and The MLFN Mean Hits.

Differences	All Runs	Best 25%	Best 10%
Actual Diff.	0.6%	1.5%	2.14%
Hypothesized Diff.	0.2%	0.5%	1%
P	.0601	.0264%	.0484

Measured in CPU time, CGPS generated an individual 90% as good as the KNN benchmark every 18 minutes.

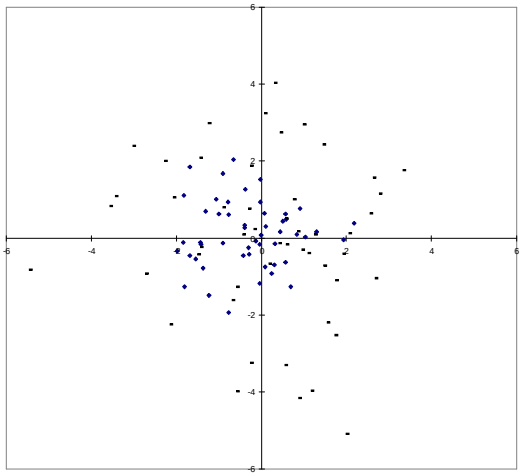

Figure 1. Sparse GAUSSIAN 3D Data Set Projected into Two Dimensions. 100 Data Points.

11. Discussion.

11.1 The Future of the IRIS Problem

The easiest conclusion to draw from our data is that the IRIS data set is probably not a very good test of any machine learning system—it is too easy. CGPS evolved programs by the end of Generation 1 that were only a few instructions long that did extremely well in classifying the IRIS data. The ease of the problem is also indicated by reported MLFN results [Katz 1996]. Of course, a finding that a machine learning system could *not* do well on the IRIS data would be quite a significant finding. Other than that, researchers should probably stop using the IRIS data as being indicative of very much of anything.

11.2 CGPS and Generalization on Sparse Data Sets.

For the most part, our other results speak for themselves. CGPS generated solutions that generalized on the other two problem sets well and quickly. It did so somewhat more effectively than the MLFN used here. CGPS did well even though we used simple functions and made no adjustments to help the system solve the particular problems presented. In short, our solutions were as free from experimenter knowledge of the solution space as we could make them.

CGPS' ability to generalize from sparse data sets was quite consistent. From the time figures noted above, with enough CPU time, CGPS was able to extract good benchmark or near benchmark quality individuals using sparse data. Of course, on the GAUSSIAN 3D and PHONEME problems, additional data was available—so in a sense, it was not necessary to utilize the extra CPU time to generate solutions that generalized well. We could just as well have used larger training sets. But in the real world, there is frequently too little data for the dimensionality of the problem. This suggests that CGPS may be a good choice for classification problems where the available data is sparse.

11.3 Relative Distribution of Generalization Results: MLFN vs. CGPS.

One of the most interesting comparisons between the CGPS and MLFN paradigms comes from examining the distribution of their relative generalization scores over all runs. Figures 2 and 3 show those distributions for the PHONEME and the GAUSSIAN 3D data sets. What jumps out is that MLFN results are spread fairly uniformly over a wide range—a normal or flattened normal distribution. The CGPS results are concentrated very heavily around the mean, have a very short low end tail, and have a long tail in the upper end of the distribution. This long tail is the reason CGPS outperformed MLFN in the high quality solution regions [Tables 3-6]. Given CGPS greater freedom of representation than MLFN, this tightly bunched distribution is, intuitively, the opposite of what we expected.

More fundamentally, the relative uniformity in CGPS results from run to run occurred despite the rather radical set of parameter changes that we made among runs. This suggests that mere repetition of runs with different parameters and random seeds may not a particularly good approach to using CGPS. The same search space appears to keep being explored over and over [*See* Mahoud 1995]. This implies that recent research where information gleaned from one run is used to improve the performance on subsequent runs may well be a *very* important future direction for research [Whigham 1995, Zannoni & Reynolds 1996].

Acknowledgments

We would like to thank the ELENA partners for assembling the ELENA databases and the benchmark data.

We also thank the reviewers and William Langdon, whose questions and comments were very helpful.

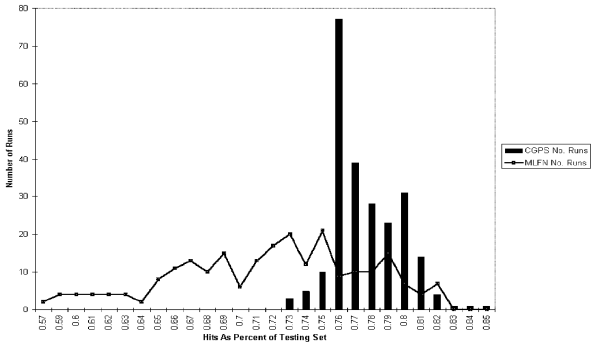

Figure 2. Phoneme Recognition Problem. Comparative Histograms of all CGPS Runs (bars) and MLFN Runs (lines) Sorted by Generalization Performance.

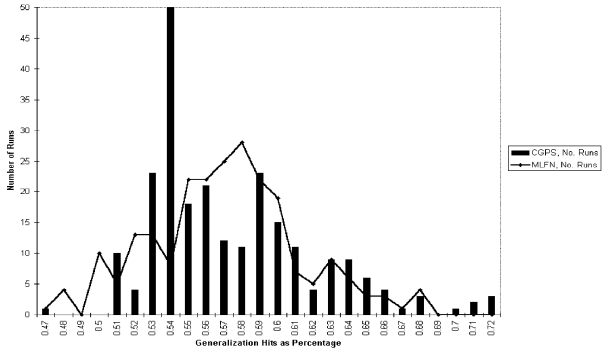

Figure 3. Gaussian 3D Problem. Comparative Histograms of all CGPS Runs (bars) and MLFN Runs (lines) Sorted by Generalization Performance.

Jeffrey Katz of Scientific Consulting, Inc. helped us configure the parameters on his N-Train Neural Network system for benchmarking purposes.

We would also like to acknowledge support from the Ministerium fur Wissenschaft und Forschung des Landes Nordrhein-Westfalen, under grant I-A-4-6037.I.

Bibliography

Crow, E., Davis, F. and Maxfield, M. 1960. *Statistics Manual*, NY, NY. Dover Publications, Inc.

ELENA Partners, The. Jutten, C., Project Coordinator 1995. Esprit Basic Research Project Number 6891, Document Number R3-B1-P.

Katz, J. 1996. Designer and owner of Scientific Consultant Services, Inc., proprietor of N-Train neural network software. Personal communication.

Koza, J. 1992. *Genetic Programming,* Cambridge, MA: MIT Press.

Mahoud, S. 1995, *Niching Methods for Genetic Algorithms*, IlliGAL Technical Report No 95001. University of Illinois. Dept. of General Engineering.

Masters, T. 1995, *Advanced Algorithms for Neural Networks*. NY, NY, John Wiley and Sons Inc. Pages 335-376.

McCormick, D. and Katz, J. 1992. *N-TRAIN Neural Network Development System, Users Manual, V 1.02*.

Nordin, J.P. 1994. A Compiling Genetic Programming System that Directly Manipulates the Machine Code. In *Advances in Genetic Programming*, K. Kinnear, Jr. (ed.), Cambridge MA: MIT Press.

Nordin, J.P., Francone, F. and Banzhaf, W. 1996. Explicitly Defined Introns and Destructive Crossover in Genetic Programming. *Advances in Genetic Programming 2*, K. Kinnear, Jr. (Editor), Cambridge MA: MIT Press.

Nordin, J.P., Banzhaf W. 1995, Complexity Compression and Evolution. In *Proceedings of Sixth International Conference of Genetic Algorithms,* Morgan Kaufmann Publishers, Inc.

Nordin, J.P., Banzhaf, W. 1995a. Evolving Turing Complete Programs for a Register Machine with Self Modifying Code. In, *Proceedings of Sixth International Conference of Genetic Algorithms,* Morgan Kaufmann Publishers, Inc.

Rao, B, Gordon, D. & Spears, W. 1995. For Every Generalization Action, Is There Really an Equal and Opposite Reaction. Analysis of the Conservation Law for Generalization Performance. In Prieditis, A. and Russell, S. (editors). *Proceedings of the Twelfth International Conference on Machine Learning.* San Francisco, CA: Morgan Kaufmann Publishers, Inc. Pages 471-9.

Whigham, P. 1995. Gramatically Based GP. In Rosca, J. (editor). *Proceedings of the Genetic Programming Workshop of the 12th Annual Machine Learning Conference*.

Wolpert, D. H. 1994. *Off-Training Set Error And A Priori Distinctions Between Learning Algorithms*, Technical Report, Santa Fe Institute, Santa Fe, N.M.

Zannoni, E. and Reynolds, R., Extracting Design Information from Genetic Program Using Cultural Algorithms. In *Proceedings for the Fifth Annual Conference on Evolutionary Programming*, 1996.