A Comparison of Genetic Programming and Neural Networks in Medical Data Analysis

Markus Brameier   Wolfgang Banzhaf

Fachbereich Informatik
Universität Dortmund
44221 Dortmund, GERMANY
email: brameier,banzhaf@ls11.informatik.uni-dortmund.de

Abstract

We apply an interpreting variant of linear genetic programming to several diagnosis problems in medicine. We compare our results to results obtained with neural networks and argue that genetic programming is able to show similar performances in classification and generalization even when using a relatively small number of generations. Finally, an efficient algorithm for the elimination of introns in linear genetic programs is presented.

1 Introduction

Genetic programming (GP) has been formulated originally as an evolutionary method for breeding programs using expressions from the functional programming language LISP [5]. We employ a new variant of linear genetic programming (LGP) using sequences of instructions of an imperative programming language. More specifically, the method operates on genetic programs being represented as linear sequences of C instructions. One of the strength of LGP is that introns, i.e. dispensible instructions that do not affect program behavior, can be removed before a genetic program is executed during the fitness calculation. This does not cause any changes to the individuals in the population but only results in an enormous speedup. Introns are also found in biological genomes, where they appear in DNA of eucaryotic cells. It is interesting to note that these natural introns are removed, too, before proteins are synthesized.

In this paper genetic programming is applied to medical data analysis. In particular LGP is tested on 6 diagnosis problems that have been taken from the PROBEN1 benchmark set of real world problems originally established for neural networks [8]. The application is a demonstration of the abilities of genetic programming in data mining, where general mathematical descriptions of regularities in data are to be found without preprocessing. For supervised learning tasks this normally means to find classifiers that generalize from a set of learned data to a set of unknown data.

The main objective of this paper is to show that for the problems discussed genetic programming is able to achieve very similar classification rates and generalization performance than is possible with neural networks.

So far GP has not been used very extensively for medical applications [3]. This stands in contrast to neural networks which are increasingly being seen as an alternative to classical statistical methods in this area. Ripley and Ripley [10] review several neural network techniques in medicine including methods for diagnosis and prognosis tasks, especially survival analysis. Most applications of neural networks to medicine refer to classification tasks.

2 Genetic Programming

Evolutionary algorithms mimic aspects of natural evolution to optimize a solution towards a defined goal. Following Darwin's principle of natural selection differential fitness advantages are exploited in a population to lead to better solutions. Different research subfields of evolutionary algorithms have emerged, such as *genetic algorithms* [4], *evolution strategies* [11], and *evolutionary programming* [2]. In recent years these methods have been applied successfully to a wide spectrum of problem domains, especially in optimization. A general evolutionary algorithm can be summarized as follows:

Algorithm (Evolutionary Algorithm):

1. The population of individual solutions is initiated to a random content.
2. Individuals are selected from the population randomly and are compared with respect to their fitness. The fitness measure defines the problem the algorithm is

expected to solve.

3. Only fitter individuals are modified by the following genetic operations:
   - Identical *reproduction*
   - Exchange of a single unit in an individual at a random position (*mutation*)
   - Exchange of substructures between two individuals (*recombination*)
4. The currently best individual represents the best solution found so far.

A comparatively young and growing research area in this context is *genetic programming* [5]. Genetic programming uses the mechanisms behind natural selection for the evolution of computer programs. The approach has been formulated originally using tree structures as individuals that were represented by variable length LISP S-expressions. The inner nodes of these trees are functions while the leafs are terminals that mean input variables or constants. The operators applied to generate variants, i.e. recombination and mutation, must guarantee syntactic closure during evolution. In other words, no syntactically incorrect programs are allowed to be generated. Figure 1 illustrates the recombination operation in *tree-based* genetic programming.

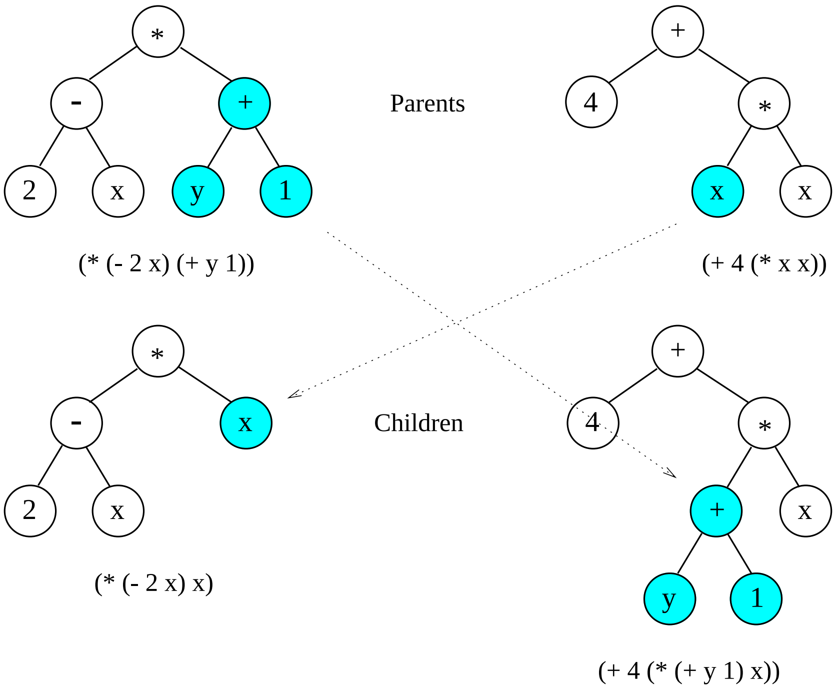

Figure 1: Crossover in tree-based GP

In recent years, the scope of genetic programming has expanded considerably and now includes evolution of linear and graph-like representations of programs as well, in addition to tree representations [1].

2.1 Linear Genetic Programming

In the experiments described below we use *linear* genetic programming, a genetic programming approach with a linear representation of individuals. Its main characteristic in comparison to tree-based genetic programming is that not expressions of a functional programming language (like LISP) but programs of an imperative language (like C) are evolved.

Linear genetic programming has been introduced by Nordin using machine code programs [7, 1]. In AIMGP (*Automatic Induction of Machine code by Genetic Programming*) individuals are directly manipulated as binary machine code in memory and directly executed without passing an interpreter during fitness calculation. This results in a significant speedup compared to interpreting systems. A drawback of AIMGP systems is their restricted portability.

Our LGP system implements an interpreting variant of linear genetic programming. An individual program is represented as a variable length string composed of simple C instructions. Each C instruction is encoded in 4 bytes holding the operation identifier, indices of the participating variables and constant values. This representation allows an efficient recombination of the programs (see figure 2) as well as an efficient interpretation during the fitness calculation. The maximum number of instructions per individual is restricted here to 256. In most cases this has been experienced to be a sufficient size of programs. Thus, one individual uses at most 1K of memory producing a system which is quite memory efficient.

An excerpt of a linear individual program looks like:

```
void ind(v)
  double v[8];
{
  ...

  v[0] = v[5] + 73;
  v[7] = v[0] - 59;       (I)
  if (v[1] > 0)
  if (v[5] > 21)
    v[4] = v[2] * v[1];
  v[2] = v[5] + v[4];     (I)
  v[6] = v[0] * 25;       (I)
  v[6] = v[4] - 4;
  v[1] = sin(v[6]);
  if (v[0] > v[1])        (I)
    v[3] = v[5] * v[5];   (I)
  v[7] = v[6] * 2;
  v[5] = v[7] + 115;      (I)
  if (v[1] <= v[6])
    v[1] = sin(v[7]);
}
```

(I) stands for intron instruction (see section 2.2).

The *instruction set* of the system is composed of arithmetic operations, binary operations, conditional branches and function calls. All these instructions operate on indexed variables

v_i or integer constants c from the *terminal set*. Currently, the maximum number of input variables is restricted to 256 and the constants range from 0 to 255. The general C notation of each instruction class listed below shows that – except for the branches – all instructions implicitly include an assignment to a variable. In tree-based genetic programming, these side-effects have to be incorporated explicitly.

Arithmetic Operations: op $\in \{ + - * / \}$

$v_i = v_j$ op v_k;

$v_i = v_j$ op c;

Binary Operations: op $\in \{ \ll \gg \& \mid \}$

$v_i = v_j$ op v_k;

$v_i = v_j$ op c;

Conditional Branches: op $\in \{ > \leq \}$

if (v_j op v_k)

if (v_j op c)

Function Calls: f $\in$ { sin cos sqrt exp log }

$v_i = \mathrm{f}(v_j)$;

In order to guarantee syntactic closure partially defined operations and functions are protected by returning a constant artificially defined value (here 1) for all undefined inputs. Because binary operations are restricted to integer variables they are not used in the experiments reported here. If a condition is false only the instruction directly following the branch instruction is skipped. This treatment of conditionals has enough expressive power because leaving out or executing an instruction can deactivate preceding effective code or reactivate preceding introns respectively (see section 2.2).

The evolutionary algorithm of the system realizes a simple steady-state 2×2-tournament selection. Figure 2 illustrates the two-point string crossover used in LGP for recombining the two winners of the tournament. A segment of random position and length is selected in each of the two parents for the exchange. If one of the children would exceed the maximum length crossover with equally sized segments is performed. Crossover points only occur *between* instructions but not within. *Inside* the instructions the mutation operation ensures that only instructions from the allowed instruction set are created with valid ranges of operator identifiers, variable indices and constants. Exchanging a variable by a single mutation may have an enormous effect on the program flow (see section 2.2). There is a range in which the integer constants are allowed to change. It is controlled by the *mutation step size* parameter. Only tournament winners are mutated.

2.2 Removing Introns at Runtime

In nature introns denote DNA segments in genes with information that is not expressed in proteins. The existence of introns in eucaryotic genomes may be explained in different ways: (i) Since the information for one gene is often located on different exons (gene parts

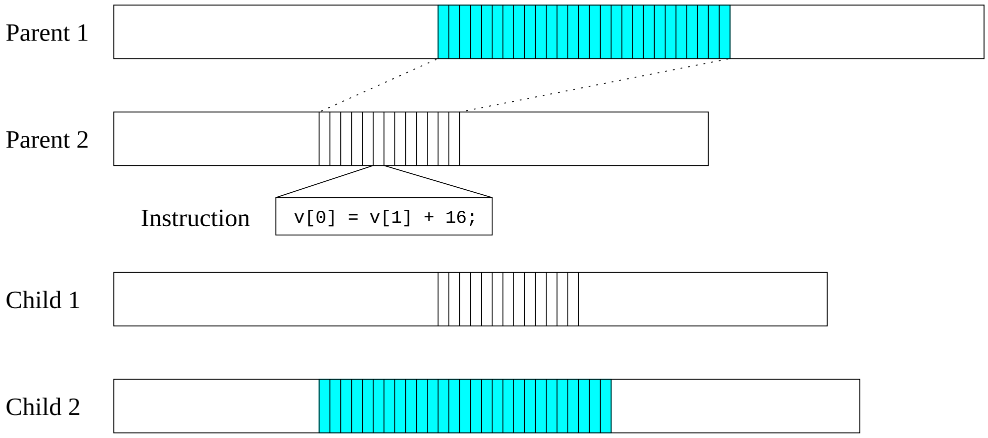

Figure 2: Crossover in linear GP

that *are* expressed) introns may help to reduce the number of destructive recombinations between chromosomes by simply reducing the probability that the recombination points will fall within an exon region [12]. In this way complete protein segments encoded by specific exons are more frequently mixed than interrupted during evolution. (ii) Perhaps even more important for understanding the evolution of higher organisms is the realization that introns allow a new protein to be tested while enabling the organism to retain the original genetic information.

After the DNA is copied the introns are removed from the resulting *messenger*-RNA that actually is participating in gene expression, i.e. protein biosynthesis. A biological reason for the removal of introns might be that the genes are more efficiently translated during protein biosynthesis in this way. Without being in conflict with ancient information held in introns, this might have an advantage, presumably through decoupling of DNA size from direct evolutionary pressure.

In analogy, an *intron* in a genetic program may be defined as a program part without any influence on the calculation of the output for all possible inputs. Other intron definitions common in genetic programming would postulate this to be true only for the fitness cases. Introns act as redundant code segments that protect advantageous building-blocks from being destroyed by the crossover operator.

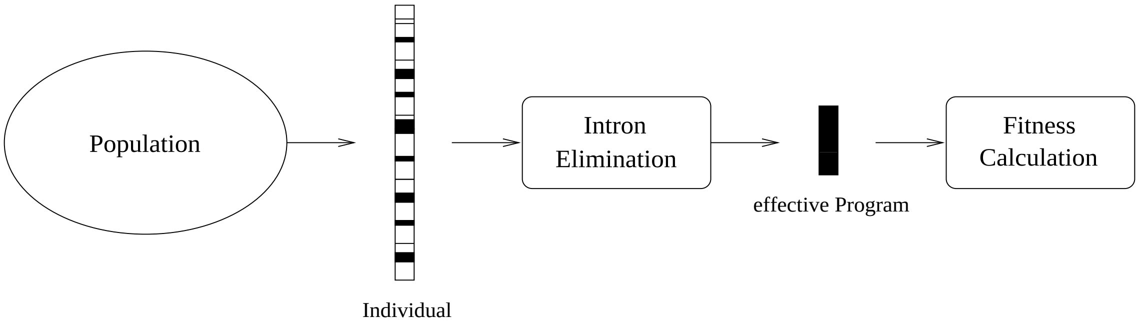

Figure 3: Intron elimination in linear GP

The program structure in linear genetic programming allows introns to be detected and eliminated much easier than in tree-based genetic programming. In our system all *struc-*

tural introns, i.e. instructions that emerge from manipulating variables which have no influence on the output at that position, are removed from a genetic program before evaluating fitness cases. This is done by copying all *effective instructions*, i.e. non-introns, to a temporary program buffer. It does *not* affect the representation of the individuals in the populations (see figure 3).

The following algorithm detects all structural introns in a linear genetic program. In the example program from section 2.1 all instructions marked with an `I` are structural introns provided that the outputs are stored in variable v_0 and v_1.

Algorithm (Intron detection):

1. Let set V contain all variables that have an influence on the result at the current program position.
   Set $V = \{ v_i \mid v_i \text{ is output variable } \}$.
   Start at the last program instruction and move backwards.
2. Mark the next assignment with *destination* variable $v_i \in V$.
   If such an instruction is not found, $\rightarrow$ 5.
3. If the assignment directly follows a branch mark the branch too
   else remove v_i from V.
4. Insert the *operand* variables v_j of new marked instructions in V (if not already contained). $\rightarrow$ 2.
5. Stop. All *non-marked* instructions are introns.

All *marked* instructions are moved to form the effective program. The algorithm needs linear runtime $O(n)$ at worst where n is the maximum length of the genetic program. Actually, detecting and removing introns from a program only requires about the same time as calculating one fitness case. The more fitness cases are calculated the more this computational overhead will pay off.

By not executing the intron instructions during fitness evaluation a large amount of computation time is saved. A good estimate of the overall speedup is the factor

$$\frac{1}{1 - p_{intron}}$$

where p_{intron} denotes the average intron percentage of a genetic program. This percentage figure of all individuals in the population is computed by the algorithm as a side-effect and can be used for further statistical analysis. In most of the runs documented below an average intron rate of about 80% has been observed. As a result, a speedup factor of about 5 can be achieved.

3 Benchmark Problems

Most GP papers today present performance results of a new method only for a very selected number of problems. There is still a lack of a standard set of benchmark problems for genetic programming. Such a set would give the researchers the opportunity for a better comparability of their published methods and results. An appropriate benchmark set should be composed of artificially generated data sets as well as real data sets taken from real problem domains.

The drawback with synthetic benchmark problems is that a simple exact solution is already known a-priori and that the data sets are artificially composed of sufficient information to evolve this solution. A setting like this normally says nothing about the generalization performance of a method and how the results would hold in similar, more realistic domains. In order to make the results obtained with artificial models more realistic and also to improve the generalization ability, noise has to be added in the data generation process. One advantage, however, of artificial data sets compared to real world data is that the characteristics of the data are in most cases exactly known, making it easier to rate problems by difficulty in relation to the tested method.

A good reason, on the other hand, for testing on real data is that it guarantees results that are relevant for at least the tested problem domains. Several data sets from different domains should be tried to increase the confidence that the tested method is not depending on a specific problem or composition of data. In general new techniques in genetic programming are not yet tested thoroughly enough on real world problems.

4 Medical Data Analysis

Genetic programming has not been applied very extensively for analyzing medical data until now. Gray et al. [3] report from an early application of GP in cancer diagnosis where the results had been found to be better than with a neural network. But the dimension of the data sets was rather small compared to problem complexities dealt with here.

In this examination genetic programming is applied to 6 medical problem domains using 3 data sets from each domain. All data are taken from an existing collection of benchmark problems, Proben1 [8], that originally has been established for neural networks and that holds only data sets from real world problems. The results obtained with one of the fastest learning algorithms for feed-forward neural networks (RPROP) accompany the Proben1 benchmark set to serve as a direct comparison with other methods. Comparability and reproducibility of the results is guaranteed by careful documentation of the experiments. Following the benchmarking idea and in order to increase the confidence in our own results with linear genetic programming we have adopted the results for neural networks from [8].

The main objective here in general is to realize the fairest comparison possible between genetic programming and neural networks in medical data analysis and in medical diagnosis. We will show that for all problems discussed the performance of GP in generalization comes very close to or is even better than the results documented for neural networks in [8]. The relatively small number of runs per data set can only give an order of magnitude comparison. In addition the results for both methods might not be the absolutely best because all experiments were run directly on raw data, i.e. data that had not undergone preprocessing.

Table 1 gives a brief description of the diagnosis problems and the diseases that are to be predicted. Diagnosis problems always describe classification tasks that are much more common in medicine than regression problems.

Problem	Diagnosis task
cancer	benign or malignant breast tumor
diabetes	diabetes positive or negative
gene	intron-exon, exon-intron or no boundary in DNA sequence
heart	diameter of a heart vessel is reduced by more than 50% or not
horse	horse with a colic will die, survive or must be killed
thyroid	thyroid hyperfunction, hypofunction or normal function

Table 1: Diagnosis problems

To improve the comparability of results, all PROBEN1 problems are equally represented. Each data set is organized as a sequence of independent example vectors divided into input and output values (supervised learning). All input values are restricted to the continuous range [0,1] except for the gene data set where only -1 or 1 is used. Using a binary *1-of-m output encoding* each output represents one of the m classes distinguished in the definition of a problem and exactly one output takes the value 1 while the others are set to 0.

It is characteristic for medical data that they almost always suffer from unknown attributes. For that reason in most of the data sets missing inputs had to be artificially completed, e.g. 30% in case of the horse data set. In contrast, heart (originally named heartc in [8]) is clean from any missing data.

Table 2 gives an overview of the specific complexity of each problem, i.e. the number of examples, inputs, output classes as well as the number of problem attributes the inputs are subdivided into.

Problem	Attributes	Inputs	Output classes	Examples
cancer	9	9	2	699
diabetes	8	8	2	690
gene	60	120	3	3175
heart	13	35	2	303
horse	20	58	3	364
thyroid	21	21	3	7200

Table 2: Problem complexities

5 Experimental Setup

Genetic Programming

For each data set at least 30 runs have been performed with LGP, all starting with the same configuration but with a different random seed. Table 3 lists the parameter settings used for all problems and data sets. No special parameter adjustments have proven necessary for the different problems.

In all cases the population is subdivided into 10 demes each holding 500 individuals. Migration between demes is organized in a ring topology in that every deme has a fixed successor. After each generation the individuals of every deme are sorted by fitness and a certain percentage of best individuals, determined by the *migration rate*, emigrates into the successor deme where it replaces the worst individuals. Migration rates of 5 to 7 percent have been found to produce good results.

Population size	5000
Number of demes	10
Migration rate	5-7%
Classification error weight	1.0
Maximum number of generations	200
Crossover probability	90%
Mutation probability	90%
Mutation step size for constants	± 5
Maximum program size	256 instructions
Initial maximum program size	25 instructions
Function set	$\{ + - * / \sin \exp \text{if}> \text{if}\leq \}$
Terminal set	$\{0,..,255\} \cup \{v_0,..,v_{n-1}\}$
Random seed	system time

Table 3: Parameter settings (LGP)

For benchmarking reason the partitioning of all PROBEN1 data sets is fixed. The *training set* always includes the first half of the examples from a data set, the next quarter is defined as the *validation set* and the last quarter is the *test set*. For some problems the composition of the training, validation and test set can differ significantly in their representation of the problem space.

The fitness of an individual program is always evaluated over the complete training set. After each generation, the error of the best-so-far individual is calculated using the validation set. From these individuals the one with minimum validation error is tested on the test set once *after* the training is over. Note that this is not necessarily the individual with minimum test error!

Throughout this paper, *fitness* F of an individual program p is calculated as the *mean square error* (MSE) between the predicted output o_{ij}^{pred} and the desired output o_{ij}^{des} for all n training examples and m outputs. Additionally, a *classification error* (CE) is computed as the number of incorrectly classified examples. To guarantee a fair comparison, the *winner-takes-all* classification method has been adopted from [8]. It simply designates the class with the highest output as response class. The classification error is added to the fitness while its influence is controlled by a weight parameter w (see table 3).

$$F(p) = \frac{1}{n \cdot m} \sum_{i=1}^{n} \sum_{j=1}^{m} (o_{ij}^{pred} - o_{ij}^{des})^2 + w \cdot CE = MSE + w \cdot CE$$

Because only classification problems are dealt with in this contribution the *test classification error* characterizing the generalization performance and the generation in which the individual with the minimum validation error appeared, i.e. the *effective training time* or *learning speed*, are of main interests.

Neural Networks

The experimental results in [8] have been achieved using multi-layer perceptrons (MLPs) without shortcut connections. Different numbers of hidden units and hidden layers (one or two) have been tried to find out the best net architecture for each data set. The method applied for training was always RPROP learning [9], a backpropagation variant about as fast as Quickprop but with less ajustments of the parameters necessary. For the RPROP parameter settings and the special network architectures, see [8].

The generalization performance on the test set was computed for that state of the network which had minimum validation error during training. The number of epochs trained until this state denotes the learning speed of neural networks.

6 Results and Comparison

	GP							
Problem	Validation CE			Test CE			effective Generations	
	best	*average*	*stddev*	*best*	*average*	*stddev*	*average*	*stddev*
cancer1	1.71	2.45	0.34	0.57	2.18	0.59	26	24
cancer2	0.57	1.39	0.40	4.02	5.72	0.66	26	25
cancer3	1.71	2.62	0.45	3.45	4.93	0.65	17	11
diabetes1	20.31	22.19	1.09	21.35	23.96	1.42	23	14
diabetes2	21.35	23.21	1.33	25.00	27.85	1.49	28	25
diabetes3	25.52	26.69	0.65	19.27	23.09	1.27	21	15
gene1	7.81	11.16	2.30	9.21	12.97	2.24	77	21
gene2	9.07	12.93	2.30	8.45	11.95	2.15	90	20
gene3	7.18	10.77	2.11	10.09	13.84	2.09	86	14
heart1	7.89	10.53	2.38	18.67	21.12	2.02	17	14
heart2	14.47	18.58	2.39	1.33	7.31	3.31	20	14
heart3	15.79	18.81	1.47	10.67	13.98	2.03	21	18
horse1	28.57	32.40	2.22	23.08	30.55	2.24	18	16
horse2	29.67	34.30	2.65	31.87	36.12	1.95	19	16
horse3	27.47	32.65	1.94	31.87	35.44	1.77	15	14
thyroid1	0.83	1.31	0.34	1.28	1.91	0.42	55	18
thyroid2	1.11	1.62	0.31	1.44	2.31	0.39	64	15
thyroid3	0.89	1.47	0.23	0.89	1.88	0.36	51	14

Table 4: Classification errors (in percent) and learning speed (rounded) of GP

Tables 4 and 5 show the results obtained with genetic programming (GP) and neural networks (NN) respectively. The best and the average classification error (CE) of all GP runs are documented on the validation and test set for each medical data set, together with the standard deviation. Most interesting is a comparison with the test classification error of neural networks, reprinted from [8]. Unfortunately, the classification results on the validation set and the results of the best runs are not specified there.

Our results prove that in general genetic programming is able to reach a quite similar generalization performance as was possible for multi-layer perceptrons using RPROP learning.

	NN			
Problem	Test CE		effective Epochs	
	average	*stddev*	*average*	*stddev*
cancer1	1.38	0.49	95	115
cancer2	4.77	0.94	44	28
cancer3	3.70	0.52	41	17
diabetes1	24.10	1.91	117	83
diabetes2	26.42	2.26	70	26
diabetes3	22.59	2.23	164	85
gene1	16.67	3.75	101	53
gene2	18.41	6.93	250	255
gene3	21.82	7.53	199	163
heart1	20.82	1.47	30	9
heart2	5.13	1.63	18	9
heart3	15.40	3.20	11	5
horse1	29.19	2.62	13	3
horse2	35.86	2.46	18	6
horse3	34.16	2.32	14	5
thyroid1	2.38	0.35	341	280
thyroid2	1.91	0.24	388	246
thyroid3	2.27	0.32	298	223

Table 5: Classification errors (in percent) and learning speed (rounded) of NN

For the gene problem the test classification errors (on average and standard deviation) have been found to be even remarkably better with GP.

For both, the GP and the NN runs considered, one can see that the more difficult a problem proves to be the lower is the resulting learning speed (section 5). However, the number of effective generations shows lower variation with respect to the different problem domains than the number of effective epochs. Average and standard deviation of the learning speed have been found to be much less variable, i.e. independent from the problem domain, with GP here.

In our experiments the learning speed could significantly be reduced by using demes (as described in section 5), without, however, leading to worser results in classification. In the case of the heart problem a comparable series of runs without demes took about 2.4 times longer based on the average effective training time. The main reason for this behavior may be the migration strategy. By migrating, i.e. reproducing, best individuals into different demes of the population, learning accelerates when these individuals are further developed simultaneously in different demes. Koza et al. have already shown [6] that demes do not only reduce the *absolute runtime* if run in parallel but have a positive effect on the *relative training time* (on generation basis) in general.

For some problems, especially diabetes, the best results have emerged without using the conditional branches. Other problems like gene have worked significantly better with branches. This might be due to the fact that if branches are not really necessary for a good solution they promote rather specialized, i.e. less generalizing, solutions.

An advantage of genetic programming is that it does not require any form of specific architecture selection like multi-layer feed-forward neural networks. In contrast to neural networks, GP is not only good at predicting outcomes in medicine but may also provide explanations for the diagnosis by allowing an analysis of the resulting genetic programs. In general, this property makes genetic programming less opaque than neural networks.

7 Future Research

A *standard set of artificial and real benchmark problems* for genetic programming should be a main objective of future research in the genetic programming community. But a set of benchmark problems is not enough to guarantee a better comparability and reproducibility of results published by different researchers. A single parameter that is not published or an ambiguous description can make an experiment irreproducible and may let the results obtained with a method differ greatly. In many contributions either comparisons with other methods are not given at all or experiments with the methods compared to had to be reimplemented first. In order to make a direct comparison of published results easier a set of *benchmarking conventions* has to be defined, along with the benchmark problems. These conventions would describe standard ways of setting up and documenting an experiment, as well as measuring and documenting the results. A step in this direction has been done by Prechelt for neural networks [8].

By using demes in genetic programming we experienced that the best generalization on the validation set is reached long before the final generation. Wasted training time can be saved if runs are stopped earlier. Appropriate *stopping rules* that monitor the progress in fitness and generalization over a period of generations are to be defined in this context.

Finally, better predictions on the medical data might be possible with genetic programming and neural networks by combining the predictions of several good solutions or by applying cross validation methods that allow the validation data being used for training, too.

8 Conclusion

We have reported on linear genetic programming, a genetic programming approach using a linear individual representation. An efficient algorithm for the elimination of introns has been presented. We have argued that this method results in a significant speedup of the evolutionary process.

Linear genetic programming has been applied successfully to a number of classification problems in medicine. We have experimentally shown that genetic programming can obtain performance similar to neural networks, even in a relatively small number of generations. We think that the different benchmark data sets are of sufficient diversity to encourage the use of genetic programming in other real problem domains.

Acknowledgements

This research was supported by the Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 531, project B2.

References

[1] W. Banzhaf, P. Nordin, R. Keller and F. Francone (1998) *Genetic Programming — An Introduction. On the automatic Evolution of Computer Programs and its Application.* dpunkt/Morgan Kaufmann, Heidelberg/San Francisco.

[2] L.J. Fogel, A.J. Owens and M.J. Walsh (1966) *Artificial Intelligence through Simulated Evolution.* Wiley, New York.

[3] H.F. Gray, R.J. Maxwell, I. Martinez-Perez, C. Arus and S. Cerdan (1996) *Genetic Programming for Classification of Brain Tumours from Nuclear Magnetic Resonance Biopsy Spectra.* In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, Stanford University, CA, MIT Press, Cambridge.

[4] J. Holland (1975) *Adaption in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI.

[5] J. Koza (1992) *Genetic Programming.* MIT Press, Cambridge, MA.

[6] J. Koza, D. Andre (1995) *Parallel Genetic Programming on a Network of Transputers.* Stanford University Computer Science Department, Technical Report STAN-TR-CS-95-1542.

[7] P. Nordin (1994) *A Compiling Genetic Programming System that Directly Manipulates the Machine-Code.* In K.E. Kinnear, editor, *Advances in Genetic Programming*, MIT Press, Cambridge.

[8] L. Prechelt (1994) PROBEN1 — *A Set of Neural Network Benchmark Problems and Benchmarking Rules.* Technical Report, University of Karlsruhe.

[9] M. Riedmiller and H. Braun (1993) *A direct adaptive method for faster backpropagation learning: the RPROP algorithm.* In *Proceedings of the IEEE International Conference on Neural Networks*, San Francisco, CA.

[10] B.D. Ripley and R.M.Ripley (1997) *Neural Networks as Statistical Methods in Survival Analysis.* In *Artificial Neural Networks: Prospects for Medicine*, edited by R. Dybowski and V. Grant, Landes Biosciences Publishers.

[11] H.P. Schwefel (1995) *Evolution and Optimum Seeking.* Wiley, New York.

[12] J.D. Watson, N.H. Hopkins, J.W. Roberts, J.A. Steitz and A.M. Weiner (1987) *Molecular Biology of the Gene.* Benjamin/Cummings Publishing Company, Inc.