A Dynamic Fitness Function Applied to Improve the Generalisation when Evolving a Signal Processing Hardware Architecture

Jim Torresen

Department of Informatics, University of Oslo
P.O. Box 1080 Blindern, N-0316 Oslo, Norway
jimtoer@ifi.uio.no
http://www.ifi.uio.no/~jimtoer

Abstract. Evolvable Hardware (EHW) has been proposed as a new method for designing electronic circuits. In this paper it is applied for evolving a prosthetic hand controller. The novel controller architecture is based on digital logic gates. A set of new methods to incrementally evolve the system is described. This includes several different variants of the fitness function being used. By applying the proposed schemes, the generalisation of the system is improved.

1 Introduction

There are many roads into making embedded systems for signal processing. The traditional method is to be running software on a Digital Signal Processor (DSP). A new alternative method is to apply digital logic gates assembled using evolution – named Evolvable Hardware (EHW). There are many applications for signal processing systems. One - implied in this paper, is prosthetic hand control.

To enhance the lives of people who have lost a hand, prosthetic hands have existed for a long time. These are operated by the signals generated by contracting muscles – named electromyography (EMG) signals, in the remaining part of the arm [1]. Presently available systems normally provide only two motions: Open and close hand grip. The systems are based on the user adapting *himself* to a fixed controller. That is, he must train himself to issue muscular motions trigging the wanted action in the prosthetic hand. Long time is often required for rehabilitation.

By using EHW it is possible to make the *controller* itself adapt to each disabled person. The controller is constructed as a pattern classification hardware which maps input patterns to desired actions of the prosthetic hand. Adaptable controllers have been proposed based on neural networks [2]. These require a floating point CPU or a neural network chip. EHW based controllers, on the other hand, use a few layers of digital logic gates for the processing. Thus, a more compact implementation can be provided making it more feasible to be installed inside a prosthetic hand.

Experiments based the EHW approach have already been undertaken by Kajitani et al [3]. The research on adaptable controllers is based on designing a controller providing six different motions in three different degrees of freedom. Such a complex controller could probably only be designed by *adapting* the controller to each dedicated user. It consists of AND gates succeeded by OR gates (Programmable Logic Array). The latter gates are the outputs of the controller, and the controller is evolved as one complete circuit. The simulation indicates a similar performance as artificial neural network but since the EHW controller requires a much smaller hardware it is to be preferred.

One of the main problems in evolving hardware systems seems to be the limitation in the chromosome string length [4, 5]. A long string is normally required for representing a complex system. However, a larger number of generations is required by genetic algorithms (GA) as the string increases. This often makes the search space becoming too large. Thus, work has been undertaken to try to diminish this limitation. Various experiments on speeding up the GA computation have been undertaken – see [6].

Incremental evolution for EHW was first introduced in [7] for a character recognition system. The approach is a divide-and-conquer on the evolution of the EHW system, and thus, named *increased complexity evolution*. It consists of a division of the *problem* domain together with incremental evolution of the hardware system. Evolution is first undertaken individually on a set of basic units. The evolved units are the building blocks used in further evolution of a larger and more complex system. The benefits of applying this scheme is both a *simpler* and *smaller* search space compared to conducting evolution in one single run. The goal is to develop a scheme that could evolve systems for complex real-world applications.

In this paper, it is applied to evolve a prosthetic hand controller circuit. A new EHW architecture as well as how incremental evolution is applied are described. Further, new dynamic fitness functions are introduced. These should improve the generalization performance of gate level EHW and make it a strong alternative to artificial neural networks.

The next two sections introduce the concepts of the evolvable hardware based prosthetic hand controller. Results are given in Section 4 with conclusions in Section 5.

2 Prosthetic Hand Control

The research on adaptable controllers presented in this paper provides control of six different motions in three different degrees of freedom: Open and Close hand, Extension and Flection of wrist, Pronation and Supination of wrist. The data set consists of the same motions as used in earlier work [3], and it has been collected by Dr. Kajitani at National Institute of Advanced Industrial Science and Technology (AIST) in Japan. The classification of the different motions could be undertaken by:

– **Frequency domain:** The EMG input is converted by Fast Fourier Transform (FFT) into a frequency spectrum.
– **Time domain:** The absolute value of the EMG signal is integrated for a certain time.

The latter scheme is used since the amount of computation and information are less than in the former.

The published results on adaptive controllers are usually based on data for non-disabled persons. Since you may observe the hand motions, a good training set can be generated. For the disabled person this is not possible since there is no hand observe. The person would have to by himself distinguish the different motions. Thus, it would be a harder task to get a high performance for such a training set but it will indicate the expected response to be obtainable by the prosthesis user. This kind of training set is applied in this paper. Some of the initial results using this data set can be found in [8].

2.1 Data Set

The collection of the EMG signals are undertaken using three sensors for each channel. The difference in signal amplitude between the two of them, together with using the third as a reference, gave the resulting EMG signal. The absolute value of the EMG signal is integrated for 1 s and the resulting value is coded by *four* bits. To improve the performance of the controller it is beneficial to be using several channels. In these experiments *four* channels were used in total, giving an input vector of 4 x 4 = 16 bits. A subset of the training set input – consisting of preprocessed EMG signals, is given in Fig. 1. For each motion, 10 samples are included.

The *output* vector consists of one binary output for each hand motion, and therefore, the output vector is coded by *six* bits. For each vector only *one* bit is "1". Thus, the data set is collected from a disabled person by considering one motion at a time in the following way:

1. The person contracts muscles corresponding to one of the six motions. A personal computer (PC) samples the EMG signals.
2. The key corresponding to the motion is entered on the keyboard.

For each of the six possible motions, a total of 50 data vectors are collected, resulting in a total of: 6 x 50 = 300 vectors. Further, *two* such sets were made, one to be used for evolution (training) and the others to be used as a separate test set for evaluating the best circuit *after* evolution is finished.

3 An Architecture for Incremental Evolution

In this section, the proposed architecture for the controller is described. This includes the algorithm for undertaking the incremental evolution. This is all based on the principle of *increased complexity evolution*.

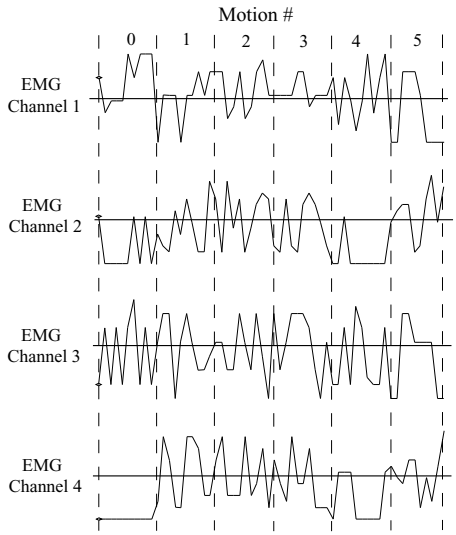

Fig. 1. EMG signals from the training set. 10 samples of data for each motion.

The architecture is illustrated in Fig. 2. It consists of one subsystem for *each* of the six prosthetic motions. In each subsystem, the binary inputs $x_0 \ldots x_{15}$ are processed by a number of different units, starting by the AND-OR unit. This is a layer of AND gates followed by a layer of OR gates. Each gate has the same number of inputs, and the number can be selected to be two, three or four. The outputs of the OR gates are routed to the Selector. This unit selects *which* of these outputs that are to be counted by the succeeding counter. That is, for each new input, the Counter is counting the number of *selected* outputs being "1" from the corresponding AND-OR unit. Finally, the Max Detector outputs which counter – corresponding to *one* specific motion, is having the largest value. Each output from the Max Detector is connected to the corresponding motor in the prosthesis. If the Counter having the *largest* value corresponds to the correct hand motion, the input has been correctly classified. One of the motivations for introducing the selectors is to be able to adjust the *number* of outputs from each AND-OR unit in a flexible way. A scheme, based on using multi-input AND gates together with counters, has been proposed earlier [9]. However, the architecture used in this paper is distinguished by including OR-gates, together with the selector units involving incremental evolution. The incremental evolution of this system can be described by the following steps:

1. **Step 1 evolution.** Evolve the AND-OR unit for each subsystem *separately* one at a time. Apply *all* vectors in the training set for the evolution of each subsystem. There are no interaction among the subsystems at this step, and the fitness is measured on the output of the AND-OR units.
2. **Step 2 evolution.** Assemble the six AND-OR units into one system as seen in Fig. 2. The AND-OR units are now fixed and the *Selectors* are to

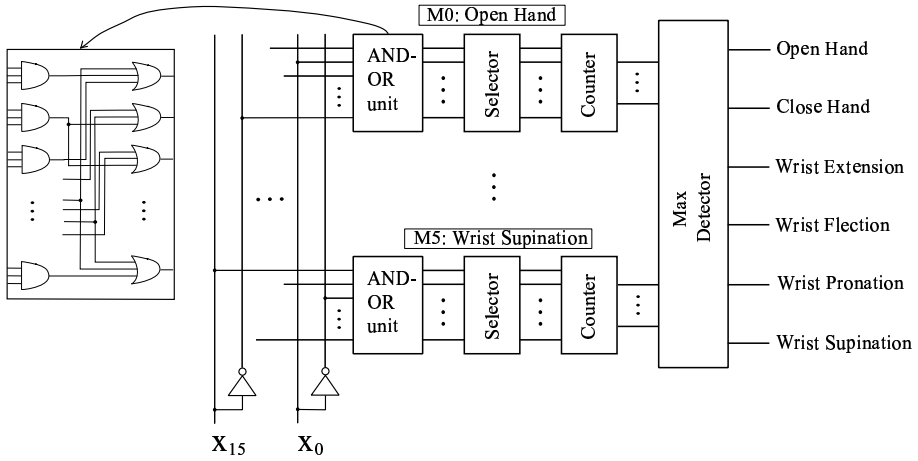

Fig. 2. The digital gate based architecture of the prosthetic hand controller.

be evolved in the assembled system – in one common run. The fitness is measured using the same training set as in step 1 but the evaluation is now on the output of the Max Selector.

3. The system is now ready to be applied in the prosthesis.

In the first step, subsystems are evolved separately, while in the second step these are evolved together. The motivation for evolving separate subsystems – instead of a single system in one operation, is that earlier work has shown that the evolution time can be substantially reduced by this approach [6, 7].

The layers of AND and OR gates in one AND-OR unit consist of 32 gates each. This number has been selected to give a chromosome string of about 1000 bits which has been shown earlier to be appropriate for GA. A larger number would have been beneficial for expressing more complex Boolean functions. However, the search space for GA could easily become too large. For the step 1 evolution, each gate's *inputs* are determined by evolution. The encoding of each gate in the binary chromosome string is as follows:

Input 1 (5 bit)	Input 2 (5 bit)	(Input 3 (5 bit))	(Input 4 (5 bit))

As described in the previous section, the EMG signal input consists of 16 bits. Inverted versions of these are made available on the inputs as well, making up a total of 32 input lines to the gate array. The evolution is based on gate level building blocks. However, since several output bits are used to represent one motion, the signal resolution becomes increased from the two binary levels.

For the step 2 evolution, each line in each selector is represented by *one* bit in the chromosome. This makes a chromosome of 32 x 6 bits= 192 bits. If a bit is "0", the corresponding line should *not* be input to the counter, whereas if the bit "1", the line *should* be input.

3.1 Fitness Measure.

In step 1 evolution, the fitness is measured on all the 32 outputs of each AND-OR unit. As an alternative experiment, we would like to measure the *fitness* on a limited number (16 is here used as an example) of the outputs. That is, each AND-OR unit still has 32 outputs but only 16 are included in the computation of the fitness function, see Fig. 3. The 16 outputs not used are included in the chromosome and have *random* values. That is, their values do not affect the fitness of the circuit. After evolution, all 32 outputs are applied for computing the performance. Since 16 OR gates are used for fitness computation, the "fitness measure" equals 16.

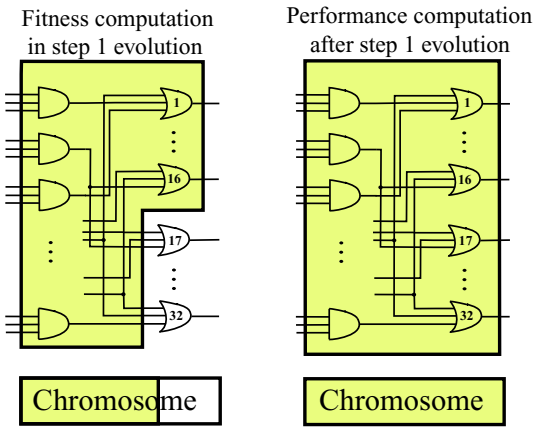

Fig. 3. A "fitness measure" equal to 16.

This could be an interesting approach to improve the generalisation of the circuit. Only the OR gates in the AND-OR unit are "floating" during the evolution since all AND gates may be inputs to the 16 OR gates used by the fitness function. The 16 "floating" OR-gates then provide additional combination of these *trained* AND gates.

3.2 Original Fitness Function

The fitness function is important for the performance of GA in evolving circuits. For the step 1 evolution, the fitness function – applied for each AND-OR unit separately, is as follows for the motion m ($m \in [0, 5]$) unit:

$$F_1(m) = \frac{1}{s} \sum_{j=0}^{50m-1} \sum_{i=1}^{O} x + \sum_{j=50m}^{50m+49} \sum_{i=1}^{O} x + \frac{1}{s} \sum_{j=50m+50}^{P-1} \sum_{i=1}^{O} x \qquad \text{where } x = \begin{cases} 0 \text{ if } y_{i,j} \neq d_{m,j} \\ 1 \text{ if } y_{i,j} = d_{m,j} \end{cases}$$

where $y_{i,j}$ in the computed output of OR gate i and $d_{m,j}$ is the corresponding target value of the training vector j. P is the total number of vectors in the

training set ($P = 300$). As mentioned earlier, each subsystem is trained for one motion (the middle expression of F_1). This includes outputting "0" for input vectors for other motions (the first and last expressions of F_1).

The s is a scaling factor to implicit emphasize on the vectors for the motion the given subsystem is assigned to detect. An appropriate value ($s = 4$) was found after some initial experiments. The O is the number of outputs included in the fitness function and is either 16 or 32 in the following experiments (referred to as "fitness measure" in the previous section).

The fitness function for the step 2 evolution is applied on the complete system and is given as follows:

$$F_2 = \sum_{j=0}^{P-1} x \qquad \text{where } x = \begin{cases} 1 \text{ if } d_{m,j} = 1 \text{ and } m = i \text{ for which } \max_{i=0}^{5}(Counter_i) \\ 0 \text{ else} \end{cases}$$

This fitness function counts the number of training vectors for which the target *output*[1] being "1" *equals* the *id* of the counter having the maximum output.

3.3 Variable Fitness Function in Step 1 Evolution.

Instead of measuring the fitness on a fixed number of output gates in step 1 evolution, the number can be varied throughout the evolution. As depicted in Fig. 4, it is here proposed to *increase* the number of outputs included in the fitness function as the evolution passes on.

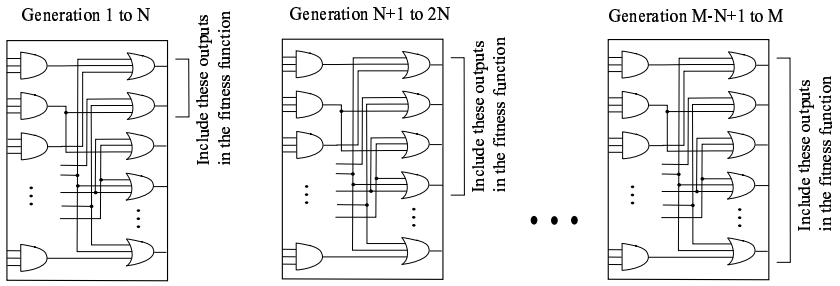

Fig. 4. Variable fitness function throughout the step 1 evolution.

The number of outputs applied by the fitness function is kept constant for N generations, before the number of outputs is *doubled*. This continues for a total of M generations. The benefit of this approach is that evolution is concentrated on a smaller number of outputs in the beginning of the evolution. In a way, it is a variant of the *increased complexity evolution*. The performance of this approach is reported in the results section.

A variant of this approach is by introducing one *more* step of evolution. After finishing the original step 1 evolution with a fitness measure equal to 16

[1] As mentioned earlier only *one* output bit is "1" for each training vector.

as described in Section 3.1, the AND-gates and the OR-gates covered by the fitness function F_1 could be fixed, and a new step of evolution could evolve the 16 "floating" OR gates. This is followed by the already described step 2 evolution of the selectors. No experiments have been undertaken by this approach.

These methods could increase the performance, but not necessarily, since the step 2 evolution already removes the bad performing OR gates. Further, this would reduce the generalisation improvement described in Section 3.1.

3.4 Modified Evolution of Selector Bits.

Instead of evolving *all* the selector bits in step 2 evolution, it is here proposed a scheme where only a *limited* number of them is evolved. Some of the output lines from each AND-OR unit that were floating in step 1 evolution are now kept permanently connected and is not evolved in step 2 evolution. This enhances the generalisation effect of these gates in the complete system. In this way, the test set performance could be improved since overfitting on the training set could be reduced. The chromosomes for the two systems reported in the results section – consisting of fixing 8 selectors at a time, are illustrated in Fig. 5. The dark areas in the figure indicate the part of the chromosome which is ignored during fitness computation. The is due to these selectors being permanently on.

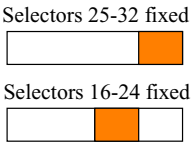

Fig. 5. Step 2 evolution where some of the selector lines are not evolved.

Earlier experiments based on a fitness measure less than 32 did not result in a test set improvement in average (however, the best runs indead leaded to improvement) [8]. This was when all selector lines were evolved in step 2. Due to the possible overfitting, the number of generations had to be kept small. If not all the selector bits are evolved, this problem is reduced. The step 2 evolution can be continued for a larger number of generations without the same risk of overfitting on the training data.

3.5 The GA Simulation

Various experiments were undertaken to find appropriate GA parameters. The ones that seemed to give the best results were selected and fixed for all the experiments. This was necessary due to the large number of experiments that would have been required if GA parameters should be able vary through all the experiments. The preliminary experiments indicated that the parameter setting was not a major critical issue.

The simple GA style – given by Goldberg [10], was applied for the evolution with a population size of 50. For each new generation an entirely new population

of individuals is generated. Elitism is used, thus, the best individuals from each generation are carried over to the next generation. The (single point) crossover rate is 0.8, thus the cloning rate is 0.2. Roulette wheel selection scheme is applied. The mutation rate – the probability of bit inversion for each bit in the binary chromosome string, is 0.01. For some of the following experiments, other parameters have been used, but these are then mentioned in the text.

The proposed architecture fits into most FPGAs. The evolution is undertaken off-line using software simulation. However, since no feed-back connections are used and the number of gates between the input and output is limited, the real performance should equal the simulation. Any spikes could be removed using registers in the circuit.

For each experiment presented in the Section 4, four different runs of GA were performed. Thus, *each* of the four resulting circuits from step 1 evolution is taken to step 2 evolution and evolved for four runs.

4 Results

This section reports the experiments undertaken to search for an optimal configuration of the prosthetic hand controller. They will be targeted at obtaining the best possible performance for the *test* set.

Type of system	# inp/gate	Step 1 evolution			Step 1+2 evolution		
		Min	Max	Avr	Min	Max	Avr
A: Fitness measure 16 (train)	3	63.7	69.7	65.5	71.33	76.33	73.1
A: Fitness measure 16 (test)	3	50.3	60.7	55.7	44	67	55.1
B: Fitness measure 32 (train)	3	51	57.7	53.4	70	76	72.9
B: Fitness measure 32 (test)	3	40	46.7	44.4	45	54.3	50.1
C: Direct evolution (train)	4	56.7	63.3	59.3	-	-	-
C: Direct evolution (test)	4	32.7	43.7	36.6	-	-	-

Table 1. The results of evolving the prosthetic hand controller in different ways.

Table 1 shows the main initial results – in percentage correct classification [8]. These experiments are based on applying the original fitness function defined in Section 3.2. Several different ways of evolving the controller are included. The training set and test set performances are listed on separate lines in the table. Each gate in the AND-OR unit has three or four inputs. The columns beneath "Step 1 evolution" report the performance after only the *first* step of evolution. That is, each subsystem is evolved separately, and afterwards they become assembled to compute their total performance. The "Step 1+2 evolution" columns show the performance when the *selector units* have been evolved too (step 2 of evolution). In average, there is an improvement in the performance for the latter. Thus, the proposed *increased complexity evolution* give rise to improved performances.

In total, the best way of evolving the controller is the one listed first in the table. The circuit evolved with the best *test set* performance obtained 67% correct classification. The circuit had a 60.7% test set performance after step 1

evolution[2]. Thus, the step 2 evolution provides a substantial increase up to 67%. Other circuits didn't perform that well, but the important issue is that it has been shown that the proposed architecture provides the *potential* for achieving high degree of generalization.

A feed-forward neural network was trained and tested with the same data sets. The network consisted of (two weight layers with) 16 inputs, 40 hidden units and 6 outputs. In the best case, a test set performance of 58.8% correct classification was obtained. The training set performance was 88%. Thus, a higher training set performance but a lower test set performance than for the best EHW circuit. This shows that the EHW architecture holds good generalisation properties.

The experiment B is the same as A except that in B all 32 outputs of each AND-OR unit are used to compute the fitness function in the step 1 evolution. In A, each AND-OR unit also has 32 outputs but only 16 are included in the computation of the fitness function as described in Section 3.1. The performance of A in the table for the step 1 evolution is computed by using *all* the 32 outputs. Thus, over 10% better training set as well as the test set performance (in average) are obtained by having 16 outputs "floating" rather than measuring their fitness during the evolution.

Each subsystem is evolved for 10,000 generations each, whereas the step 2 evolution was applied for 100 generation. These numbers were selected after a number of experiments. The circuits evolved with direct evolution (E) were undertaken for 100,000 generations[3]. The training set performance is impressive when thinking of the simple circuit used. Each motion is controlled by a *single* four input OR gate. However, the test set performance is very much lower than what is achieved by the other approaches.

4.1 Variable Fitness Function in Step 1 Evolution.

Table 2 includes the results when the fitness function is changed throughout step 1 evolution. Two and four fitness functions are used which correspond to changing the fitness function one and three times during the evolution, respectively. Except for the number of generations, the GA parameters are the same as for the earlier conducted step 1 evolution experiments. Both the training set as well as the test set performance are improved – for all variants of the fitness function, compared to applying all output lines for fitness computation throughout the evolution. This is seen by comparing Table 2 to B in Table 1. In the first two lines of Table 2, the evolution is run for the same number (10000) of generations as B in Table 1. However, the performance is not better than that obtained in A.

The average test set performance is very high for the step 1 evolution when *four* fitness functions are applied. It is about 10% higher than B in Table 1. However, the test set performance is reduced after step 2 evolution. This could

[2] Evaluated with all 32 outputs of the subsystems.

[3] This is more than six times 10,000 which were used in the other experiments.

Type of system	Total # generations	Step 1 evolution			Step 1+2 evolution		
		Min	Max	Avr	Min	Max	Avr
Two fitness functions (train)	10k	50.3	58.7	55.2	69	72	70.6
Two fitness functions (test)	10k	49.7	54.7	51.6	49	56.3	53.0
Two fitness functions (train)	20k	51.7	61.7	56.0	70	78.3	73.9
Two fitness functions (test)	20k	45.7	55.3	50.6	47	56.3	52.8
Four fitness functions (train)	10k	57	62.3	60.2	67.7	71.3	69.5
Four fitness functions (test)	10k	50.3	57.3	54.5	47.3	56	53.5
Four fitness functions (train)	40k	55.3	64.7	60.9	69.3	77	71.9
Four fitness functions (test)	40k	50	59	55	49.7	59.7	53.4

Table 2. The results of evolving the prosthetic hand controller with a fitness function changing throughout the evolution.

be explained by overfitting on the training set. Further, the total number of generations (10k versus 20k and 40k) do not very much influence the performance.

4.2 Modified Evolution of Selector Bits

The first experiment is based on re-running the step 2 evolution of A, but now with fixing 8 selector bits at a time, and evolving for 500 generations. As seen in Fig. 6 the performance is improved – compared to the original scheme, for all runs when selectors 25-32 were fixed. However, when fixing selectors 17-24 it was less. Each column is the average of the four step 2 runs. Thus, the *Run #* on the x axis is the number of the four different circuits evolved in step 1 evolution.

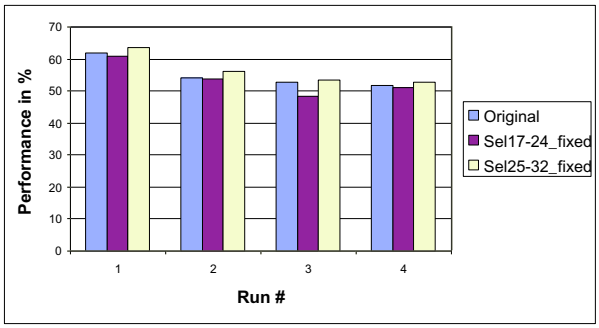

Fig. 6. Results for the test set when fixing 8 selectors permanently on during step 2 evolution (500 generation evolution).

Having selectors 25-32 fixed give an average test set performance of 56.4%. This is slightly better than the best earlier result (55.7%). Thus, in average there is not a large improvement or decrease in performance. This indicate first, that the scheme evolving *all* the selector bits (the original scheme) do not suffer much from overfitting the training data. Second, depending on the random values of the "floating" OR gates, there is a potential of improvement by not evolving all of the selectors.

Evolving for 100 generations provided in average about the same performance as the original scheme. The original scheme, which was evolved only for 100 generations, did not benefit from being evolved for *more* than 100 generations. In another experiments *four* selectors were fixed. However, these experiments gave less performance.

5 Conclusions

In this paper, an EHW architecture for pattern classification including incremental evolution has been introduced. Several different fitness functions have been applied. The results indicate that the proposed schemes are able to improve the generalization performance.

References

1. R.N. Scott and P.A. Parker. Myoelectric prostheses: State of the art. *J. Med. Eng. Technol.*, 12:143–151, 1988.
2. S. Fuji. Development of prosthetic hand using adaptable control method for human characteristics. In *Proc. of Fifth International Conference on Intelligent Autonomous Systems.*, pages 360–367, 1998.
3. I. Kajitani and other. An evolvable hardware chip and its application as a multifunction prosthetic hand controller. In *Proc. of 16th National Conference on Artificial Intelligence (AAAI-99)*, 1999.
4. W-P. Lee et al. Learning complex robot behaviours by evolutionary computing with task decomposition. In Andreas Brink and John Demiris, editors, *Learning Robots: Proc. of 6th European Workshop, EWLR-6 Brighton*. Springer, 1997.
5. X. Yao and T. Higuchi. Promises and challenges of evolvable hardware. In T. Higuchi et al., editors, *Evolvable Systems: From Biology to Hardware. First Int. Conf., ICES 96*. Springer-Verlag, 1997. Lecture Notes in Computer Science, vol. 1259.
6. J. Torresen. Scalable evolvable hardware applied to road image recognition. In *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*. Silicon Valley, USA, July 2000.
7. J. Torresen. A divide-and-conquer approach to evolvable hardware. In M. Sipper et al., editors, *Evolvable Systems: From Biology to Hardware. Second Int. Conf., ICES 98*, pages 57–65. Springer-Verlag, 1998. Lecture Notes in Computer Science, vol. 1478.
8. J. Torresen. Two-step incremental evolution of a digital logic gate based prosthetic hand controller. In *Evolvable Systems: From Biology to Hardware. Fourth Int. Conf., ICES'01)*. Springer-Verlag, 2001. Lecture Notes in Computer Science, vol. 2210.
9. M. Yasunaga et al. Genetic algorithm-based design methodology for pattern recognition hardware. In J. Miller et al., editors, *Evolvable Systems: From Biology to Hardware. Third Int. Conf., ICES 2000*. Springer-Verlag, 2000. Lecture Notes in Computer Science, vol. 1801.
10. D. Goldberg. *Genetic Algorithms in search, optimization, and machine learning.* Addison Wesley, 1989.