Speech Sound Discrimination With Genetic Programming

Markus Conrads[1] Peter Nordin[1 2] and Wolfgang Banzhaf[1]

[1] Dept. of Computer Science, University of Dortmund, Dortmund, Germany
[2] Dacapo AB, Gothenburg, Sweden

Abstract. The question that we investigate in this paper is, whether it is possible for Genetic Programming to extract certain regularities from raw time series data of human speech. We examine whether a genetic programming algorithm can find programs that are able to discriminate certain spoken vowels and consonants. We present evidence that this can indeed be achieved with a surprisingly simple approach that does not need preprocessing. The data we have collected on the system's behavior show that even speaker-independent discrimination is possible with GP.

1 Introduction

Human speech is very resistant to machine learning approaches. The underlying nature of the time-dependent signal with its variable features in many dimensions at the same time, together with the intricate apparatus that has evolved in human speech production and recognition make it one of the most challenging – and at the same time most rewarding – tasks in machine pattern recognition.

The strength of Genetic Programming [Koz92] is its ability to abstract an underlying principle from a finite set of fitness cases. This principle can be considered the essence of the regularities that determine the appearance of concrete fitness cases. Genetic programming tries to extract these regularities from the fitness cases in the form of an algorithm or a computer program. The particular implementation of genetic programming that we use enables us to evolve computer programs at the very lowest level, the machine code level.

Despite their potential complexity it is obvious that regularities rule human speech production, otherwise oral communication would never be possible. These regularities, however, must already reside in the "raw" time signal, i.e. the signal void of any preprocessing, because any method of feature-extraction can only extract what is already there.

The question, then, that we investigate in this paper is whether it it possible for a genetic programming system to extract the underlying regularities of speech data from the raw time signal, without any further preprocessing or transformation.

2 Classification Based on Short-Time Windows

2.1 The Goal

At an abstract level, the task of the genetic programming system is to evolve programs based on the time signal that are able to distinguish certain sounds. Since the task is complicated by the fact that at reasonable sampling rates there is a huge amount of data, it is necessary to impose certain restrictions.

For classification we shall use a short window for the time signal. The size of this window will be 20 ms. At those window sizes the (transient) speech signal can be regarded as approximately stationary. Thus classification of more complex transient signals from longer time periods will not be studied here.

Some of the challenges with our approach will be summarized in the sections to follow.

2.2 Size of the Data Set

When using the time signal as a basis for classification, the system will have to cope with very large data sets. The samples used here have been recorded with a sampling rate of 8 kHz, which means that for each 20 ms window of the signal we have 160 sampled values.

It would be very hard, if not outright impossible, to evolve a GP-program with 160 independent input variables. There would be far too many degrees of freedom, which could easily lead to over-fitting. Besides, the programs would have to become very complex: Reading such a large input and filtering out distinctive features for classification requires a very large amount of instructions.

In many automatic speech recognition systems of today this problem is solved by calculating a feature-vector. Applying a Fourier transform, a filter bank, or other feature extraction measures does not only emphasize the important features, it also drastically reduces the amount of data to be fed into the learning system.

In this study, however, we will not allow any of these measures, as we want to show that genetic programming is able to find discriminating features in the time signal *without* any preprocessing applied to it. We must find a way, though, to reduce the input to the GP-program without reducing or changing the data.

2.3 Scanning the Time Signal

The solution to the problem of the previous section we embarked on is the following. If it is not possible to give the program all data at once, we shall feed them step by step: The time signal is scanned from start to end. The GP program is expected to run through an iteration where in every iteration step it gets only *one* sampled value plus the current position of the sampled value. The program then produces an *output vector* which consists of different components. Additionally, in every iteration step the program gets the output vector of the

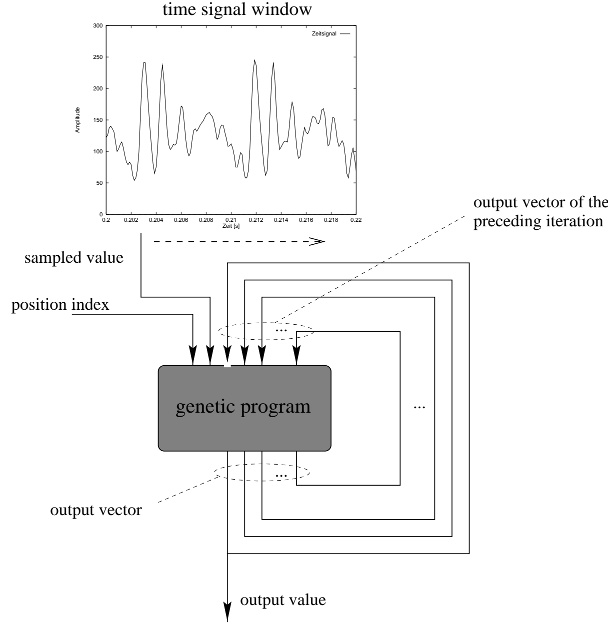

Fig. 1. Data flow when classifying a time signal window

preceding iteration. For the first iteration, we initialize this vector to 0 in each component. Figure 1 illustrates our approach.

The feedback from the output is important. It makes the behavior of the program in iteration n dependent not only on the current input but also on the behavior in the previous step, $n - 1$, which again depends on what happened in the step before and so on. With this method, the output produced in step n depends on *all* sampled values read up to that moment.

Thus, the output of the last iteration is influenced by all sampled values in the scanned time signal window. This output can be used for classification of the signal. For this classification we use only one component of the output vector, leaving it up to the system to use the others for itself on its way to solve the problem.

Relying on such an approach dramatically reduced the number of inputs to the program from a three-digit number to a one-digit number, without applying any preprocessing to the data.

2.4 Calculating the Fitness

In the present study, we only examine classification into two classes. Experiments using more than two classes have been very discouraging so far and will not be considered here.

In order to distinguish two classes $\mathcal{A}$ and $\mathcal{B}$ we determine a target value for each class, $T_{\mathcal{A}}$ and $T_{\mathcal{B}}$. Here we use the values 10,000 and 0. This means that if

the output value of the program after the last iteration is bigger than 5,000, the scanned sample will be classified as class $\mathcal{A}$, otherwise as class $\mathcal{B}$ sample.

Fitness of a program is calculated using the *error*, i.e. the absolute value of difference between the target value and the actual output of the program. The sum of all errors for the presented training instances, the so called *fitness cases*, is the fitness of the individual. Thus, the fitness value is the lower the better, the best possible fitness value is 0.

2.5 Improving Generalization with Stochastic Sampling

When choosing fitness cases, it is important that they represent the problem in the best possible way. The goal is to evolve programs that can *generalize*. It is not very useful to have programs that work good on known data, the fitness cases, but fail on unknown data (overfitting).

In order to avoid overfitting, how can we choose the "right" fitness cases? We will have to choose a sample of fitness cases that is adequate to the problem. This means in practice that we will need many fitness cases. Using a large amount of fitness cases for evaluating the fitness has, on the other hand, a serious disadvantage: It will require a very long time to train the system.

There is a method, however, to maintain good generalization abilities by using a large number of fitness cases while not requiring that much computing power, *stochastic sampling*[1] [NB95c,NB97]: From a large set of fitness cases a small sample is chosen randomly before a fitness evaluation takes place. Fitness evaluation is performed only on the small sample, and hence is less expensive than evaluation on the entire fitness case set.

Thus, no program from the population can really "know" the data it will be confronted with. There are not many fitness cases when fitness is calculated, and memorizing does not help, because at the next evaluation a completely different set is presented. Thus, programs must develop a strategy for how to cope with many *different* possible inputs. They are forced to really *learn* instead of just to memorize. Learning in this context means to perform an *abstraction process*.

Of course, at any moment a relatively good program may be in "bad luck" and be competing against a program that might work worse on most other fitness cases. But seen over many tournaments in the evolutionary process, only those programs will survive that have developed a strategy to cope with more fitness cases than other programs.

A further advantage of this method is that the total amount of fitness cases can be increased arbitrarily, without slowing down the system because the number of fitness cases used for each fitness evaluation remains the same. It can be expected, that a larger set of fitness cases will result in better generalization because the generated GP programs will have to be able to cope with an even larger set of data.

Reports in the literature [GR94] on the failure of this stochastic selection method for training do not conform to our experiences.

[1] A similar method called *Random Subset Selection*, which uses a slightly different way of selecting the fitness cases, has been described in [GR94].

j fitness-case no.	C_{in} input-sample	C_{out} desired output
0	sample of an [a]	10000
1	sample of an [a]	10000
$\vdots$	$\vdots$	$\vdots$
$M/2-1$	sample of an [a]	10000
$M/2$	sample of an [i]	0
$M/2+1$	sample of an [i]	0
$\vdots$	$\vdots$	$\vdots$
$M-1$	sample of an [i]	0

Table 1. Template of a fitness case table used to evolve a program that can distinguish between [a] and [i]. A simple measure to help the system succeed in the classification task is to provide approximately the same amount of fitness cases for each class.

2.6 Applying Stochastic Sampling to Speech Sound Discrimination

The classifiers that we want to evolve will work on a short time window. The speech sounds that the system will need to classify are vowels, fricatives or nasals. These are sounds which can be articulated for a longer time than 20 ms and hence can be regarded as approximately stationary.

If we record a speaker producing an [a] for half a second and sample the signal with a frequency of 8 kHz, we get 4000 sampled values. Therefore we might put the 20 ms window on the time signal at 4000 different positions. Because the signal always changes at least slightly over time, we hence gain 4000 different fitness cases out of one half second of recording. The large amount of possible fitness cases is ideal for stochastic sampling: Calculating all fitness cases would be far too expensive, but their large number will be very helpful for avoiding overfitting. Before fitness of an individual program with respect to a certain sample of fitness cases is evaluated, a window is layed upon the sample at a random position.

In order to ensure that programs do not over-adapt to a certain act of speaking, fitness should be calculated using samples of different acts of speaking. It is advisable to establish a fitness case table for registering which samples are being used and which class they belong to, i.e. what the desired output should be. Table 1 shows a fitness case table used to evolve an "[a]-[i]-classifier" program that can distinguish between [a] and [i]. Fitness case tables for evolving other classifiers can be constructed in similar ways. It is important to note, that the sounds to be classified do not contain significant time-structured features. For instance, for classifying sounds like diphthongs or plausives, the method would have to be modified.

If instead of one-speaker samples those of many speaker are used for the training, it is also possible to reach speaker-independent phone recognition[2]. Samples recorded from many different speakers enable the system to abstract from speaking habits of a single or a few speakers.

3 A Run Example

In this section we shall demonstrate the dynamics of our system with an example run. The phenomena documented here have been observed in many other runs, so the documentation should be considered representative. The behavior of the system applied to other phone discrimination problems is qualitatively similar, though different recognition rates and fitness values will occur in each case.

3.1 The GP-System and its Parameters

We used the *Automatic Induction of Machine Code by Genetic Programming* (AIMGP) system[3] for this problem. AIMGP is a linear GP system that directly manipulates instructions in the machine code. We call it a "linear" system because the representation of the evolved programs is instruction-after-instruction, and therefore linear as opposed to the hierarchical, tree-based structure in most common GP systems.[4] Theoretically, any GP system could be used for this problem, and there are already many [BNKF98]. The problem addressed here, however, is quite complex and requires a lot of computing power. The advantage of AIMGP compared to other GP systems is its speed (at least a factor of ten compared to other existing GP systems) and it is therefore better suited for the challenge of this task. In addition, the representation of individuals is very efficient in AIMGP, giving us the opportunity to run large populations (10,000 individuals) on a single processor system (SPARC), which turned out to be necessary to solve this problem [Nor97].

The parameter settings are shown in Table 2. The choices shown have generated good results, although a parameter optimization was not done. Our goal was not to succeed with parameter tweaking, but in principle.

3.2 The Fitness Case Table

In our example run we want to evolve a program that can distinguish between [a] and [i]. The fitness case table consists of two samples of each vowel, spoken by the same speaker (see Table 3). So, the system is set to develop a speaker-dependent classifier.

---

[2] Phones are subunits of phonemes.

[3] formerly known as CGPS [Nor94,NB95b]

[4] The linear representation seems to have merits of its own, not only regarding the speedup due to directly executing machine code instructions. [Nor97]

Parameter	Setting
population size:	10 000
selection:	tournament
tournament size:	4 (2 x 2)
maximum number of tournaments:	5 000 000
mutation-frequency (terminals):	20 %
mutation-frequency (instructions):	35 %
crossover-frequency:	100 %
maximum program size:	256 instructions
maximum initial size:	30 instructions
maximum initial value of terminal numbers:	100
number of input-registers:	3
number of output(memory)-registers:	1
number of ADFs:	0
instruction set:	addition subtraction multiplication left shift right shift bitwise logical OR bitwise logical AND bitwise logical XOR
termination-criterion:	exceeding the maximum number of tournaments

Table 2. The Koza tableau of parameter settings for AIMGP

Sample	Target Value
sample 1 of an [a]	10000
sample 2 of an [a]	10000
sample 1 of an [i]	0
sample 2 of an [i]	0

Table 3. Fitness case table for developing a program that can distinguish between [a] and [i]. All recordings have been taken from the same speaker.

3.3 Fitness Development

In Figure 2 the fitness development is shown for evolving an [a]-[i]-classifier. Due to the fitness value being calculated as the sum of errors (see section 2.4), the most interesting fitness value is 20,000. We call it the *critical value*. Caused by the structure of the fitness cases, this value can be reached by programs that have a constant output of 5,000, no matter what the input might be. This trivial “solution” leads to a recognition-rate of 50% in a two-class discrimination problem.

As soon as the mean fitness falls below the critical value the ability for discriminating the two classes is indicated. For a program to fall below the critical value is far from trivial. The critical value is a local optimum, because other programs that produce a non-constant output might easily be wrong and go extinct. For this reason it is important to set the population size large enough, so that some of the programs with non-constant output are still able to survive. These programs might eventually breed later on and escape the local optimum.

Therefore, the progression of the mean fitness during the first 1,000,000 tournaments in Figure 2 is particularly interesting. One can see it falling very fast to the critical value, but not falling below. For about 400,000 tournaments the mean fitness stagnates at the critical value. Afterwards it suddenly increases again, reaches its maximum at 600,000 tournaments and then descends, falling below the critical value at about 850,000 tournaments. Fitness keeps decreasing, until it reaches a value of about 5,000.

How can this progression be explained? At first, the local optimum is quickly found. Then, for about 400,000 tournaments, nothing seems to happen if we observe mean fitness only. But if we observe fitness of the best program, it becomes apparent that something is going on indeed. Best fitness is considerably lower than mean fitness and heavy fluctuations are observable. Recall that the system works with stochastic sampling. The same individual may have very different fitness-values, depending on which fitness case is chosen. Especially in the first phase of the evolutionary process this can lead to heavy fluctuations in fitness when observing a special individual. Note also, that the best individual is not always the same, so the fact that it consistently remains below the critical value (from the outset) does not mean that one individual has already found a solution.

But why is it that the mean fitness increases again sharply? Is there something wrong with the process, is the population getting worse? No, just the opposite. A raise in average fitness indicates that the system is about to discover or already has discovered a solution. This solution, however, is not very robust. It can be destroyed by crossover or mutation events, and also due to bad-luck resulting from stochastic sampling. At this point, a look at the best fitness reveals that it is not changing that much anymore during this period, giving an additional hint that a solution has been found already. As evolution continues, the individuals that have found the discriminating features are busy breeding and becoming more robust. They will finally take over the whole genetic pool.

3.4 Development of Program Length

In Figure 2 we depict the length development of programs over the same time period as that of Figure 2. One can see immediately that length increases dramatically at about 1 million tournaments. The limit for program length is set to 256. This limit is reached by both the average program length and the length of the best individual after about 2 million tournaments.

The interesting point about this dynamics is that the dramatic grow of program length happens at the time when fitness decreases. This means that long

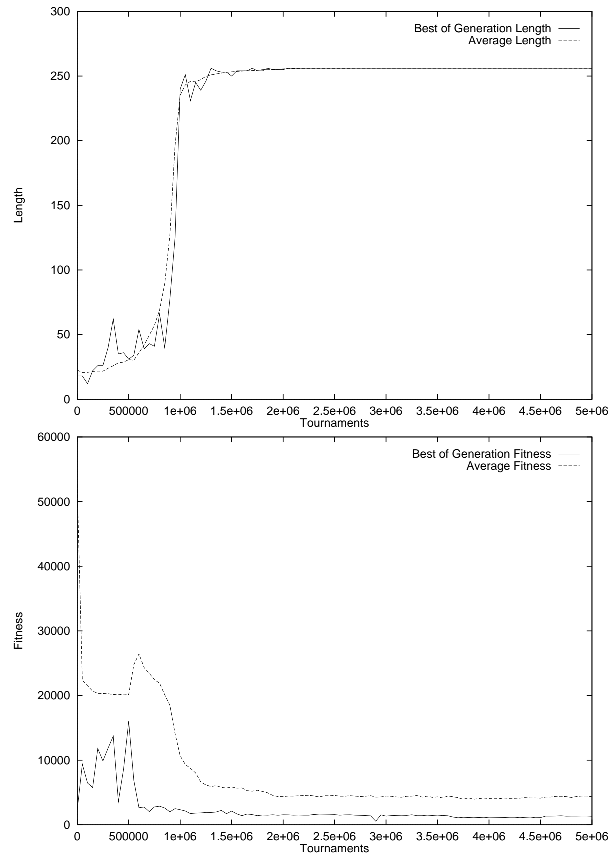

Fig. 2. The development of the program lengths (above) compared with the fitness development (below) when evolving a [a]-[i]-classifier. The x-axis gives the amount of tournaments performed. One tournament is equivalent to four fitness evaluations, 5000 tournaments are equivalent to one generation.

programs have an advantage compared to short ones. We might want to conclude that longer programs are more resistant against destructive mutation or crossover as we have pointed out elsewhere [NB95a,BNKF98] in more details.

3.5 Performance of the Evolved Individual

As a measure of the performance of the evolved individual we calculate the classification rate: Before the run, ten samples of each vowel had been recorded by the same speaker. Two samples have been used for evolving the classifier already. The eight samples left are unknown to the system. These samples are now used to calculate the classification rate of the best-of-run program. Each sample is about 0.5 s long which means that we can lay about 4000 time windows

of 20 ms upon this sample. So, having 8 samples per vowel, we obtain 64 000 *validation cases*.

We calculate the recognition rate by determining the percentage of validation cases on which the classifier made the correct decision. The best individual of the run documented in this section had a recognition rate of 98.8 %. When tested on unknown samples spoken by different speakers, the program still had a recognition rate of 93.7 %. This is quite surprising, given the fact that the program had been evolved based on data taken from one speaker only. Thus, there is strong evidence for a very good generalization ability of our system.

3.6 Behavior of the Evolved Program

Because the performance of the evolved program is relatively good, it would be interesting to know what it actually does. When looking at the program code, though, one quickly discovers that it is not easy to understand. Even after introns, i.e. code segments that do not affect the output, are removed, the remaining code still consists of about 60 instructions. When the program is converted into closed expressions the result is also not very intuitive. After all, GP develops programs by evolution, not by intuition.

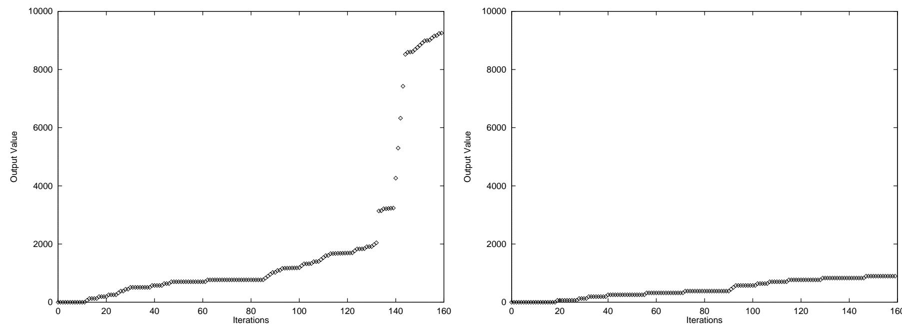

Fig. 3. The dynamics of the output value of an [a]-[i]-classifier working on a sample of an [a] (left side) resp. an [i] (right side).

We therefore approach this problem empirically by studying program behavior. Figure 3 shows the dynamics of the output-value when the classifier scans an [a] and an [i], respectively. Each dot represents the output-value after one of the 160 necessary calls during scanning a 20 ms time window. For an [a], the target-value is 10 000 (case A), for an [i] it is 0 (case B).

In both figures one can see a monotone increase of the output value. However, there is a sudden step in case A from 2000 up to 8000. After this step the value is increasing steeper than before. In case B no such step can be observed.

This transition occurred in all evolved classifiers we have investigated. Sometimes there was even more than one step. The time of the transition is not the

same for all input data. The transition may be delayed, depending on the position of the time-window. Probably this avoids a dependence on a phase-lag caused by the position of the time-window.

Some hints how the program might work can be obtained from *linear digital filters*. These filters work in a similar fashion as the evolved classifiers. The section 3.8 will discuss this topic in more detail.

3.7 An Undiscovered Bug

When looking at the closed expressions which evolved, we made a very astonishing discovery: Only one of the memory-registers (originally 6), intended to be used for feedback of the output (see section 2.3), was employed by the system! The only memory register used was the one that simultaneously served as single output register. That register was used subsequently for calculating the fitness of the individual program, after all 160 samples had passed through it.

Unfortunately, the reason for this parsimonious behavior of programs was a bug in our system! All memory registers (except the one also serving as output) were erased or at least changed in some way between calls and were therefore of no use in feedback.

This bug seems amazingly severe. Instead of the seven potentially independent feedback signals that we intended to provide originally, the system had only one. Although the results presented here have been produced by this unintentionally weak system, they are pretty good. The system had learned to cope with this weakness and did not use the additional registers in feedback, for their lack of stability. Instead, they were put to use as additional calculation registers for storage of provisional results *within* the program.

This whole episode can be seen as another example for the robustness of the GP paradigm, which has been observed by other researchers before. Kinnear notes on this topic in [Kin94]:

> *"Few of us have experience with truly robust computer algorithms or systems. Genetic Programming is such a system, and through the same power that allows it to search out every loophole in a fitness test it can also sometimes manage to evolve correct solutions to problems despite amazingly severe bugs."*

3.8 Linear Digital Filters and Evolved Classifiers

How Do Linear Digital Filters Work? Digital filters work by extracting certain frequencies from the time signal: A linear digital filter computes a sequence of output-points y_n from a sequence of input-points x_k, using the formula (from [PFTV88]):

$$y_n = \sum_{k=0}^{M} c_k x_{n-k} + \sum_{j=1}^{N} d_j y_{n-j} \tag{1}$$

The $M+1$ coefficients c_k and the N coefficients d_j are fixed in advance and will determine the filter response. The filter produces a new output using the current input and the M earlier input-values before, and also using the N outputs produced earlier. If N is 0 the filter is called non-recursive or *finite impulse response* (FIR)-filter and the former output does not affect the current output. If $N \neq 0$ the filter is called recursive or *infinite impulse response* (IIR)-filter.[5]

What is the use of these filters? They are used to extract certain frequencies in a signal and to dampen others. This behavior is expressed by the *filter response function*. For linear digital filters it reads [PFTV88]:

$$\mathcal{H}(f) = \frac{\sum_{k=0}^{M} c_k e^{-2\pi i k (f \Delta)}}{1 - \sum_{j=1}^{N} d_j e^{-2\pi i j (f \Delta)}} \tag{2}$$

With this formula one can calculate the filter response if the c and d coefficients are given. In order to *design* a filter, however, one must go the other way: The c's and d's are to be determined for a desired filter-response. For further information on filter design, the reader is referred to [HP91,MK93].

Are The Evolved Programs Digital Filters? When looking at the algorithm for digital filters, they appear similar to our GP-approach for classifying speech data. In an IIR-filter there is a *feedback* of the output. The N formerly produced outputs are being used to produce the current output. The data-flow is very similar to the one shown in Figure 1. There, too, several output-values are being fed back, although the output-values are all from the same iteration.

The GP system, however, certainly does not evolve a linear response. Instead, it uses bit- and shift-operations plus multiplication operations all of which result in non-linear behavior. Irrespective of the nature of the filter, formula (1) suggests that GP-evolved programs might be able to extract features from a time signal in quite a similar way. Although linear digital filters provide a hint, the resulting programs are much more complicated than a simple linear digital filter.

4 Experimental Results

4.1 The Quality of the Testing-Data

For the experiments documented here sound-samples of several speakers have been recorded. The recording was done with the standard sound-device of a SPARCstation 10. Samples are in 8 kHz μ-law format. They have been converted to a linear 8 bit format. Except for this conversion the data had not been changed in any way. The recordings are not free from noise. Although there are no background voices on the recordings, the fan is audible to the human ear.

[5] Infinite impulse response means, that a filter of this kind may have an response that extends indefinitely (due to the feedback of the output). In contrast, finite impulse response filters have a definite end to their response if there is no input-signal anymore.

In summary, the quality of the recordings is not very high. With a sampling rate of 8 kHz only signal frequencies below 4 kHz can be played back reliably. For certain speech sounds this is definitely not enough. One cannot, for instance, distinguish between [s] and [f] when talking on the phone, a transmission of comparable quality.

4.2 Speaker-Dependent Vowel Discrimination

For vowel discrimination we used the vowels [a], [e], [i], [o] and [u]. For each pair of vowels we evolved classifiers, all in all ten different types of classifiers. In the speaker-dependent case all recordings used to train the system and to test its performance were spoken by the same person.

For determining the recognition rate we used the eight unknown samples of each of the two vowels the classifier is able to distinguish. Each classifier scans a time window of 20 ms. This window is shifted over the entire sample. In each iteration the window starts 0.12 ms later, which corresponds to exactly one sampled value at a sampling rate of 8 kHz. The number of hits, i.e. correct classification answers, is counted and a percentage is computed. Two samples of different classes form a pair. The hit-rate of a pair is calculated as the mean of the hit-rates of each sample in the pair. The longer sample is cut off in order to have an equal number of chances to classify correctly.

For each classifier we get eight pairs. The classification rate is computed as the mean of the hit-rates of all eight pairs. Table 4 shows the recognition rates for all possible vowel combinations. For each vowel combination five GP runs have been done. Table 4 shows the best value that could be reached in these runs.

	[a]	[e]	[i]	[o]	[u]
[a]	—	93.1	99.1	75.0	50.0
[e]	—	—	85.2	74.0	79.1
[i]	—	—	—	89.4	76.1
[o]	—	—	—	—	75.4
[u]	—	—	—	—	—

Table 4. The recognition rates for vowel classification in the speaker-dependent case (in %)

Looking at these values it becomes apparent, that there are obviously some combinations that make it easier to classify than others. These combinations are [a] and [e] and also [a] and [i], where recognition rates of more than 90 % could be obtained (for [a] and [i] we even obtained over 99 %, which can be considered perfect). For all other combinations we have recognition rates of over 70 %. An exception is the combination [a] and [u], where no solution could be found.

We think these results may even be improved by doing more runs or optimizing GP run parameters.

4.3 Speaker-Independent Vowel Discrimination

As said earlier, in order to evolve a classificator that can discriminate vowels irrespective of the speaker, it is necessary to have training data spoken by as many people as possible. For this test series we had training and testing data spoken by six persons. For reliable speaker-independence this is, of course not enough diversity. However, our aim was to show that speaker independence is possible at all. With six different speakers, the system will already have to cope with a lot of in-class-variance.

The recognition rates were determined in the same way as before. The only difference was that training and testing samples have been recorded from six *different* speakers.

Table 5 shows the best recognition rates that could be obtained. It is somewhat surprising that the combinations with best results are not always the same as in the speaker-dependent case. [a] and [o], as well as [a] and [u] could be discriminated very well in this case although this was not possible in the speaker-dependent case. On the other hand results for [a] and [e] as well as [e] and [i] are worse.

	[a]	[e]	[i]	[o]	[u]
[a]	—	72.2	92.2	88.6	96.4
[e]	—	—	78.9	61.3	69.7
[i]	—	—	—	81.1	65.2
[o]	—	—	—	—	60.0
[u]	—	—	—	—	—

Table 5. The recognition rates for vowel discrimination in the speaker-independent case (in %)

Speaker-independent vowel discrimination is more difficult, because the in-class variance is much higher. Therefore, probably more runs would have been necessary to reach the same level of recognition rates. However, it is clear already from these limited results that distinctive features could be found by GP that were exploited for discrimination.

4.4 Discriminating Voiced and Unvoiced Fricatives

In the third task we examined, discrimination between voiced and unvoiced fricatives there are now more than one phones in each class. In this test series there are three each; the voiced fricatives [z], [ʒ] and [v], and their unvoiced

pendants [s], [ʃ] and [f]. Three phones in each class mean a still larger in-class variance than before.

The recognition rates can be read from Table 6. One can see that the value for all combinations with [ʒ] is at about 60 %, while the rates are much higher in all other combinations. The reason could be that [ʒ] differs too much from [z] and [v]. Recall that the recordings have been sampled with 8 kHz, a quality that makes it difficult even for humans to distinguish between certain sounds. Signals that contain high frequencies loose information and fricatives have high frequency formants.

	[z]	[ʒ]	[v]
[s]	90.4	59.6	81.6
[ʃ]	90.6	59.5	81.2
[f]	90.5	60.0	81.3

Table 6. The recognition rates classification of voiced and unvoiced fricatives (in %). Only one classifier used.

In the other cases the GP system has found a way to separate the classes. Recognition rates of 80 to 90 % could be obtained.

4.5 Discriminating Nasal and Oral Sounds

Like in the discrimination task of voiced versus unvoiced sounds this problem has several sounds in one class. In our test series we evolved programs that can discriminate the nasal sounds [n] and [m] from the vowels [a] and [e]. This is another example of large in-class variance, since in the sections 4.2 and 4.3 the vowels [a] and [e] were *discriminated*, while here they have to fall into the same class.

Table 7 shows our results. Again, the system was able to find distinctive features that can be applied to all different phones in the two classes.

	[a]	[e]
[n]	89.3	78.8
[m]	97.0	86.4

Table 7. The recognition rates for classification of oral versus nasal sounds (in %). Only one classifier used.

5 Conclusions

Looking at our experimental results, the most important conclusion is that in every case it was possible for the GP system to evolve programs able to filter out certain discriminating features from the time signal. Even if in some cases the recognition rates were not optimal, there was always a tendency toward discrimination.

Thus, the question that we started with; Is it possible for genetic programming to extract an underlying law from the raw time signal in order to make a classification? can be positively answered. Indeed, the system has extracted regularities, regularities at least, that underlie a large part of the fitness cases, and that are helpful for discriminating the two classes.

In the speech recognition literature, it is often stated that the time signal would not be suitable for classification. It would not contain the needed features in an easily accessible way. When looking at the time signal one can indeed be amazed, how similar signals of different phones can look in some cases, and how different signals of the same phone can look in another time window. But this is clearly no limitation to the GP system.

What is particularly interesting in the GP system is the way, in which it classifies. Most methods used in speech recognition (or more general pattern recognition) perform some kind of comparison of a feature-vector with some reference-vector. Nothing like that is happening here. The evolved programs are (after removing the introns) much too short to memorize reference vectors. They *have to* use the regularities residing in the data to discriminate classes.

Another interesting aspect of this approach is, that the evolved classifiers are very fast, for various reasons:

- No additional preprocessing is needed;
- The programs are already represented in machine code;
- Introns[6] (code that does not effect the output) can be removed from the final program, resulting in a speedup of approximately a factor of 5.

It is certainly too early to say how far this approach might lead, but we see at least some potential for applications to be developed along these lines. The evolution of machine code results in very rapidly executing code that can - by virtue of being the native language of the CPU - perform well even on weak platforms.

[6] In chapter 7 of [BNKF98] further information on the emergence of introns in GP can be found.

ACKNOWLEDGMENT

Support has been provided by the DFG (Deutsche Forschungsgemeinschaft), under grant Ba 1042/5-1.

References

[BNKF98] Wolfgang Banzhaf, Peter Nordin, Robert Keller, and Frank D. Francone. *Genetic Programming — An Introduction.* dpunkt/Morgan Kaufmann, Heidelberg/San Francisco, 1998.

[GR94] Chris Gathercole and Peter Ross. Some training subset selection for supervised learning in genetic programming. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature III*, volume 866 of *Lecture Notes in Computer Science*, pages 312–321, Jerusalem, 9-14 October 1994. Springer-Verlag, Berlin, Germany.

[HP91] Richard A. Haddad and Thomas W. Parsons. *Digital signal processing. Theory, applications, and hardware.* Electrical engineering, communications, and signal processing series. W H Freeman, New York, NY, 1991.

[Kin94] K. E. Kinnear, editor. *Advances in Genetic Programming.* MIT Press, Cambridge, MA, 1994.

[Koz92] John R. Koza. *Genetic Programming – On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, 1992.

[MK93] Sanjit K. Mitra and James F. Kaiser, editors. *Handbook for digital signal processing.* A Wiley-Interscience publication. Wiley, New York, 1993.

[NB95a] Peter Nordin and Wolfgang Banzhaf. Complexity compression and evolution. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 310–317, Pittsburgh, PA, 15-19 July 1995. Morgan Kaufmann, San Francisco, CA.

[NB95b] Peter Nordin and Wolfgang Banzhaf. Evolving Turing-complete programs for a register machine with self-modifying code. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 318–325, Pittsburgh, PA, 15-19 July 1995. Morgan Kaufmann, San Francisco, CA.

[NB95c] Peter Nordin and Wolfgang Banzhaf. Genetic programming controlling a miniature robot. In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 61–67, MIT, Cambridge, MA, 10–12 November 1995. AAAI, Menlo Park, CA.

[NB97] Peter Nordin and Wolfgang Banzhaf. An on-line method to evolve behavior and to control a miniature robot in real time with genetic programming. *Adaptive Behavior*, 26:107 – 140, 1997.

[Nor94] Peter Nordin. A compiling genetic programming system that directly manipulates the machine code. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, Cambridge, MA., 1994.

[Nor97] Peter Nordin. *Evolutionary Program Induction of Binary Machine Code and its Applications.* PhD thesis, University of Dortmund, M"unster, 1997.

[PFTV88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C – The Art of Scientific Computing.* Cambridge University Press, Cambridge, 1988.