19

Automated Parameter Tuning for Interpretation of Synthetic Images

David J. Montana

We have been developing an expert system that will automatically detect, characterize, associate and classify passive sonar signals. This system consists of a few different modules, each of which contains a large number of parameters. The importance of selecting these parameters optimally has led us to develop methods of automatically tuning these parameters which are much better than manual techniques. These automated methods have succeeded largely because they allow the system to learn (i.e., improve with experience) and adapt (i.e., change its behavior in response to changing conditions). In this chapter we describe three different automated tuning methods for three different modules of the expert system. The first involves selecting the thresholds of detection rules to yield optimal performance; the second is optimization of the parameters in an algorithm for detection and tracking of multiple signals in an image; the third is optimal selection of weights in a neural network used for classification. In all three approaches, genetic algorithms play a central role.

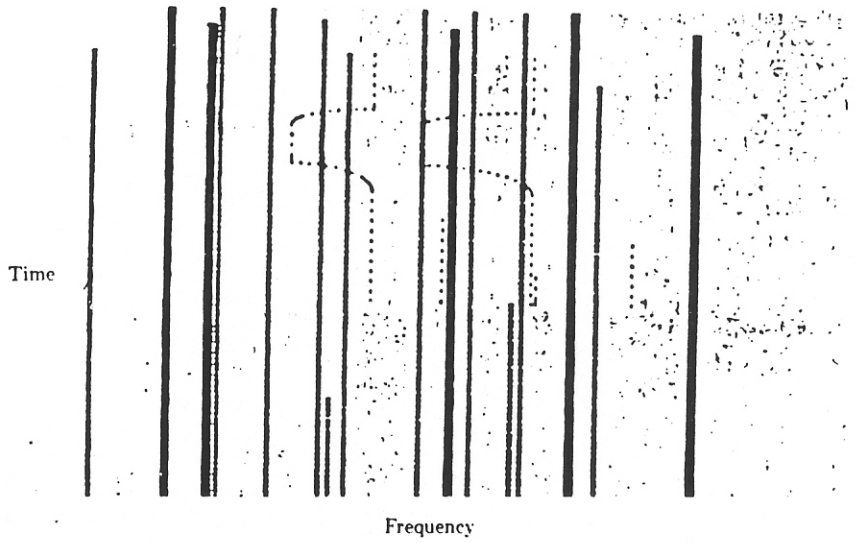


Figure 19.1: A simulated sonogram (with noise suppressed). Reprinted from Nii et al (1982).

INTRODUCTION

We start with a very brief overview of the expert system and the task it performs. (For a more detailed description of the passive sonar understanding problem and an expert system architecture suited to this problem, see Nii et al (1982).) We then examine the need for automated tuning techniques to allow the system to learn and adapt. Finally, we discuss the importance of genetic algorithms for implementing such mechanisms.

The Expert System

Our expert system operates on processed passive sonar data that it receives from an independent signal-processing module. This module transforms the incoming hydrophone data into images called sonograms. Figure 19.1

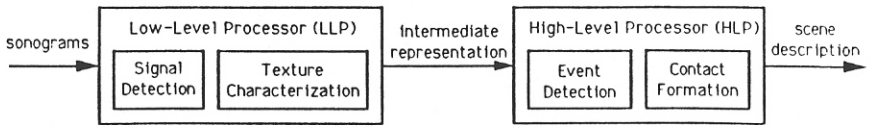


Figure 19.2: Functionality of the expert system

shows a simulated sonogram. A sensor system (such as the one from which our system receives data) produces a set of sonograms that correspond to sensing from different locations and/or in different directions. Sonar analysts read these sonogram images to identify the acoustic signatures of vessels that may be emitting sound in the area. Our expert system attempts to perform the same function.

The expert system performs four basic subtasks, which in order of increasing abstraction are (1) signal detection: determine which parts of the image correspond to emitted sound, and not just noise (known as figure-ground separation in general image analysis); (2) texture characterization: compute parameters that characterize the visual texture of the detected signals; (3) event detection: determine the location of "interesting" features of the signals; and (4) contact formation: group the signals that came from the same source into clusters (known as contacts) and attempt to classify and geographically track the contacts. These subtasks are distributed among two loosely coupled subsystems called the Low-Level Processor (LLP) and High-Level Processor (HLP). The LLP performs signal detection and texture characterization. Its primary inputs are processed data, and its primary outputs are data structures that contain signal locations and computed texture parameters. The HLP performs event detection and contact formation. It takes as inputs the LLP outputs and creates a scene description of the acoustic sources. Hence, the LLP outputs serve as an intermediate-level representation of the information in a sonogram from which the HLP forms a high-level representation. (This intermediate-level representation for sonar image understanding is analogous to the 2 1/2 D sketch for natural image understanding, see Marr 1982.) This functionality is illustrated in Figure 19.2.

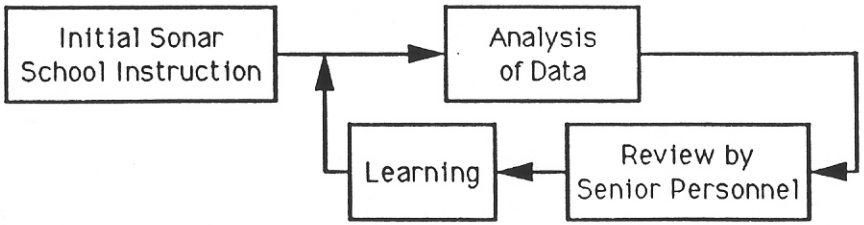
The HLP is similar in architecture and functionality to HASP/SIAP (Nii et al 1982), one of the early examples of a blackboard system. Architecturally, the HLP has three main components: a global database for storing information received from the LLP plus its own inferences; rules for forming new inferences based on the data in the database; and a control structure for invoking rules at the appropriate time. We discuss particular components of the system in greater detail as required in subsequent sections.

The Need for Learning and Adaptation

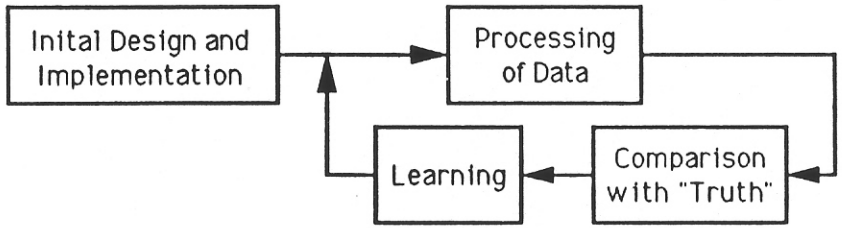
Passive sonar data are very complex. Mathematical models generally do not capture all the characteristics of this data, and those that come close do not yield easily to mathematical analysis. Therefore, when building a system for analyzing sonar data, we have two distinct but interrelated tasks: to build the system and to tune it. The tuning process has received little attention in the past, despite its importance to the success of the system.

Upon examining the tuning process, we have reached three basic conclusions. First, the nature of the data requires that tuning be an ongoing process. Because of the wide range of conditions, signal types, and scenarios, any system tuned on a finite amount of data will eventually encounter a new situation for which its performance is substandard. If the system cannot improve based on this experience, then it will repeat the same mistakes in the future. As an example, consider a system tuned under low-traffic conditions. When it first encounters high traffic, it will inevitably fail. The system must subsequently learn to handle high traffic or be considered inadequate. Figure 19.3 illustrates this approach to system development and the similarities between this approach and the way human sonar analysts learn to perform the same task.

The second conclusion is that two types of learning are involved. The first type is algorithmic tuning. Playing real data through the system will highlight shortcomings and conceptual bugs in the underlying algorithms that must be fixed. At the present time, this type of learning is best done by the human developers with the aid of tools on the machine. The second type of learning is parameter tuning. Our system contains a large number of parameters whose settings greatly influence performance. Choosing the best values for these parameters is generally difficult for a human for a number of reasons, including (1) interactions between parameters, (2) the difficulty humans have mentally juggling large amounts of data to make statistically based decisions, (3) the long time required to evaluate a single set of parameters, and (4) the difficulty of changing the parameters to meet new system specifications. We have therefore been developing methods



(a) Process of Professional Development for Human Analyst



(b) Process of Expert System Development

Figure 19.3: Comparison of development paths of a human sonar analyst and our expert system.

whereby the machine can learn appropriate parameter settings. We discuss these in the next three major sections of this chapter.

The third conclusion is that parameter adaptation (i.e., changing parameter values in response to changing conditions) can greatly improve system performance. For example, the parameter settings that are optimal for high-traffic data are different from those that are optimal for low-traffic data. Hence, a system that uses one appropriate set of parameter values for high traffic and another for low traffic will outperform a system that uses a single set of parameter values under both conditions.

The Importance of Genetic Algorithms

Although (as we will see below) we have used different approaches to automating the tuning of different pieces of our expert system, in each case we have used a genetic algorithm for performing optimization. The properties of genetic algorithms that have made them well suited to our applications are the following. First, they generally find nearly global optima in complex spaces. This is important because the search spaces for our problems are highly multimodal, a property that leads hill-climbing algorithms to get stuck in local optima. Second, genetic algorithms do not require any form of smoothness. This is important because in two cases (rule threshold optimization and tracking parameter optimization) the search space is discontinuous. Third, considering their ability to find global optima, genetic algorithms are relatively fast, especially when tuned to the domain on which they are operating. (They are especially fast when distributed across different machines on a network as discussed later in this chapter .) In all cases, the speed of the optimization algorithm is one of the prime factors determining the practicality of the approach. Fourth, tuning a genetic algorithm for a particular domain is relatively easy because genetic algorithms consist of a general format with some variable components chosen to meet the needs of the domain. The five variable components are

1. A way of encoding solutions to the problem on chromosomes.
2. An evaluation function that returns a rating for each chromosome given to it.
3. A way of initializing the population of chromosomes.
4. Operators (e.g., mutation and crossover) that may be applied to parents when they reproduce to alter their genetic composition.
5. Parameter settings for the algorithm, the operators, etc.

```
If:   (< (average-kludginess signal) 20000)
      (> (lossage-derivative signal time) 0.01)
      (> (friendliness-at-time signal time) 5.5)
Then: (declare-event-type-foo signal time)
```

Figure 19.4: Example event detection rule packet.

OPTIMIZATION OF THRESHOLDS IN EVENT DETECTION RULES

Some particularly important rules in the HLP are those that detect events. Their function is to decide whether or not a certain signal has a certain type of event at a certain time. They thus perform pattern recognition, distinguishing between positive and negative examples of different types of events. An example of an event detection rule is shown in Figure 19.4. Note that the conditions of the rule consist of tests of real-valued functions of the database against real-valued thresholds. These thresholds are parameters whose values can be varied to optimize detection performance. In this section we examine how to automate the process of selecting values for these thresholds.

We start by discussing ROC curves. We then describe the two key components of our optimization procedure: (1) genetic algorithms and (2) windowing. Finally, we examine the results and benefits of using this automated approach.

Receiver Operating Characteristic (ROC) Curves

For the detection problem, there are two basic types of errors: a missed detection (classifying a positive example as a negative one), and a false alarm (classifying a negative example as a positive one). These two types of errors give rise to two different and competing measures of detection performance: probability of detection (P_d), the fraction of positive examples classified correctly, and probability of false alarm (P_f), the fraction of negative examples classified incorrectly. Many detection algorithms have parameters that can be changed to yield different performance and thus a different pair of performance measures (P_d, P_f). A realizable (P_d, P_f) is called Pareto optimal if there exists no other realizable (P_{d1}, P_{f1}) for which $P_{d1} \geq P_d$ and $P_{f1} \leq P_f$ and at least one of these inequalities is a strict

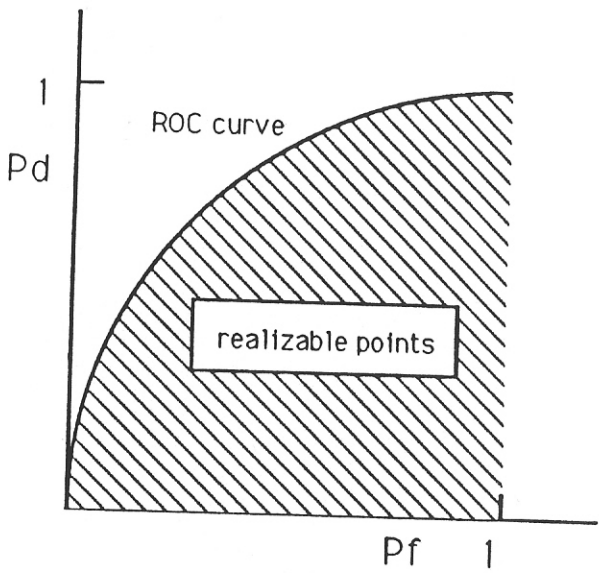


Figure 19.5: Example of an ROC curve.

inequality. The set of all Pareto optimal pairs (P_d, P_f) forms a curve in $P_d - P_f$ space called an ROC curve. An example of an ROC curve is shown in Figure 19.5.

Genetic Optimization of Rule Thresholds

In this section we describe the various components of our genetic algorithm for rule threshold optimization. The algorithm is implemented using an early version of Lawrence Davis's OOGA code.

Encoding: We use a real-valued encoding scheme instead of the traditional binary one. An individual consists of a list of the thresholds in the order in which they appear in the rule. The possible values that a threshold may take are both range-limited and quantized. As one of the functions of our rule editor, the developer specifies the maximum value, minimum value, and step size for a particular threshold. The developer picks these parameters based on knowledge of typical values of the corresponding statistic. This approach is necessary because the statistics used in the rules have a wide range of typical values. For instance, one statistic may generally have

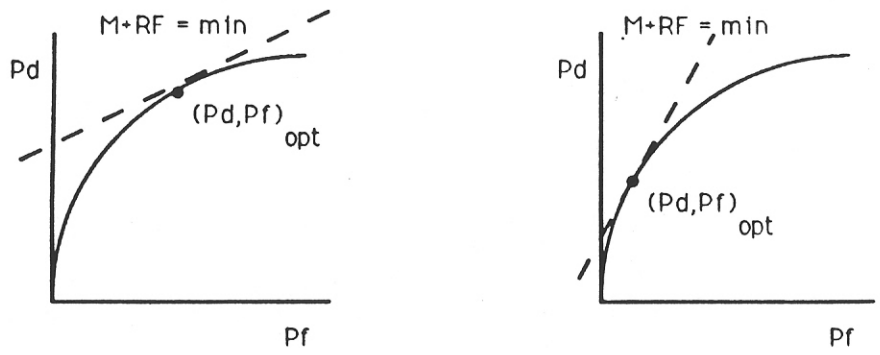


Figure 19.6: Two different values of R yielding two different optimal points along the ROC curve.

values between 5 and 15 while another may typically have values between 0.001 and 0.004. Knowing this information makes the genetic algorithm much more effective because it does not have to waste its time searching out of a statistic's range or on a scale that is insignificant with respect to the statistic. Shaefer (1987) describes a genetic optimization technique called ARGOT which automatically handles the large dynamic range of parameters by adapting the representation during a run.

Evaluation Function: A training database of positive and negative examples exists for each event type. (How we construct this database is the topic of the next subsection, "Windowing.") An automatic scoring function loops through all examples of the appropriate event type in the training database and counts the number of missed detections, M, and the number of false alarms, F, for a given set of rule thresholds. Let N_p be the total number of positive examples and N_n be the total number of negative examples. Then an estimate of P_d is $1 - M/N_p$, and an estimate of P_f is F/N_n. Note that the training database is stacked with particularly difficult cases owing to the windowing procedure (see "Windowing"). Hence, the calculated P_d and P_f are not good absolute estimates of general rule performance. In particular, the calculated P_f is orders of magnitude greater than the actual P_f of the rules. However, these scores do provide a good way to compare relative performance of rules, and they can be computed fairly quickly and effortlessly.

The evaluation function is defined as $M + RF$, where R is a parameter

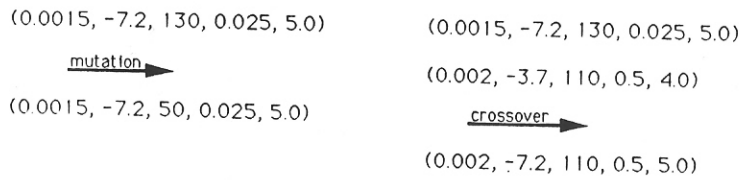


Figure 19.7: The genetic operators mutation and crossover.

whose value is selected by the developer. The choice of R specifies the operating point on the ROC curve (assuming that the genetic algorithm does indeed find global optima). Figure 19.6 shows how two different choices of R lead to two different points on the ROC curve. Allowing the developer to choose the value of R provides him with the capability of choosing an arbitrary operating point on the ROC curve. This capability is crucial to the success of the system. (Schaffer (1985) describes a system called VEGA which tries to find multiple points distributed over the ROC curve rather than just a single point.)

Initialization: The threshold settings for each individual in the initial population are randomly selected from the admissible set.

Operators: We use the two basic genetic operators, crossover and mutation, suitably modified for the particular representation scheme. Our mutation operator creates a child that is the same as the parent in all locations except one or more randomly selected ones. The threshold values of the child in these locations are chosen randomly from their allowable sets (see Figure 19.7a). (The property that the child must be different from the parent is what Davis (1989) calls "guaranteed".)

Our crossover operator takes two parents and creates a single child. It selects each threshold value of the child by randomly choosing one of the two parents and using the corresponding threshold value in that parent (see Figure 19.7b). This is an example of uniform crossover; we have chosen to use this type of crossover based on results reported in Syswerda (1989).

Parameters: The values of a number of parameters can greatly influence the performance of the algorithm. Some of the important parameters are as follows.

PARENT-SCALAR: This parameter determines with what probability each individual is chosen as a parent. The second-best individual is

PARENT-SCALAR times as likely as the best to be chosen, the third-best is PARENT-SCALAR times as likely as the second-best, and so forth. The value was linearly interpolated between 0.92 and 0.89 over the course of a run.

OPERATOR-PROBABILITIES: This list of parameters determines with what probability each operator in the operator pool is selected. These values were initialized so that mutation and crossover had equal probabilities of selection. An adaptation mechanism changes these probabilities over the course of a run to reflect the performance of the operators in a manner described in Davis (1989).

POPULATION-SIZE: This self-explanatory parameter was set to 50.

GENERATION-SIZE: This parameter tells how many children to generate for each iteration (and how many current population members to delete). It was set to one. In the terminology of Syswerda (1989), this makes our genetic algorithm a steady-state genetic algorithm rather than a generational replacement algorithm. The advantage of having the generation size be as small as possible is to be able to immediately incorporate better individuals created via reproduction into the reproductive process as potential parents.

Windowing

It may be hard to find many positive examples of a particular event type, but there is no shortage of negative examples. In fact, so many negative examples are encountered in even a relatively short time slice of continuous sonar data that it is computationally infeasible to optimize the rule thresholds with all these examples in the training database. We therefore require a way to select a training database which contains only a small subset of all the examples encountered but which is representative of the full set of examples. Windowing (Quinlan 1979) is a process which allows us to choose such a subset.

Windowing works according to the following steps: (1) Randomly select a small subset of the examples called the window (which we call the training database); (2) Train the algorithm on the window; (3) Search through the full training set for incorrectly classified examples and add them to the window if they are not already there; (4) If new examples were added to the window, repeat from step (2). This process is illustrated in Figure 19.8. (Note the similarity in form between the windowing procedure shown in Figure 19.8 and the general system development process shown in Figure 19.3. Windowing is an example of this general approach.)

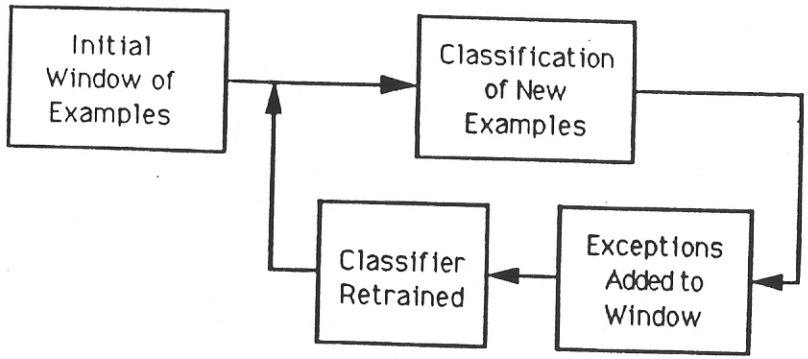


Figure 19.8: Windowing procedure.

We have devised a user interface (described in Montana 1990) that allows a sonar analyst to view the decisions of the rules and add misclassifications into the training database. We have thus been able to perform numerous passes through the windowing cycle. As we do so, the training database becomes more representative of the full range of data, and hence event detection performance continually improves: that is the system learns with experience. Results concerning this learning process are given in the following section.

Results and Benefits

We have discovered a number of results about the rule threshold optimization procedure described above. These are described in detail in Montana (1990) and are summarized here. One result involves how the genetic algorithm scales as a function of the number of thresholds in a rule; we have found that this function is approximately linear and that the number of evaluations required is approximately 100 to 150 times the number of thresholds. Our quantization of the thresholds is such that each threshold on the average represents around five binary dimensions. Hence, the scaling factor is 20 to 30 times the binary dimension.

A second result concerns a comparison between automated and manual tuning. An early version of the system has rule thresholds set by hand by human developers. A later version of the system has rule thresholds determined using the automated technique described in this section. A comparison of the results is shown in Figure 19.9. Note that (1) the automated

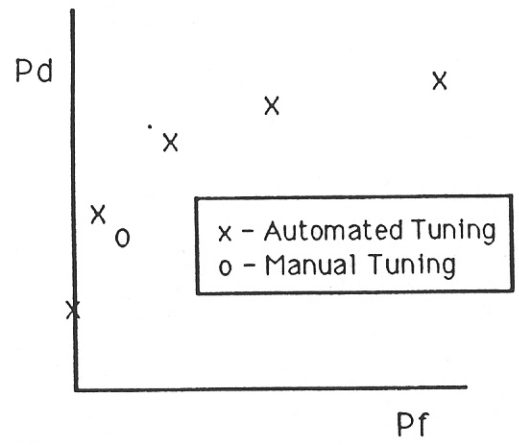


Figure 19.9: Automated tuning versus manual tuning.

approach generated an operating point that is better in a Pareto sense than the manual technique, and (2) the automated approach generated a full ROC curve of operating points simply by choosing different tradeoffs between P_d and P_f and running the genetic optimizer for each tradeoff (note that the time-consuming part of the automated approach is generating the training database via windowing), while the manual approach generated only a single point (and would require starting from scratch to generate a different point).

By enabling the user to easily change the operating point on an ROC curve, the automated technique provides a number of benefits. One is the ability to adapt to changes in system specifications. A second benefit is the ability to have event detection rules with different confidences (derived by changing the tradeoff between P_d and P_f) in the same system (see Montana 1990). In this case, higher level modules can make the ultimate detection decisions based on the confidence associated with detections as well as the existence or lack of confirming evidence from other sources. A third benefit is the ability to adapt to changing conditions. For instance, high-traffic conditions require a lower P_f than low-traffic conditions; hence, tuning rule thresholds for high traffic means putting greater weight on P_f than for low traffic. The system can switch in the appropriate rule thresholds as required.

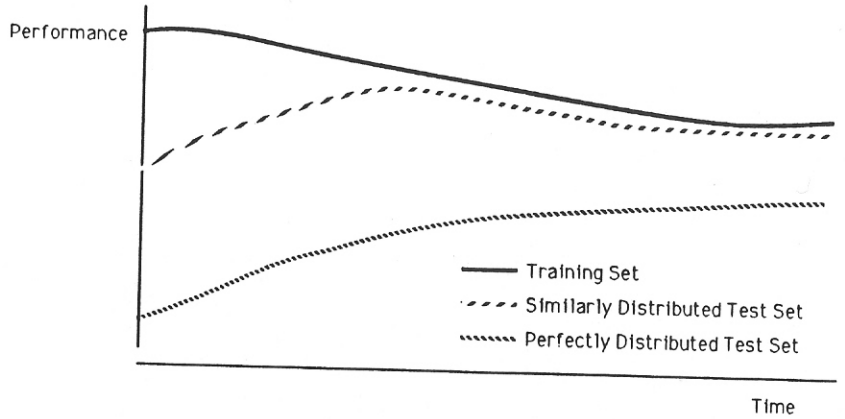


Figure 19.10: Evolution of performance with time.

A final result concerns how the system improves with experience, that is learns. Figure 19.10 presents a graph of system performance versus time on three different types of data. The training data are a randomly selected subset of the full training database. The similarly distributed test data are the data from the full training database that were not chosen as part of the training data. The perfectly distributed training data in theory are a set of examples representative of all possible signals and conditions, but in practice they represent a large amount of independent data. True performance is measured with respect to the perfectly distributed training data. Learning occurs as the training data become more and more like perfectly distributed data, and hence true performance improves.

OPTIMIZATION OF SIGNAL DETECTION PARAMETERS

The signal detection module detects the presence of (possibly simultaneous) signals in a sonogram and determines their location in the image. Most of the details of the algorithm do not matter for the purposes of this chapter.

Even so, we do need to convey some understanding of the large number of parameters in the algorithm, the variety of functions of these parameters, and how these parameters interact. There are four basic parts to the algorithm, each with its own associated parameters: (1) track creation: decide where to start new tracks; (2) track extension: decide how existing tracks proceed through the sonogram (this includes a track smoothing filter with situation-dependent filter parameters); (3) track validation: decide which sections of which tracks to consider to be a true signal rather than noise; and (4) track deletion: decide where the tracks end. Selecting these parameters appropriately is crucial to the success of the algorithm. We have been able to consolidate into 20 significant parameters what once was a larger number of partially redundant parameters. In this section we discuss a method for automatically finding an optimal set of parameters.

The genetic algorithm used to optimize these parameters is the same as that used for rule threshold optimization with a few key differences, which we discuss below. The first is the scoring function used to evaluate a set of parameters, which is similar but more complex than the evaluation criterion for rule thresholds. The second is the fact that the evaluation computations are distributed across multiple computers on a network. We conclude the chapter with a discussion of results and benefits.

The Scoring Function

As is the case for rule threshold optimization, we need a training database against which to score the performance of a set of signal detection parameters. We call this the "truth tracks" database, a phrase that refers to the fact that we are not just detecting the presence of signals but are also tracking their location in the sonogram. Because of the need to track signals, examples in the truth tracks database must contain desired outputs that are much more than simply yes or no (i.e., detect or reject). Instead, they consist of (1) a list of all signals, (2) for each signal, a list of all pixels in which it is visible, (3) a characterization of each signal, and (4) locations of events which should be detected by the event detection rules. (Signal characterizations and events are needed to help weight different signal detection errors according to their importance.) To get all this information, especially that at the pixel level, is very difficult and tedious. To make the process of "truthing" the data much simpler, we have developed a graphical, menu-driven tool for entering and editing data in the truth tracks database.

The scoring function compares the outputs of the signal detection module on some gram regions with the truth tracks for these gram regions to compute a single performance score. This score is analogous to the $M+RF$ score for event detection rules. However, the signal detection score is much more complex for the following three reasons. First, in addition to the basic tradeoff between detections and false alarms, we must consider other performance criteria, such as how close the computed track locations are to the truth track locations. Second, there are great differences in the importance of different signals and pieces of signals. For example, it is particularly important to detect and track correctly the pieces of signals near events. This means the scoring function must weight all errors based on their importance. Third, the outputs of the signal detection module are not an end in themselves but rather are data to feed into higher-level processing modules. This makes scoring signal detection performance partially subjective in the sense that it depends on opinions of how different types of errors affect the higher-level processing.

The score is computed in two stages. The first stage is to compute a correspondence map between pixels of truth tracks and pixels of computed tracks. This map is defined as follows. A pixel of a truth track corresponds to a pixel of a computed track if and only if (1) the computed track pixel is the closest such pixel to the truth track pixel, (2) the truth track pixel is the closest such pixel to the computed track pixel, and (3) the computed track is tracking the truth track at some point. This leaves some truth track pixels unassociated with computed track pixels and vice versa. We call a truth track or computed track unassociated if all its pixels are unassociated.

The second stage of the evaluation computation is to calculate from the correspondence map a small number of performance measures (analogous to P_d and P_f for event detection rules) which can then be combined using a weighted sum to give a single score. The developer chooses the weighting coefficients before the start of an optimization run. We currently use the following six performance measures.

- Missed pixels: weighted sum (with a weighting scheme based on importance) over all truth tracks of unassociated pixels.
- Added pixels: weighted sum over associated computed tracks of unassociated pixels.
- Trash pixels: sum over unassociated computed tracks of all pixels.
- Breaks: weighted sum over associated truth tracks of the number of corresponding tracks minus one.
- Jumps: weighted sum over associated computed tracks of the number of corresponding tracks minus one.

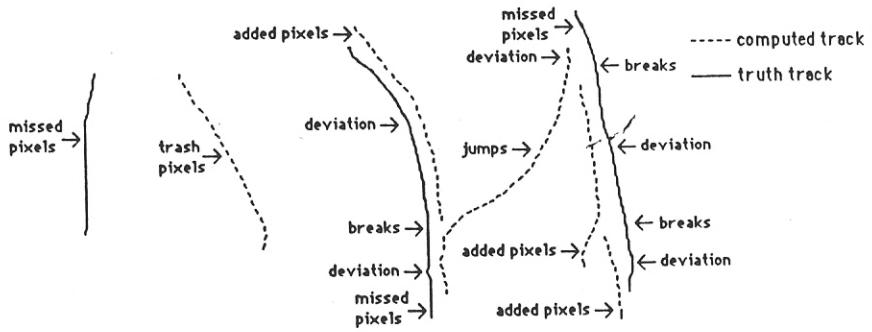


Figure 19.11: Examples of contributions to the different performance measures.

- Deviation: a weighted sum over associated truth track pixels of the square of the distance from its corresponding computed track pixel.

Figure 19.11 presents examples of contributions to each of these scores.

Distributed Evaluations

There are two reasons for distributing the evaluations across machines. First, the signal detection code runs only on SUN computers, whereas the scoring and genetic algorithm code run only on Symbolics computers. The easiest solution to this incompatibility is to have the evaluation function on a Symbolics tell a SUN to run the signal detection algorithm with parameters as given in a given individual. The evaluation function then reads the computed tracks to the Symbolics for the scoring function. A second, more basic reason for distributing the evaluations is speed. Running the signal tracking algorithm on a gram piece typically takes a few minutes, and the scoring function can take an additional minute. If, as an example, we are using three gram pieces, then evaluating each individual might take 10 to 15 minutes, and evaluating many individuals quickly adds up to unmanageably long runs.

We distribute the processing as follows. There are two resource pools,

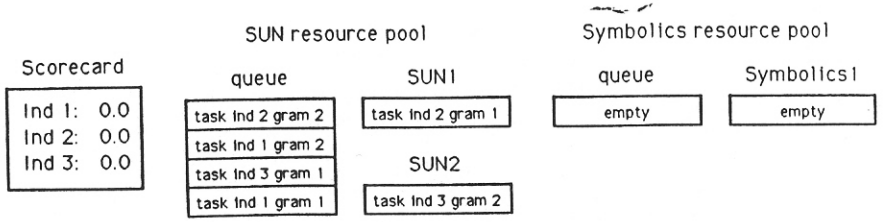
one for SUN and one for Symbolics. Each resource pool contains a list of available hosts and a queue of processes waiting for a host. As hosts become available, processes are taken off the queue and allowed to run. The evaluation function generates one process for each individual and gram piece. Each process first seizes a SUN from the resource pool to run the signal detection algorithm on its gram piece with parameters given by its individual. After this finishes, the process seizes a Symbolics to score the outputs of the signal detection module. When the scoring is finished, the process increments the cumulative score for its individual. This procedure is illustrated in Figure 19.12.

One interesting issue is how the degree of parallelism affects choices of parameters for the genetic algorithm. The only parameter we have found necessary to change with the degree of parallelism is the generation size. For the reason explained in "Genetic Optimization of Rule Thresholds," making the value of this parameter smaller means that the genetic algorithm requires fewer evaluations before convergence. However, with parallelism available, making the generation size smaller also means increasing the average time needed per evaluation. The reason for this is the need for synchronization between generations. Before the genetic algorithm can create the individuals of a new generation, it needs to finish evaluating those of the old generation. During this synchronization period, the evaluation function cannot achieve full parallelism. Since the time needed for synchronization does not change with the generation size, the larger the generation size the larger the average amount of parallelism. When running with two grams in the training database, three SUN's, and one Symbolics, we have found five to be a good value for the generation size.

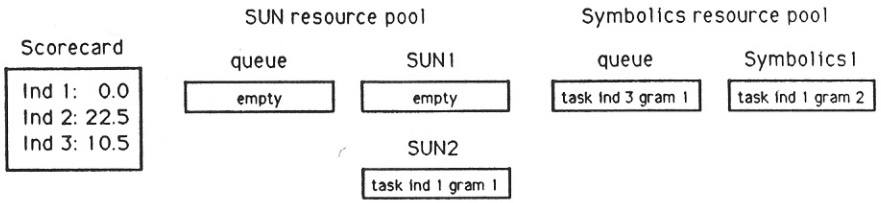
(Note that the optimal solution (which we have yet to implement) to the tradeoff between using good individuals in the reproduction process as soon as possible and obtaining maximal parallelism is to do away with the concept of generations. Instead, let the reproduction and evaluation processes be asynchronous, that is (1) create a new individual for evaluation as soon as there is a free resource to start the evaluation process and (2) as soon as an individual is finished being evaluated, insert it into the population, eliminating the worst population member to make room. In this case, computing resources are always fully utilized while individuals can reproduce as soon as they are evaluated.)

Results and Benefits

The signal detection parameter optimization procedure is a more recent development than the rule threshold optimization procedure. Hence, there



(a) Configuration immediately after starting evaluations



(b) Configuration at a later time

Figure 19.12: Distributing the genetic algorithm.

are few results documenting its performance. We have experimented with it on a relatively small training database with the following results. First, it takes on the order of 500 evaluations for the genetic algorithm to converge. (This is a rule of thumb and has not been statistically validated.) This corresponds to 25 evaluations per real-valued dimension or five evaluations per binary dimension.

Second, with appropriate weights, the optimization routine produces parameters that are better than the manually tuned parameters on the training database in the following respects: (1) its overall score is better (as it should be or else the optimization is not finding a global optimum), (2) each of its six individual scores is better, and (3) most importantly, developers familiar with the expert system agree that its outputs are better. (Recall that there is no truly objective measure of performance because these are only intermediate results.) Work continues to determine whether we can get similar improvements with a larger training database.

Another benefit is the ability to easily alter the tradeoff between the different performance measures in computing a single score as well as the relative weightings of the different signals and pieces of signals in computing the performance measures. Plans for this capability include different parameter sets for different conditions and multiple signal detection routines running on the same data with different parameter sets (thus tuned for detecting particular types of signals).

TRAINING OF NEURAL NETWORKS FOR TEXTURE CHARACTERIZATION

The present texture characterization module computes known and understood parameters of a signal to provide clues to the higher level modules. We have been experimenting to see if we can instead perform classification directly from the texture. We have used neural networks as an underlying classification structure and have trained them on a database of examples of different types of signal textures. Training a neural network means selecting its free parameters so as to optimize its performance (measured by a chosen evaluation function) on the training database. The standard method for training neural networks is called backpropagation (Rumelhart 1986) and is a variant on gradient search. However, for our problem the complexity of the search space caused backpropagation to continually get stuck in local optima far in value from the global optimum. For obtaining better optima, we have used a genetic algorithm for training neural

networks. In this section we describe this genetic training algorithm and the results obtained with it. The material in this section also appears in Montana and Davis (1989).

Neural Networks

Neural networks generally consist of five components:

1. A directed graph known as the network topology whose arcs we call links.
2. A state variable associated with each node.
3. A real-valued weight associated with each link.
4. A real-valued bias associated with each node.
5. A transfer function for each node which determines the state of a node as a function of (a) its bias ψ, (b) the weights, w_i of its incoming links, and (c) the states, x_i, of the nodes connected to it by these links. This transfer function usually takes the form $f(\sum w_i x_i - \psi)$ where f is either a sigmoid or a step function.

A feedforward network is one whose topology has no closed paths. Its input nodes have no arcs to them, and its output nodes have no arcs away from them. All other nodes are hidden nodes. When the states of all the input nodes are set, all the other nodes in the network can set their states as values propagate through the network. The operation of a feedforward network consists of calculating outputs given a set of inputs in this manner. In a layered feedforward network, any path from an input node to an output node traverses the same number of arcs. The nth layer of such a network consists of all nodes that are n arc traversals from an input node. A hidden layer is one which contains hidden nodes. Such a network is fully connected if each node in layer i is connected to all nodes in layer $i+1$ for all i. We have restricted ourselves to working with layered, feedforward networks. Figure 19.13 shows such a network.

The Genetic Algorithm

We now describe the five components of our genetic algorithm for training neural networks.

Encoding: The weights (and biases) in the neural network are encoded in order as a list (see Figure 19.13).

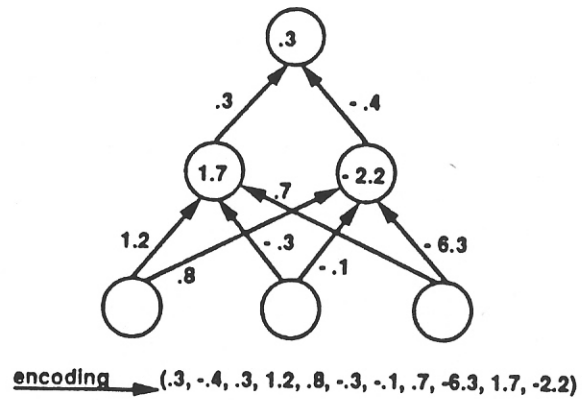


Figure 19.13: Encoding a network on a Chromosome.

Evaluation Function: The weights on the chromosome are assigned to the links in a network of a given architecture, the network is run over the training set of examples, and the sum of the squares of the errors is returned from each example.

Initialization: The entries (i.e., weights) of the initial members of the population are chosen at random with a probability distribution given by $e^{-\|x\|}$, that is a two-sided exponential distribution with a mean of 0.0 and a mean absolute value of 1.0. This is different from the initial probability distribution of the weights generally used in backpropagation, which is a uniform distribution between -1.0 and 1.0. Our probability distribution reflects the empirical observation that optimal solutions tend to contain predominantly weights with small absolute values but that they can have weights with arbitrarily large absolute values. We therefore seed the initial population with genetic material that allows the genetic algorithm to explore the range of all possible solutions but that tends to favor the most likely solutions.

Operators: We created a large number of different types of genetic operators. The goal of most of the experiments we performed was to find out how different operators perform in different situations and thus to be able to select a good set of operators for the final algorithm. The operators can be grouped into three basic categories: mutations, crossovers, and gradients. A mutation operator takes one parent and randomly changes some of the entries in its chromosome to create a child. A crossover operator takes two parents and creates one or two children containing some of the

genetic material of each parent. A gradient operator takes one parent and produces a child by adding to its entries a multiple of the gradient with respect to the evaluation function. We now discuss each of the operators individually, one category at a time.

UNBIASED-MUTATE-WEIGHTS: For each entry in the chromosome, this operator will with fixed probability $p = 0.1$ replace it with a random value chosen from the initialization probability distribution.

BIASED-MUTATE-WEIGHTS: For each entry in the chromosome, this operator will with fixed probability $p = 0.1$ add to it a random value chosen from the initialization probability distribution. We expect biased mutation to be better than unbiased mutation for the following reason. Right from the start of a run, the parents chosen tend to be better than average. Therefore, the weight settings in these parents tend to be better than random settings. Biasing the probability distribution by the present value of the weight should then give better results than a probability distribution centered on zero.

MUTATE-NODES: This operator selects N noninput nodes of the network that the parent chromosome represents. For each ingoing link to these N nodes, the operator adds to the link's weight a random value from the initialization probability distribution. It then encodes this new network on the child's chromosome. The intuition here is that the ingoing links to a node form a logical subgroup of all the links in terms of the network's operation. By confining its changes to a small number of these subgroups, it will make its improvements more likely to result in a good evaluation. In our experiments, $N = 2$.

MUTATE-WEAKEST-NODES: This operator uses a quantity we called node strength. Node strength is different from the concept of error used in backpropagation. For example, a node can have zero error if all its output links are set to zero, but such a node is not contributing anything positive to the network and is thus not a strong node. We define the strength of a hidden node in a feedforward network as the difference between the evaluation of the network intact and the evaluation of the network with that node lobotomized (i.e., with its output links set to zero). We have devised a more efficient way to calculate node strength, but it is not discussed here.

The operator MUTATE-WEAKEST-NODES takes the network that the parent chromosome represents and calculates the strength of each hidden node. It then selects the M weakest nodes and performs a mutation on each of their ingoing and outgoing links. This mutation is unbiased if the node strength is negative and biased if the node strength is positive. It then encodes this new network on a chromosome as the child. The intuition behind this operator is that some nodes are not only not very useful to the network but may actually be hurting it. Performing mutation on these nodes is more likely to yield bigger gains than mutation on a node that is already doing a good job. Note that since this operator will not be able to improve nodes that are already doing well it should not be the only source of diversity in the population. In our experiments, $M = 1$.

CROSSOVER-WEIGHTS: This operator puts a value into each position of the child's chromosome by randomly selecting one of the two parents and using the value in the same position on that parent's chromosome.

CROSSOVER-NODES: For each node in the network encoded by the child's chromosome, this operator selects one of the two parents' networks and finds the corresponding node in this network. It then puts the weight of each ingoing link to the parent's node into the corresponding link of the child's network. The intuition here is that networks succeed because of the synergism between their various weights, and this synergism is greatest among weights from ingoing links to the same node. Therefore, as genetic material gets passed around, these logical subgroups should stay together.

CROSSOVER-FEATURES: Different nodes in a neural network perform different roles. For a fully connected, layered network, the role that a given node can play depends only on which layer it is in and not on its position in that layer. In fact, we can exchange the role of two nodes A and B in the same layer of a network as follows. We can loop over all nodes C connected (by either an ingoing or outgoing link) to A (and thus also to B) and we can exchange the weight on the link between C and A with that on the link between C and B. Ignoring the internal structure, we see that the new network is identical to the old network. That is, given the same inputs they will produce the same outputs.

The child produced by the previously discussed crossovers is greatly affected by the parents' internal structures. The CROSSOVER- FEATURES operator reduces this dependence on internal structure by doing the following. For each node in the first parent's network, it tries to find a node in the second parent's network which is playing the same role by showing a number of inputs to both networks and comparing the responses of different nodes. It then rearranges the second parent's network so that nodes playing the same role are in the same position. At this point, it forms a child in the same way as CROSSOVER-NODES. The greatest improvement gained from this operator over the other crossover operators should come at the beginning of a run before all the members of a population start looking alike.

HILLCLIMB: This operator calculates the gradient for each member of the training set and sums them together to get a total gradient. It then normalizes this gradient by dividing by the magnitude. The child is obtained

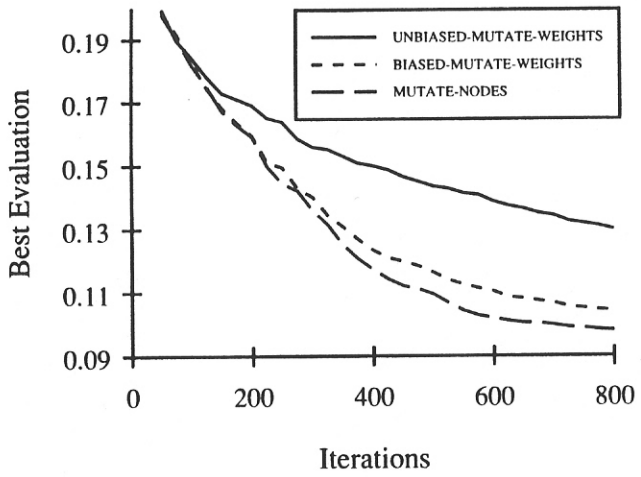


Figure 19.14: Comparison of mutations.

from the parent by taking a step in the direction determined by the normalized gradient of size step-size, where step-size is a parameter that adapts throughout the run in the following way. If the evaluation of the child is worse than the parent's, step-size is multiplied by the parameter step-size-decay = 0.4; if the child is better than the parent, step-size is multiplied by step-size-expand = 1.4. This operator differs from backpropagation in the following ways: (1) weights are adjusted only after calculating the gradient for all members of the training set, and (2) the gradient is normalized so that the step size is not proportional to the size of the gradient.

Parameters: The parameters were the same as for rule threshold optimization.

Results

We have run a series of experiments discussed in detail in Montana and Davis (1989). We summarize the results here. First, comparing the three general-purpose mutations resulted in a clear ordering (see Figure 19.14):

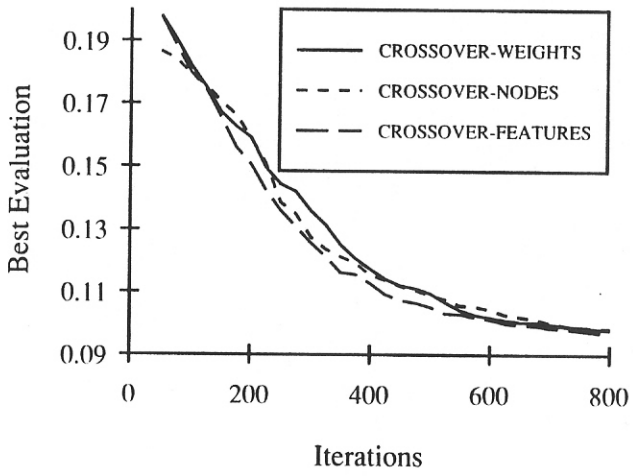


Figure 19.15: Comparison of crossovers.

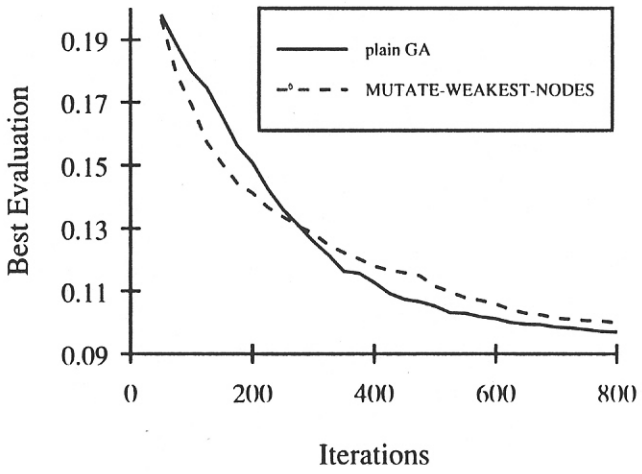


Figure 19.16: Effect of MUTATE-WEAKEST-NODES.

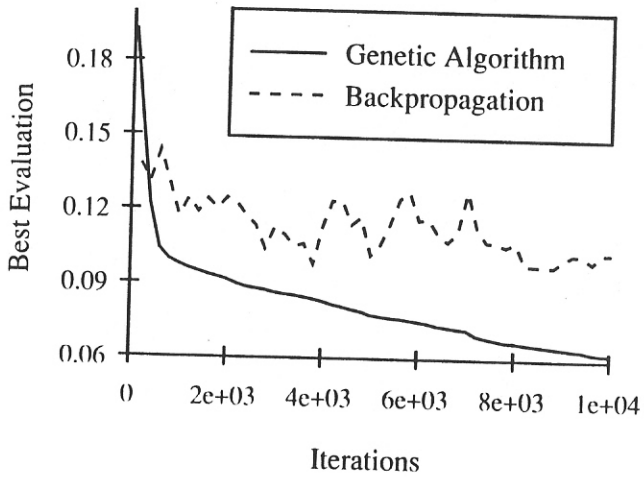


Figure 19.17: Comparison of genetic algorithm with backpropagation.

1. MUTATE-NODES,
2. BIASED-MUTATE-WEIGHTS, and
3. UNBIASED-MUTATE-WEIGHTS.

Second, a comparison of the three crossovers produced no clear winner (see Figure 19.15). Third, when MUTATE-WEAKEST-NODE was added to a mutation and crossover operator, it improved performance only at the beginning; after a certain small amount of time, it decreased performance (see Figure 19.16). Fourth, a hill-climbing mode with just a HILLCLIMB operator and a population of size two gave temporarily better performance than MUTATE-NODES and CROSSOVER-NODES but would quickly get stuck at a local minimum. Finally, a genetic training algorithm with operators MUTATE-NODES and CROSSOVER-NODES outperformed backpropagation on our problem (see Figure 19.17).

CONCLUSIONS

Automated parameter tuning has provided a big boost to the performance of our expert system and promises to do so even more in the future. The expert system consists of four different modules, and we have applied automated parameter tuning to three of them: (1) optimizing rule thresholds for the event detection module, (2) optimizing algorithm parameters for the signal detection module, and (3) optimizing the free parameters of a neural network for the texture characterization module. In each case, genetic algorithms have played a key role.

ACKNOWLEDGMENTS

Thanks are due to the following people: David Davis, for his tutelage in the field of genetic algorithms, his recognition of the potential for genetic algorithms for these parameter optimization problems, and the use of his OOGA code; Steve Milligan, for suggesting the ideas of optimizing rule thresholds and applying neural networks to texture characterization; Fred White, for writing code to support the rule threshold optimization process; and Ken DeJong, for his useful comments.

REFERENCES

Davis, L. (1989). Adapting operator probabilities in genetic algorithms. *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 61–69.

Marr, D. (1982). *Vision.* San Francisco: W.H. Freeman and Co.

Montana, D. J. (1990). Empirical learning using rule threshold optimization for detection of events in synthetic images. To appear in *Machine Learning.*

Montana, D. J. and Davis, L. (1989). Training feedforward neural networks using genetic algorithms. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 762–767.

Nii, H. P. et al (1982). Signal-to-symbol-transformation: HASP/SIAP case study. *AI Magazine* 3(2), pp. 23–35.

Quinlan, J. R. (1979). Discovering rules by induction from large numbers of examples: a case study. In D. Mitchie (ed.), *Expert Systems in the Microelectronic Age.* Scotland: Edinburgh University Press.

Rumelhart, D., Hinton, G. and Williams, R. (1986). Learning representations by backpropagating errors. *Nature* 323, pp. 533–536.

Schaffer, J. D. (1985). Multiple objective optimization with vector evaluated genetic algorithms. *Proceedings of the First International Conference on Genetic Algorithms*, pp. 93–100.

Shaefer, C. G. (1987). The ARGOT strategy: adaptive representation genetic optimizer technique. *Proceedings of the Second International Conference on Genetic Algorithms*, pp. 50–58.

Syswerda, G. (1989). Uniform crossover in genetic algorithms. *Proceedings of the Third International Conference on Genetic Algorithms*, J. David Schaffer, editor, pp. 2-9. Morgan Kaufmann.