Machine Learning, 5, 427–450 (1990)

Empirical Learning Using Rule Threshold Optimization for Detection of Events in Synthetic Images

DAVID J. MONTANA (DMONTANA@BBN.COM)
Bolt, Beranek and Newman, Inc., 70 Fawcett Street, Cambridge, MA 02138

Abstract. We have developed an expert system for interpretation of passive sonar images. A key component of the system is a group of event detection rules whose conditions consist of tests against thresholds. Due to the complexity, variability and clumpiness (i.e., tendency towards highly nonuniform distribution) of the data, tuning these thresholds for good performance under all conditions is a difficult task. We have implemented a procedure for learning rule thresholds whereby the detection capability of each rule continually improves as more and more data is played through the system. The learning procedure contains the following components: 1) a windowing mechanism that adds exceptions (i.e., false alarms and missed detections) into a training database of positive and negative examples and 2) a genetic algorithm to optimize the thresholds with respect to the training database. The genetic training algorithm allows the developer to explicitly choose an operating point on the Receiver Operating Characteristic (ROC) curve of a rule. Experiments have verified 1) the superiority of this automated approach to selecting rule thresholds over manual techniques and 2) the improvement of rule performance with experience.

Keywords. Learning, image interpretation, detection, optimization, genetic algorithms, thresholded rules

1. Introduction

We start with a very brief overview of the full 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 describe the structure of the event detection rules (the tuning of which will be the focus of the paper). Finally, we discuss the necessity of learning mechanisms to allow the expert system to improve with experience.

1.1. The expert system

Our expert system operates on processed passive sonar data which it receives from an independent signal-processing module. This module transforms the incoming raw data into multiple synthetic images. Figure 1 shows a simulated sonar image. Sonar analysts read these images to identify the signatures of nearby vessels. Certain features of sonar images are particularly helpful in the deciphering process, and we call such features "events." Based on interviews with experts, we have compiled a short list of different types of events that they use when analyzing sonar images and which our expert system must therefore be able to detect.

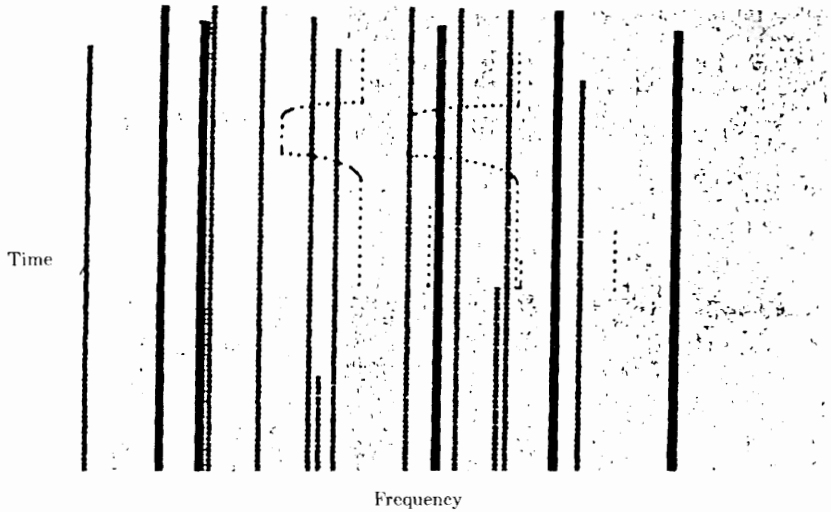


Figure 1. A simulated sonar image (with noise suppressed) reprinted from (Nii et al., 1982).

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 which characterize the visual texture of the detected signals,
3. event detection: determine the location of events, and
4. contact formation: group the signals which 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 which 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 vessels in the area. Hence, the LLP outputs serve as an intermediate-level representation of the information in a sonar image 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 (Marr, 1982).) This functionality is illustrated in Figure 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. In the next section we discuss the rules in detail.

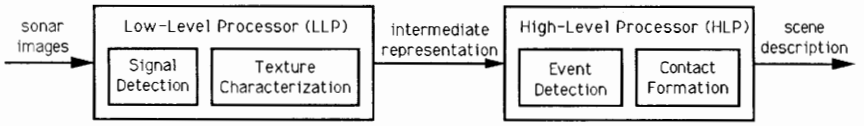


Figure 2. The functionality of the expert system.

1.2. The structure of rules

The rules in the HLP have a hierarchical structure. Individual rules are grouped together into rule packets. The rules in a packet always have the same arguments. When a rule packet is invoked, the rules in that packet are run in sequence until one fires or they all fail to fire. Thus, when the consequents of the rules contained in a packet are the same (as they are for event detection rules), the conditions of the rule packet implement a disjunction of conjunctions, which is a popular structure for empirical learning (Michalski, Mozetic, Hong & Lavrac, 1986). Related rule packets are grouped together into rule sets. Rule sets can easily be turned on and off in order to change the functionality of the system. (One use of this capability described in Section 5.2 is to adapt system performance to varying conditions by having multiple event detection rule sets and using the one appropriate for the present conditions.)

Some particularly important rules 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 are thus performing pattern recognition, distinguishing between positive and negative examples of different types of events. Two examples of event detection rules are shown in the rule packet of Figure 3. (For reasons explained in Section 4.2, the event detection rule packets in our system usually differ from that in Figure 3 in that they only contain one rule.) Note that the conditions of the rules consist of tests of real-valued functions of the database against a real-valued threshold. These thresholds are parameters whose values can be varied to optimize detection performance, and this paper focuses on an automated method for selecting these values.

```
EXAMPLE-RULE-PACKET (signal time)
"detects foos with confidence 0.5"

If: (< (average-kludginess signal) 20000)
    (> (lossage-derivative signal time) 0.01)
Then: (declare-foo signal time 0.5)

If: (< (average-kludginess signal) 10000)
Then: (declare-foo signal time 0.5)
```

Figure 3. An example event detection rule packet.

The example packet in Figure 3 operates as follows. When invoked, it will first calculate the average kludginess of the signal, where some lower-level routine(s) determines how we measure this quantity. If the value is less than the threshold 20000, then the packet continues with the next condition in this rule; otherwise, it moves on to the next rule. If all the conditions of either rule are true, then the packet invokes the consequent, whose effect is to modify the database to indicate that a foo has been detected on the specified track at the specified time with confidence 0.5.

All packets which detect events of different types with the same confidence are grouped together into a single rule set. (Figure 4 show the rule hierarchy for event detection rules.) The reason for this grouping is primarily to support the adaptation mechanism described in Section 5.2. (Adaptation requires turning on and off all event detection rule packets which detect their different event types with the same confidence.)

1.3. The need for learning

Passive sonar data is very complex. Mathematical models generally do not capture all of the characteristics of this data, and the ones that come close do not yield easily to mathematical analysis. Therefore, when building a system for analyzing sonar data, there are two distinct but interrelated tasks: building the system and tuning 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 two basic conclusions. First, due to the nature of the data, tuning must 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 5 illustrates this approach to system development and the similarities between this approach and the way that human sonar analysts learn to perform the same task. This iterative approach to development as applied to selection of event detection rule thresholds is embodied by the windowing procedure described in Sections 2.1 and 3.3.

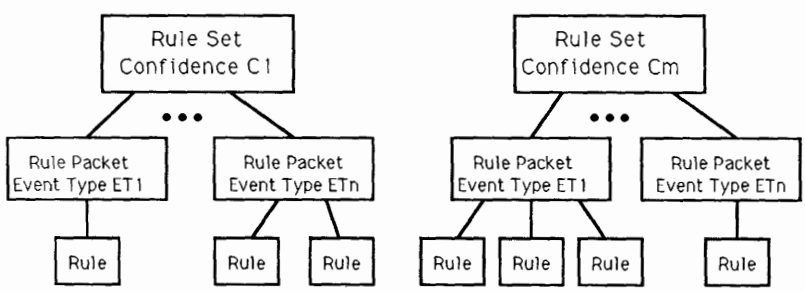
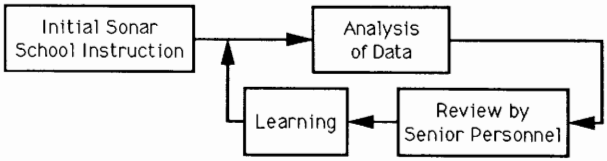
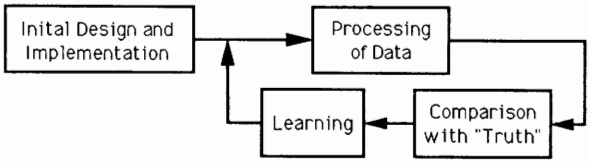


Figure 4. The rule hierarchy for event detection.



(a) Process of Professional Development for Human Analyst



(b) Process of Expert System Development

Figure 5. A comparison of development paths of a human sonar analyst and our expert system.

The second conclusion is that there are two types of learning involved. The first is algorithmic tuning (a rough equivalent of knowledge acquisition). Playing data through the system will highlight shortcomings and conceptual bugs in the underlying algorithms which 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. (In Section 5.3 we discuss one such tool.) The second type of learning is parameter tuning (a rough equivalent of skill refinement). Our system contains a large number of parameters whose settings greatly influence performance. Examples of such parameters are the thresholds in the event detection rules. The parameters settings not only need to improve (i.e., learn) with experience but also need to change (i.e., adapt) in response to changes in system specifications (see Section 2.2), underlying algorithms, and surrounding conditions (see Section 5.2). Choosing the best values for these parameters is generally a hard problem for a human for a number of reasons including the interaction between parameters and the difficulty of objectively comparing different sets of parameters (see Section 4.2). We have therefore been developing methods whereby the machine can learn appropriate parameter settings. Some of our work on learning the parameters of a neural network for texture characterization is described in (Montana & Davis, 1989). Optimizing the parameters of a signal tracking algorithm is discussed in (Montana, in press). In this paper we focus on how the system learns the thresholds for event detection rules.

2. Motivation for our approach

In this section, we motivate our approach to the problem of learning to detect events. We start by describing two important requirements of any solution: 1) windowing and 2) the ability to pick an arbitrary operating point on the receiver operating characteristic (ROC) curve of the underlying classifier. We then discuss why other popular classification algorithms are not appropriate for our problem. Finally, we examine why genetic algorithms are well suited to our approach.

2.1. The need for windowing

Windowing is a method for making nonincremental empirical learning more efficient on a very large training set. (Windowing is generally unnecessary for incremental learning algorithms such as the one described in (Schlimmer & Granger, 1986); its function is to allow nonincremental techniques to work on problems which would otherwise only be computationally feasible using incremental techniques.) It was introduced by Quinlan (1979) for use with the ID3 algorithm, but it can be applied to any pattern classification algorithm. It works according to the following steps:

1. Randomly select a small subset of the examples called the window (which we sometimes refer to as the training database to avoid confusion with the user interface mechanism),
2. Train the learning algorithm on the window,
3. Search through the full training set for exceptions (i.e., incorrectly classified examples) and add them to the window if they are not already there, and
4. If there were new examples added to the window, repeat from step (2).

This process is illustrated in Figure 6. (Note the similarity in form between the windowing procedure shown in Figure 6 and the general system development process shown in Figure 5. Windowing is an example of this general approach.)

The idea is that training the algorithm multiple times on small training sets is computationally less expensive than training once on a very large training set. The success of windowing depends on the ability to find a small number of representative examples which when used as a training set yield the same results as using the full training set. This ability in turn depends on the data; there are some large training sets which satisfy this property and some which do not. Wirth and Catlett (1988) examine the costs and benefits of using windowing with ID3 on a test suite of eight different classification tasks and conclude that there are no real benefits to windowing. However, in our work on detecting sonar events, we have found windowing to be essential. The reasons for this are not related to the particular learning algorithm employed but rather to properties of the data which we now describe.

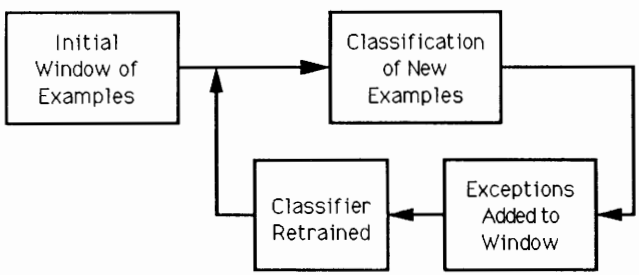


Figure 6. The windowing procedure.

For one, the full database of examples is not only extremely large but constantly increasing. Our system keeps on encountering new data; we ignore it at the peril of continually repeating the same mistakes since there are always signal types and listening conditions not adequately represented in previous data. Hence, we must constantly retrain the system to include the new data, and it is not computationally possible to be continually retraining using all the data as a training set.

Secondly, examples which occur close in time are likely to be very similar. We refer to this property as "clumpiness." Clumpiness of the data arises due to the slow rate at which the scene changes. Hence, examples which occur close in time are likely to be from the same source, located at approximately the same position, and viewed approximately under the same conditions. Because of the similarity between so much of the data, there is generally a relatively small subset of examples which is representative of the full set of examples, which is the key condition for windowing to succeed.

2.2. *Receiver operating characteristic (ROC) curves*

For the detection problem, there are two basic types of errors. One is a missed detection (classifying a positive example as a negative one), and the other is 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 which 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_{d_1}, P_{f_1}) for which $P_{d_1} >= P_d$ and $P_{f_1} <= P_f$ and at least one of these inequalities is a strict inequality. The set of all Pareto optimal pairs (P_d, P_f) forms a curve in P_d-P_f space called a ROC curve. An example of a ROC curve is shown in Figure 7.

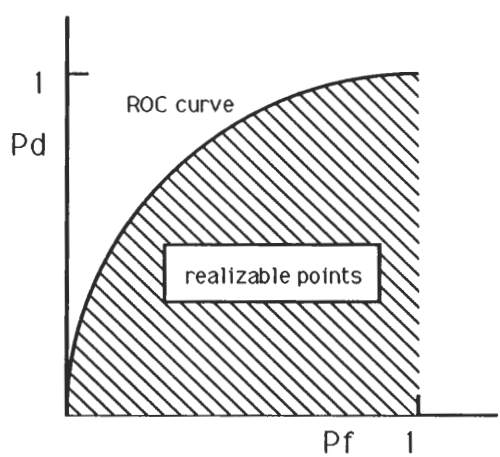


Figure 7. An example of a ROC curve.

The ability to explicitly control the tradeoff between P_d and P_f for event detection rules is key to the success of our system. The operating concept for the system defines the criterion by which we evaluate the relative merits of different points in P_d-P_f space. (For example, in the first version of the system, we were primarily interested in not looking foolish by making too many obviously false calls. This translated into a much greater emphasis on lowering P_f than on raising P_d. In the second release, the emphasis shifted much more towards raising P_d.) For any reasonable evaluation criterion (i.e., one for which, all else being equal, it is always better to increase P_d and decrease P_f), the operating point which optmizes this criterion lies on the ROC curve. Hence, the ability to select a specific point on the ROC curve allows for optimal detection performance.

Other benefits of the ability to explicitly trade off between P_d and P_f are the potential for multiple confidences (see Section 5.1) and adaptation to changing conditions (see Section 5.2).

2.3. Selecting an empirical learning technique

Most empirical learning algorithms have two basic components, a family of classifiers and a training algorithm. The training algorithm searches through the space of members of the family of classifiers for one which classifies well a set of training examples and which is likely to correctly classify new examples. Different approaches to empirical learning use different families of classifiers and different training algorithms.

One group of algorithms, the best known of which is AQ (Michalski et al., 1986), uses what they call decision rules as their family of classifiers, where they define the left-hand side of a decision rule as a disjunction of conjunctions. (Note the difference in terminology here: we define the left-hand side of a rule as just a conjunction of conditions, which is standard nomenclature for rule-based systems. Our rule packet (assuming that the contained rules have the same consequents) is the equivalent of their decision rule.) The AQ training algorithm uses heuristic search to find a logical expression which is as small as possible which classifies all the training examples correctly.

A second group of algorithms, which includes ID3 (Quinlan, 1979) and CART (Breiman, Friedman, Olshen & Stone, 1984), uses decision trees as their family of classifiers. The basic ID3 training algorithm employs heuristic search to find a decision tree which is as small as possible which classifies the training examples correctly. CART additionally uses a method for pruning back these trees for better generalization based on a technique known as cross-validation. Decision trees are equivalent to decision rules: every decision tree can be written as a decision rule and vice versa (Quinlan, 1987). Hence, decision tree algorithms and decision rule algorithms use the same family of classifiers; the difference is the inductive biases of the training algorithms.

A third group of algorithms, which includes backpropagation (Rumelhart, Hinton & Williams, 1986) and perceptrons (Rosenblatt 1959), uses feedforward neural networks of a fixed topology (with variable weights and biases) as their family of classifiers. Their training algorithms search through the space of weights and bias for those which minimize an error criterion. This criterion measures the difference between the desired outputs of the network on the training examples and the actual outputs. Backpropagation uses a gradient

search for multilayer networks. Perceptrons use heuristic search for a single layer network. Another approach uses genetic search for multilayer networks (Montana and Davis, 1989).

Each of these approaches has its advantages and disadvantages. Which approach is the best depends on the problem to be solved. In our approach, none of these seemed applicable for the following reasons.

First, the problem requires that the classifier have a straightforward explanation facility for telling a human the reason it made a particular classification decision. We need such an explanation facility because the event detector is part of a larger, imperfect, and evolving system. Knowledge of why the event detector is failing or succeeding has been crucial to the improvement of the overall system. For example, Section 5.3 describes a tool for evaluating the importance of the classification features used by the event detection rules which works because of the existence of a simple explanation facility. This tool has allowed us to find better classification features, which in turn has improved the performance of the event detection rules. Neural networks do not have any satisfactory explanation facility.

Second, as discussed in Section 2.2, our system requires the ability to choose an arbitrary point on the ROC curve when training a detection algorithm. AQ, perceptrons, ID3, and CART do not provide such a capability (although CART does provide some limited ability to trade off between P_d and P_f in the pruning process).

As described in Section 1.2, the family of classifiers for our approach is all decision rule packets (the equivalent of AQ's decision rules) of fixed form with variable thresholds. This is a much more limited space than that of AQ, which can have decision rules of variable form with variable thresholds. Fixing the rule packet form in our approach is analogous to fixing the network topology in the neural net approach. The search space is R^n, where n is the number of free parameters (in our case rule thresholds). The training algorithm is a genetic algorithm for reasons given in Section 2.4. It minimizes a user-defined error criterion which specifies the tradeoff between P_d and P_f (see Sections 2.2 and 3.2). This approach provides both a simple explanation facility and the ability to explicitly choose an operating point on the ROC curve (see Section 3.2). It is thus the only approach of those described which meets the requirements of the problem (see Figure 8).

Requirements / Algorithms	explanation facility	arbitrary point on ROC curve
AQn (rule induction)	▨	
ID3 (decision trees)	▨	
Perceptron		
Backpropagation		▨
Rule Threshold Opt'n	▨	▨

Figure 8. A matrix of algorithms versus problem requirements.

2.4. Genetic algorithms

Genetic algorithms are a family of algorithms for optimization and learning. We use a genetic algorithm to optimize the rule thresholds of a rule packet (see Section 3.2), i.e., as the training algorithm for our empirical learning procedure. The properties which make genetic algorithms well suited for this task are the following. First, they generally find nearly global optima in complex spaces. This is important because the search space for our problem is highly multimodal, a property which leads hillclimbing algorithms to get stuck in local optima. Secondly, they do not require any form of smoothness. This is important because our search space is discontinuous, consisting of discrete steps between areas with a constant value. Thirdly, considering their ability to find nearly global optima, genetic algorithms are relatively fast, especially when tuned to the domain on which they are operating. This is important to our problem because windowing requires us to continually reoptimize the thresholds as the training database receives new data.

Their name alludes to the features they share with biological evolution. These include i) a population of individuals, ii) reproduction as a means of creating new individuals, and iii) survival of the fittest. We assume that the reader is familiar with how genetic algorithms evolve a population of better and better individuals. If not, a good introduction is (Goldberg, 1988). We do mention here that there are five variable components to a genetic algorithm which are used to adapt the general format to a particular domain. These 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.

In Section 3.2, we describe how these components are defined for our threshold optimization genetic algorithm.

3. The learning system

In this section, we describe various aspects of the piece of our system devoted to learning of rule thresholds. First, we examine a user interface for windowing which allows a sonar analyst to easily add exceptions into the training database. We then detail the genetic algorithm used for threshold optimization. Finally, we describe the iterative procedure by which the rule thresholds continually improve.

3.1. Windowing user interface

A good user interface is a key component of any computer system which interacts with people, especially those people not familiar with computers. As an example consider the

importance of rule editors to the development of expert systems. They make it easier to enter knowledge into the system and thus aid in the more efficient development of more effective systems. The windowing user interface plays the role of the rule editor in our learning system: it provides the mechanism by which a domain expert can impart his knowledge to the system. Making this process easy is crucial to the collection of a training database of sufficient quantity and variety.

An important feature of our windowing interface is that it does not require the human analyst to classify every example but rather to just point out the mistakes that the system makes. Our sonar data consists of an endless stream. Classifying every example would mean a workload that would remain constant over time. However, the job of pointing out exceptions (i.e., misclassifications) has a workload that decreases with time as the system improves.

We are not the first to use interactive windowing; one earlier version is a tool called Interactive ID3 described in (Shapiro, 1987). Our user interface differs from others in how it is tailored to our problem domain. There are two basic mechanisms by which the user can add examples to the training database, one for examples which the rules call events and one for examples which they do not call events. For the former, a scrollable *peek* window contains a list of all called events along with identifying information such as event type and position in the image (see Figure 9). The events in this peek window are mouse-sensitive; hence the analyst can instruct the system to save any event into the training database with a mouse click. The system asks the analyst whether this is a positive or negative example before writing it out.

A different procedure is used for examples which the rules do not call events because the number of such examples is generally far too numerous for display in a window and because nonevents are not saved in the HLP global database. In this case, the analyst must display the appropriate sonar image and click on the position in the image of the missed event. The system prompts for information such as event type and whether the example is positive or negative and then writes out the example.

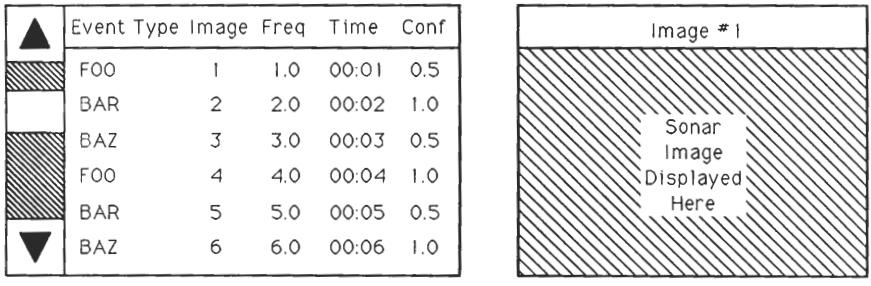


Figure 9. The windowing user interface.

3.2. *Genetic optimization of rule thresholds*

In this section we describe the various components of our genetic algorithm for rule threshold optimization.

1. *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 which 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 such a wide range of typical values. For instance, one statistic may generally have values between 5 and 15 while another may typically have values between .001 and .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 which is insignificant with respect to the statistic. It is also aesthetically pleasing to have thresholds which are round numbers. (Note that using a binary encoding would have forced the number of steps for each threshold to be a power of two, and this would have yielded unappealing step sizes.)

2. *Evaluation Function*: An automatic scoring function loops through all examples 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 due to the windowing procedure (see Section 2.1); hence, the calculated P_d and P_f are not good absolute estimates of general rule performance. In particular, the calculated P_f is generally 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 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 10 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; as discussed in Section 2.2, this capability is crucial to the success of the system.

Note that there are many other possible optimization criterion besides a linear combination of M and F. One such criterion is $f(M, F)$, where $f = \infty$ when $F > F_o$ and $f = M$ when $f \leq F_o$ and where F_o is some constant threshold. This optimization criterion leads to a set of thresholds which maximize P_d subject to the constraint of P_f being below some fixed value. Interchanging the role of M and F in the previous optimization criterion yields a set of thresholds which minimize P_f subject to P_d being above some fixed value. In fact, any real-valued function $f(M, F)$ can serve as an optimization criterion although the choice of $f(M, F)$ can significantly effect the speed of convergence of the genetic

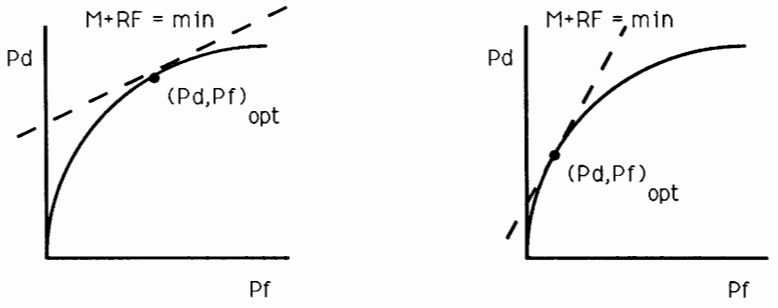


Figure 10. Two different values of R yielding two different optimal points along the ROC curve.

algorithm. (For example, for constrained optimization problems such as the ones just described, experimental evidence indicates that well-chosen, graded penalty functions will outperform the hard-limiting penalty functions given above (Richardson, Palmer, Liepins & Hilliard, 1989).)

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

4. *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 a randomly selected one. The threshold value of the child in this location is chosen randomly from its allowable set (see Figure 11). We have chosen our mutation to change exactly one value each time because the rules generally have had no more than a dozen thresholds and sometimes as few as three or four (and we usually have only one rule in an event detection rule packet for reasons described in Section 4.3). Hence, this mutation operator can create sufficient diversity while ensuring that a child is never the same as its parent (this latter property 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 11). This is an example of uniform crossover; we have chosen to use this type of crossover based on results reported in (Syswerda, 1989).

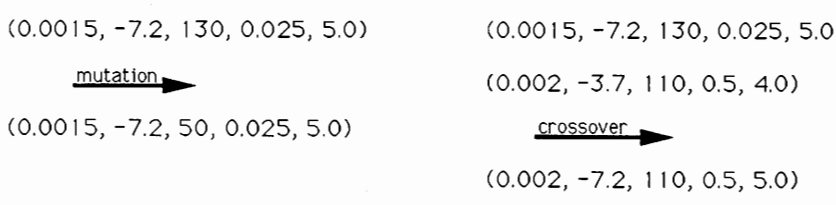


Figure 11. The genetic operators mutation and crossover.

5. *Parameters*: There are a number of parameters whose values can greatly influence the performance of the algorithm. We now discuss some of the important parameters individually.

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, etc. 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. (We choose to always delete the worst member of the population.) In the terminology of Syswerda (1989), this makes our genetic algorithm a steady-state genetic algorithm rather than a generational replacement algorithm. Syswerda (1989) explains the advantages of a SSGA.

3.3. The learning process

The learning process occurs as follows. We initialize the training database by turning off the rules, playing data through the system, and collecting a relatively large number of positive examples. We start with only positive examples because they are so much less common than negative examples that we need to initially stack the training database with them to ensure statistical significance. We then execute the windowing loop described in Section 2.1. Training the system consists of running the genetic optimization algorithm once for each type of event. We choose the tradeoff constant R in the optimization criterion to give appropriate detection performance, where the developer is the subjective judge of appropriateness. Sometimes the developer must make multiple runs for one event type to find a value of R which produces acceptable performance. The retained system is then run on some data, and the exceptions (i.e., false alarms and missed detections) are added into the training database using the interfaces described in Section 3.1.

At times we add the following twist to the windowing loop to increase the efficiency with which we extend the training database. Our run-time system allows rules which detect the same type of event with different confidences (see Section 5.1). So, when we retrain the system, we create two rule sets, one which detects events with high confidence (i.e., low P_f) and one which detects events with low confidence (i.e., high P_d). When running the data through the system, we add to the training database all negative examples called events with any confidence and all positive examples which either are not called events or are called with low confidence. Note that there is an overlapping region where both

positive and negative examples are saved. This helps greatly to increase the rate at which we collect examples for the training database and eliminates the dilemma between tightening the thresholds to catch more missed detections and loosening the thresholds to catch more false alarms.

4. The results

In this section, we present four different types of results concerning the performance of our learning system. The first set of results details the properties of our genetic training algorithm. The second set of results investigates the dependence of detection performance on the number of rules in a detection rule packet. The third set of results describes how the detection performance of the rules improves with time. The fourth set of results compares the performance of our machine learning procedure for setting rule thresholds with the performance of humans setting the thresholds by hand.

4.1. Training algorithm properties

There are two properties of our genetic training algorithm which are of particular interest: i) its time of convergence as a function of problem complexity, and ii) its ability to find global optima. Two factors influencing problem complexity are the number of thresholds (one measure of the dimensionality of the search space) and the number of examples. We have performed experiments to attempt to quantify these properties. (Note that in all of these experiments we keep the value of $R = 0.7$ constant. Recall from Section 3.2 that R is the weighting of false alarms relative to missed detections.)

The first set of experiments examines the relationship between time of convergence and number of thresholds. Each of these experiments consists of performing the following procedure for a particular event type. First, run the threshold optimization algorithm ten times on the corresponding detection rule recording the best current value as a function of the number of individuals evaluated. Average the values of these ten runs together to get an average best value as a function of the number of evaluations. Say that the number of evaluations required for convergence is that point at which the value of the average run was first within ϵ of the final value of that run. (Note that ϵ is held constant for all experiments.) This is an estimate of time of convergence of the training algorithm. Now, repeat this procedure with the following difference: for each run, take out of the rule one condition (and thus one threshold) selected randomly. The number of evaluations required for convergence for this average run is an estimate of time of convergence with one less threshold than in the original rule. Continue in this manner taking out two thresholds, three thresholds, and so on until the last average run is working with only two thresholds. This provides an estimate of time of convergence as a function of the number of thresholds.

We have performed this experiment four times for four different event types giving the results pictured in Figure 12. There are a few conclusions we can draw from these results. First, averaging over ten runs did not sufficiently eliminate noise. Functions that should have been monotonically increasing were clearly not due to the noise. Second and more

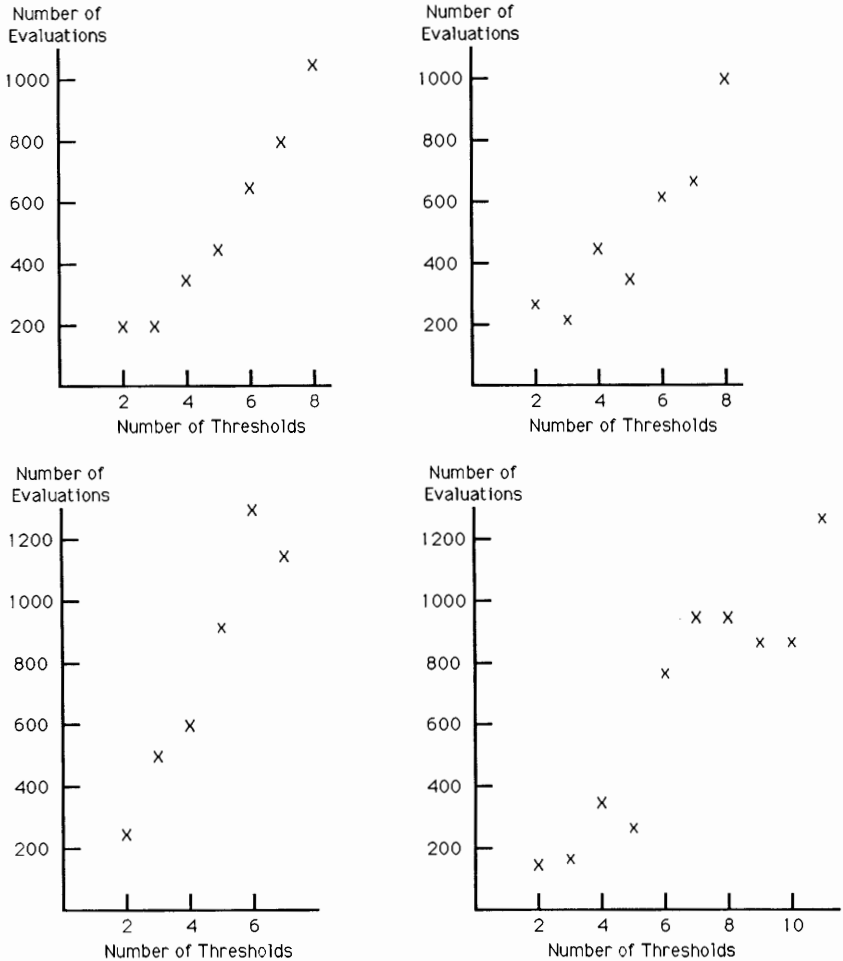


Figure 12. Plots of number of evaluations before convergence versus dimension of search space for four different event types.

important, the dependence of time of convergence on dimension of the search space is close to linear at least for the range of dimensions in the experiment. We should note that the binary dimension of the search space is on the average about five times the number of thresholds, so that from a binary point of view these search spaces are reasonably large.

The second set of experiments examines the relationship between time of convergence and number of examples in the training database. These experiments use the same methodology as the first set of experiments except that they vary the number of examples instead of the number of thresholds. The results are shown in Figure 13. Notice that the time of convergence tends to increase linearly with the number of examples until a certain

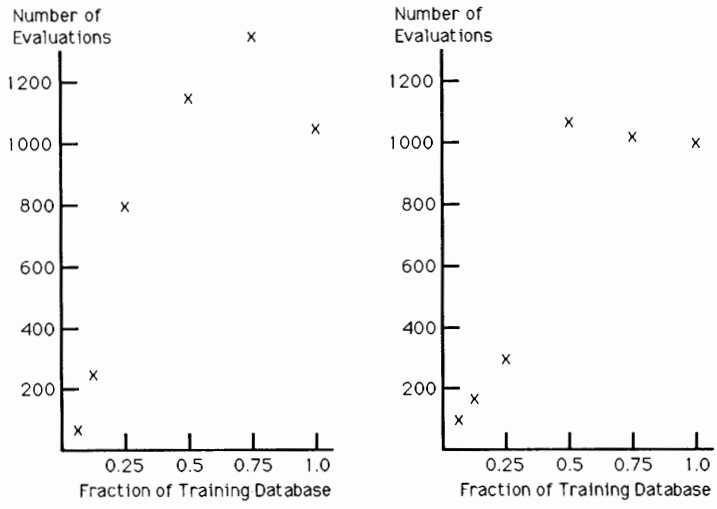


Figure 13. Plots of number of evaluations before convergence versus training set size for two different event types.

point where it reaches a plateau. A possible explanation for this is that adding more examples increases the complexity of the search space; however, in a manner intuitively similar to the Sampling Theorem, a discrete space has a maximum complexity. Note also that the amount of compute time required for an evaluation is approximately proportional to the number of examples. Hence, the time required to run the training algorithm is proportional to the number of evaluations times the number of examples.

The third set of experiments examines the ability of the training algorithm to find nearly global optima. These experiments consisted of performing ten runs of the training algorithm on the same data and recording the evaluation of the best individual from each run. The results are shown in Figure 14. Note that they are almost all different values, which clearly indicates that this genetic algorithm does not generally converge on the absolute global optimum if the search space is sufficiently complex. However, observe that the seven best values are all within a range of each other which is equal to the penalty for one false alarm, 0.7, and less than the penalty for one missed detection, 1.0. Since there are hundreds of examples, this is an insignificant difference. The other three values differ from the best value by small but significant amounts. Upon examining the individuals associated with these values, we found them to be in a completely different part of the search space from the other seven. The main conclusion from these results is that to obtain a final set of thresholds for use in the field it is best to run the training algorithm a few times to make sure the optimum is nearly global; however, for the windowing process one run of the training algorithm is usually sufficient.

[Note that from this point on the results are all qualitative rather than quantitative due mainly to the sensitive nature of the data.]

Run Number	1	2	3	4	5	6	7	8	9	10
Best Value	56.1	57.6	56.3	56.7	56.0	56.3	56.2	57.3	56.2	57.9

Figure 14. The best values from ten different runs on the same data with $R = 0.7$.

4.2. One rule versus multiple rules

Recall from Section 1.2 that a rule packet can have more than one rule in it. These rules are combined disjunctively, thus making the rule packet a disjunction of conjunctions of conditions. The developer has the choice of how many rules to include. We have performed some experiments to determine the optimal number of rules. These experiments are analogous to those described in (Gorman & Sejnowski, 1988) for determining an optimal number of hidden units of a neural network. The procedure is as follows. Hold aside a fraction of the training database for a test set. Train rule packets with different numbers of rules on the training set. Then test their generalization on the test set. We have run this experiment for two different event types. Qualitative results from one of the experiments are shown in Figure 15. Observe that, as expected, adding more rules improves performance on the training set. However, there is no significant performance difference on the test set. According to these experiments, for our data using one rule per packet offers the advantages of maximum simplicity and predictability of generalization without any disadvantages. This is why we generally use only one rule in event detection rule packets.

For these experiments as well as those described in the next two sections, we measure performance as follows: compute $M + RF$ (see Section 3.2 for the meaning of these terms) on the indicated set of data and normalize it by $N_p + RN_n$, where N_p is the number of positive examples in the data and N_n is the number of negative examples.

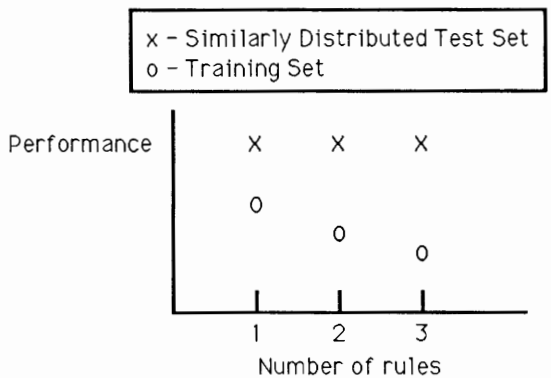


Figure 15. Plot of performance ($M + RF$) versus the number of rules.

4.3. Detection performance as a function of time

There are three different kinds of data on which to evaluate detection performance: i) training data, ii) test data chosen from the same distribution as the training data, which we call *similarly distributed* test data, and iii) test data which has the same distribution as the data which the system encounters over a long period of operation, which we call *perfectly distributed* test data. Similarly distributed test data is easy to obtain by randomly dividing available data into a training set and a test set. Using similarly distributed test data is a good way to test the generalization of a trained classifier. The problem is that the training data is not distributed in the same way as the data the system encounters over a long period in the field. The fielded system sees a vast variety of different signals and conditions while the training data tends to focus on a subset of these. Hence, performance on similarly distributed test data is generally not a good indicator of performance over a long period in the field. However, performance on perfectly distributed test data is a good indicator of performance in the field. The problem with perfectly distributed data is that it is very difficult to create such a set of data, especially one which is of sufficiently small size that we can train a detection algorithm on it.

Windowing can be viewed as the process of making the training database more and more like a filtered version of perfectly distributed data, where by *filtered* we mean that those examples which are redundant in terms of training a classifier (which constitute the vast majority) are omitted. Learning occurs as the training database gets closer and closer to this ideal and thus the trained classifier can handle correctly a larger range of signals and conditions. Figure 16 shows a qualitative graph of performance versus time for these three different kinds of data on which to measure performance. Of course, we do not have a true set of perfectly distributed test data; what we do instead is use all the data available to us (a relatively large amount) as a substitute. Notice the following about the graphs. First, the performance on the training set is for the most part monotonically decreasing. As the training set grows larger in both number of examples and variety of examples, the training problem becomes increasingly harder. Second, the performance on the similarly distributed test set becomes ever closer to the performance on the training set. This is because our training algorithm works with a fixed number of free parameters which are determined with ever greater statistical significance as the amount of training data grows larger. Third, the performance on the perfectly distributed test data consistently increases with time as the training data becomes more representative of perfectly distributed data. It is in this sense that our system is learning.

4.4. Threshold optimization: manual versus automated

There are two kinds of comparisons we can make between our machine learning system and humans. The first is a comparison between the detection performance of the automated system and that of a human analyst. In some sense this is the bottom line. The problem is that, as described in Section 1.2, the event detection rules which the learning system works on are a relatively small part of the full automated system. The event detection rules depend fully on the Low-Level Processor (LLP) to detect and characterize the narrowband

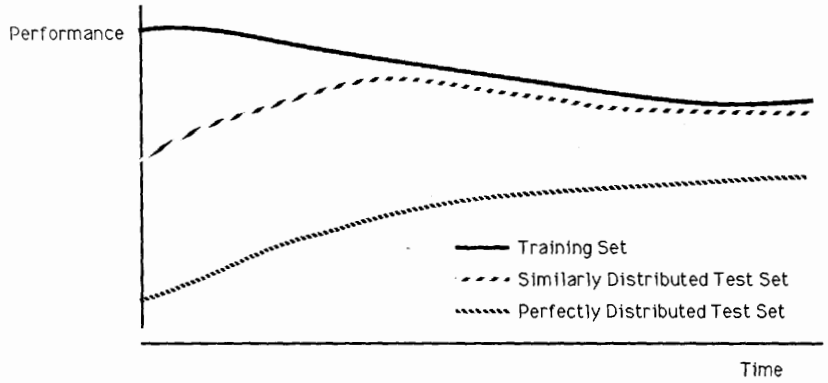


Figure 16. The evolution of performance with time.

signals. Not surprisingly, the LLP is a poor substitue for the human low-level visual system, generally producing a degraded version of the information present in the sonar image (although we are working on applying technology similar to that described here to improve the LLP (Montana & Davis, 1989; Montana, in press)). Comparisons between human and machine detection performance serve primarily to highlight the disparity between the LLP and human vision.

The second type of comparison is between machine learning of detection rules and manual tuning of these rules by human developers. The first release of our system contains rules which were tuned manually while the next release has rules generated by the learning system. The LLP remained functionally the same between releases. Hence, differences in the detection performance of these two releases indicate the improvements to our system of using machine learning rather than manual tuning.

When initially evaluating the benefits of the machine learning approach, we performed the following experiment. Take the data on which the rules from the first release were tuned. Continually perform the windowing procedure for a single event type on this data until no more examples are being added to the training database. Optimize the rule for detecting this event type multiple times using different values of R, the tradeoff between P_d and P_f. Evaluate the detection performance of this rule on the full set of data. The results are shown in Figure 17.

These results indicate first of all that the machine learning approach generated rules with better performance. By adjusting R appropriately, this approach could find an operating point which has both higher P_d and lower P_f than the hand-tuned rules. The second advantage of the machine learning approach is its ability to easily change its operating point along the ROC curve. As stated in Section 2.2, a goal for the second release was to raise the rules' P_d possibly at the expense of raising P_f. With the hand-tuning approach, this would have meant starting from scratch and reperforming a very labor-intensive task. In fact, it would have been harder the second time because higher P_d and higher P_f require the mental juggling of more examples. A third advantage of the machine learning approach not shown in the graph is that the rule packet was much smaller and thus generalized better

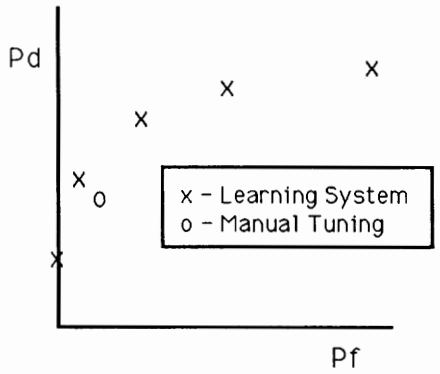


Figure 17. A comparison of the learning system with manual tuning.

to new data. Human developers tended to add new conditions and rules into the packet to handle individual examples, and as this practice continued the rule packet became large. A fourth big advantage of the machine learning approach is its ability to keep improving with experience. Human developers could only mentally juggle a relatively small amount of data before they became overloaded and hence could not continue to improve the rules (in the sense of performance on a perfectly distributed test set) after a certain point. This contrasts with the continual improvements generated by the learning system described in Section 4.3.

5. Additional benefits

Using the approach to learning described above, the system is able to successfully detect sonar events, improve its performance with experience, and trade off between high P_d and low P_f in a way consistent with system specifications. We now describe some additional benefits of our approach: multiple confidences, adaptable detection performance, and a tool for understanding the importance of each classification feature.

5.1. Multiple confidences

By altering the tradeoff between high P_d and low P_f, we can create rules which detect events with different levels of confidence. A rule whose thresholds optimize an error criterion with P_f weighted heavily compared to 1 - P_d (i.e., where the tradeoff constant R is large) will be relatively unlikely to call false alarms. Hence, any events detected by such a rule are true events with a high confidence. Alternatively, events detected by a rule with R small are true events with a lower confidence. In our run-time system, we include rules which look for the same types of events with different confidences. The confidence of a particular event is the highest confidence associated with a rule which detected it. We have already

discussed how this helps for data collection. More importantly, having a confidence associated with each detected event allows the higher-level rules a lot more flexibility in using these events to make other inferences.

5.2. Adaptation to varying conditions

Different environmental conditions dictate detection performance for the rules. As an example, consider the difference between high-traffic and low-traffic conditions. The former requires a much lower P_f to satisfy system performance requiring false alarm rates to be below a threshold. Another example is the effect of outside sources of information. Such outside information can cause us to expect to see events on certain sonar images; for these images, we can raise the P_d without losing confidence in our detections. These different requirements translate into different values for the rule thresholds. Using our optimization routine, we can create sets of threshold settings appropriate for each of these situations. As described in Section 1.2, all the rule packets appropriate for a particular situation are grouped into a rule set. These rule sets are disabled when the corresponding situation is not in effect and are enabled when the situation is in effect.

5.3. Classification features evaluation

As mentioned in Section 1.3, the system tuning process includes improvement of the statistics functions used to define classification features as well as the learning of the rule thresholds. The general form of rules makes it easy to determine the effects of each feature on the performance of a rule. When a rule does not fire, the features responsible for this are all those that are not within their corresponding thresholds. We have created a tool for evaluating the marginal contribution of a feature to a rule which works as follows: i) find all examples in the training database for which the rule fails to fire solely because of the feature (i.e., all examples which would be called events if the feature were ignored), ii) let N, the negative score, be the number of positive examples in this set, iii) let P, the positive score, be the number of negative examples in this set, iv) the net contribution of the feature is RP-N, where we recall that R is the tradeoff constant in the optimization function.

We interpret the results of this tool as follows. If the net contribution of a feature is negative, then the optimization has failed because it should have been able to set the thresholds in such a way as to effectively ignore the feature (i.e., $N = P = 0$) and thus do better. In the rare situations when this occurs, it is often the case that the developer has overly constrained the values for the corresponding threshold through his choice of range and step size. However, it also can be that the genetic algorithm has failed to find the global optimum and should be run again. If the net contribution of a feature for many runs over different training databases is always zero, then this is a worthless feature and should not be used in the rule. Note that because of the multimodal nature of the search space and the everchanging training database, the fact that a feature has zero contribution on one particular run does not merit discarding it.

6. Conclusion

Detection of events in sonar data is difficult due to the complexity, variability and clumpiness of the data. We have successfully employed a new type of empirical learning for detecting such events. It uses thresholded rules grouped into rule packets as its underlying decision structure. A genetic training algorithm optimizes the thresholds of these rules for performance on a training database of examples. Since genetic algorithms can be used to optimize with respect to arbitrary error criteria, we have the ability to explicitly dictate system performance specifications to the training algorithm. We have chosen to use an error criterion which is a linear combination of 1 - P_d and P_f with the relative weighting of these two terms used to specify the desired operating point along the ROC curve of the rule. We can use this capability to create multiple rule sets which detect events with different confidences or which are tuned for different conditions.

The learning process we have used employs windowing as a method for getting a handle on the huge and ever increasing amount of data passing through our system. Iterating through the windowing loop (of (i) gathering more examples for the training database and (ii) retraining) allows the rules to improve with experience.

Acknowledgments

Thanks are due to the following people: Dave Davis, for his tutelage in the field of genetic algorithms and the use of his genetic algorithm software; Steve Milligan, for suggesting the idea of rule threshold optimization and allowing me to work on it; Fred White, for writing code to support the optimization process; and Ken De Jong, for his comments, which have helped to greatly improve the paper.

References

Breiman, L., Friedman, J., Olshen, R., & Stone, C. (1984). *Classification and regression trees*. Monterey, CA: Wadsworth.

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

Goldberg, D. (1988). *Genetic algorithms in search, optimization, and machine learning*. Reading, MA: Addison-Wesley.

Gorman, R. & Sejnowski, T. (1988). Analysis of hidden units in a neural network trained to classify sonar targets. *Neural Networks*, Vol. 1, 75–89.

Marr, D. (1982). *Vision*. New York: W.H. Freeman and Company.

Michalski, R., Mozetic, I., Hong, J., & Lavrac, N. (1986). The multi-purpose incremental learning system AQ15 and its testing application to three medical domains. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 1041–1045).

Montana, D. & Davis, L. (1989). Training feedforward neural networks using genetic algorithms. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 762–767). San Mateo, CA: Morgan Kaufmann.

Montana, D. (in press). Automated parameter tuning for synthetic image interpretation. In L. Davis (Ed.), *The genetic algorithms handbook*.

Nii, H.P., et al. (1982). Signal-to-symbol-transformation: HASP/SIAP case study. *AI Magazine, 3*, 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 micro-electronic age.* Edinburgh University Press.

Quinlan, J.R. (1987). Generating production rules from decision trees. *Proceedings of the Tenth International Conference on Artificial Intelligence* (pp. 304–307).

Richardson, J., Palmer, M., Liepins, G., & Hilliard, M. (1989). Some guidelines for genetic algorithms with penalty functions. *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 191–197). San Mateo, CA: Morgan Kaufmann.

Rosenblatt, R. (1959). *Principles of neurodynamics.* New York: Spartan Books.

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

Schlimmer, J., & Granger, R. (1986). A case study of incremental concept induction. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 502–507).

Shapiro, A. (1987). *Structured induction in expert systems.* Maidenhead, U.K.: Addison-Wesley.

Syswerda, G. (1989). Uniform crossover in genetic algorithms. *Proceedings of the Third International Conference on Genetic Algorithms* (pp. 2–9). San Mateo, CA: Morgan Kaufmann.

Wirth, J. & Catlett, J. (1988). Experiments on the costs and benefits of windowing in ID3. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 87–99). San Mateo, CA: Morgan Kaufmann.